JackTrip-WebRTC: Networked music experiments with PCM stereo audio in a Web browser

Matteo Sacchetto
Politecnico di Torino
Turin, Italy
matteo.sacchetto@polito.it

Antonio Servetti
Politecnico di Torino
Turin, Italy
antonio.servetti@polito.it

Chris Chafe
CCRMA Stanford University
Stanford, California
cc@ccrma.stanford.edu

ABSTRACT

A large number of web applications are available for videoconferencing and those have been very helpful during the lockdown periods caused by the COVID-19 pandemic. However, none of these offer high fidelity stereo audio for music performance, mainly because the current WebRTC RTCPeerConnection standard only supports compressed audio formats. This paper presents the results achieved implementing 16-bit PCM stereo audio transmission on top of the WebRTC RTCDataChannel with the help of Web Audio and AudioWorklets. Several measurements with different configurations, browsers, and operating systems are presented. They show that, at least on the loopback network interface, this approach can achieve better quality and lower latency than using RTCPeerConnection, for example, latencies as low as 50-60 ms have been achieved on MacOS.

1. INTRODUCTION

We are witnessing a large adoption of web based audio/video communication platforms that run in a web browser and can be easily integrated with the Web environment. Most of these solutions are in the context of videoconferencing, but the recent limitations to people's mobility, due to the risk of contagion with the COVID-19 virus, encouraged the extension of such platforms to also serve in the context of Networked Music Performance (NMP). Most of the videoconferencing applications are based on WebRTC's media streams [2], which represent the standard solution to peer-to-peer low latency audio and video conferencing. The main problem is that media streams do not give much control over the total latency they introduce, which may be rather high. In NMP latency is a key factor, in fact if we want to achieve real-time interaction between musicians, one-way latency should be kept under 30 ms [10] (with the knowledge that the actual value may vary depending on the genre, the tempo and the instruments played).

Current constraints in WebRTC's media streams are the reasons why almost all the NMP tools are standalone and native applications that run directly on the operating system of choice in order to benefit from fast access to system calls and complete customization of the software implementation. Among those tools JackTrip [3], Soundjack [5], LOLA [7], and UltraGrid [8] take advantage of both UDP transmission and uncompressed audio format to reduce the overall latency to the minimum. These softwares can be precisely configured in order to choose the minimum audio packet size, playout buffering, and redundancy that allow high quality communication while limiting the end-to-end latency. By contrast, real-time communications based on WebRTC's media streams have very limited configuration options and employ audio compression to reduce the communication bitrate. Encoding and decoding may account for 20 ms or more of additional delay as reported in the experiment with the Aretusa NMP software[12].

One solution to the media stream configuration problem is to avoid media streams all together and instead use the RTCDataChannel which is more commonly used for non-multimedia data. This approach was investigated in 2017 by researchers at Uninett, Otto J. Wittner et al. [13] and proved to be feasible. However, the limited audio processing capability of the *ScriptProcessorNode* [1] limited the overall performance with regard to latency. Nevertheless, they suggested that the emerging AudioWorklet API [6] would probably provide an alternative which would improve upon their results since it would provide independent real-time threads and enable more efficient and consistent audio processing.

In our present work we have taken inspiration from the Uninett work and have implemented an alternative approach to peer-to-peer high quality and low latency audio communication. Exploiting both the WebRTC's RTCDataChannel and the AudioWorklet API we developed a WebRTC application named JackTrip-WebRTC as a sibling to the popular JackTrip software for live music performance over the Internet[1]. The application is released as open source software at https://github.com/JackTrip-webrtc/JackTrip-webrtc. In this paper several configurations, browsers and operating systems are tested to characterize the performance of this solution as compared to the traditional approach based on media streams.

This paper is organized as follows: after a brief introduction to the WebRTC architecture, with particular attention to the transmission layer, in Section 2, we provide a description of the proposed alternative for very low latency WebRTC in Section 3. Measurements of the achieved performance are presented in Section 4 and Section 5 for both end-to-end latency and jitter. Additional functionalities in-

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

Web Audio Conference WAC-2021, July 5–7, 2021, Barcelona, Spain.

