POLITECNICO DI TORINO

Dipartimento di Automatica e Informatica

Master degree course in Ingegneria Informatica (Computer Engineering)

Master Degree Thesis

JackTrip-WebRTC

Networked music performance with web technologies

Supervisor
Prof. Antonio Servetti

Co-Supervisor
Prof. Chris Chafe
(CCRMA - Stanford University)

Candidate
Matteo Sacchetto
matricola: 263113

Academic year 2019 – 2020

Ai miei genitori, a mio fratello Marco, al mio amico di sempre Alin e a tutti gli amici che mi hanno supportato e sopportato durante questi anni

Summary

We are witnessing a large adoption of web based audio/video communication platforms that run in a browser and can be easily integrated with the web environment. Most of these solutions are in the context of videoconferencing, but the recent limitation to people mobility encouraged the extension of such platforms also to the context of networked music performance. This master thesis focuses on showing an alternative approach to peer-to-peer high quality and low latency audio streaming by exploring an unconventional solution based on WebRTC's DataChannels, instead of WebRTC's MediaStreams.

Uncompressed audio is extracted from WebRTC's MediaStream, processed by Web Audio API's AudioWorklets and sent on WebRTC's DataChannels in order to have more control on the overall transmission protocol and to avoid possible communication delays introduced by compression, filtering and other audio processing MediaStreams may perform.

This approach is compared to the classical one to measure its performance and advantages or drawbacks in a networked music performance context. The comparison between the two approaches is carried out by considering different configurations of the getUserMedia function call and by measuring the local mouth-to-ear latency each configuration leads to in both solutions (network latency is not involved in this analysis).

The measurements show that our custom solution usually has a lower latency than the standard MediaStream solution, but the measured latency (50-100ms) is still above the 30ms a networked music performance context requires. Next it shows the measurements performed on the different steps of the audio chain, where we discovered that the first step of the audio chain is where most of the latency is introduced. Unfortunately we can not influence the latency introduced by that step in any other way than by appropriately configuring the getUserMedia call, which we already did.

So, for now this solution can not replace other solutions which already exist, like

JackTrip, for real-time networked music performance, but it still has some value, since it can work as a preparation step for tools like that one, and for introducing more people to the networked music performance world, thanks to the ease of use web applications have.

The last part of this thesis illustrates some other functionalities, like loopback, stereo support and statistics, which are relevant in a networked music performance scenario.

Acknowledgements

First, I would like to thank my supervisor, Professor Antonio Servetti, for his availability, his professionalism, his kindness and for all the help he gave me during the development of this project. Being able to exchange ideas, share thoughts and discuss, with Professor Servetti, possible solutions to problems which arose during the course of this project was of great help.

Next, I would like to thank my co-supervisor, Professor Chris Chafe, for his availability and for the enthusiasm shown towards this project. Being able to talk, share ideas with him and to learn all the cavets related to networked music performance from the creator of JackTrip, the reference application when we consider networked music performance, has been crucial in order to develop this web application and all the associated functionalities. His extensive knowledge about the argument really helped me during the development of the main functionalities of this project, functionalities which I would have probably never thought of.

Next I would like to say a big thank you to all other people involved in this project and all the people which showed interest towards this project. Seeing the interest and enthusiasm of those people towards this project really motivated me.

A big thanks goes to my parents, my brother Marco, my best friend Alin and all other friends which have been there during these five years. They were really supportive and motivated me to keep working hard. If I got to where I am is mostly thank to all the support I received from them.

Last but not least, I would like to say a special thanks to my friends Federico and Vito, which are the guys with which I teamed up to complete all the projects assigned over the last two years. A special thanks goes also to Giulia, part of this team during the first of these last two years. If I managed to reach the end of this master degree course it is also thanks to them.

Contents

List of Tables

List of Figures

"Without music,
life would be a mistake."

[F. Nietzsche, Twilight of the Idols]

Chapter 1

Introduction

Web browsers are one of the most used applications nowadays, everyone is aware of them and uses them on a daily basis. In fact, web browsers are the main interaction point between users and the internet. For those reasons web browsers are always evolving and implementing new features and frameworks.

One of the frameworks that was introduced in 2011 and was later adopted by all major web browsers is WebRTC [1] - "An open framework for the web that enables Real-Time Communications (RTC) capabilities in the browser". Thanks to WebRTC, web browsers are able to support peer-to-peer connections and to exchange audio and video through MediaStreams [2] and raw data through RTC-DataChannels [3]. WebRTC nowadays became the standard for creating anything which involves a web browser, a P2P connection and audio/video streaming.

In fact WebRTC is actually used by some of the big companies (like Microsoft, Google, Facebook, ...) in some parts of their web applications to actually allow their web-applications to perform also audio and video calls. Some of the most famous applications which take advantage of WebRTC are Google Meet, Discord, Facebook Messenger [4], Zoom [5], Skype (the web-app version) [6]. Even the Virtual Classroom system which PoliTo started using this year (BigBlueButton) is based on WebRTC [7]. Here [8] is a list of other web-applications based on WebRTC.

The main problem of WebRTC is that it was created with in mind the idea of performing audio/video calls in a peer-to-peer fashion, and of being able to stream

audio/video, so by using WebRTC out of the box without any particular configuration you obtain an app which is focused on streaming mainly a human voice[1] and has a latency[2] in the order of 150-250ms. While this is something usable for any standard audio/video call with a browser (this is actually what all the above mentioned applications use WebRTC for) our main focus is networked music performance and here things are a bit different. In networked music performance, latency is a key factor and, as reported by [9], it should be kept under 30ms for real-time playing, then, depending on the genre and the instruments, the latency constraint can be relaxed a bit, but the important thing to remember is that latency is a very important factor if we want to achieve real-time interaction between musicians.

Another problem of WebRTC is the lack of configurability of some of its elements. The main element which lacks some of the configuration that would be necessary in a networked music performance scenario is the MediaStream [2]. MediaStreams act as a black box to programmers, in the sense that they are easy to use and simply work every time you need to create an audio/video stream between two peers, but you have no control on how they work, you can not control the algorithm they use to send data and you have no control on the buffers they use (whether they are sending buffers or playback buffers). This means that you have no control on the actual latency they introduce due to processing and buffering.

For all the reasons mentioned above, MediaStreams are not recommended in the context of networked music performance, since we need a more fine grained control over the communication mechanism, which we do not have with MediaStreams.

An important API introduced around the same years as WebRTC is the Web Audio API [10], which defines a set of nodes which can then be chained in order to obtain a full audio chain to process input audio data. It allows the selection of the source node, the selection of the intermediate nodes, like audio effects or custom defined nodes, and it allows you to decide which is the destination of this audio chain. This API allows developers to create some custom audio processing within a web browser and with this API it is even possible to create a DAW [11] using only a web browser. One of the key factor of Web Audio API is its ability to interact with WebRTC's MediaStreams for both input and output, in fact through the use of two special nodes: MediaStreamAudioSourceNode [12] and MediaStreamAudioDestinationNode [13] we are able to go from WebRTC

[1] I mean that it focuses mainly on mono sources and applies some audio processing, like echo cancellation and/or noise suppression

[2] I'm referring to local latency, which means that it involves only acquisition and processing latencies and does not involve any network latency

MediaStreams to Web Audio API nodes, and vice-versa.

This versatility and the fact that it has been developed specifically to process audio, so it has some well defined constraints a node must satisfy, make it something interesting to explore and which could be helpful for a networked music performance context.

The last resource we must consider when we are talking about networked music performance is JackTrip [14] - "A System for High-Quality Audio Network Performance over the Internet". JackTrip is a command line program which uses Jack [15] to manage audio connections and allows people to connect together, through a peer-to-peer or a Client-Server mechanism, and play together as if they are in the same room. In terms of performance and latency JackTrip is the best we can do, since it runs as a native application and it is developed in C++, a programming language focused on performance. The main problems JackTrip has are ease of use and the complexity of its installation process. Since JackTrip is a command line tool, in order to use it you need to know and remember the options you actually need among all available options and you need to be familiar with the terminal of your OS, which is something not all users are familiar with and know how to use. This is actually the main reason why we decided to develop this project.

The main idea of this project, called **JackTrip-WebRTC** [16], is to create an equivalent of JackTrip, using only web technologies, in particular using WebRTC to handle peer-to-peer connections, and to exchange audio and video among peers and using the Web Audio API for any custom audio processing required. Our main focus is to transmit very low latency audio between peers, for this reason we will use peer-to-peer connections, in that way we are able to reach the lowest network transmission delay between peers. Furthermore we will use WebRTC's DataChannel for exchanging uncompressed audio between peers, in this way we have more control over the transmission protocol and delays introduced, and we are able to avoid any additional delay that may be introduced by MediaStreams, due to encoding, filtering, resampling, etc. The last important thing is that we will use UDP as the transport protocol of the DataChanel in charge of sending audio data, in this way we are able to avoid any delay that may be introduced due to packet re-transmissions, segmentation, etc.

In order to extract uncompressed audio from WebRTC MediaStreams, which we will still need to use for some steps of the audio chain, we will use a custom audio node. In the Web Audio API there is an interface which can be used to define custom audio processing nodes, and it is called AudioWorklet [17].

Before AudioWorklets, which were introduced officially around mid-2014, if you wanted to create a custom node of the Web Audio API, you needed to implement

the ScriptProcessorNode interface [18], which defines a input buffer and an output buffer. That interface defines one parameter, which is the buffer size and has to be a power of 2 between 256 and 16384.

The main problem of this interface is that the node you obtain is executed in the main thread and so it does not define a strict timing of the call of its *onaudioprocess* function. If the main thread is busy doing something else then the onaudioprocess function will be called late, and that is something which is not great in case of real-time audio processing.

AudioWorklets overcome this problem thanks to how they are designed, in fact each time you create a new AudioWorklet you are actually creating two different elements: an AudioWorkletNode [19], which is executed in the main thread and is the element that will be connected with other nodes of the Web Audio API, and an AudioWorkletProcessor [20], which is executed in a different thread and it is the one in charge of executing the actual audio processing. Thanks to the fact that the AudioWorkletProcessor is executed in a separate thread than the main one, the *process* function, which is the function in charge of executing the audio processing, is called with a well defined timing and its call does not affect and it is not affected by the main thread. Moreover AudioWorklets define a fixed buffer size of 128 samples, which is half the minimum buffer size of the ScriptNodeProcessor. All of this combined together allows the AudioWorklet to perform some ultra low latency audio processing, which is exactly what we need in this project.

Chapter 2

WebRTC Application Architecture

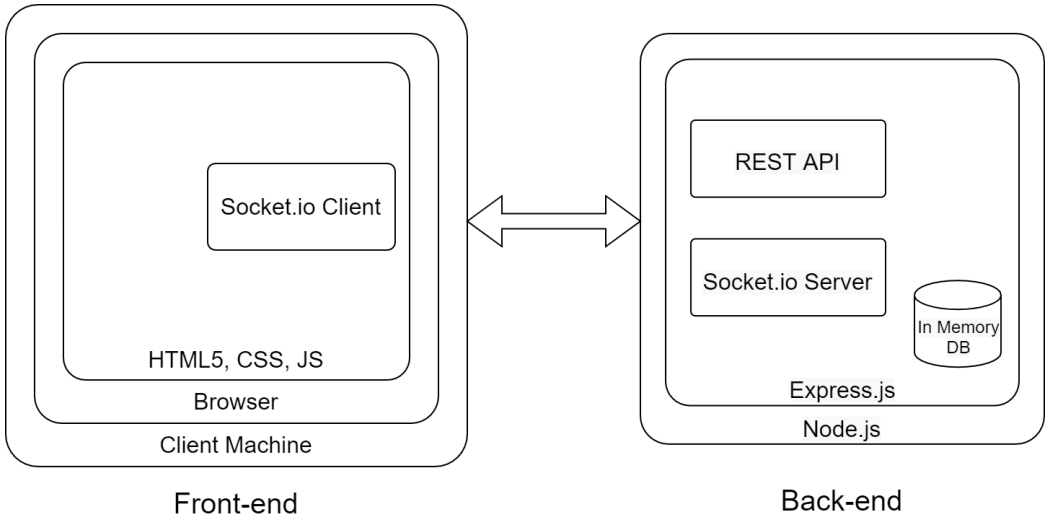

Figure 2.1. Web-application Architecture

This application works with a peer-to-peer mechanism and uses WebRTC to create peer-to-peer connections. The application front-end is the part of the application in charge of creating those peer-to-peer connections and of performing all required processing to exchange audio and video between peers and play them back. In order to create peer-to-peer connections though, WebRTC requires the presence of a **signaling server**, in charge of allowing peer discovery and negotiation of all

necessary information to create the P2P connection. The back-end of the application is the part in charge of implementing the signaling server and of providing some additional functionalities. The signaling server itself is not enough to allow the creation of the P2P connection in every situation since, due to the fact that peers may be behind a NAT, a direct P2P connection may require some additional information or may even not be possible. In order to overcome the problems due to NATs, WebRTC uses the ICE framework [21], which defines a series of techniques to allow two machines to communicate as directly as possible. In order to support ICE we need to have two external additional servers: the STUN server and the TURN server.

