Web-Based Networked Music Performances via WebRTC: A Low-Latency PCM Audio Solution

Web-Based Networked Music Performances via WebRTC: a Low-Latency PCM Audio Solution*

Matteo Sacchetto,[1] Paolo Gastaldi,[3] Chris Chafe,[4] Cristina Rottondi,[1] and Antonio Servetti[2]
(matteo.sacchetto@polito.it) (paolo.gastaldi@studenti.polito.it) (cc@ccrma.stanford.edu) (cristina.rottondi@polito.it)
(antonio.servetti@polito.it)

[1]*Department of Electronics and Telecommunications, Politecnico di Torino, Italy*
[2]*Department of Control and Computer Engineering, Politecnico di Torino, Italy*
[3]*Department of Energy, Politecnico di Torino, Italy*
[4]*Center for Computer Research in Music and Acoustic, Stanford University, California, USA*

Nowadays, widely used videoconferencing software has been diffused even further by the social distancing measures adopted during the SARS-CoV-2 pandemic. However, none of the web-based solutions currently available support high fidelity stereo audio streaming, which is a fundamental prerequisite for networked music applications. This is mainly due to the fact that the WebRTC *RTCPeerConnection* standard or web-based audio streaming do not handle uncompressed audio formats. To overcome that limitation, we discuss an implementation of 16-bit PCM stereo audio transmission on top of the WebRTC *RTCDataChannel*, leveraging Web Audio and AudioWorklets. Results obtained with multiple configurations, browsers, and operating systems show that the proposed approach outperforms the WebRTC *RTCPeerConnection* standard in terms of audio quality and latency, which in our best case to date has been reduced to only 40 ms between two MacBooks on a local area network (LAN).

0 Introduction

Web-based audio/video (A/V) streaming platforms for videoconferencing services have become ubiquitous, in part thanks to their easy integration within the web environment. The majority of web-based videoconferencing solutions leverage Web Real-Time Communication (WebRTC) media streams [1], which represents the standard approach to peer-to-peer low-latency A/V streaming. The recent social distancing countermeasures imposed to mitigate the spreading of the SARS-CoV-2 pandemic have further fostered the extension of such platforms to Networked Music Performance (NMP) applications which support real-time musical interaction between musicians performing together from multiple geographical locations as if they were in the same room.

Unfortunately, media streams do not allow for fine-grained control over the latency introduced by A/V acquisition, processing and buffering, which is of pivotal importance for the perceived quality of experience in NMP applications. Indeed, to ensure the necessary synchronisation and interplay between participants who perform music together in real-time, the one-way Mouth-to-Ear (M2E) latency should ideally not exceed the 30 ms threshold [2] (the reader is referred to [3] for a thorough discussion on how such threshold is influenced by the genre, the tempo and the instrumental characterization of the musical piece).

For this reason, NMP platforms are typically conceived as standalone, native applications that run directly on an operating system of choice in order to benefit from fast access to system calls and customized software implementation. The vast majority of those tools, e.g. JackTrip [4], Soundjack [5], LOLA [6], and UltraGrid [7], exploit UDP transmission of uncompressed audio streams to minimize the M2E latency. These software programs allow settings for audio packet size and de-jitter buffering to strike the best trade-off between quality of the audio streaming and perceived latency.

Conversely, real-time communication based on WebRTC media streams offers limited configuration options and adopts compressed audio formats to limit the bitrate, at the price of introducing additional latency during the audio encoding/decoding process: such delay may reach 20 ms or more, as measured in experiments with the Aretusa NMP software [8]. To overcome the lack of configurability of WebRTC media streams, i.e., the *RTCPeerConnection*, an alternative option is to avoid their usage and to exploit instead the *RTCDataChannel*, which is usually adopted for non-multimedia data. The feasibility of such an approach

*To whom correspondence should be addressed, e-mail: antonio.servetti@polito.it

was first explored in 2017 by researchers at Uninett, Otto J. Wittner et al. [9]. However, the limited audio processing capability of the *ScriptProcessorNode* [10] introduced a significant amount of latency. The new AudioWorklet API [11] promises to consistently reduce such latency contribution, as it provides independent real-time threads and enables more efficient audio processing.

Inspired by such possibilities, in this study we implemented an alternative approach to peer-to-peer high quality and low-latency audio communication, which exploits both the WebRTC *RTCDataChannel* and the AudioWorklet API. We developed a WebRTC application named JackTrip-WebRTC as a sibling to the popular JackTrip software[1] for NMP over the Internet. The application is released as open source software on GitHub[2]. In the remainder of this paper, we present the low-latency application architecture, investigate its performance for several configurations, browsers, and operating systems and benchmark it against the traditional approach based on media streams.

A preliminary version of this study appears in [12]. With respect to [12], the novel contributions presented in this paper are:

- A new WebRTC implementation that adopts Shared Array Buffer (SAB) instead of the Message Channel (MC).
- A substantial scalability improvement with respect to the previous implementation, which enabled the connection of at most three peers, thus greatly limiting the usage in practical scenarios. The current version can instead scale up to tens of connected peers, provided that enough bandwidth and CPU power are available.
- An extensive performance assessment of the proposed implementation, including the comparison of its performance to that of widely available web-based audio communication platforms such as Jitsi and Google Meet.

The remainder of the manuscript is structured as follows: after an overview of the related literature in Section 1 and a brief introduction to the WebRTC architecture in Section 2, we describe our proposed solution for low-latency WebRTC communications in Section 3. Experimental results are discussed in Section 4 and Section 5, respectively for the overall M2E latency and for each layer of the audio chain, whereas Section 6 reports the assessment of the application jitter and scalability for multi-peer communications. Finally, Section 7 concludes the paper.

1 Related Work

The recent isolation imposed by the pandemic and the increased need of interacting and working remotely have fostered the adoption of simple and easy-to-use software tools for videoconferencing. In almost every modern web browser, those solutions leverage the WebRTC standard, which was jointly proposed by the World Wide Web Consortium (W3C) and by the Internet Engineering Task Force (IETF). WebRTC provides an open and royalty-free standard for real-time audio/video acquisition and peer-to-peer multimedia and data transmission. Nowadays, most of the services that were originally developed with proprietary protocols, such as Cisco's WebEx or Zoom, have a WebRTC alternative implementation that allows users to join directly from their browser without downloading any software [13].

In [14] the authors provide an extensive comparison of the performance of four videoconferencing applications (Zoom, Microsoft Teams, VoiceLessonsApp, and FaceTime) in terms of audio fidelity for the real-time transmission of musical content. By analyzing the introduced distortions in both time and frequency domain, they conclude that, though all the considered platforms introduce artifacts or noise, Zoom provides the highest fidelity to dynamics and spectral characterization of the streamed signals. Nevertheless, we should note that the web-based Zoom alternative that leverages the WebRTC implementation does not provide the same audio quality level.

To overcome the scalability limits of the P2P structure of WebRTC – that does not allow scaling up to hundreds of connected peers – in [15] the authors propose a system that efficiently extends such structure with synchronized mixing and broadcasting of the audio/video streams to large audiences with adequate real-time performance. [16] proposes an alternative to improve bandwidth efficiency and audio quality for speech communications with an adaptive bitrate switching algorithm that selects the most suitable bitrate and operating mode of the Opus codec[17], thus lowering the impact of bursty and random packet loss conditions.