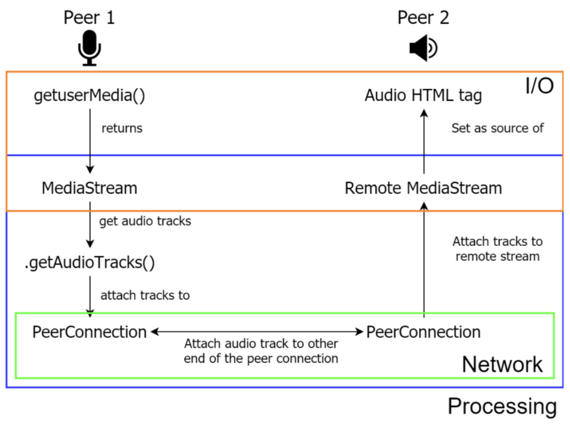


Figure 1: Traditional WebRTC application structure with the RTCPeerConnection.

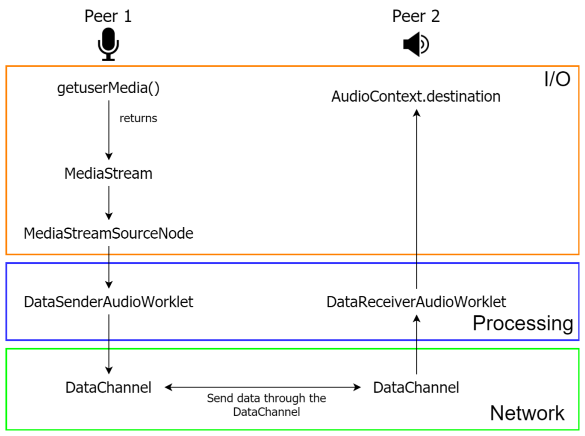


Figure 2: Custom WebRTC application structure with the RTCDataChannel.

troduced for the NMP scenario are described in Section 6.

2. ARCHITECTURE OF WEBRTC

In a traditional WebRTC application, media streams have a fundamental role because they are used both for media acquisition/playback, and for exchanging multimedia data with the communication layer, i.e. the RTCPeerConnection.

Figure 1 represents the conventional WebRTC application structure: the audio application uses getUserMedia to acquire a media stream that represents the audio feed from the user's microphone and it attaches the audio signal to the RTCPeerConnection that takes care of the media delivery through the network. The same happens on the peer in the reverse order. Here, at the receiver, the output element is commonly an HTML5 audio or video tag in the web page.

The RTCPeerConnection channel implements several communication mechanisms on top of the UDP transport layer in order to provide good real-time communication quality. However, these mechanisms are not optimized for ultra low latency ($<$ 30 ms), but for mid-range low latency ($<$ 150 ms) to satisfy the requirements of real-time turn taking discourse communication. As a consequence they often trade off an increase in latency for a reduction of the transmission bitrate by employing audio codecs, or for an increase in the transmission robustness by employing congestion control algorithms, playout buffering, error correction, etc.. RTCDataChannel provides an alternative that is data agnostic and by default does not introduce any additional processing. The RTCDataChannel transport layer is built on the Stream Control Transmission Protocol (SCTP) [11] that allows configurable delivery semantics that range from reliable and ordered delivery to unreliable and un-ordered transmission. The latter mode is the preferred one as it bypasses all the communication overhead introduced by the RTCPeerConnection channel and achieves the lowest possible communication latency.

The substitution of RTCDataChannel for the RTCPeerConnection channel is not straightforward. In the following section we present our method and in Sec. 4 document the performance achieved in terms of latency reduction.

3. VERY LOW LATENCY WEBRTC APPLICATION STRUCTURE

The very low latency JackTrip-WebRTC application structure that is shown in Fig. 2 is derived from the classical structure of a WebRTC application presented in Fig. 1.

While the web browser, as a native application, can directly access the low level OS API to implement custom functionalities, for security reasons the web application is forced to use only the JavaScript API made available by the browser. As a consequence, in the investigation of alternative configurations for latency reduction, we can only modify the application structure if the JavaScript API provides us with alternative implementations for a given task. Some of the traditional bottlenecks could not be removed or substituted with different implementations, so we limited ourselves to investigation of settings that were available which could contribute to the lowest possible latency. In Section 4 we present the tests which led us to the definition of the best configuration for both the getUserMedia and the AudioContext objects.