2.1 Back-end

Even though the connection is peer-to-peer you still need at least one server, which is the **signaling server**. Once the peer-to-peer connection is created, the signaling server is no more involved, all the communication data will be sent directly from one peer to the other one.

WebRTC does not specify any characteristic this server should have, other than allowing connected peers to discover each others and exchange signaling information. It can be implemented with any available technology and can use any available transport mechanism for the signaling information. It even doesn't need to understand the signaling message content, since the only thing that it needs to do is deliver those messages to the other peer.

In this application the signaling server is a Node.js [22] server that relies on WebSockets [23] for real-time communication with the browser. To create the web server I am using the framework Express.js [24], a easy to use Node.js framework which makes really easy the process of creating a web server. I then used the Socket.io [25] server library to enable support for WebSockets in the web server. The beauty of Socket.io is that it integrates easily with an Express-based web server and it supports the concept of rooms by default, concept which we will use.

In fact, in the signaling server I also implemented the concept of room, a sort of virtual place where peers can meet and talk together. All the peers which connect to same room are able to discover each other and exchange all required information to setup the peer-to-peer connection. Through WebSockets we are able to identify events like connection to the room and disconnection from the room, and signal those events to all the other clients connected to the same room. Each room is independent, this means that a peer connected to room a will **only** see the other peers connected to the same room and it will not be aware of the fact that there are other rooms with other people connected. Each room has a

random unique identifier which is assigned by the server on room's creation. After a certain period of inactivity the room is then deleted from the server and will not be available anymore (the amount of time is customizable through an environment variable of the application). Rooms are stored in a sort of in memory DB, which is actually implemented as a simple associative array. This means that if the server is restarted all the already created rooms are no more present. This is not a problem since server restart should not happen frequently, and new rooms are easy to create.

2.1.1 STUN server

Session Traversal Utilities for NAT (STUN) [26] is a protocol which provides a tool to deal with NATs. It defines a mechanism that a peer can use to determine the public IP address and port allocated by a NAT that corresponds to its private IP address and port. It also provides a way for an endpoint to keep a NAT binding alive. A STUN server is simply a server which implements this protocol.

With a STUN server a peer is able to find out the public IP address and port which have been allocated to it by the NAT. Since this binding is kept alive thanks to the STUN server, the peer can try to create the P2P connection with the other peer using that information.

Some STUN server are already available to be freely used. The one we decided to use in this project is: `stun.l.google.com:19302`

As we can see in Figure 2.2, with a STUN server the process of creating the P2P connection is almost the same as if it was not present, with the only difference that the IP addresses used by the peers to create the connection are their public IP addresses[1], obtained thanks to the STUN server, instead of their private ones.

As reported here [27] the STUN protocol works with the following types of NAT: full cone NAT, restricted cone NAT, and port restricted cone NAT. Any connection involving peers behind one or more of the NATs listed above can be successfully created using just the STUN server. But, if at least one of the peers is behind a symmetric NAT then the STUN server may not be enough in order to create the P2P connection because, since the IP address of the STUN server is different from the one of the other peer, the NAT mapping that the STUN server sees is different from the one that the other peer will see. In those cases, in order to allow the communication between the two peers we need a TURN server.

[1]The public IP address is the one of the router which is connected with the internet, address which is used by the NAT during the translation process

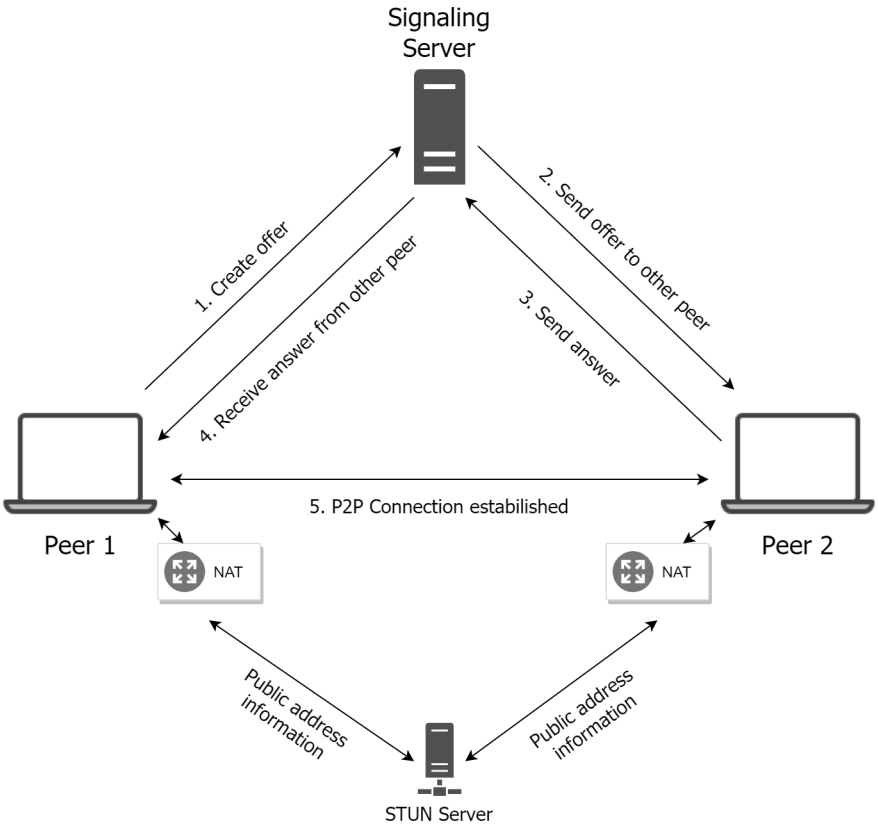

Figure 2.2. Peer discovery and negotiation through the signaling server with a STUN server

Here [28, 29] you can find a full explanation of the characteristics of the different types of NAT available.

2.1.2 TURN server

Traversal Using Relays around NAT (TURN) [30] is a protocol which allows two hosts behind NATs (called TURN clients) to communicate with each other passing through a server (called TURN serer) which acts as a relay. This protocol is used by the ICE framework every time a direct communication through a P2P connection is not possible due to the fact that both peers are behind a NAT which does not work with STUN. Using the TURN server we are actually converting the P2P connection, which will be otherwise impossible, in a client-server connection, which works with any type of NAT. Moreover, TURN servers have some additional functionalities to deal with firewalls, in particular TURN server are able to convert a UDP stream into a TCP one, in order to bypass firewall rules which filter out

UDP packets.

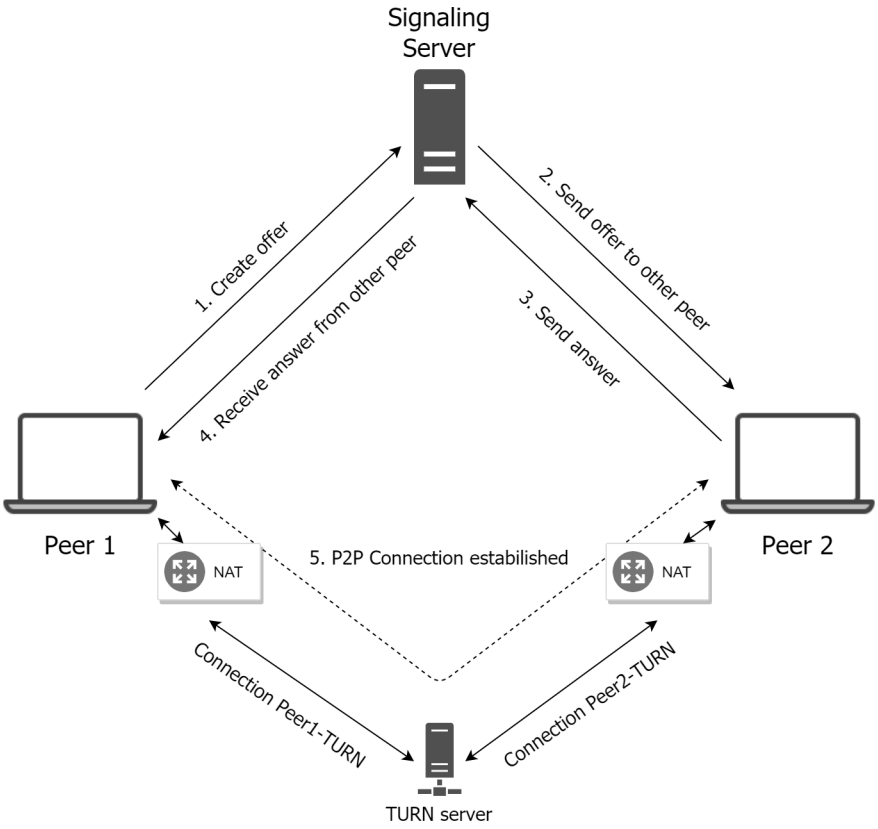

Figure 2.3. Peer discovery and negotiation through the signaling server with a TURN server

In Figure 2.3 we can see that from the point of view of the clients the connection is similar to a standard P2P connection, but in reality the P2P connection is emulated by two client-server connection. One thing to note is that, in our case the TURN server is needed only to allow the P2P connection to be always created, but in terms of performance in this solution we have a higher end-to-end latency than in previous solution, this is due to the fact that we have a higher RTT between clients, since each packet needs to go to the TURN server in order to be then delivered to the other peer. So, depending on the actual location of the TURN server with respect to the location of the peers that want to communicate, the additional latency can vary from not noticeable to 200-300ms.

TURN servers are not available for free, since the they can be quite heavy on the CPU in order to do all required packet processing for all connections which go through it. For this reason you can either buy a service from one of the TURN

server providers or you can deploy one yourself. We opted for the second solution.

During the setup of the P2P connection, ICE first tries to make a connection using the host address obtained from a device's operating system and network card (If the peer is behind a NAT then this connection will fail). If that fails, ICE then obtains the peer's external IP address using a STUN server, and tries to make a connection using the external address. It that fails too, then traffic is routed via the TURN server. In this way we are always able to create a connection and the connection we create is the one which provides better performance and lower latency.

2.2 Front-end

The application front-end is executed in the context of a web browser and has been developed in plain HTML5, CSS3 and JS (ES6). To develop the front-end functionalities I exploited WebRTC [1] and the Web Audio API [10], frameworks/libraries already provided and supported by most of web browsers. In addition to that, I used some third party libraries, in order to simplify the development process. In particular I used the Socket.io [25] client library, to easily set-up WebSockets, necessary to implement the signaling mechanism, moreover I used JQuery [31] and Bootstrap 4 [32] to simplify the creation of the web application GUI.

After creating a room through the GUI, or after having received by another peer a link to a room, the application front-end connects to the server through WebSockets. Once it has connected successfully to the room, it will receive the list of all the clients already connected. For each of the connected clients it will begin the negotiation phase, so it will send the SDP offer to that peer, through the signaling server, and it will receive the SDP answer from that peer. Then, the ICE framework will perform all of its steps, and will send all the information to the local peer, which then needs to send them to the other peer. Once that also the ICE framework finished, both peers have all necessary information to successfully establish the peer-to-peer connection, so the negotiation phase finishes.