The same authors in [18] highlight the limits of the WebRTC de-jitter buffer behavior in enabling low-latency audio communications, even in presence of negligible network delay: measurements show that WebRTC default applications may lead to latency levels that approach the ITU-T Rec. G.114 [19] thresholds. In order to significantly decrease the WebRTC communication latency, the authors in [9] envisioned bypassing the WebRTC standard algorithm for the de-jitter buffer and all the overhead introduced by the *RTCPeerConnection* channel.

The alternative investigated in that study is to rely on the *RTCDataChannel* that, although designed for data (and not media) communications, can be configured to exclude reliable and ordered delivery algorithms and to allow a UDP-like unreliable and unordered transmission. That implementation did not achieve the expected results because the support for efficient and timely management of audio data in the browsers was still in its infancy [20] and new improvements were still to be introduced [21][22], but it laid the foundations for the present work.

[1] https://www.jacktrip.org/

[2] The code repository is available on GitHub https://github.com/jacktrip-webrtc/jacktrip-webrtc/. The *main* branch contains the code of the JackTrip-WebRTC version presented in the conference paper [12] while the *experimental* branch contains the code of the enhanced version discussed in this paper.

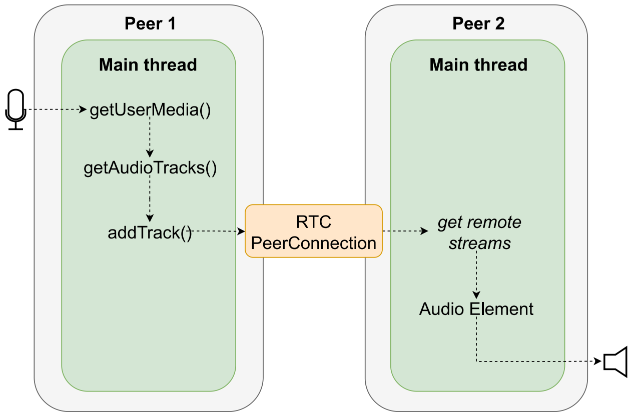


Figure 1: Traditional WebRTC application structure with the *RTCPeerConnection*.

2 Background

2.1 Architecture of a WebRTC videoconferencing application

The three major components of the architecture of a WebRTC videoconferencing application are the *MediaStream*, *RTCDataChannel*, and *RTCPeerConnection* objects. *MediaStream*, together with the *getUserMedia* method, is used to acquire real-time media streams, such as the device camera or the microphone, for rendering or further processing (e.g., by means of the Web Audio API). The *RTCDataChannel* is a transport mechanism used for sending arbitrary data such as control messages or files. It is implemented through the Stream Control Transmission Protocol (SCTP) [23], which is a low-latency, message-oriented, multi-streaming protocol based on UDP capable of providing also reliable, in-sequence transport of messages with congestion control and packet retransmission similarly to TCP. *RTCPeerConnection* is considered the main part of the WebRTC specification, as it enables audio and video communication between the peers. It is built on top of the RTP/SRTP protocol, but it also provides multimedia codecs, bandwidth management, and includes several signal processing algorithms.

In a WebRTC application, media streams play a key role, since they are used both for media acquisition/playback and for exchanging multimedia data through the network. Figure 1 depicts the typical WebRTC application structure.

At the sender, the web application uses the *MediaDevices.getUserMedia()* method to acquire data from a media input, i.e., the audio feed from the internal microphone or the input of a dedicated audio device, as a media stream. Note that the *getUserMedia()* method takes a *MediaStreamConstraints* object, whose properties are of fundamental importance to set up the media acquisition process and minimize the overall system latency (together with the properties that define the setup of the *RTCPeerConnection*), as we experimented and described in [12]. The *getAudioTracks()* method is then used to return an audio track from the *MediaStream* and the *RTCPeerConnection.addTrack()* method is finally used to add the new media track to the set of tracks that will be transmitted to the other peers. The *RTCPeerConnection* component contains all the logic to manage the real-time media stream delivery to the network.

An almost symmetric structure is implemented at the receiver peer where, when a new track is connected, a callback retrieves the remote media stream and attaches it to an HTML audio element for playback.

2.2 Limitations of Videoconferencing Tools in NMP Scenarios

Designed before the social distancing experienced during the SARS-CoV-2 pandemic, the WebRTC protocol is mainly optimized for speech-centered, turn-taking scenarios. The criteria that govern the selection of multimedia codecs and of the audio processing algorithms focus on preserving network bandwidth at the expenses of higher M2E latency and on preserving intelligibility at the expenses of audio fidelity.

In fact, although the WebRTC protocol could support any multimedia codec, the standardization process defined a limited set of formats to ensure compatibility. Among them, the frame-based perceptual codecs Opus [17] and G.711 [24] are mandatory. Conversely, the use of uncompressed PCM, which would ensure high-fidelity quality and introduce no algorithmic delay, is not supported. Moreover, as reported in Sec. 4, even with all the settings optimized to achieve the lowest latency, a WebRTC application based on the *RTCPeerConnection* still introduces too high delay, (60 ms in the best case), to be considered a suitable tool for NMP applications.

Another main drawback exhibited by the vast majority of videoconferencing applications, when adopted in NMP scenarios, is that adaptive noise suppression (ANS), echo cancellation (EC) and automatic gain control (AGC) are enabled by default. Such features are useful to increase the intelligibility of speech signals, but alter the naturalness of the sounds produced by musical instruments or by singers' voices, and introduce distortions in sound transients and intentional loudness dynamic variations. Furthermore, all such functionalities introduce non-negligible processing delays.

For some videoconferencing software products, it is difficult or even impossible to disable such features. As an example, Google Meet and Microsoft Teams do not offer the possibility to disable AGC. In Jitsi, disabling AGC is possible only by tinkering with the URL parameters, since there is no setting option directly available in the user interface. Skype introduced such possibility only very recently (previously, a modification in the Windows system registry was necessary to disable AGC).

To substantiate the above claims, using the same audio material provided in [14] online[3] we analyzed the audio transmitted by several videoconferencing applications. The

[3]The media files used in the article are available from https://www.ianhowellcountertenor.com/preliminary-report-testing-video-conferencing-platforms