In switching from the RTCPeerConnection channel to the RTCDataChannel we are required to implement two additional procedures: i) in the transmitter we need to extract the raw audio from the media stream object returned by the getUserMedia method, ii) in the receiver we need to reverse this process and create a new playable stream from the raw audio data received from the RTCDataChannel. Both of these procedures can be implemented by means of the Web Audio API and the AudioWorklet interface.

At the transmitter the media stream is fed as an audio source into the audio processing graph using the *createMediaStreamSource* method of the AudioContext that 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. The audio processing, in this case, is limited to accessing the raw audio data and packetizing it for transmission. This task is performed by an AudioWorklet that is composed 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. 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. However the AudioWorkletNode and the AudioWorkletProcessor are executed in different threads and the browser needs to be charged with the overhead of exchanging audio data between the two threads by means of a MessageChannel.

The AudioWorklet in the transmitter we named DataSender (Fig. 2). It extracts uncompressed audio from the source media stream and creates the packets for delivering on the RTCDataChannel. On the receiver side, we named our complementary AudioWorklet DataReceiver and use it to retrieve audio packets from the RTCDataChannel and manage a playout buffering queue that compensates for network jitter.

In order to find the configuration that can achieve the lowest latency, the application structure can be further customized considering different audio outputs. In Section 4 we compare the results of using the *AudioContext.destination* object that maps to the system default output device or a *MediaStreamAudioDestinationNode* that can be configured with a specific audio output device directly from the application.

4. LATENCY MEASUREMENTS

The rationale for the implementation of a network layer based on the RTCDataChannel rather than on a RTCPeerConnection channel is mainly due to the desire to achieve lower end-to-end delay. In the context of networked music performance, as well as in other multimedia communication scenarios, this delay is sometimes called mouth-to-ear latency. To measure it, we setup a recording environment where we were able to record both the sound acquired by the sending input device and the sound emitted by the peer receiving output device. In order to factor out lags due to variable network delays both sender and receiver are running on the same computer as shown in Figure 3.

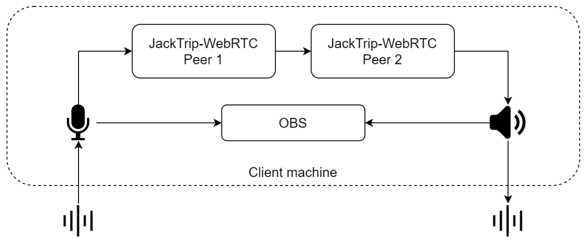


Figure 3: Mouth-to-ear measurement process.

Using an impulsive sound as input, and muting the audio input of the receiving peer to avoid feedback, we were able to measure the mouth-to-ear latency for different configurations, browsers, and operating systems. We tested two *getUserMedia* configurations:

- **Basic**: *getUserMedia* default values for audio and video properties.

- **Low-latency**: *getUserMedia* with *autoGainControl*, *echoCancellation*, *noiseSuppression* set to `false`, and *latency* set to `0`

Both configurations have been tested using the default AudioContext parameters. However some documentation exists that suggests that, depending on the browser, its version, and the operating system, lower latency can be achieved by setting the *latencyHint* property to `0` [9]. We had inconsistent results using the different setups so we preferred to report measurements using the `latencyHint:'interactive'` setting that is referred in the specifications [1] as the one that should *provide the lowest audio output latency possible without glitching.*

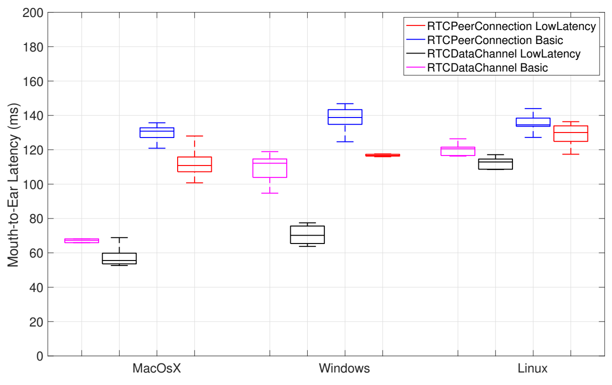
All measurements were performed on three different operating systems: MacOS (v10.14.6) with Core Audio, Windows 10 (v2004) with High Definition Audio, and Ubuntu Linux (20.04) with PulseAudio[2]. Windows and Linux were running on the same machine, an ASUS N551JX with an i7-4720HQ (4 cores / 8 threads) processor and 16 GB LPDDR3 1600 MHz RAM. MacOS was running on a MacBook Pro 2018 with an i7-8559U (4 cores / 8 threads) processor and 16 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.87 and Firefox v.84. In addition for Chrome we enabled the flag *"Use realtime priority thread for Audio Worklet"* in order to achieve better audio stream stability, and for Firefox we enabled the flag *"media.setsinkid.enabled"* to be able to select the audio output device directly from the web browser, without needing to change the system settings.