From now on, all the communication will only interest the front-end of the application, and will no more involve the signaling server, except for signaling the disconnection of a peer from the room. So, now each peer will proceed to create all the necessary elements to send data to the other peers and to extract and process the uncompressed audio samples.

In order to deal with more than two peers, I implemented a **full mesh** network topology, in this way each client can connect to each other directly, allowing for the lowest possible latency among each pair.

Chapter 3

Low-latency Media Transmission Through The DataChannel

In the standard WebRTC solution for transmitting audio and video, MediaStreams [2] have a fundamental role, they handle all the aspects related to exchanging audio and video information among peers, from acquisition to playback.

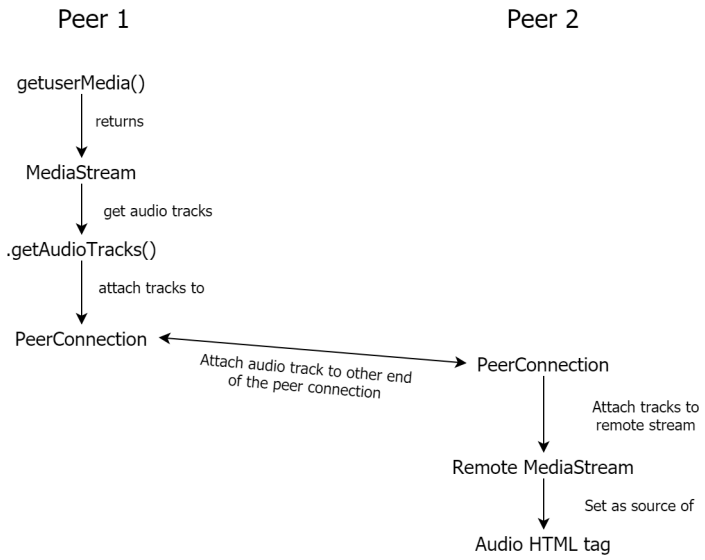

Figure 3.1. MediaStream transmission chain

In order to implement the standard WebRTC audio/video transmission chain, shown in Figure 3.1, we need to perform a call to the `getUserMedia()` function of the `navigator` object. To that function we can pass a configuration object which specifies which devices we need access to and which constraints they should respect. Once we perform the call to the `getUserMedia()` a prompt will request the user to grant access to microphone and/or webcam. Once the users grants the application access to the requested devices, the `getUserMedia()` will return a MediaStream containing the MediaTracks associated to those devices. We will then need to simply extract all the MediaTracks contained in the MediaStream and attach them to the PeerConnection object. By doing so we are actually sharing the MediaTrack with the other peer, which will just need to recreate a MediaStream and attach it to an HTML audio tag.

While this solution is really easy to implement, it has a major issue: the programmer has **no control** over the media transmission process. That means that the programmer has no control over latency introduced by the transmission process, but latency is the main factor we must keep under control if we want to implement a web-application focused on network music performance. Since our focus is transmitting low latency audio among peers, we will use the standard MediaStream solution for exchanging video among peers, but we need to find a different solution to exchange audio, and the solution we found takes advantage of **DataChannels**. From now on we will only focus on audio transmission.

3.1 DataChannel transmission architecture

A DataChannel [3] represents a network channel which can be used for bidirectional peer-to-peer transfers of arbitrary data. With DataChannels we should have more control over the transmission process, since we can select the type of transport protocol to use, TCP or UDP, and the actual creation of the packets is responsibility of the programmer. But, even if we decide to not use MediaStreams in the transmission process, we still end up having them in at least one step of the transmission chain. In fact, the only way a web browser allows the application to gain access to the audio input device is by performing a call to the `getUserMedia()` function, which, as already said previously, returns a MediaStream. This means that, in order to implement a solution using DataChannels we need 2 things:

- At the **transmitter**: a way to **extract uncompressed audio data from the MediaStream** obtained through the `getUserMedia()` call.
- At the **receiver**: a way to **play the uncompressed audio** received through the DataChannel.

Luckily we can exploit **AudioWorklets** [10] from Web Audio API to solve both problems, an interface which can be used to define custom audio processing nodes that are executed in a separate thread in order to provide very low latency audio processing[1]. An AudioWorklet is actually made of two different elements:

- An **AudioWorkletNode** [19]: a custom node of the Web Audio API which is necessary in order to allow the AudioWorklet to be chained with other nodes of the Web Audio API. This part of the AudioWorklet is executed in the main thread of our web application.
- An **AudioWorkletProcessor** [20]: the actual audio processor, in charge of executing the custom audio processing required. This part of the AudioWorklet is the one executed in a separate thread. It has an input buffer of 128 samples and it has a main function, called `process`, which is called automatically by the browser at a fixed timing rate, depending on the selected sample rate.

The AudioWorkletNode and the AudioWorkletProcessor are executed in two different threads, so we need a mechanism to exchange data between the two elements. JavaScript is single-threaded, so it doesn't deal with the creation of threads nor their synchronization, thus all of this is demanded to the web browser executing the code. This means that it is also responsibility of the web browser to provide a mechanism to exchange data between the two threads, so between the AudioWorkletNode and the AudioWorkletProcessor. This problem is solved thanks to MessageChannels [33], which define a two-way pipe with a port at each end. This is an event based mechanism which allows us to easily exchange data between the AudioWorkletNode and the AudioWorkletProcessor and signal when data is ready. The AudioWorklet interface deals with the creation of a MessageChannel itself, so when we create the AudioWorklet we find the MessageChannel already created and ready to be used.

In this application I have implemented two different AudioWorklets:

- A **DataSender**: is the AudioWorklet in charge of extracting uncompressed audio from the MediaStream, creating a packet containing that data with some additional information and delivering it to the DataChannel, in order to be sent to the other peer.
- A **DataReceiver**: is the AudioWorklet which receives from the DataChannel

[1] AudioWorklets are still a rather new technology, for this reason they are still not supported by all browsers. As it is right now (2020), Firefox 76+, Chrome 66+ and any Chromium-based browser from version 66+ are the only web browsers that supports them. Compatibility with all major web browsers can be checked here: https://caniuse.com/?search=audioworklet

the packets generated by the other peer, and it is in charge of extracting the audio samples from the packets and playing them back. It manages a queue where packets are stored based on their sequence number and filters out old packets. This AudioWorklet will not start audio playback until it receives a certain number of packets, which we will call: **playout buffer size**.

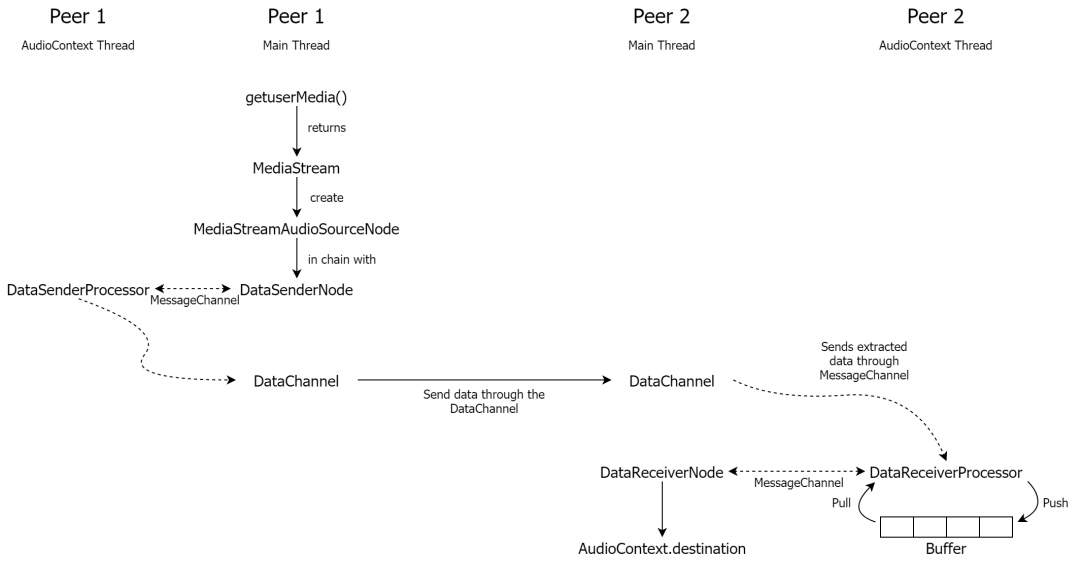

Figure 3.2. DataChannel transmission chain

The Figure 3.2 shows the full DataChannel transmission chain with the assumptions that the peer-to-peer connection has already been created and that also the creation of the DataChannel has been fully completed. As we can see, this audio chain is more complex than the standard one (3.1), but this also means that we have more control over the various steps of the audio chain and the latency they introduce.

First thing we need to do, as for the standard audio chain, is to gain access to the audio input device. In a browser this is only possible through the call to the `getUserMedia()` function. Once that the user allows the application to access the device, the `getUserMedia()` will return the MediaStream containing the MediaTrack associated to that device. In order to connect the MediaStream to the DataSenderNode we need a way to convert the MediaStream to a node of the Web Audio API. Luckily, the Web Audio API has a node which has that exact purpose: the **MediaStreamAudioSourceNode** [12]. We create that node by calling the `createMediaStreamSource()` function of the `AudioContext` object, and by passing it the MediaStream we obtained previously. The object returned by that function is the MediaStreamAudioSourceNode of the Web Audio API,

node which can then be chained with other nodes of the Web Audio API, and since the DataSenderNode is one of them, we can connect the MediaStreamAudioSourceNode with it. The Web Audio API works in a lazy way, in the sense that if an audio chain is not connected to any destination it will not start its computation until it will be connected to one of them, in order to save resources, so in order to allow this part of the audio processing chain to start we need to connect the DataSenderNode to a destination node of the Web Audio API or to another node which is connected to a destination node. The Web Audio API provides a default destination node, which is accessible through the `destination` property of the `AudioContext` object, and we can connect the DataSenderNode to this destination in order to allow the processing to start. Once the processing starts the DataSenderProcessor begins to extract the audio samples from the MediaStream, it creates a packet containing those samples and it delivers them to the DataSenderNode, through the MessageChannel, in order to be delivered to the other peer through the DataChannel. On peer 2 side the audio chain needs to do the opposite of what is done on peer 1 side. So on peer 2 the audio chain is smaller, since it is only composed of the DataReceiverNode which is then connected to the default destination of the Web Audio API. When the DataChannel receives the packet generated by peer 1, it will deliver it to the DataReceiverProcessor, through the MessageChannel. The DataReceiverProcessor will check if the packet is before or after the last one played. If it is after the last packet played, but it is still within the receiving window[2] then the packet will be added to the circular buffer, otherwise it will be discarded, as it arrived too late or too early. The DataReceiverProcessor, is also in charge of audio play back, in fact, each time its `process()` function is called, the DataReceiverProcess will extract the current packet from the circular buffer and will assign its samples to the `output` object. **NOTE**: in all of the above audio chains we considered only audio going from peer 1 to peer 2, but everything is valid also for the opposite direction.

3.1.1 Circular buffer

Within the DataReceiverProcessor I implemented a queue where I store all the packets coming from the DataChannel. This queue is implemented as a circular buffer where the index of each packet in the queue is predetermined.

$$index = packet\_n \bmod window\_size \tag{3.1}$$

[2] I'm referring to the closed interval $[last\_packet\_n, last\_packet\_n + window\_size]$, where **last_packet_n** is the sequence number of the last packet played, and **window_size** is the size of the buffer

The **packet_n** is the sequence number of the current packet and **window_size** is the size of the circular buffer. By implementing it this way I can access each position with a $\Theta(1)$ cost, regardless of the sequence number of the packet, moreover I am able to avoid any shift or search within the buffer, operations which would introduce a $O(n)$ cost. In parallel to the queue, we have a marker array which keeps tracks of which positions of the buffer have data and which are empty. In this way I can check if a certain packet is present or not with a $\Theta(1)$ cost, and if it is present I can easily extract it and play it's samples.