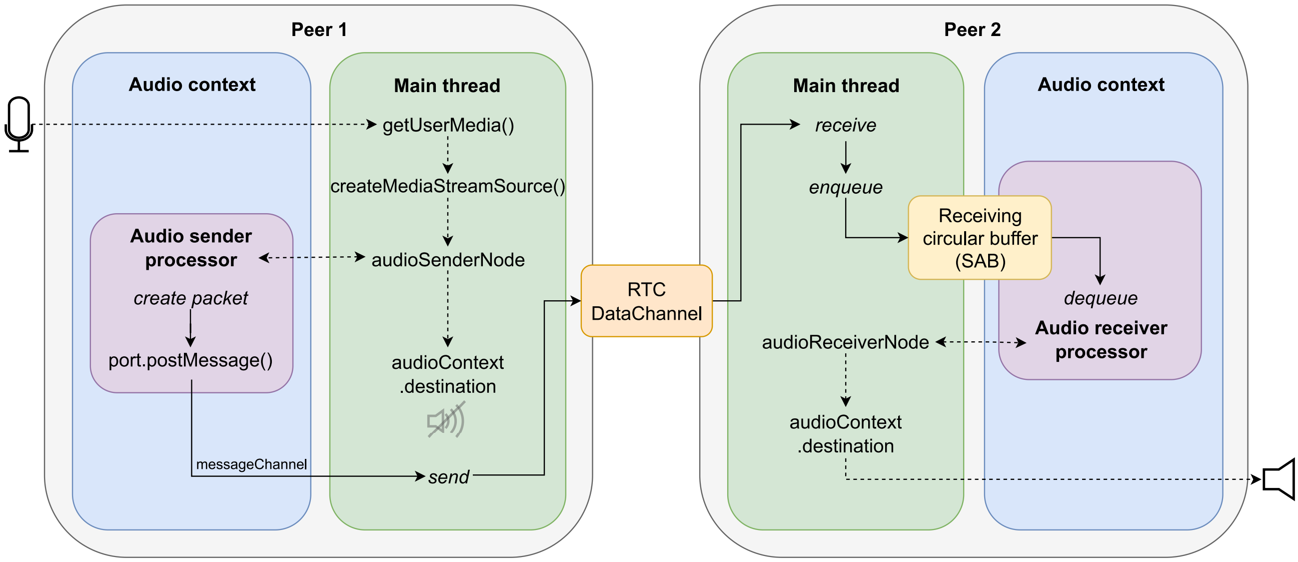


Figure 2: Custom application structure of the WebRTC application with the *RTCDataChannel* and the Shared Array Buffer (SAB) instead of the *RTCPeerConnection*.

obtained results are comparable to those reported in [14]: Figure 3 shows an example of the alteration of the amplitude envelope of a bowed violin, where the intensity of various segments of the excerpt has been completely reshaped by the AGC, thus altering the original signal. In addition, in several occurrences sustained notes by cello or violin were suppressed because misinterpreted as noise by the ANS algorithm.

Even when ANS, EC and AGC are disabled, all the videoconferencing applications still adopt some kind of audio compression to reduce the media bitrate, thus de-facto reducing the audio quality. Figure 3 also shows the comparison of the frequency content of the original signal with the compressed one obtained with Google Meet or Jitsi; Google Meet seems to preserve less accurately the quality of the signal, especially in the high frequency range. The same happens with all the other videoconferencing software except for Zoom, which provides the so-called "original audio" option that can transmit the audio with minimal alterations of its spectral content. However, we must note that the "original audio" option is available only in the native Zoom application, whereas it is not provided in the web-based client application[4].

Finally, the logic that automatically controls the bandwidth usage and the latency of the de-jitter buffer, (required to mitigate the variability in packet inter-arrival times) is designed to maximize speech quality at the expense of introducing additional latency (even up to 150 ms). Such latency can be tolerated in interactive speech communications, but substantially hinders remote music performances.

Figure 4 shows the results of several experiments with popular videoconferencing software: the measured M2E delays are in the range of 80-180 ms between two computers in the same local area network. Note that, when allowed by the software, we tested both the default configuration and a low-latency (LL) configuration where most of the audio processing features that increase the M2E delay were disabled[5]. Being based on WebRTC, Jitsi and Google Meet show similar latency figures, on the order of 120 ms, when the default configuration is used. In its low-latency configuration, Jitsi latency is reduced by about 30 ms, and almost the same reduction is obtained with Zoom. We note that Zoom presents the highest latency among all the applications because it always works in a client/server mode, thus relying on the transmission through a server on the Internet, instead than in a peer-to-peer mode with a direct connection between the devices.

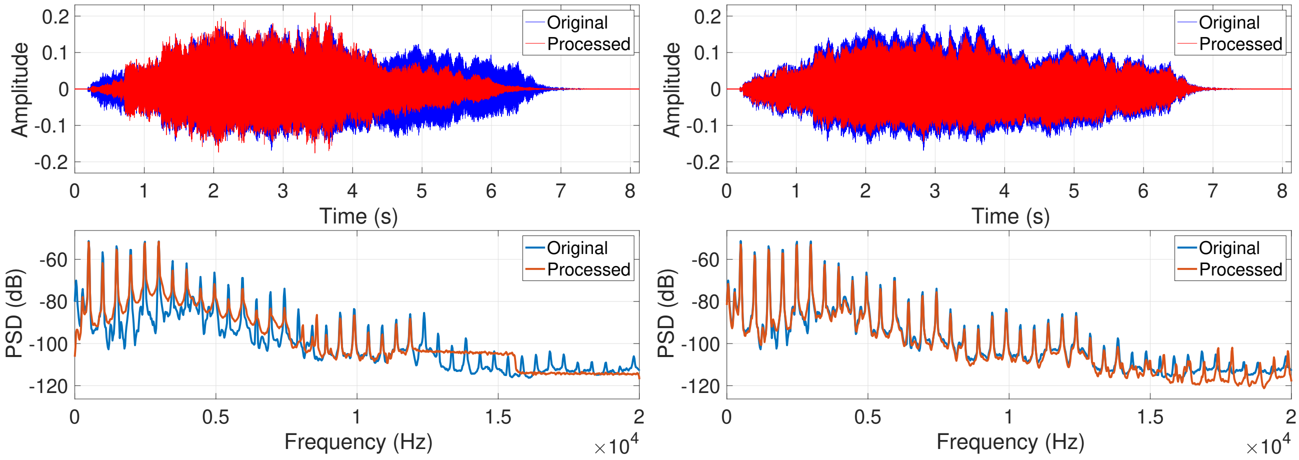
3 The proposed low-latency WebRTC application

3.1 Operational Principles

The structure of our proposed low-latency WebRTC application is depicted in Fig. 2. It is derived from the classical structure of a WebRTC application, which is reported in Fig. 1. Note that, in contrast with the popular NMP applications cited in Section 0 that, being native applications, can directly access the low level OS API to implement custom functionalities, a web application is forced to use only

[4]Comparison of Zoom Desktop with Zoom Web Client https://support.zoom.us/hc/en-us/articles/360027397692

[5]Jitsi allows disabling audio processing, automatic EC, ANS, AGC, high pass filtering by providing such information in the URL as reported in https://www.homepages.ucl.ac.uk/~rtnvrmp/JitsiStereo.html. Skype allows disabling ANS and AGC from the *audio & video* preference panel. Zoom has a special feature called *original audio* that disables all audio processing and improves audio coding as reported in https://blog.zoom.us/high-fidelity-music-mode-professional-audio-on-zoom/. Google Meet does not allow any customization, thus only the default configuration has been used.



(a) Google Meet (default with AGC, NS, AEC) (b) Jitsi (with AGC, NS, AEC disabled)

Figure 3: Distortions due to automatic audio gain (top) and perceptual codec compression (bottom) in Google Meet and Jitsi with Chrome. The blue line represents the original signal and the red one the processed signal, both in the time and frequency domains. The audio trace is a sample of a bowed violin playing quietly (-12 dB) the pitch B4.

the JavaScript API made available by the browser, for security reasons. It follows that, in our attempt to identify alternative WebRTC configurations with the aim of reducing latency, we can modify the application structure only if the JavaScript API provides us with alternative implementations for a given task.