Measurements in Fig. 4 show that the basic configuration, i.e., the default configuration for *getUserMedia*, introduces a quite high delay, well above one hundred milliseconds, in all the scenarios. Each column summarizes the results of at least ten sessions and the boxplot represents the maximum, 75th percentile, median, 25th percentile, and minimum values (from top to bottom). As expected, a large part of this delay is able to be removed with a configuration favoring lower latency, i.e., by disabling advanced filters on the audio input: automatic gain control, automatic echo cancellation, and automatic noise suppression. Their removal is not detrimental in a networked music scenario because musicians wear headsets to prevent the echo feedback and prefer to avoid automatic gain control during the performance. Interestingly, the approach with the RTCDataChannel shows a significantly lower mouth-to-ear delay with respect to the architecture with the RTCPeerConnection. This is partly because the RTCDataChannel does not implement most of the features included in the RTCPeerConnection for transmission robustness or bitrate reduction. And partly because the AudioWorklet data exchange by means of the MessageChannel is very effective and it does not add a significant amount of latency. Note that in both the approaches a playout buffer of about 20 ms is included in the latency measurements (i.e., eight packets for the RTCDataChannel implementation).

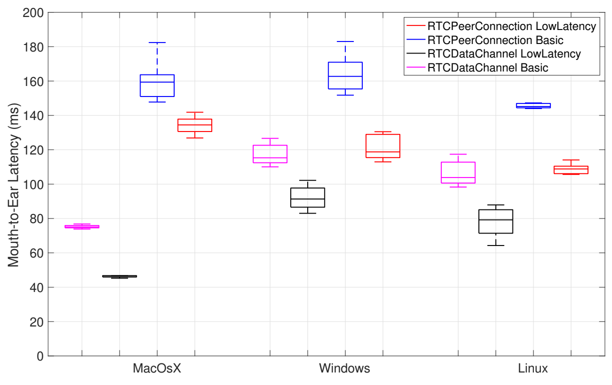
4.1 Audio chain latency analysis

The improvement achieved in the reduction of the end-to-end delay is due both to the implementation of the transmission layer by means of the RTCDataChannel and to a proper configuration of the different layers of the audio chain. In this section we introduce measurements to further investigate how each of those layer contributes to the overall delay.

[2]The default system setup has been used in the three OSs. Reduction in latency may be achieved with a custom setup in particular with Linux and PulseAudio.



(a) Chrome v.87



(b) Firefox v.84

Figure 4: Mouth-to-ear latency measurements for the two approaches: RTCDataChannel and RTCPeerConnection. The two *getUserMedia* configurations, i.e. basic and low-latency, are tested with Chrome and Firefox on different operating systems. AudioContext *latencyHint* property is set to *interactive* and the default audio output device is used.

As depicted in Figure 2 the audio chain consists of three layers:

A. The **input/output layer** that gets a media stream from the input audio device or reproduces the media stream on the output audio device.

B. The **processing layer** that moves audio data from the media stream to the main thread and vice versa. Here a *playout buffer*, that introduces additional latency, is implemented in the receiver to compensate for network jitter.

C. The **network layer** that on one side sends each audio buffer of 128 samples as an independent packet through the RTCDataChannel and on the other side handles the received packets to the processing layer playout buffer.

In addition, two different output configurations can be selected by means of the Web Audio API:

0. **AudioContext.destination** which corresponds to the system default output device.

1. **MediaStreamAudioDestinationNode** which enables selection of an output device that is different from the system default.