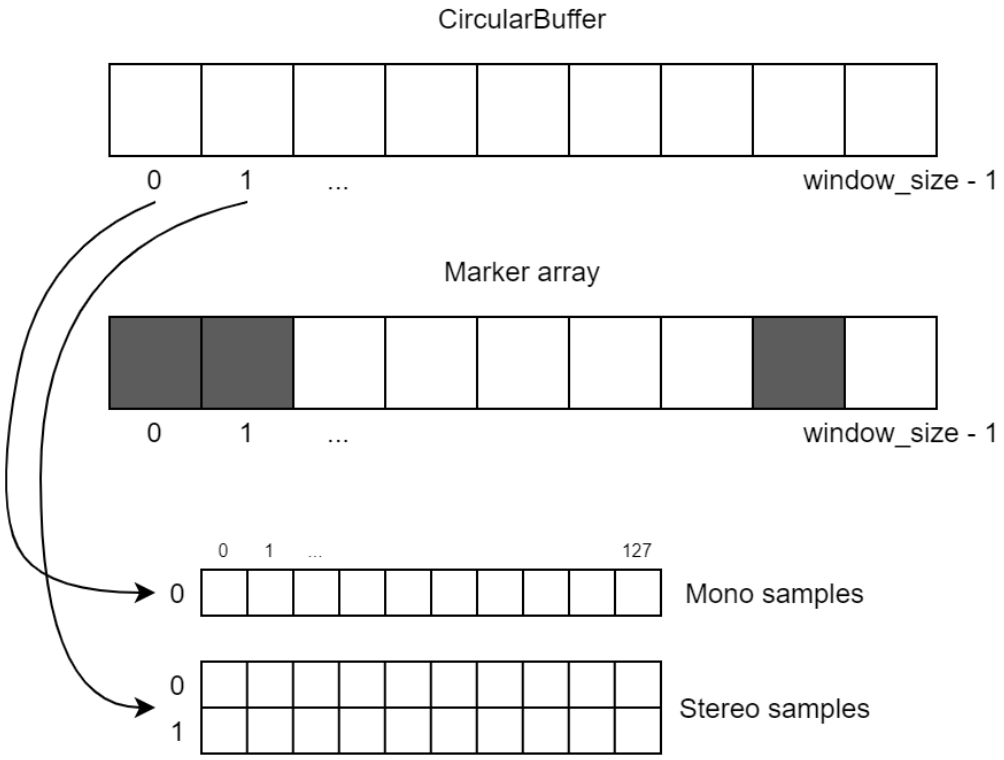

Figure 3.3. DataReceiver circular buffer

In each position of the queue I store the samples received through the DataChannel. Samples can be of two different types: **mono** or **stereo**, depending on the type of source selected. The queue has been developed to support both type of samples (actually it can support samples coming from sources with any type of multi-channel set-up) since in each position of the queue I store an array the same size as the number of channel present in the received packet (1 for mono, 2 for stereo), and at each index of this array I store the Float32Array [34] containing the 128 samples of the relative channel. In Figure 3.3 there is a graphical representation of all of this.

3.1.2 Packet structure

The packet sent over the DataChannel is a binary packet with the following structure.

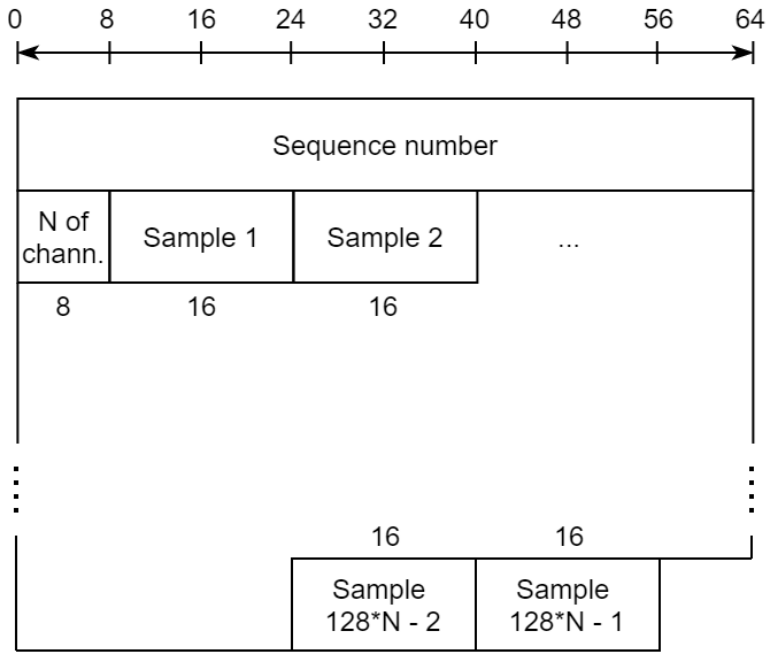

Figure 3.4. Packet structure

The first field contains the sequence number, a 64 bit `BigInt` [35] incremental number. Even though `BigInt` allows us to represent correctly, in JavaScript, integers bigger than the maximum safe integer `Number.MAX_SAFE_INTEGER` ($2^{53} - 1$), going above that value may lead to an unexpected behaviour, due to the fact that internally I am converting the `BigInt` field back into a `Number` variable. Realistically though, this value is big enough to not be reachable by any continuous peer-to-peer communication, since, if we consider a sample rate of 48 KHz and a input buffer of 128 samples, we are generating an audio packet every 2.67ms, which means that in order to reach the maximum value of $2^{53} - 1$, we would need around 764 643 years. The second field contains a 8 bit integer representing the number of audio channels of the audio source selected by the sender. Since the application supports only mono and stereo sources, the only valid values this field can assume are 1 and 2. After this field we find the payload, which contains the actual audio samples. The payload is represented as a continuous array of samples, where each 128 samples changes the audio channel the samples refers to (so, samples from 0 to 127 refer to first audio channel, samples from 128 to 255

refer to the second audio channel and so on). In the Web Audio API context each sample is represented by a 32 bit float value, which varies in the range $[-1, 1]$. Since packets are generated quite frequently (each 2.67ms), in order to reduce the required bandwidth, we decided to convert the 32 bit float samples to 16 bit integer ones during the creation of the packets. In order to achieve this I needed to convert the interval $[-1, 1] \subset \mathbb{R}$ to the interval $[-32768..32767] \subset \mathbb{Z}$. To do so I used a simple normalization formula

$$s_{16} = \left\lfloor \left(\frac{s_{32} + 1}{2} \right) \cdot 65535 - 32767 \right\rfloor, \quad s_{32} \in [-1, 1] \subset \mathbb{R},\ s_{16} \in [-32768, 32767] \subset \mathbb{Z} \quad (3.2)$$

Once the packet reaches the receiver, the audio samples need to be converted back to 32 bit float when extracted from the packet, in order to be compatible with the Web Audio API. To do so I used the following formula (which does the opposite of what 3.2 did)

$$s_{32} = \left(\frac{s_{16} + 32767}{65535} \right) \cdot 2 - 1, \quad s_{16} \in [-32768, 32767] \subset \mathbb{Z},\ s_{32} \in [-1, 1] \subset \mathbb{R} \quad (3.3)$$

By having the conversion performed by the functions which create the packet and extract data from the packet, all this process is completely transparent to both the DataSender and the DataReceiver AudioWorklets, so in case we need to change again this conversion process we would need only to modify the class which represent the Packet, without needing to change anything in the AudioWorklets.

The packet is implemented as an ArrayBuffer [36] with a size of 265 Bytes for mono sources and 521 Bytes for stereo sources. If we consider a sample rate of 48 KHz and a buffer of 128 samples, we are generating 375 packets/s, this means that the bandwidth required to transfer audio packets from one peer to the other is of 776.36 Kbit/s for mono sources and 1.49 Mbit/s for stereo sources.

The ArrayBuffer type is among the types which implement the Transferable [37] functionality. This means that its ownership can be transferred between different context, in particular, in our case, it means that it can be transferred from the AudioWorklet thread context to the main thread and vice-versa, when we send it through the MessageChannel. This functionality allows us to save time when we transfer the packet between those context, since instead of creating a new object with the same content, we can actually transfer directly that object, saving the time it takes to create the new object and copy the content to it.

3.1.3 Communication architecture

The overall communication architecture between any two peers is made of three different connections:

- **MediaStream**: it is used to exchange video information between the peers. A MediaStream is mono-directional, so we have one MediaStream for each peer.

- **DataChannel (UDP)** - audio: it is used to exchange packets containing the uncompressed audio. A DataChannel is bi-directional so we have only one audio DataChannel for each pair. This DataChannel is configured to send binary data, in the form of an ArrayBuffer.

- **DataChannel (TCP)** - control: this DataChannel is used to exchange control information, and since we need this information to be always delivered to the other peer this DataChannel is set to use TCP as transport protocol. The control information we send is rather small, so we can afford sending it as a JSON object. Also this DataChannel is bi-directional so we have only one for each pair.

The control DataChannel is necessary in the DataChannel trasmission architecture, due to the fact that some information that usually is intrinsic in the MediaStream data structure, is no more available when we move to the DataChannel solution. An example of control information which is sent through the control DataChannel is the status of the audio input device (active or muted), whenever the status changes we send a message to the other peer.

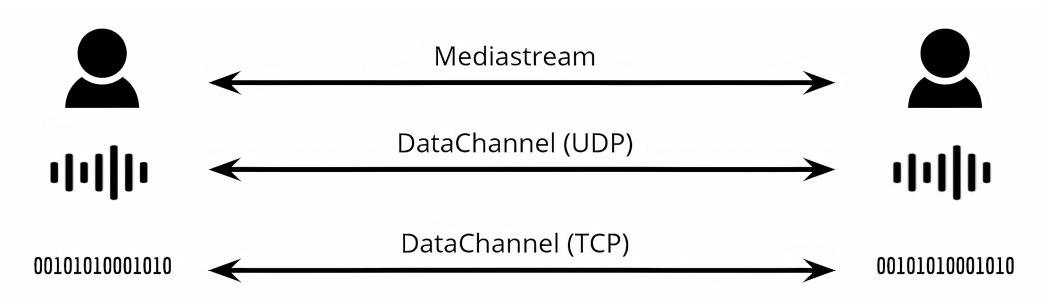

Figure 3.5. Communication Architecture

3.1.4 GetUserMedia and AudioContext

Now, that the transmission chain is under our control, we need to focus on the parts of the web application which introduce latency, and those parts are the MediaStream and the Web Audio API nodes. We can not directly control the latency introduced by a MediaStream or by each individual node of the Web Audio API, but we can influence their behavior through the configuration of the `getUserMedia()` and the `AudioContext`.

In particular, for MediaStreams associated to input devices, we can reduce the latency they introduce through an appropriate configuration of the `getUserMedia()` function call. The `getUserMedia()` function accepts as parameter a configuration object, where we specify the type of devices we want to obtain access to and the requirements they should satisfy. The configuration parameter is a JavaScript object with 2 attributes: `audio` and `video`. For each of this attributes we can either specify the value `true` if we just want to access the relative input device, or we can specify a JavaScript object where we define the constraints that we want the input device to satisfy, the `getUserMedia()` will look for the device and the settings that will satisfy the required constraints. If it can not find any device which satisfies those constraints, then it will throw an error. Here [38] are more details about the `getUserMedia()` function. A full list of all the constraints which can be specified for the two attributes can be found here [39].

We will focus on fine-tuning only the configuration of the audio constraints, while we will simply require access to the camera, this means that the configuration will look something like this: `{video: true, audio: ...}`. Regarding the audio constraints, the ones we will focus on experimenting with are the following:

- **autoGainControl**: specifies whether automatic gain control is preferred or required.
- **echoCancellation**: specifies whether or not echo cancellation is preferred and/or required.
- **latency**: specifies the latency or range of latencies which are acceptable and/or required.
- **noiseSuppression**: specifies whether noise suppression is preferred and/or required.
- **sampleRate**: specifies the sample rate.
- **sampleSize**: specifies the bit depth.