In order to switch from an architecture based on the *RTCPeerConnection* channel to an architecture based on the *RTCDataChannel*, we have to resort to the Web Audio API and to the AudioWorklet interface. It is in fact necessary to tamper with the WebRTC *MediaStream* object in order to access the raw audio content and route it to and from the *RTCDataChannel*. In particular, two custom AudioNodes need to be implemented: one in the transmitter, to extract the raw audio from the *MediaStream* object

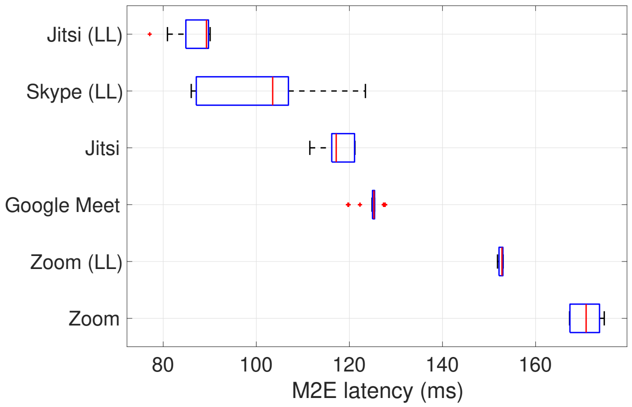


Figure 4: Mouth-to-Ear latency measured for common videoconferencing software, both native and web-based (Chrome v.98), between two MacBooks connected to the same Ethernet switch. When available, a low-latency (LL) software configuration without audio processing has been selected. For each configuration, ten sessions of the duration of one minute have been measured.

returned by the *getUserMedia* method, and one in the receiver to reverse the process.

More in detail, at the transmitter the *MediaStream* is fed as an audio source into the audio processing graph using the *createMediaStreamSource* method of the *AudioContext*, which creates a *MediaStreamAudioSourceNode* (Fig. 2), i.e., an audio node whose media is retrieved from the specified source stream. Such a node can then be chained with other nodes of the Web Audio API for further processing.

In this case, the required audio processing is limited to accessing the raw audio data and packetizing it for transmission. This task is performed by an AudioWorklet that consists of two objects: an *AudioWorkletNode* that allows the AudioWorklet to be connected with the other nodes of the Web Audio graph, and an *AudioWorkletProcessor* that will be in charge of executing the audio processing job. We name the custom AudioWorklet in the transmitter *AudioSender* (Fig. 2). It extracts uncompressed audio from the source *MediaStream* and creates the packets to be delivered on the *RTCDataChannel*. Low-latency requirements can be satisfied by the *AudioWorkletProcessor* because it is executed in a separated thread and called each 128 samples, i.e., every 2.6 ms at 48 kHz.

On the receiver side, the custom AudioWorklet is named *AudioReceiver*. Here the audio data retrieved from the *RTCDataChannel* needs to be managed by a proper de-jitter buffer before playout. The *AudioWorkletProcessor* is then in charge of transferring the audio frames from the de-jitter buffer to its outputs by means of its *process* method. Finally, the associated *AudioWorkletNode* is connected to the *AudioContext.destination* to enable the rendering of the audio on the selected output device.

However, since the *AudioWorkletNode* and the *AudioWorkletProcessor* are executed in different threads, the browser has to handle the overhead of exchanging audio data between the two of them. Two different methods can be used to transfer data between the main thread and the audio thread:

0. The **Message Channel**[6] interface which allows the two threads to exchange data using messages between one another through pipes with a *port* on each end.
1. The **Shared Array Buffer**[25] object which represents a pre-allocated memory block that can be accessed by both threads for fast data transfer.

In our preliminary implementation [12], we described an approach based on the common asynchronous Message Channel – already available in the web browsers from a long time – that was conceived to allow two separate scripts running in different browsing context of the same document to communicate between each other passing messages through a two-way pipe named *port*. This approach showed to be suboptimal for real-time audio because it suffers from high latency, and it requires repeated memory allocation for the copy of the data, thus strongly limiting the reliability and scalability of the system.

In the new implementation presented in this paper, we substitute the Message Channel with a Shared Array Buffer that lives between the page main thread and the AudioWorklet thread and that enables more efficient data communication between them. Here, a single (de-jitter) buffer is allocated once, and it is reused for each data exchange, since it can be accessed from both threads with proper coordination, as illustrated in [22].

For multi-thread synchronization, Atomics[7] methods are used to ensure correct concurrent access to the shared buffer. On one side, in the main thread, the *enqueue* procedure consists of *mixing* every audio frame received from each peer into the buffer [26]. This operation is implemented with a simple addition in the appropriate position, and it is performed with the *Atomics.add* method. The position is given by the current frame index and the peer offset. The peer offset is computed when the first frame from that peer is received and it is equal to the current read position minus the de-jitter offset[8] (see [27] for a description of the de-jitter buffer logic). On the other side, in the AudioWorklet thread, the *dequeue* procedure consists of *extracting* successive audio frames from the buffer for playout. The reading position is always incremented by one and the values of the current frame are substituted by zeros so that in case of lost frames a silence is played out. These operations can be performed in a single instruction by the *Atomics.exchange* method.

[6]https://developer.mozilla.org/en-US/docs/Web/API/MessageChannel

[7]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Atomics

[8]Since the de-jitter buffer is a circular buffer, every position is computed *modulo* its size

4 Latency Measurements

4.1 Experimental Settings

To measure the M2E latency, we set up a testing environment where we are able to accurately quantify the delay between the sound acquired by the sending input device and the sound emitted by the receiving output device. In order to monitor and better control the network delay, we perform the measurements between two different devices (peers) connected throughout a wired connection to the same switch of a local area network, as shown in Fig. 5.

Moreover, to exclude behavioral differences due to the use of different audio devices, all measurements use the same USB audio interface: the Behringer UMC404HD. This choice is also motivated by the fact that this platform is targeted at professional musicians, and so we consider the use of an external audio interface mandatory. Additionally, measurements performed in a non professional setting, using the internal audio card of the devices, have already been presented in [12]. Measurements are collected using custom developed software running on an external device which is calibrated every time to avoid the introduction of any processing latency. This allows to accurately measure the whole audio chain delay, including the delay introduced by the audio interfaces themselves. The measurement device emits an impulsive sound and records the time elapsed between the peak value of the emitted sound and the corresponding peak of the received one.

As reported in [12], the first crucial step in order to reduce the overall latency is to appropriately configure the *getUserMedia* call. By disabling all the additional processing performed on *MediaStreams*, i.e., AGC, ANS and EC, it is possible to decrease the latency by tens of milliseconds. Both the *RTCPeerConnection* and *RTCDataChannel* based solutions may benefit from this latency reduction. For this reason, all the measurements of this work are performed with the settings as in Listing 1, i.e., *autoGainControl*, *echoCancellation*, *noiseSuppression* set to `false`, and *latency* set to `0`.

Listing 1: Low-latency constraints for WebRTC *getUserMedia*.

```
const constraints = {
  video: false,
  audio: {
    autoGainControl: false,
    echoCancellation: false,
    latency: 0,
    noiseSuppression: false,
    sampleRate: 48000,
  }
};
const mediaStream =
  navigator.mediaDevices.getUserMedia(constraints);
```

All measurements are performed on three different operating systems: macOS (v10.14.6 and v12.2.1) with Core Audio, Windows 10 (v20H2) with Windows Audio, and Ubuntu Linux (20.04, kernel 5.15) with PulseAudio[9]. Win-

[9]All three OSes have been used with the default system setup. However, further reduction in latency may be achieved with a custom setup, in particular with Linux.