In the following we provide the results of three measurements setups where the first one (A0) uses only the input/output layer with the default output device[3], the second one (C0) uses the full audio chain with the default output device, and the third one (C1) adds to the overall system the ability to select a custom output device by means of the MediaStreamAudioDestinationNode. The different setups with increased features allow us to analyze the delay introduced by each additional feature to the entire end-to-end delay.

Figure 5 reports the measurement results for the three setups (A0, C0, C1) that we have tested for different operating systems and browsers. To ensure loss free playback a playout buffer size of eight packets has been used both in setup C0 and C1, that, itself, introduces an additional delay of about 21.36 ms. It clearly appears that the configuration with the default output device is the one that allows the smallest latency and that the creation of a *MediaStreamAudioDestinationNode* to be able to select a different output device significantly increases the overall delay. The drawback of working in a browser environment is evident in the latency results of the I/O layer alone. Here the simple acquisition and reproduction of audio without further processing is delayed by 30 or more milliseconds.

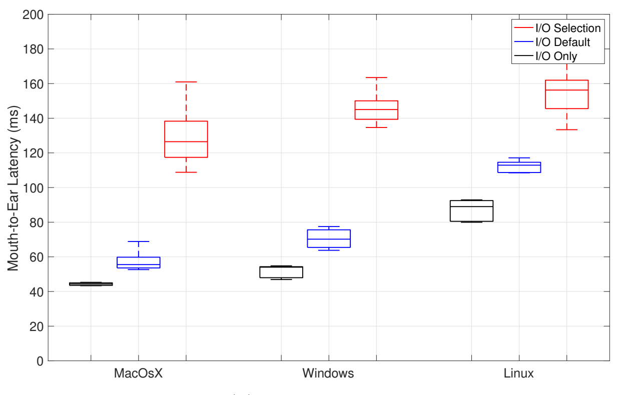
[3]In order to measure the end-to-end delay we had to directly connect the I/O layers between the transmitter and receiver, thus bypassing the lower layers.

5. JITTER MEASUREMENTS

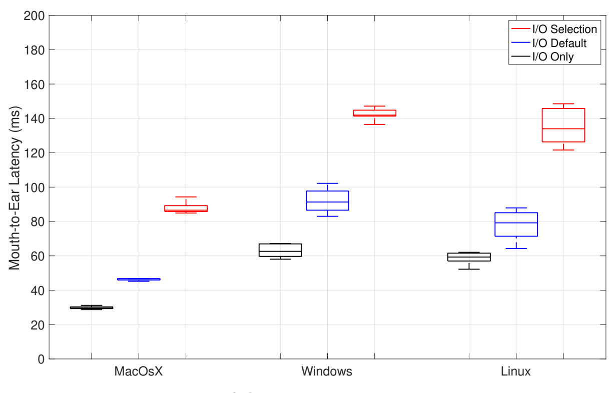
Real-time communication intrinsically suffers from varying delays in the transmission chain that need to be compensated deferring a bit the reproduction of the media samples. Varying delays can be introduced at different levels of the audio transmission chain, thus we analyzed the timing in both the acquisition and the transmission chain. By means of the high precision timestamp API available in the browser we traced the arrival time of each audio buffer at the AudioWorkletNode, both in the transmitter and in the receiver peer.

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 measured on the same computer on the loopback device). Figure 6 confirms that assumption, but it also shows that the acquisition jitter is not negligible since it can reach peaks of 5, 10 or even 15 ms. From the jitter analysis we can suggest that some amount of playout buffering is necessary even in case of local transmission in order to avoid frequent glitches and that it would need to be further increased in a real transmission scenario.

To further investigate the behaviour of the RTCDataChannel we report in Figure 7 the number of audio frames sent over the RTCDataChannel during a frame cycle (i.e., 128 samples). Every time we queue a new frame for transmission, we get the number of bytes in the sender buffer, as reported by the *bufferedAmount* property, and compute the difference from the previous call. When this value is zero, that means that a packet has not been sent in a timely man-



(a) Chrome v.87



(b) Firefox v.84

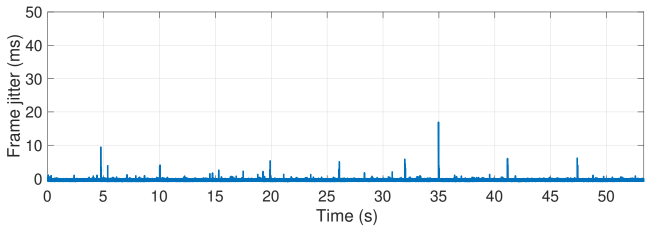
Figure 5: Mouth-to-ear latency measurements for setup A0 (I/O only), C0 (I/O default), and C1 (I/O select) with Chrome and Firefox on different operating systems.