Regarding the Web Audio API, we can influence the latency introduced by the different nodes, by appropriately configuring the `AudioContext` [40]. We can do

so by passing a configuration object during the creation of the `AudioContext`. In the configuration object we can specify two different parameters:

- **latencyHint**: specifies the type of playback that the context will be used for. It can be either a value from the AudioContextLatencyCategory [41] or a double precision float value. The user agent may or may not choose to meet this request.
- **sampleRate**: The sample rate to be used by the AudioContext, specified in samples per second. If not specified, the preferred sample rate for the context's output device is used by default.

During our measurements we will keep the sample rate fixed at 48 KHz and we will check how latency varies when we change the value set for the latencyHint parameter.

3.2 Web Application Functionalities

In this section I present all the relevant functionalities I implemented in this web application, explaining the reasons why I implemented them and why they are significant in a networked music performance focused application.

3.2.1 Loopback

The first functionality I implemented is the **loopback**. The idea behind this functionality is to give to the user an auditory feedback on how the application is behaving. In particular we have two kinds of loopback which aim to give two different types of feedback:

- **Client-server loopback**
- **Peer-to-peer loopback**

Client-server loopback

This is the first type of loopback which the user encounters. It is available in the modal present during the connection set-up, so it is available only while the user is not yet connected to the room, and it aims to give to the user a feedback on the upload/download performance of its network connection and on the selected settings (audio input device, audio output device, ..) to check if everything related to the audio chain is working correctly.

In practice, when the client-server loopback is activated, an audio chain similar to the DataChannel one used for peer-to-peer communication is created. In this

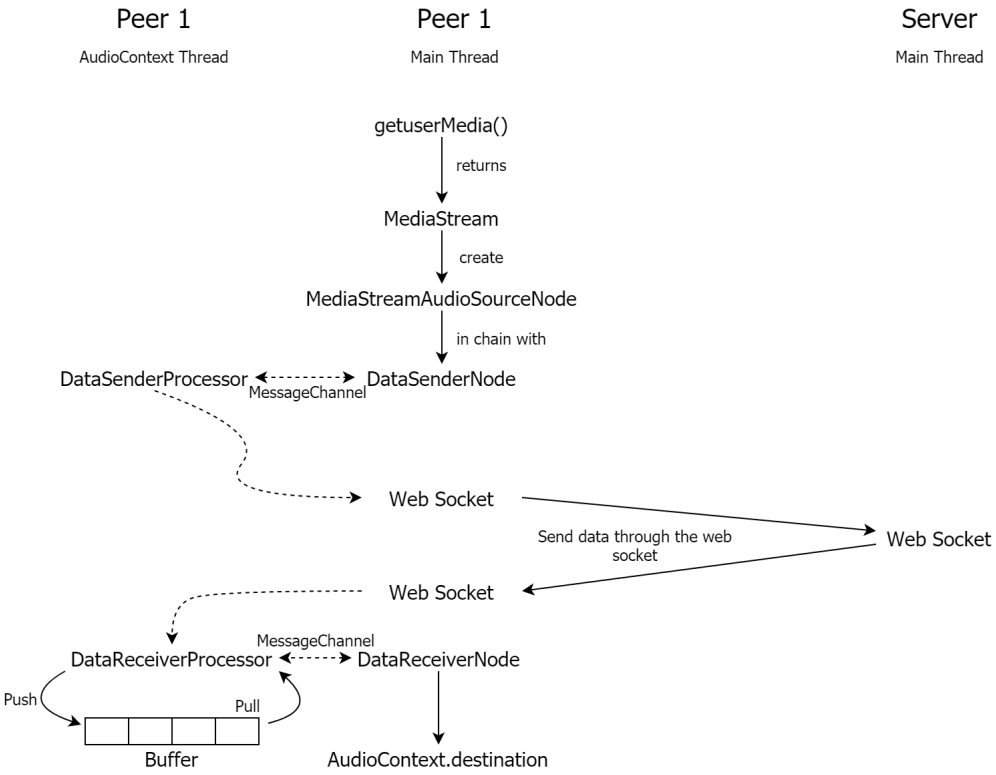

Figure 3.6. Client-Server loopback - audio chain

case, though, the peer which generates the packets is the same peers which plays them back. The full audio chain is represented in Figure 3.6. The main idea is that the client generates audio packets in the same way it does when it is connected to another peer, and sends them to the server, which simply acts as an echo server, sending back every packet received by that client. Once the client receives the packets back it puts them into the playback queue and then, once it receives as many packets as specified by the playout buffer size, it start playing back audio. The main difference between the standard DataChannel chain and the audio chain used for client-server loopback is the transmission channel. In fact, in the client-server loopback audio chain we are using WebSockets [23] instead of DataChannels. This means that the transmission protocol we are using for client-server loopback is TCP and not UDP as we do with DataChannels. This has an important side-effect on the communication mechanism because, since we are using TCP, we have some possible additional delays introduced by TCP, due to how TPC works, which means that it is possible that we experience some audio issues while using client-server loopback which are no more present once we connect with another peer. It is even possible that during the client-server loopback phase

we do not hear any audio played back, but once we connect with another peer we do not experience any issue. It is not perfect as functionality, but in most of the cases it should give the user a rough idea if everything is working, if they have enough upload/download bandwidth to support a peer-to-peer connection with at least another peer, and in case, after the peer-to-peer connection, audio is still not working, it can be used as a diagnostic tool to understand which is the peer that is not working properly. It goes by itself that, since the idea of this functionality is to test the outside network performance, the server should be located outside of the local network of the client, otherwise this test is pointless. In Figure 3.7 it is visible the GUI associated to this functionality.

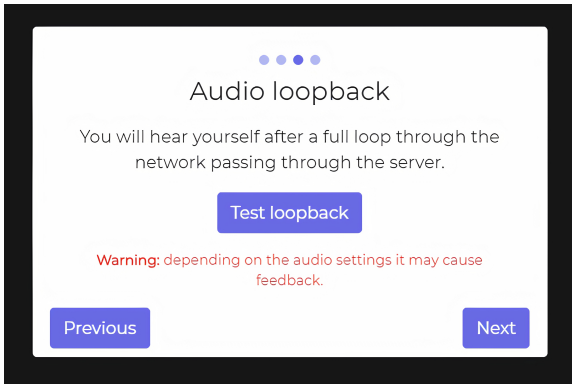

Figure 3.7. Client-Server loopback - GUI

Peer-to-peer loopback

The other type of loopback functionality present in the web application, is the peer-to-peer loopback. This functionality, as the name suggests, is available only once the user has connected to a room where there is at least another user connected, so once an actual peer-to-peer connection is established. The reason behind this functionality is to have an auditory feedback on the latency which is more-less heard by the other peer and to have a way to receive the audio back, in order to measure the total mouth-to ear latency, with also the network latency. An additional purpose of this functionality is, in case of multiple peers connected to the same room, to have a sort of mechanism to test the actual peer-to-peer connection in order to understand, in case of some audio issues, which may be the peer that is not working properly. In peer-to-peer looback, the source peer generates the packets as usual and delivers them to the other peer, which has set itself in loopback mode with respect to the first peer. When this peer receives the audio packets generated by the first peer it sends them back to the first peer, which will take care of playing them back to the user. The peer-to-peer loopback mode can be activated selectively for each of the connected peer. Two peer-to-peer

looback modes are available:

- **Network level mode**
- **Audio level mode**

Network level mode

In this mode, as soon as the packet reaches the other end of the DataChannel (the one on the peer set in loopback mode), it is sent back to the source peer. It is called network level loopback since the received packet only reaches the elements of the loopback peer involved in network transmission, and it does not go through the full audio chain of that peer. The full audio chain is represented in Figure 3.8.

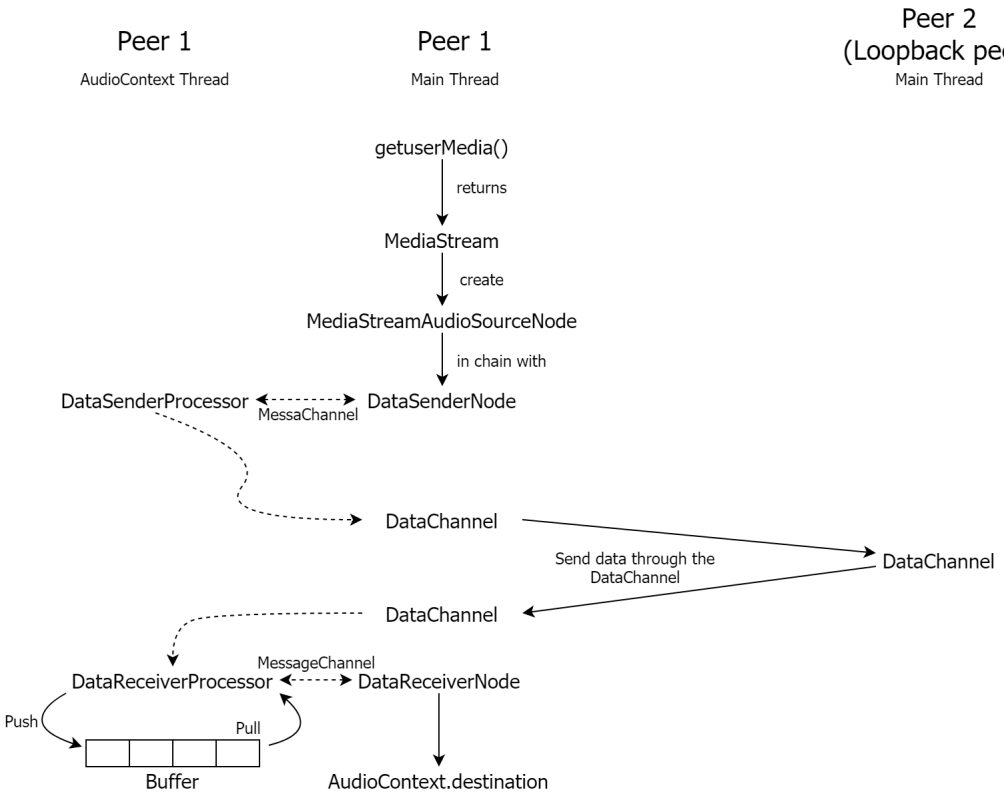

Figure 3.8. Peer-to-peer loopback - Network level mode

Audio level mode

In this mode, the packets generated by the source peer go through the entire audio chain of the loopback peer before being sent back to the source peer. In this case the audio chain is a bit more complex, since, we have the same audio chain we have

in the DataChannel solution, but on one of the peer we actually short circuited the destination with the source in order to implement the loopback mechanism. The full audio chain is shown in Figure 3.9.