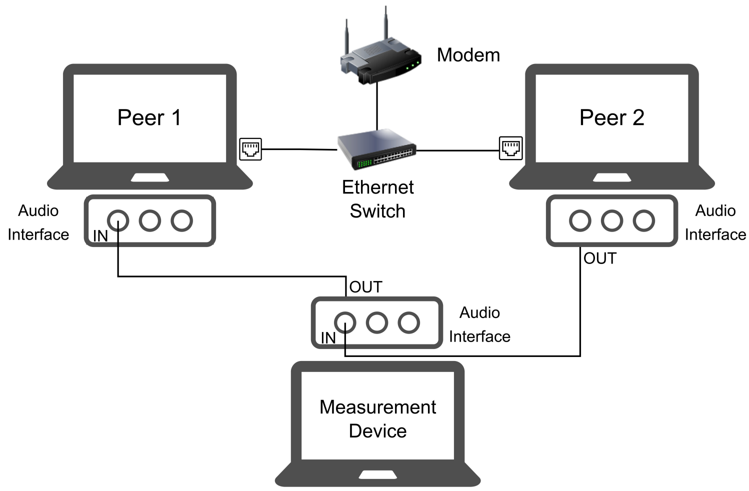


Figure 5: M2E measurement process. The measurement device computes the delay as the time difference between the time it emits the test signal and the time it receives it back from the full audio chain, which includes the audio interfaces (here represented as the rectangular objects), the peers and the network transmission between them.

dows and Linux are running on two different machines: an HP ProBook 430 G8 with an i7-1165G7 (4 cores / 8 threads) processor and 16 GB LPDDR4 3200 MHz RAM and a custom-built PC with an i9-10900 processor with 16 GB of DDR4 2133 MHz RAM. MacOS was running on two different MacBooks: a MacBook Pro 2018 with an i7-8559U (4 cores / 8 threads) processor and 16 GB LPDDR3 2133 MHz RAM, and a MacBook Pro 2020 with an i5-8257U (4 cores / 8 threads) processor and 8 GB LPDDR3 2133 MHz RAM. For each OS, we tested the latest version of the two major web browsers that currently support *AudioWorklets*: Chrome v.98 and Firefox v.97.

4.2 Overall Latency

Measurements in Fig. 6 show the comparison between the *RTCPeerConnection* and the *RTCDataChannel* implementations. Each row summarizes the results of ten sessions of the duration of one minute and the box plot represents the maximum, 75th percentile, median, 25th percentile, and minimum values.

Interestingly, the approach with the *RTCDataChannel* shows a significantly lower M2E delay with respect to the architecture with the *RTCPeerConnection*, with latency as low as 40 ms on Firefox on macOS. This result, that already includes a de-jitter buffer of about 20 ms – that may suffice in a typical network scenario – is very close to the one that could be achieved by NMP tools[10]. On the contrary the performance on other browsers and OSes is not so appealing, mostly because, at the moment, no browser can be easily configured to directly exploit the audio card low-latency (ASIO) drivers, as it is done by the native NMP applications. Nevertheless, the measurements reveal that the adoption of the AudioWorklet and the *RTCDataChannel* for low-latency audio communication may be more effective than the actual implementation of the WebRTC *MediaStream* that suffers from the algorithmic delay of audio coding and, mostly, from its propensity for using large delays in the de-jitter buffer, that, in the reported experiments, have been measured in the order of 20 to 60 ms [18].

[10]See the JackTrip Visual Studio FAQ at https://help.jacktrip.org/hc/en-us/articles/360055332753-Virtual-Studio-Frequently-Asked-Questions-FAQ- (accessed on March 17, 2022).

5 Audio Chain Latency Assessment

In this section we report further measurements to investigate how each part of the application audio chain contributes to the overall delay. From the architecture depicted in Fig. 2 we can decompose the application in three main layers:

A. The **input/output layer** that corresponds to the elements that are directly responsible for exchanging data with the audio device, i.e., *getUserMedia* and *audioContext.destination*.
B. The **processing layer** that includes the *AudioWorklets* and that moves audio data from the *MediaStream* to the main thread and vice versa. Here, in the receiver, a de-jitter buffer (that introduces additional latency) is implemented to compensate for network delay variance.
C. The **network layer** that corresponds to the *send* and *receive* procedures that responsible for exchanging data with the *RTCDataChannel* thus managing the transmission and reception of each audio frame.

We tested five *ad hoc* setups of the application. Figure 7 presents the results of the M2E latency assessments for different combinations of browsers and operating systems. As described in Sec. 4, to simulate a professional setup, we used an external audio card instead of the internal microphone and speakers. With respect to the results presented in [12] we notice that the adoption of an external audio card reduces by about 20 ms the latency on macOS. On the con-

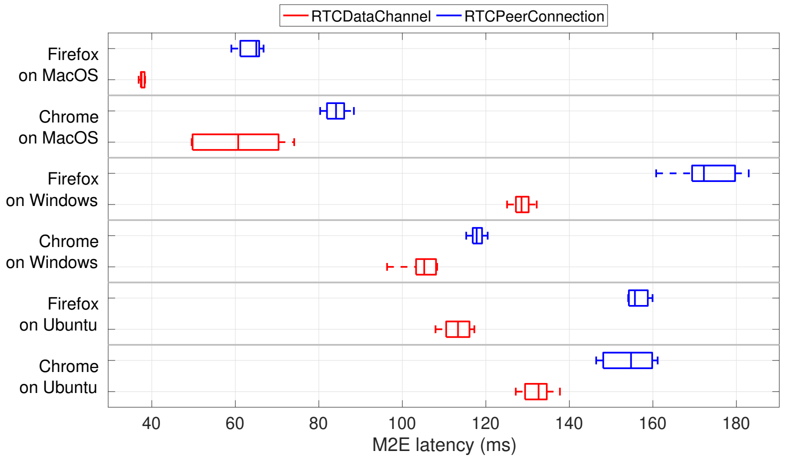


Figure 6: Mouth-to-Ear latency measurements with different browsers and OSes for the two approaches based on the *RTCPeerConnection* and the *RTCDataChannel*.

trary, we record a worse performance in the other OSes, likely due to the fact that, at the moment, other operating systems do not optimize browser access to the audio cards.

Setup A represents the simplest I/O chain that is possible to implement in a browser. Here the input layer has been directly connected to the output layer to test the baseline latency that any audio application implemented in a web browser may suffer from for just capturing and playing out the unprocessed audio, i.e., avoiding processing and networking. In this setup both browsers achieve the lowest latency on macOS, a value that is well below the one achieved on the other operating systems: about 30 ms for Chrome and close to 15 ms for Firefox. Firefox is faster than Chrome on macOS and Ubuntu, but not on Windows.

Setups B0 and B1 use both the I/O layer and the processing layer, but not the network layer: audio is captured, processed by the *AudioSender* and *AudioReceiver AudioWorklets*, and then played out within the same browser page. In this setup we must consider two implementations, one with the MC, B0, and one with the SAB, B1. As shown in Fig. 7, both implementations suffer from about the same latency. We should note, anyway, that the increase in latency with respect to the setup A is almost completely due to the presence of a de-jitter buffer in the receiver, which is needed to compensate the system delay jitter, thus allowing for loss-free playback. A buffer of eight audio frames is used, that accounts for a delay of 21.34 ms at 48 kHz.

Finally, setup C0 and C1 use the complete audio chain and connect two applications on two different devices, as in a real scenario. Here, the devices are connected on the same 1 Gbps LAN with an Ethernet cable in order to limit the effect of network delay (and its variation). For this setup, in the majority of the scenarios we note a small increase in the delay, but in two scenarios it happens that the delay is instead lower than the delay in setups B0 and B1. Given the high variability of the latency measurements and the negligible impact of the network (way below one millisecond), from these results we can assume that in this scenario also the overhead of the processing and network layer is very modest. The measurements confirm that the *RTCDataChannel* can provide timely audio delivery even if designed for data, and not audio, transfer. At the same time, the artifice of using AudioWorklet to divert audio from the media stream to the main thread and vice versa does not significantly impact the performance of the system.