ner through the network, but that it has been queued and delayed in the RTCDataChannel sender buffer. The peaks represent bursts that happen when queued packets are sent. In the current WebRTC specification the programmer cannot configure the RTCDataChannel to avoid buffering, but it seems that, even at a rate of 375 frames per second, the RTCDataChannel does it best to send the data as soon as possible.

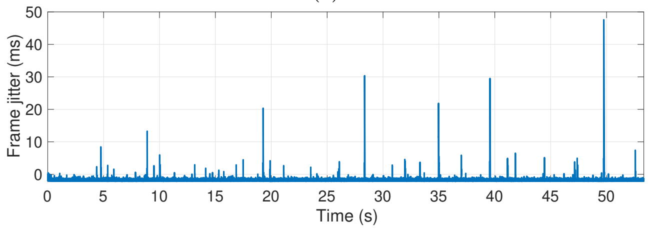
6. ADDITIONAL FUNCTIONALITIES

In JackTrip-WebRTC we implemented some additional functionality focused on characteristics of networked music performance.

An audio loopback mechanism is provided in two configurations: *client-server* and *peer-to-peer*. The client-server loopback is present only during the connection set-up and it gives the user a quick way to check if their network is able to handle the audio stream before connecting with other users.



(a)



(b)

Figure 6: Audio frame jitter measurements with Chrome v.86 at the sender, after acquisition from the input device (a), and at the receiver, after reception from the network layer (b).

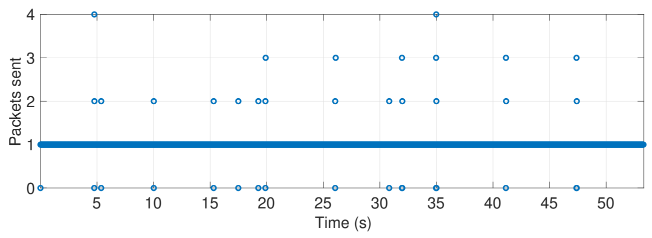


Figure 7: Number of audio frames effectively sent on the network during each frame cycle (2.67 ms) in Chrome v.87.

The peer-to-peer loopback is present only once a peer-to-peer connection is established. It aims to give the user an auditory feedback on the total mouth-to-ear latency and it can be tested with any peer participating in the same room.

For enhanced audio fidelity the application supports both mono and stereo sources. During the connection set-up it is possible to select the source type and number of audio channels. Audio samples are encoded as 16 bit PCM.

A statistics monitor provides information about the communication performance. Every second we provide a report of the average playout buffer size, the percentage of received and discarded packets, and the round-trip-time for each of the users we are connected with.

Unfortunately, when we tried to measure performance with more than two peers connected with each other, performance deteriorated rapidly. This is probably due to the necessity of running the AudioWorklet as a real-time thread. Every time a new peer connects, we create two new AudioWorklet threads, one for the sender part and one for the receiver part of the application. When the number of threads exceeds the number of threads supported by the CPU real-time processing cannot be guaranteed and we observe impaired application behavior. Perhaps changing the design so that the same two threads are able to handle additional connections may result in better scalability.

7. CONCLUSIONS

We described a WebRTC application for PCM stereo audio communication implemented on top of the RTCDataChannel in order to bypass the limitations in quality and latency present in the RTCPeerConnection. Handling of raw audio from the media stream to the RTCDataChannel and vice versa has been performed by means of the AudioWorklets that allow high performance real-time audio processing. From the measurements with different configurations, browsers, and operating systems we can report that this approach can reduce mounth-to-ear latency by tens of milliseconds with respect the one based on RTCPeerConnection (that uses compressed audio formats). However, we noticed that a quite high portion of the overall latency (i.e., between 30 and 60 ms) is introduced by the audio acquisition and reproduction layers as implemented in the web browser API. In addition, the RTCDataChannel cannot be configured to avoid buffering and sometimes this causes delays in transmission of several milliseconds. We advocate for inclusion of an uncompressed codec in the RTCPeerConnection API so as to take advantage of its minimum transmission buffering. Besides that, we expect to leverage future latency improvements in browser real-time media acquisition.