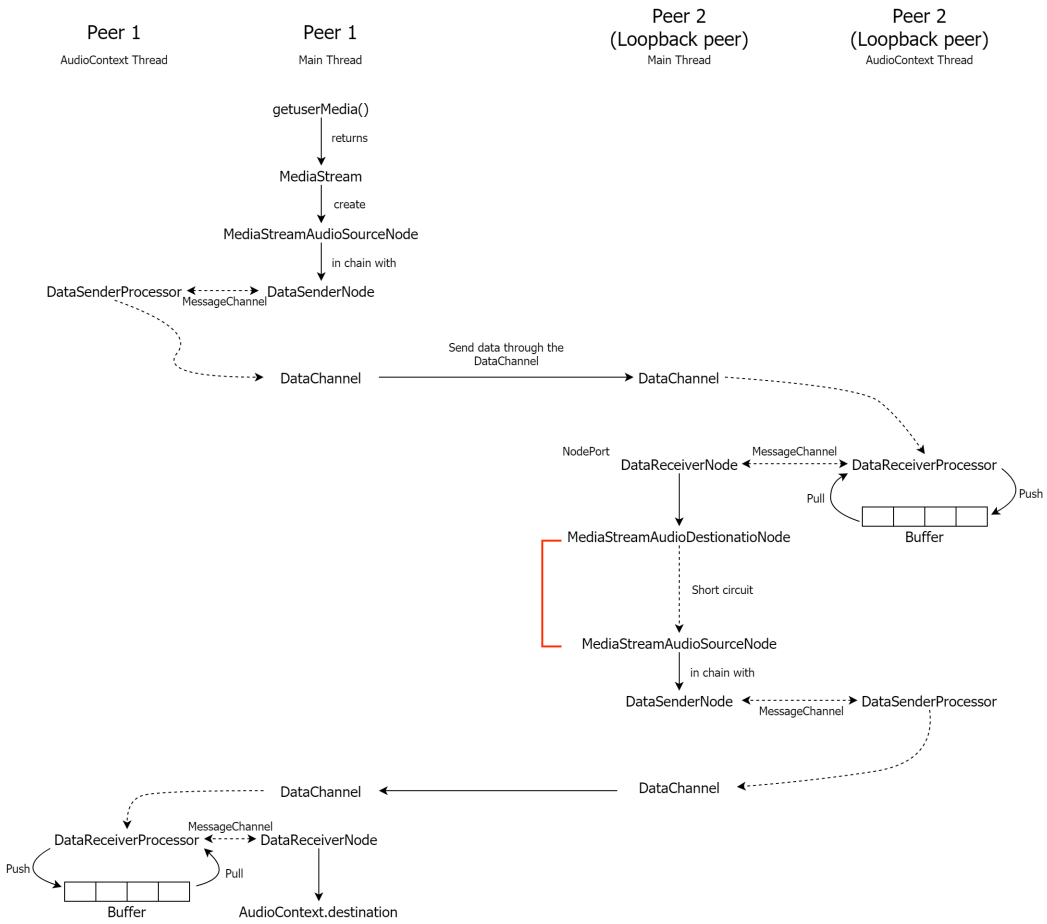

Figure 3.9. Peer-to-peer loopback - Audio level mode

In Figure 3.10 can be seen the GUI used to activate this functionality.

Through this two different modes it is possible to perform some useful latency measurements. In fact, by using the network loopback mode, if we know the round trip time (as we will see later this information is provided by the application), by performing a mouth-to-ear measurement on the source peer, we are able, once subtracted from the measured latency the round trip time, to know which is the total audio latency of the audio chain of the source peer. With the audio loopback mode instead, if we measure the total mouth-to-ear latency and we divide the result by two, we are able to estimate more or less which is the total latency of

the communication. Moreover this second mode gives the source peer an auditory feedback on the actual perceived latency during the performance.

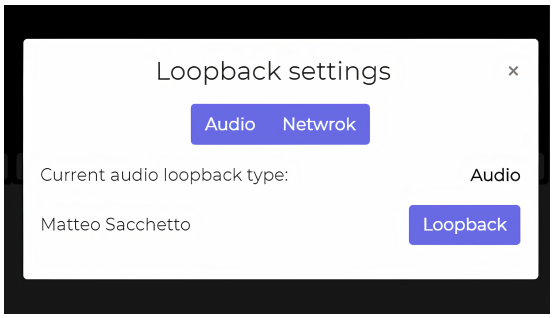

Figure 3.10. Peer-to-peer loopback - GUI

One question we have not yet fully answered: is why are all of those loopback functionality important in a networked music performance focused application? Well, as Professor Chafe explained to me, in any application which involves an exchange of audio data in real-time between different users, especially if focused on networked music performance, three steps need to be performed before creating the connection in order to make sure that everything is configured correctly and that everything is working properly. These three steps are:

1. Check that the equipment is working: this first steps is executed by the user before starting/opening the application.
2. Check that the audio routing through the application is working properly: this second step is performed by the user once the application is started. It is a check which is performed locally on the client machine.
3. Check that the network is working properly: this last step is performed by the user through an echo client or some sort of loopback mechanism. This step involves communicating with an external machine, which can be another client or can be the server

All those steps are required in order to make sure that once we connect with the other peers they are able to hear our audio, and that if they are not able to hear us, some possible source of problems are excluded from the debugging process. The loopback functionalities we implemented help us to perform the third step mentioned above. Just for completeness, also the second step is performed through the app, during the connection set-up phase, and, in order to make sure that everything is routed correctly, we have some visual and acoustic feedback for the user. To check if the webcam is working we simply show the user the image which the webcam is capturing. Instead to make sure that audio is routed

correctly we have a meter which shows the input level of the microphone and we have a button which, when clicked, plays a sine wave through the speakers. In the web application is also possible to change video/audio input device in case we want to use a different device from the default one or in case the selected one is not working properly.

3.2.2 Mono/Stereo

This next functionality is probably the most important one in the context of networked music performance, and it is the support for both **mono** and, in particular, **stereo** sources. This functionality was probably the one musicians were most exited about, but why so?

Well, if we analyze all of the video conferencing applications available today, like Zoom, Skype, Google Meet, we see that all of them have one thing in common: since they were developed only for video conferencing, all of them support only mono sources. That is reasonable, since all of this application are meant to be used only for videoconferencing where the only audio source involved is the human voice, which is mono by default.

The problem is that, especially the current situation we are living in led a lot of musicians to rely on those web application in order to exchange ideas, teach their music students, (try to) perform together, and so on, but in music, stereo is really important, and so does in the context of network music performance. From stereo audio may benefit both mono and stereo instruments, since for mono instruments, with stereo we are able to position them in the auditory field, and with stereo instruments we are able to hear all the details and colours they have. For example, if we hear the piano in mono we feel it very flat and static, while if we hear it in stereo we enjoy it much more. It immediately becomes less static and wider, since we are able to hear high notes more towards the right speaker and bass notes more towards the left one. Same is true for drums, since with stereo we are able to hear the spacial positioning of all the elements which compose a drum kit, and we are able to keep the focus on main components (bass drum and snare), which will be in the middle of the stereo image.

In order to support stereo what I needed to do was to change the packet structure as explained in section 3.1.2, and, as a consequence, I needed also to modify both the DataSender and DataReceiver AudioWorklets, with its relative circular buffer. Moreover, in the GUI I added the possibility to select the type of audio source, if mono or stereo, and in case of a mono source it is even possible to select which channel to use. One thing to note is that only the first two channel of the audio interface are supported. The GUI associated to this functionality is show in Figure 3.11

Last thing I did, partially as a consequence of the fact that we introduced stereo support, was to convert the audio samples from 32 bit float encoding to 16 bit int encoding, in order to reduce the required bandwidth, since with stereo samples we have double the data we have with mono ones. All of this is explained with more details in section 3.1.2.

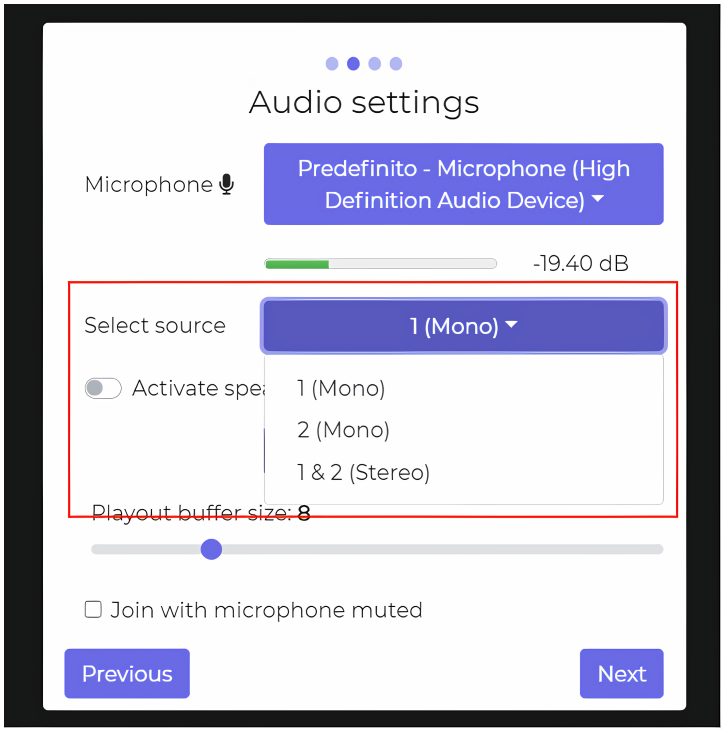

Figure 3.11. Mono/Stereo GUI

3.2.3 Statistics

The last important thing that is relevant in a web application focused on networked music performance is to have some statistics on the performance of the communication. In our case the statistics we decided to show are statistics about the quality of the communication, the round trip time, and some statistics about how the circular buffer is behaving. All the selected statistics we decided to show are visible in Figure 3.12

The first statistics shown are some information about the circular buffer behavior. In particular we are showing both graphically and numerically how many consequent packets we have currently in the buffer, on average, and which is the minimum number of packets we had in the buffer during the whole communication

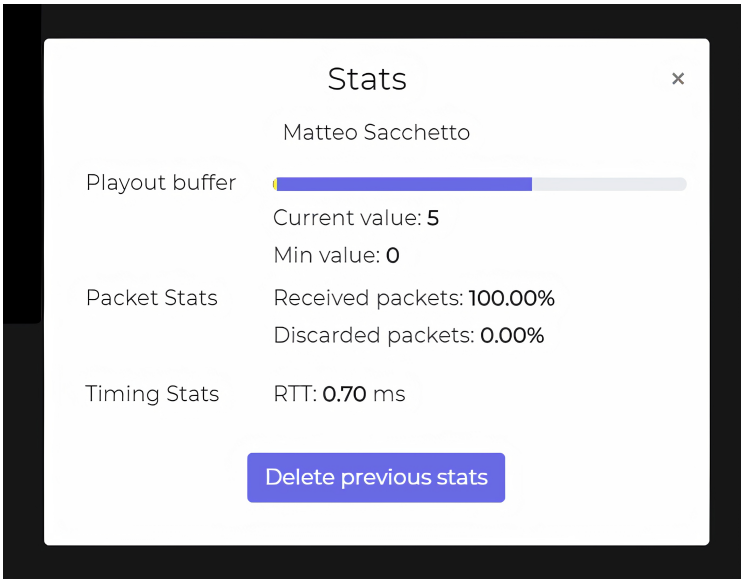

Figure 3.12. Statistics GUI

process. This statistic is updated every 500 packets, which correspond more or less to every second). The idea behind this statistic is to help the user in the process of setting the right value for the playout buffer size, by giving them some feedback on the selected value. In fact, if on average the current value is 0 then the size of the playout buffer is too small, while if it is on average 5, then it means that a good value for the playout buffer size is 5. Then it can be fine-tuned from there, but at least it gives a starting point. The graphical representation want to represent a sort of audio meter which keeps track of the minimum value.

The next statistics shown are some details about the quality of the connection. What I show here is the percentage of packets which were received correctly and therefore inserted in the circular buffer, and the percentage of packets which were discarded since they arrived too late. These percentages are calculated considering only received packets, they do not keep track of lost packets. This statistics are particularly helpful to identify network related problems like not enough upload bandwidth, network congestion, to identify machine related problems like not enough processing power, too many connections to handle, and, when combined with the previous statistics, they can help to identify if the playout buffer size is too small. This statistics are updated every second.

The last statistic shown is simply the round trip time. The round trip time is calculated in the following way. Every time a packet number multiple of 500 is created, I save its time stamp before sending it to the other peer. When the other

peer receives the packet it will send a message, through the control DataChannel, to signal the reception of the packet. The source peer, once it will receive this message, will save the time stamp and calculate the RTT as the difference between the two timestamp. Since we repeat this process every time we reach a packet number multiple of 500, it means that the RTT value is updated more-less every second. One thing to note about this mechanism of measuring the RTT is that we are using UDP during the transmission from source to destination peer and we are using TCP on the way back. This means that, sometimes the audio packet may get lost in the network, so we may not be able to update the RTT value, and that the actual RTT value can vary sometimes due to TCP delays (like re-transmission delays). On average, though, it should give a value which is representative of the actual RTT between the two peers. This statistic is helpful in order to understand which is the network latency, so if we want to perform peer-to-peer loopback measurements, we know the value which we have to subtract from the measures. Moreover, this statistic is also helpful to understand if there are problems with the network, in fact if the RTT is really high (like 2-3 s) it could indicate that the upload bandwidth of the peer we are connected with is not enough to support the communication. Instead if it is high or if it starts to increase with time it may indicate that the network is becoming congested. In both cases it can also indicate that the peer has not enough processing power to support the communication.