From the analysis of the contribution of each single layer of the software application to the overall M2E latency, it is clear that the main drawback in implementing a NMP tool in a web browser is due to the latency in the real-time acquisition (and reproduction) of the audio data that accounts for the vast majority of the overall M2E delay. This result represents a great limitation for every low-latency real-time audio processing application that would be implemented in a web browser, not only for NMP applications. However, the performance of Firefox on macOS, that has a latency close to 15 ms, is not significantly far from the one achievable by native NMP applications, that is in the order of 5–10 ms at best.

6 Application Robustness

6.1 Jitter Measurements

All the experiments discussed so far have been performed with a de-jitter buffer of eight audio frames (21.34 ms). However, the length (and delay) of the buffer can be set to different values, depending on the CPU load and the network conditions. Since varying delays can be introduced at different stages of the audio transmission chain, in the following we report additional measurements of both the acquisition and the transmission stages that illustrate the real-time characteristics of the system. Note that, in order to have the same time reference at both transmitter and receiver sides, the two peers are running on the same device, in two tabs of the same browser.

By means of the high precision timestamp API available in the browser[11], we traced the arrival time of each audio frame at the *AudioWorkletNode*, both in the transmitter and

[11] https://developer.mozilla.org/en-US/docs/Web/API/Performance/now

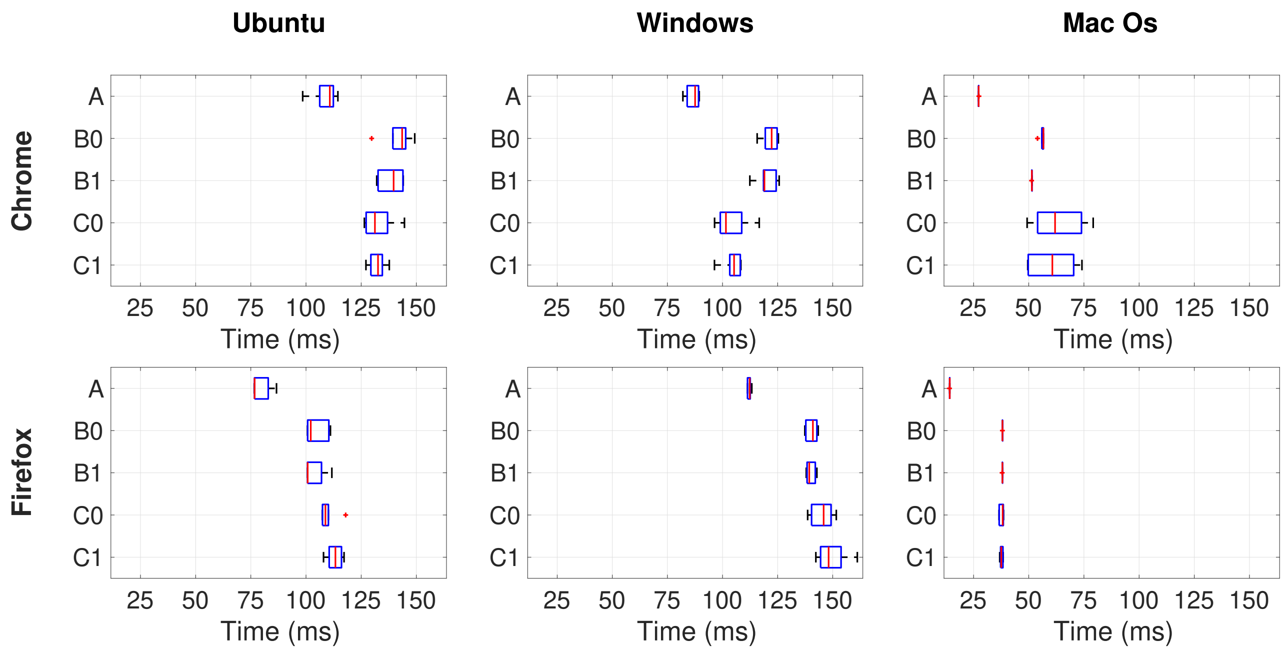


Figure 7: Mouth-to-Ear latency measurements for different setups of the audio chain with Chrome and Firefox on different operating systems. Labels correspond to the setups (A, B, C) described in Sec. 5 either with the MC (0) or the SAB (1).

in the receiver peer (those are the first places at both ends where we can trace the frame timestamps). The acquisition chain measurement includes only the jitter introduced by the *getUserMedia* method on the input device. The transmission chain measurement includes also the jitter due to the *RTCDataChannel* that is supposed to be slightly higher (even if the measurement is performed on the same computer on the loopback device).

Figure 8a shows traces of the Inter-Frame Delay Variation (IFDV) both after frame acquisition at the sender and after frame reception at the receiver. The two peers are on the same browser running Chrome or Firefox under low CPU load (only the application and some system and user background services). In a low jitter setup the IFDV value should be close to zero. Comparing the acquisition and reception traces, as expected, we see larger variation in the IFDV of the latter. This is also confirmed by Fig.8b, that reports the histogram of the probability density function for the two IFDV traces measured using Firefox on macOS. It is important to notice that with Chrome there are two most frequent series of values, +2.67 and -2.67 ms, that means that the frames are handled in pairs. This behavior can however appear in both browsers when the CPU load increases.

6.2 Scalability Evaluation

The proposed solution has been tested to verify its scalability, both with the MC and with the SAB implementation, when multiple peers are interconnected. Even if the performance, with two peers, is the same in terms of latency, as shown in Fig.7, and no losses are present, we experience a rapid reduction in performance for the MC implementation as soon as the number of connected peers increases. On the contrary, the SAB implementation does not report any loss in any of the audio streams even with as much as twenty concurrent peers.

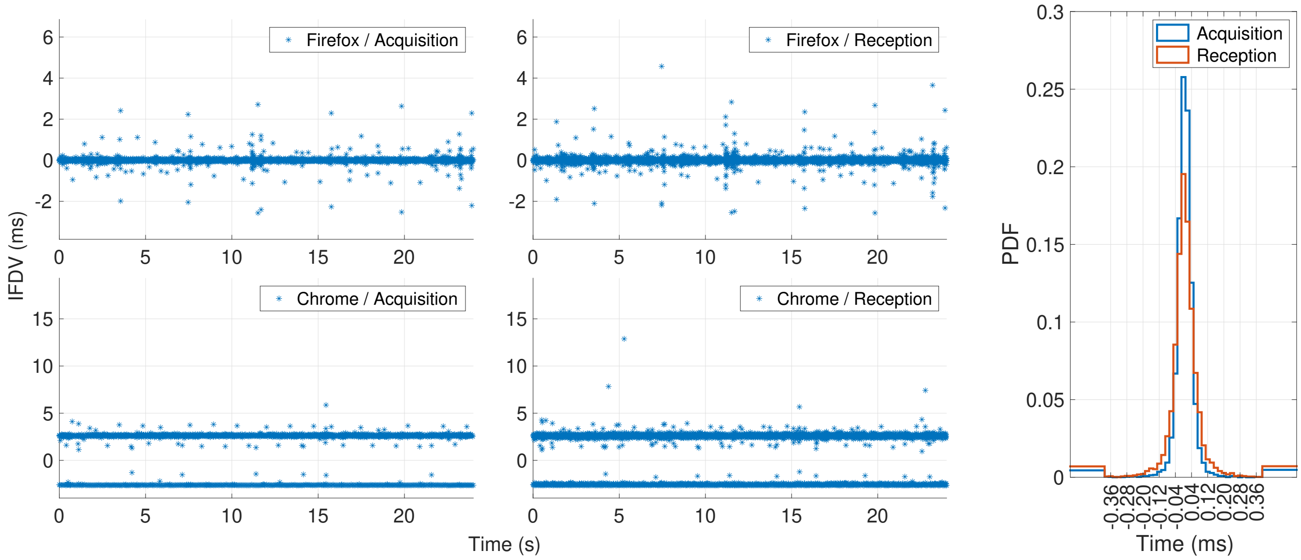
Figure 9 presents the results of the measurements in term of audio frame losses versus the number of concurrent peers (two sessions of one minute have been performed for each scenario and the average loss is reported). Here the lost frames are indeed *late losses*, that is frames that have been received too late for playback (despite the 21.34 ms de-jitter delay). Also in this case macOS shows better results than Windows with both the browsers, and Firefox outperforms Chrome in both the OSes.