Successful connections between Turin and Stanford have been performed with two peers, but a limitation seemingly due to the number of real-time threads required does not allow us to add more than three peers without degrading overall system performance. We will further investigate this problem to test if the scalability of the system can be improved. For the future, we intend to make the JackTrip-WebRTC packet format compatible with JackTrip, see Appendix A.

8. REFERENCES

[1] P. Adenot and H. Choi. Web Audio API. Technical report, W3C, June 2020. https://www.w3.org/TR/webaudio/.

[2] H. Boström, J.-I. Bruaroey, and C. Jennings. WebRTC 1.0: Real-time communication between browsers. Technical report, W3C, Dec. 2020. https://www.w3.org/TR/webrtc/.

[3] J.-P. Cáceres and C. Chafe. JackTrip: Under the hood of an engine for network audio. *Journal of New Music Research*, 39(3):183–187, 2010.

[4] J.-P. Cáceres, C. Chafe, et al. JackTrip v1.2.2. [Online]. Available: https://github.com/jacktrip/jacktrip, 2020. Accessed on: Dec. 28, 2020.

[5] A. Carôt and C. Werner. Distributed network music workshop with Soundjack. *Proceedings of the 25th Tonmeistertagung, Leipzig, Germany*, 2008.

[6] H. Choi. Audioworklet: the Future of Web Audio. In *ICMC*, 2018.

[7] C. Drioli, C. Allocchio, and N. Buso. Networked performances and natural interaction via LOLA: Low latency high quality A/V streaming system. In *International Conference on Information Technologies for Performing Arts, Media Access, and Entertainment*, pages 240–250. Springer, 2013.

[8] 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*, pages 1457–1460, 2012.

[9] J. Kaufman. AudioWorklet latency: Firefox vs Chrome. [Online]. Available: https://www.jefftk.com/p/audioworklet-latency-firefox-vs-chrome, 2020. Accessed on: Dec. 21, 2020.

[10] C. Rottondi, C. Chafe, C. Allocchio, and A. Sarti. An overview on networked music performance technologies. *IEEE Access*, 4:8823–8843, 2016.

[11] R. Stewart. Stream Control Transmission Protocol. Technical report, RFC 4960, Sept. 2007. https://tools.ietf.org/html/rfc4960.

[12] K. Tsioutas, G. Xylomenos, and I. Doumanis. Aretousa: A competitive audio streaming software for network music performance. In *Audio Engineering Society Convention 146*. Audio Engineering Society, 2019.

[13] O. J. Wittner. Live audio and video over WebRTC's datachannel. [Online]. Available: https://in.uninett.no/live-audio-and-video-over-webrtcs-datachannel/, 2017. Accessed on: Dec. 18, 2020.

APPENDIX

A. PACKET FORMATS

Packet format for both JackTrip-WebRTC and JackTrip are reported for future interoperability.

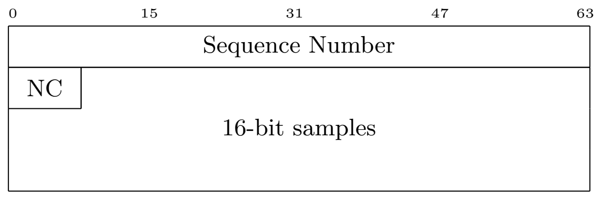


Figure 8: JackTrip-WebRTC packet format. NC: number of channels.

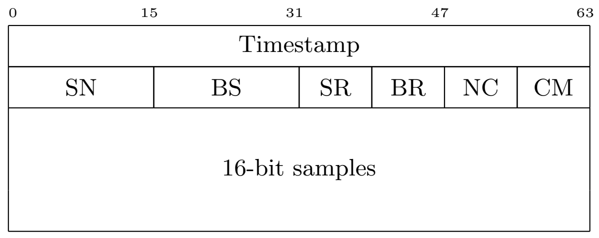


Figure 9: JackTrip packet format [4]. SN: sequence number, BS: buffer size, SR: sampling rate, BR: bit resolution, NC: number of channels, CM: connection mode.