So, to wrap up, the combination of all this stats should give some helpful feedback to identify the source of the problem in case of some issues during the communication and it can help with the fine-tuning of the playout buffer size and with mouth-to-ear latency analysis.

Chapter 4

Results

In this chapter I will present several results. I will start from the mouth-to-ear latency measurements performed in order to fine-tune the configuration of both the `getUserMedia` and the `AudioContext`. Once found the best configuration of both elements we will move on to the mouth-to-ear latency measurements performed on the various steps of the audio chain, in order to find out the latency introduced by the different elements involved in the audio chain. In the end I will focus on the analysis of all the other aspect of the communication and all the factors which may influence its performance. We will analyze how the packet acquisition/reception delay varies in different scenarios and how the different statistics in the application can help us to identify potential problems/issues. We will analyze also which may be the causes of that variation i.e, the acquisition and reception jitter in different situations and the scalability of this application, identifying which are the factors that influence it. For each of this last test I used the fine-tuned configurations of the `getUserMedia` and the `AudioContext`.

4.1 Mouth-to-ear measurements

In a networked music performance context we can define the mouth-to-ear latency as the time it takes to audio to go from when it is captured by the microphone of a device, to when it is played back by the speaker of another device, passing through the full audio processing chain of the considered audio streaming application. This measurement is the most significant one, since it gives us an idea of the latency a user of the system will perceive. In the context of networked music performance this is the latency which needs to be kept under the 30 ms limit in order to allow a natural interaction between musicians.

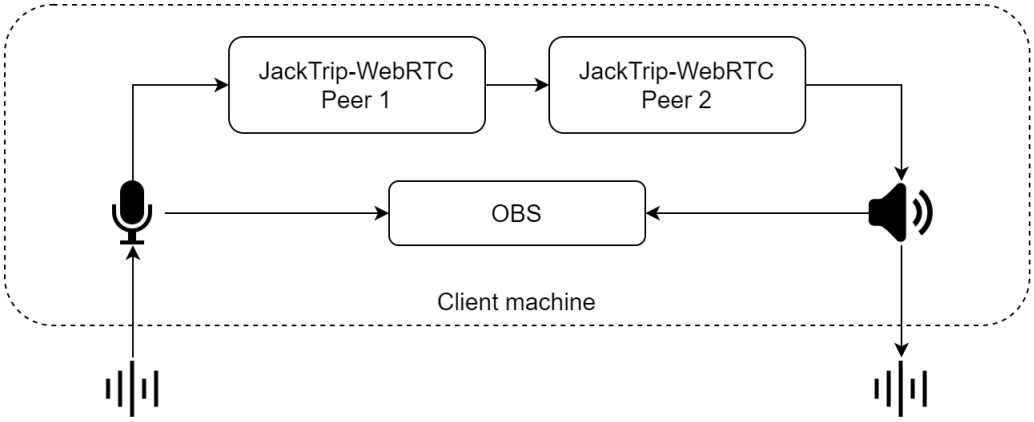

Figure 4.1. Mouth-to-ear measurement process

Since we have no control on the delay introduced by the network, and since depending on the physical location of the different peers it can vary a lot, we decided to keep network delays out of the picture. So, unless otherwise specified, all the measurements which will be presented in this and all the other sections are considering only the processing latency introduced by the full audio processing chain. In order to do so, all the measurements have been performed on the loopback interface of the same machine.

Delays have been measured by recording an abrupt sound at the input and at the output of the web application. To measure the full audio chain, from acquisition to playback, I opened two instances of the web application, this way the peer-to-peer connection is established locally to the machine and we have the full transmission chain, then I muted the audio of one of the two instances, in order to have only one peek in the audio waveform representing the audio playback. To facilitate the measurement process I used OBS [42] to record both the audio picked up by microphone and the audio played back by the web application. For ease of understanding there is a visual representation of this setup in Figure 4.1.

I then extracted the recorded audio and imported it into a DAW [11], where I measured the distance beetween the two peeks of the recorded audio, as shown in Figure 4.2.

All the measurements were performed on the same machine, which has the following characteristics:

- **CPU**: Intel i7 4720 HQ (4 cores/8 threads)
- **RAM**: 16 GB LPDDR3 (1600 MHZ)

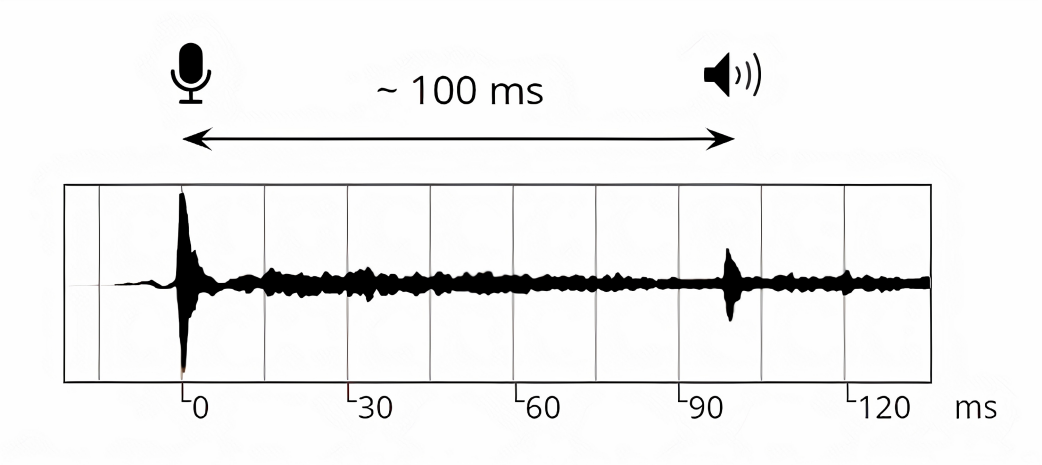

Figure 4.2. Mouth-to-ear latency measurement

I performed the measurements using the following OSes:

- **Windows 10**: version 2004 - build 19041.572
- **Pop OS 20.04** (Ubuntu derivative): version 20.04.1

And the following web browsers:

- **Chrome 86**: version 86.0.4240.111
- **Firefox 82**: version 82.0.2

All measurements have been performed using the default audio driver of each OS, so Microsoft's default driver on Windows and PulseAudio on Linux.

For each `getUserMedia` configuration and each pair (OS, web browser) I measured the latency of the standard WebRTC solution, based on MediaStreams, and the latency of the solution based on DataChannels. For DataChannels I repeated the measurement for two different values of the playout buffer size: **two** and **four** packets. We introduce the notation: DataChannel(n), to refer to a measurement of the DataChannel audio chain performed with the playout buffer size set to n.

During this first measurement phase we will only focus on the effects of the `getUserMedia` configuration on the overall processing latency, so during this first phase I will use the default configuration of the `AudioContext`, which is the following one:

```
{
    latencyHint: "interactive",
    sampleRate: 48000
}
```

Listing 4.1. Default AudioContext configuration

Once found the `getUserMedia` configuration which gives us the best results, I will proceed optimizing the `AudioContext` configuration.

4.1.1 GetUserMedia - Basic configuration

```
{
    video: true,
    audio: true
}
```

Listing 4.2. GetUserMedia - Basic configuration

I started from the basic configuration in order to have a reference point for later, when I will optimize the `getUserMedia` configuration.

Windows	Chrome	MediaStream	160 ms
		DataChannel(4)	113 ms
		DataChannel(2)	106 ms
	Firefox	MediaStream	187 ms
		DataChannel(4)	105 ms
		DataChannel(2)	103 ms
Linux	Chrome	MediaStream	125 ms
		DataChannel(4)	114 ms
		DataChannel(2)	106 ms
	Firefox	MediaStream	157 ms
		DataChannel(4)	122 ms
		DataChannel(2)	113 ms

Table 4.1. Mouth-to-ear latency measurements - GetUserMedia - Basic configuration

With the basic configuration of the `getUserMedia` function call, I measured latencies which are quite high. Let's remember that all those values represent only the processing latency introduced by the web application, without considering network delays. So if we introduce also network delays things can only get worse.

But, one interesting thing we can observe from Table 4.1, is that with the basic `getUserMedia` configuration, the DataChannel solution is able to achieve lower latencies with respect to the MediaStream solution. In particular I was able to obtain a latency which is lower by 50-80 ms, than the MediaStream latency, on Windows and by 10-40 ms on Linux. This is due to the fact that since with DataChannels we send uncompressed audio, we are able to avoid all of the additional audio processing MediaStreams perform, like encoding, decoding, etc. Moreover, with the DataChannel solution we have more control over the transmission process and the audio playback, this means that with DataChannels solution we are also able to avoid/have more control on the buffering mechanism present in both phases.

Unluckily, as we can see, all of those values are well above the 30 ms latency required by the networked music performance context in order to allow a natural interaction between musicians. This means that, if we want to reach the goal of creating a web application focused on networked music performance we need to fine-tune this configuration.

4.1.2 GetUserMedia - Advanced configuration

```
{
    video: true,
    audio: {
        autoGainControl: false,
        echoCancellation: false,
        noiseSuppression: false,
        latency: {
            min: 0.01,
            max: 0.02
        },
        sampleRate: 48000,
        sampleSize: 16,
    }
}
```

Listing 4.3. GetUserMedia - Advanced configuration

In this configuration I disabled auto gain, echo cancellation and noise suppression. Then I set a constraint on the required latency of the input device, by declaring a range of accepted values. Min and max values where set as the minimum values I could use for a range of devices (PC and smartphones) without incurring into

an `Overconstrained Error` exception. I then set constraints on the sample rate and the bit depth of the audio samples.

Windows	Chrome	MediaStream	121 ms
		DataChannel(4)	84 ms
		DataChannel(2)	78 ms
	Firefox	MediaStream	127 ms
		DataChannel(4)	84 ms
		DataChannel(2)	78 ms
Linux	Chrome	MediaStream	121 ms
		DataChannel(4)	94 ms
		DataChannel(2)	96 ms
	Firefox	MediaStream	114 ms
		DataChannel(4)	90 ms
		DataChannel(2)	91 ms

Table 4.2. Mouth-to-ear latency measurements - GetUserMedia - Advanced configuration

As we can see in Table 4.2, with a more fine-tuned configuration I was able to achieve lower latencies than with the basic configuration, and this is true for both solutions. On Windows I was able to reduce latency by 40-60 ms with MediaStreams, and by 15-30 ms with DataChannels. On Linux I was able to reduce it by 10-40 ms with MediaStreams and by 10-30 ms with DataChannels. This shows us that a fine-tuned configuration of the `getUserMedia` function call is crucial in order to reach our goal, since, not only the standard WebRTC solution benefits from it, in terms of latency, but, also the DataChannel solution does, since in both solutions the first step of the audio chain is the same, and it involves MediaStreams.

Anyway, we can see that also with this configuration the DataChannel solution is generally able to achieve a lower mouth-to-ear latency than the traditional WebRTC's solution, saving around 40 ms on Windows and around 20 ms on Linux.

Unluckily, with this configuration, the DataChannel solution was not behaving correctly in some browsers. In fact, for some reasons (may be due to the latency constraints), especially on Chrome, after a certain amount of time peers became silent and did not recover from that state. In order to find out why the peers became silent I collected, through the User Timing API [43], some timing statistics on when the packets were acquired (generated), sent, received, discarded and played back. Since each of the functions involved in that steps should be called every 2.67 ms, with a more-less precise timing, I plotted the difference between

when the function was called and when it should have been called, using as a reference point the timing of the first packet.