The combination of the AudioWorklet, the SAB and the *RTCDataChannel*, has proved to efficiently handle both the high bit-rate (1,536 kbps per peer) required for the transmission of two uncompressed 16 bit PCM audio channels at 48 kHz and the high frame rate (375 frames per second) given by using small audio frames of 128 samples each.

7 Conclusions

In the time of SARS-CoV-2 restrictions we have seen a widespread adoption of Internet software for remote music performance by conservatories, music schools, and even professional musicians. Some of these users adopted proper NMP tools, but many of them relied on videoconferencing applications that are far simpler to use, in particular if web-based. However, all the current web-based audio communication software suffers from high latency and constrained quality because the WebRTC media stream (*RTCPeerConnection*) is targeted to speech interaction and its API does not allow the programmer to customize its behavior.

To the extent of our knowledge, this is the first work that presents a web-based solution (i.e., based on Web Audio



(a) Inter-Frame Delay Variation (Firefox and Chrome on macOS)

(b) Inter-Frame Delay Variation Probability Density Function (Firefox on macOS)

Figure 8: Inter-Frame Delay Variation (IFDV) with Chrome and Firefox on macOS at the sender, after acquisition from the input device, and at the receiver, after reception from the network layer. Both sender and receiver are on the same device in order to have the same time reference.

and WebRTC) for NMP with the quality of uncompressed stereo PCM audio and with far reduced M2E latency with respect to the state of the art. On one side, the results presented in this paper show that is possible to achieve the desired audio quality, robustness, and scalability with such an implementation. On the other side, the achieved M2E latency results to be just above the requirements of NMP applications only on macOS with the Firefox web browser; other scenarios still present too high delay. Nevertheless, the current implementation may be used in some scenarios where pristine audio quality is pivotal, such as remote individual music lessons or performances with loose musical interplay.

Investigation of the different layers of the audio communication chain reveal that quite a high portion of the overall latency is due to the audio acquisition and reproduction layers that are implemented in the native code of the web browser and are not under the control of the JavaScript programmer. Thus, we advocate for reduction of browser latency in audio I/O and inclusion of uncompressed audio in the *RTCPeerConnection* API with the possibility to customize media streams for music applications.

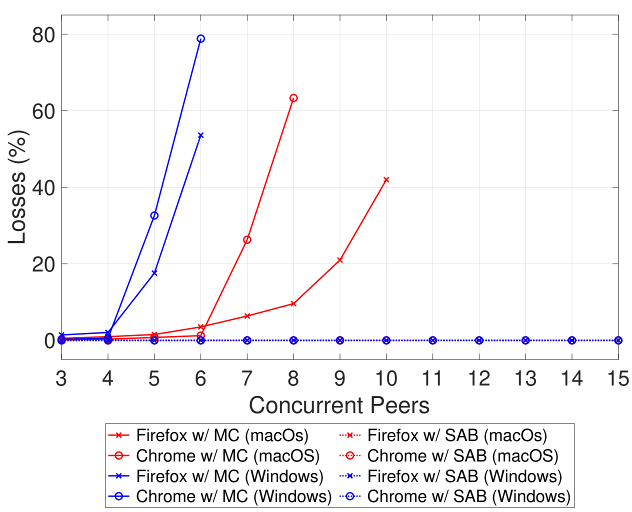


Figure 9: Percentage of lost audio frames, i.e., discarded because too late, as a function of the number of concurrent peers, when MC or SAB are used.

8 ACKNOWLEDGMENT

This work has been funded in part by the Italian Ministry of University and Research (MUR), Programme FISR COVID 2020 (project HiFiReM - High Fidelity system for Remote Music teaching with synchronised and collaborative live concert capabilities, cod. FISR2020IP_01206).

9 Bibliography

[1] H. Boström, J.-I. Bruaroey, and C. Jennings, “WebRTC 1.0: Real-Time Communication Between Browsers,” W3C (2020 Dec.), https://www.w3.org/TR/webrtc/.

[2] C. Rottondi, C. Chafe, C. Allocchio, and A. Sarti, “An Overview on Networked Music Performance Technologies,” *IEEE Access*, vol. 4, pp. 8823–8843 (2016 Dec.), https://doi.org/10.1109/ACCESS.2016.2628440.

[3] C. Rottondi, M. Buccoli, M. Zanoni, D. Garao, G. Verticale, and A. Sarti, “Feature-Based Analysis of the Effects of Packet Delay on Networked Musi-

cal Interactions," *Journal of the Audio Engineering Society*, vol. 63, no. 11, pp. 864–875 (2015 Nov.), https://doi.org/10.17743/jaes.2015.0074.

[4] 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 (2010 Nov.), https://dx.doi.org/10.1080/09298215.2010.481361.

[5] A. Carôt and C. Werner, "Distributed Network Music Workshop with Soundjack," in *Proceedings of the 25th Tonmeistertagung* (Leipzig, Germany) (2008).

[6] C. Drioli, C. Allocchio, and N. Buso, "Networked Performances and Natural Interaction via LOLA: Low Latency High Quality A/V Streaming System," in *Proceedings of the 2nd Intl. Conf. on Information Technologies for Performing Arts, Media Access, and Entertainment*, pp. 240–250 (Porto, Portugal) (2013 Apr.).

[7] P. Holub, J. Matela, M. Pulec, and M. Šrom, "UltraGrid: Low-Latency High-Quality Video Transmissions on Commodity Hardware," in *Proceedings of the 20th ACM International Conference on Multimedia*, pp. 1457–1460 (Nara, Japan) (2012 Oct.).

[8] K. Tsioutas, G. Xylomenos, and I. Doumanis, "Aretousa: A Competitive Audio Streaming Software for Network Music Performance," presented at the *146th Convention of the Audio Engineering Society* (2019 Mar.), paper 10201.

[9] O. J. Wittner, "Live Audio and Video over WebRTC's DataChannel," http://in-legacy.uninett.no:8080/live-audio-and-video-over-webrtcs-datachannel/ (accessed Dec. 18, 2020).

[10] P. Adenot and H. Choi, "Web Audio API," W3C (2020 Jun.), https://www.w3.org/TR/webaudio/.

[11] H. Choi, "Audioworklet: the Future of Web Audio," in *Proceedings of the 44th International Computer Music Conference* (Daegu, South Korea) (2018 Aug.).

[12] M. Sacchetto, A. Servetti, and C. Chafe, "JackTrip WebRTC: Networked Music Experiments in a Web Browser," in *Proceedings of the 6th International Web Audio Conference (WAC)* (Barcelona, Spain) (2021 Jul.).

[13] N. Blum, S. Lachapelle, and H. Alvestrand, "WebRTC: Real-Time Communication for the Open Web Platform," *Communications of the ACM*, vol. 64, no. 8, p. 50–54 (2021 Aug.), https://doi.org/10.1145/3454122.3457587.

[14] J. Glasner, I. Howell, K. Gautereaux, N. Perna, C. Ballantyne, and T. Nestorova, "Preliminary Report: Comparing the Audio Quality of Classical Music Lessons Over Zoom, Microsoft Teams, VoiceLessonsApp, and Apple FaceTime," https://www.ianhowellcountertenor.com/preliminary-report-testing-video-conferencing-platforms (accessed Mar. 25, 2020).

[15] D. Tang and L. Zhang, "Audio and Video Mixing Method to Enhance WebRTC," *IEEE Access*, vol. 8, pp. 67228–67241 (2020 Apr.), https://doi.org/10.1109/ACCESS.2020.2985412.

[16] M. Alahmadi, P. Pocta, and H. Melvin, "An Adaptive Bitrate Switching Algorithm for Speech Applications in Context of WebRTC," *ACM Transaction on Multimedia Computing, Communications, and Applications*, vol. 17, no. 4, pp. 133:1–21 (2021 Nov.), https://doi.org/10.1145/3458751.

[17] J. Valin, G. Maxwell, T. Terriberry, and V. Kos, "High-Quality, Low-Delay Music Coding in the Opus Codec," presented at the *135th Convention of the Audio Engineering Society* (2013 Oct.), paper 8942.

[18] O. Komperda, H. Melvin, and P. Pocta, "A Black Box Analysis of WebRTC Mouth-To-Ear Delays," *Communications - Scientific Letters of the University of Zilina*, vol. 18, no. 1, pp. 11–16 (2016 Feb.), http://dx.doi.org/10.26552/com.C.2016.1.11-16.

[19] ITU-T Rec. G.114, "One-way Transmission Time," ITU (2003 May).

[20] P. Adenot, "Web Audio API vs. Native: Closing the Gap," presented at the *1st International Web Audio Conference* (2015 Jan), https://medias.ircam.fr/x2af2f6.

[21] P. Adenot, "Web Audio API vs. Native, Closing the Gap, Take 2," in *Proceedings of the 5th International Web Audio Conference (WAC)*, pp. 178–179 (Trondheim, Norway) (2019 Dec.).

[22] P. Adenot, "Introduction to Lock-Free Programming on the Web with AudioWorklet," https://www.youtube.com/watc g4 (accessed Apr. 5, 2020).

[23] R. Stewart, "Stream Control Transmission Protocol (RFC 4960)," IETF (2007 Sep.), https://tools.ietf.org/html/rfc4960.

[24] ITU-T Rec. G.711, "Pulse Code Modulation (PCM) of Voice Frequencies," ITU (1988 Nov.).

[25] ECMA International, "ECMAScript 2017 Language Specification - SharedArrayBuffer Objects," (2017), https://www.ecma-international.org/ecma-262/8.0/#sec-sharedarraybuffer-objects.

[26] S. Chandra, S. Kumar, and B. Prasad, "Audio Mixer for Multi-Party Conferencing in VoIP," in *Proceedings of the 3rd IEEE Intl. Conf. on Internet Multimedia Services Architecture and Applications (IMSAA)*, pp. 1–6 (Bangalore, India) (2010 Jan.).

[27] ITU-T Rec. G.1021, "Buffer Models for Development of Client Performance Metrics," ITU (2012 Apr.).

THE AUTHORS

Matteo Sacchetto

Paolo Gastaldi

Chris Chafe

Cristina Rottondi

Antonio Servetti

Matteo Sacchetto is a PhD student at Department of Electronics and Telecommunications of Politecnico di Torino (Italy). He received his Master's Degree "cum laude" in Computer Engineering from Politecnico di Torino in 2020, with the Master thesis "JackTrip-WebRTC - Networked Music Performance with Web Technologies". His main research topic is networked music performance, but his interests also include WebRTC and Web Audio.

●

Paolo Gastaldi received the Master's Degree in Computer Engineering from Politecnico di Torino (Italy) in 2021. His Master thesis focused on technological optimizations and enhancements of JackTrip-WebRTC. In addition to engineering, he is a double bass student at Conservatorio di Torino.

●

Chris Chafe is a composer, improvisor, and cellist, developing much of his music alongside computer-based research. He is Director of Stanford University's Center for Computer Research in Music and Acoustics (CCRMA). In 2019 he was International Visiting Research Scholar at the Peter Wall Institute for Advanced Studies, The University of BritishColumbia;Visiting Professor at the Politecnico di Torino; and Edgard-Var'eseGuest Professor at the Technical University of Berlin.At the Institut deRecherche et-Coordination Acoustique/Musique (Paris) and The Banff Centre (Alberta), he has pursued methods for digital synthesis, music performance, and real-time internet collaboration. CCRMA's JackTrip project involves live concertizing with musicians the world over. Chafe's works include gallery and museum music installations, which are now into their second decade with "musifications" resulting from collaborations with artists, scientists, and medical doctors.Recent work includes the Earth Symphony, the Brain Stethoscope project (Gnosisong), Polar-Tide for the 2013 Venice Biennale, Tomato Quintet for the TransLife Media Festival at the National Art Museum of China, and Sun Shot played by the horns of large ships in the port of St. John's, Newfoundland.

●

Cristina Rottondi is Associate Professor with the Department of Electronics and Telecommunications of Politecnico di Torino (Italy). Her research interests include optical networks planning and networked music performance. She received both Bachelor's and Master's Degrees "cum laude" in Telecommunications Engineering and a Ph.D. in Information Engineering from Politecnico di Milano (Italy) in 2008, 2010, and 2014 respectively. From 2015 to 2018 she had a research appointment at the Dalle Molle Institute for Artificial Intelligence in Lugano, Switzerland. She is co-author of more than 80 scientific publications in international journals and conferences. She served as Associate Editor for IEEE Access from 2016–2020 and is currently Associate Editor of the IEEE/OSA Journal of Optical Communications and Networking. She is co-recipient of the 2020 Charles Kao Award, three best paper awards (Conference of Open Innovations Association (FRUCT)–International Workshop on the Internet of Sounds 2020, International Conference on Design of Reliable Communication Networks 2017, and IEEE Green Computing and Communications Conference 2014), and one excellent paper award (International Conference on Ubiquitous and Future Networks 2017).

●

Antonio Servetti has been an Assistant Professor with the Department of Control and Computer Engineering of the Politecnico di Torino (Italy) since 2007. He received M.S. and Ph.D. degrees in Computer Engineering from the Politecnico di Torino in 1999 and 2004 respectively. In 2003 Dr. Servetti was a Visiting Scholar supervised by Prof. J.D. Gibson at the Signal Compression Laboratory of the University of California, Santa Barbara, where he worked on selective encryption for speech transmission over packet networks.His research focuses on speech/audio processing, multimedia communications over wired and wireless packet networks, and real-time multimedia network protocols. With the advent of video and audio support in HTML5, his interests also include multimedia Web applications, WebRTC, Web Audio, and HTTP adaptive streaming.