What I noticed was that, for some unknown reason, there was an increasing delay introduced already during the packet acquisition step, delay which was not decreasing with time. This increasing delay caused packets to be generated late, with respect to when they should have been generated, so they were actually sent and received late and for this reason they were discarded by the receiving peer. Since this delay was always increasing without decreasing, after a certain amount of time all the packets received by the other peer were discarded, since they were considered invalid because received too late, and since all the packets were discarded the peer generating packets actually becomes silent to the ears of the receiving peer.

In Figure 4.3 I am plotting the difference between the actual timing of the creation of each packet and the ideal one, of a capture of the first 20 000 packets (53 s), performed on Windows 10 with Chrome 85, where the `getUserMedia` configuration is the one reported above (4.3), and the playout buffer size is set to 4. On the x-axis it is reported the packet number, on the y-axis it is reported the difference in ms between the actual timing of the packet creation and the ideal one, which represents the actual delay introduced in the full audio chain.

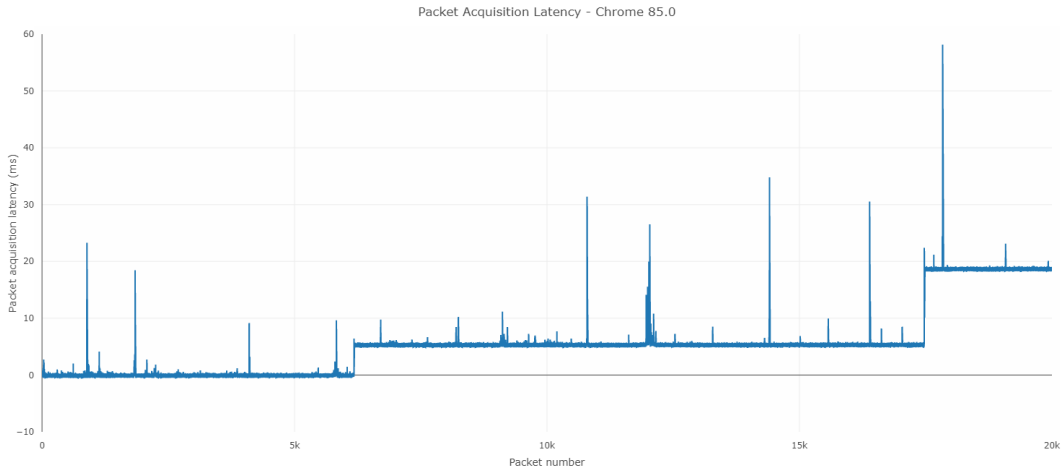

Figure 4.3. Packet Acquisition Latency - Chrome 85

As we can see from the above plot, after around 6 000 packets there is the first step, which introduces around 6ms of delay, and after around 15 000 packets there is the second one, which introduces a total delay of around 19ms. As it is clearly visible in the plot, once the delay is introduced it is there and it does not decrease with time. Since the playout buffer size is set to 4, we can tolerate the loss of

up to four packets, which means that we are able to bear up to around 10 ms of delay. Unluckily after around 15 000 we go above the 10 ms limit, which means that from that point on all the packets generated by this peer will be discarded by the other one and so that means that this peer will become silent from the point of view of the other peer. Unluckily, since we are generating 375 packets/s, we reach the 15 000th packet after only 40s of audio streaming. Increasing the size of the playout helps to delay the moment when all packets generated by a peer are discarded by the others, but it is not a solution, since the delay keeps increasing, potentially reaching some very high values like 300 ms, as shown in Figure 4.4, value reached after only 53 000 packets (2 m 21 s). So, in order to solve this issue I had to find what caused this strange behaviour.

After a lot of debugging and testing I found out that the issue was probably caused by the above `getUserMedia` configuration (4.3), which introduced those delays directly in the call of the `process` function of the DataSenderProcessor, function which is executed in a different thread with respect to the main JavScript thread, and whose call is scheduled, by the browser, to be every 2.67 ms (this time depends on the selected sample rate, which is 48 KHz in our case).

In order to solve this issue I had to change the `getUserMedia` configuration to one less restrictive.

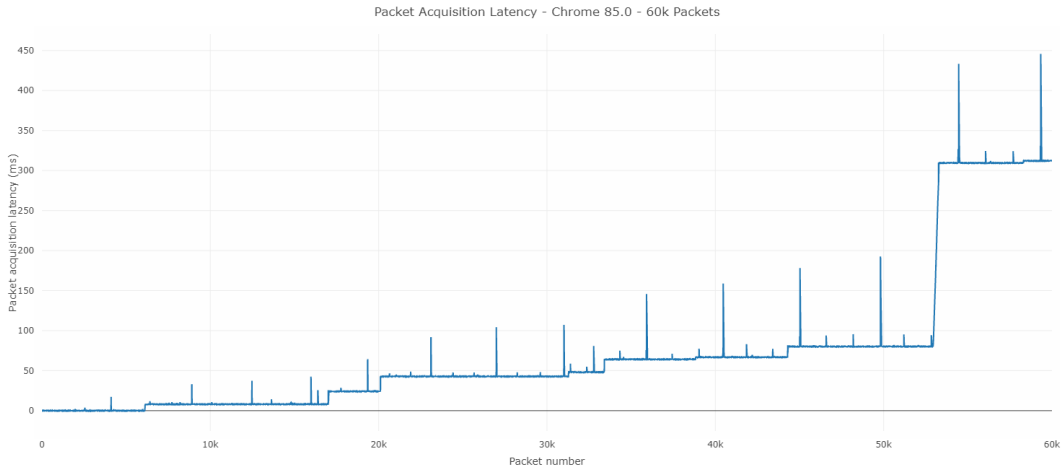

Figure 4.4. Packet Acquisition Latency - Chrome 85 - 60 000 Packets

4.1.3 GetUserMedia - Best configuration

```
{
    video: true,
    audio: {
        autoGainControl: false,
        echoCancellation: false,
        noiseSuppression: false,
        latency: 0
    }
}
```

Listing 4.4. GetUserMedia - Best configuration

After a lot of testing this is the configuration I found which solved the issue we were experiencing before. In this configuration I am actually relaxing the constraint on latency, in fact, by setting `{latency: 0}` we are actually letting the browser select the lowest value of latency it can, without incurring in the `Overconstrained Error` exception. Moreover, I removed the last part of the previous configuration (4.3), since it was not contributing in reducing the latency, but it may have contributed in the misbehaviour I was experiencing. As in the previous configuration I kept disabled auto gain, echo cancellation and noise suppression, since by doing so we are able to avoid unnecessary audio processing and so save precious time.

Windows	Chrome	MediaStream	102 ms
		DataChannel(4)	83 ms
		DataChannel(2)	78 ms
	Firefox	MediaStream	115 ms
		DataChannel(4)	89 ms
		DataChannel(2)	83 ms
Linux	Chrome	MediaStream	115 ms
		DataChannel(4)	99 ms
		DataChannel(2)	95 ms
	Firefox	MediaStream	117 ms
		DataChannel(4)	83 ms
		DataChannel(2)	93 ms

Table 4.3. Mouth-to-ear latency measurements - GetUserMedia - Best configuration

As can be seen in Table 4.3, with this configuration I was able to achieve similar

or slightly lower latencies than with configuration 4.3. Again, as we already saw with previous two configurations, with the DataChannel solution I was able to obtain a latency which is about 20-30ms lower than the MediaStreams solution. But, most importantly, with this configuration I was able to solve the issue I was experiencing on Chrome with the previous configuration (4.3). In fact, as can be seen in Figure 4.5, with this configuration the delay stays constant.

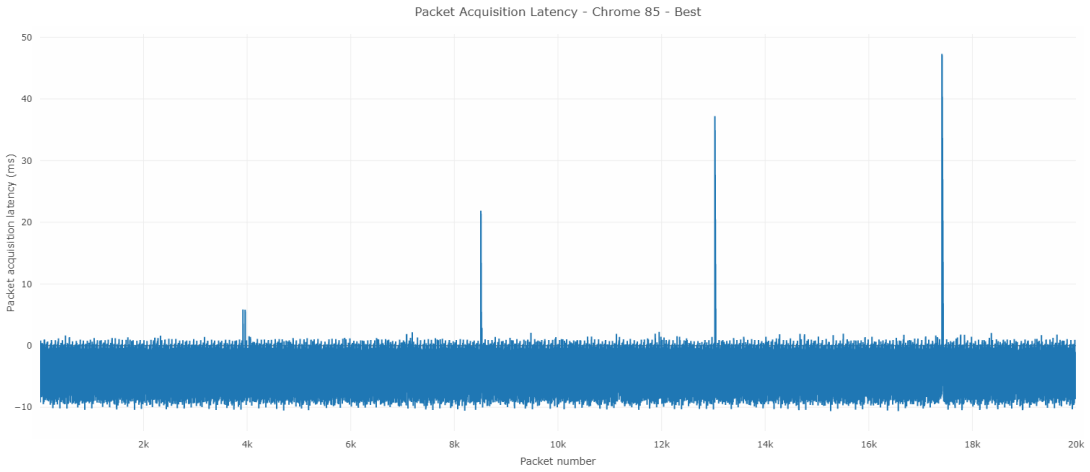

Figure 4.5. Packet Acquisition Latency - Chrome 85 - Best configuration

Now that I've finished fine-tuning the `getUserMedia` configuration, we can focus on fine-tuning the AudioContext configuration

4.1.4 AudioContext - Best configuration

```
{
    latencyHint: 0,
    sampleRate: 48000
}
```

Listing 4.5. AudioContext - Best configuration

It is not actually well documented but, after some research and some suggestions from other people involved in this project, I discovered that by setting `{latencyHint: 0}` we are actually telling the browser to select, for the AudioContext, the lowest possible latency it can handle. To check which of the two configurations (4.1 or 4.5) leads to a lower latency value for the `AudioContext`, I checked the value of the `baseLatency` attribute of the `AudioContext` object, and

discovered that, by setting `{latencyHint: 0}` we are able to achieve in Chrome a `baseLatency` of 0.00267 s, while with `{latencyHint: "interactive"}` we obtain a value of `baseLatency` of 0.01 s. On Firefox both the configuration lead to the same result: a value of `baseLatency` equal to 0 s. So, generally, with `{latencyHint: 0}` we should be able to achieve a lower or equal latency than with the default configuration. Since we are focusing on reducing the latency at its lowest, this is the configuration which should give the best results.

Windows	Chrome	DataChannel(8)	54 ms
		DataChannel(4)	45 ms
		DataChannel(2)	37 ms
	Firefox	DataChannel(8)	85 ms
		DataChannel(4)	74 ms
		DataChannel(2)	65 ms
Linux	Chrome	DataChannel(8)	105 ms
		DataChannel(4)	107 ms
		DataChannel(2)	98 ms
	Firefox	DataChannel(8)	105 ms
		DataChannel(4)	83 ms
		DataChannel(2)	97 ms

Table 4.4. Mouth-to-ear latency measurements - AudioContext - Best configuration

As we can see in Table 4.4, the configuration of the AudioContext is another crucial step. In fact, just by setting `{latencyHint: 0}` in the configuration of the AudioContext, I was able to reduce the total mouth-to-ear latency by 40 ms on Chrome and by 15 ms on Firefox on Windows 10. One note about Linux, as we can see from the above results, the `AudioContext` configuration seemed to not have any effect on the mouth-to-ear latency measured on Linux, but this is probably because, as described in the introduction of this section, on Linux we are performing measurements using PulseAudio, which, by default uses a high value for the size of its internal buffer. Probably by configuring PulseAudio appropriately or by using something else, like JACK [15], to manage audio instead of PulseAudio, latency should reduce.

Anyway, with this configuration we reached a very good latency result, since I was able to achieve a latency of under 50 ms using Chrome on Windows 10. 50 ms is not 30 ms but it is still impressive given the fact that it was achieved through a web browser.

One thing to note is that, while all the measurements where performed with a **playout buffer size** of 2 and 4, in order to be able to compare the different

configurations, those values are actually quite low.

As we can see from Figures 4.5 and 4.6, we have a jitter of around 10ms, so in order to avoid glitches we need at least a playout buffer size of 4.

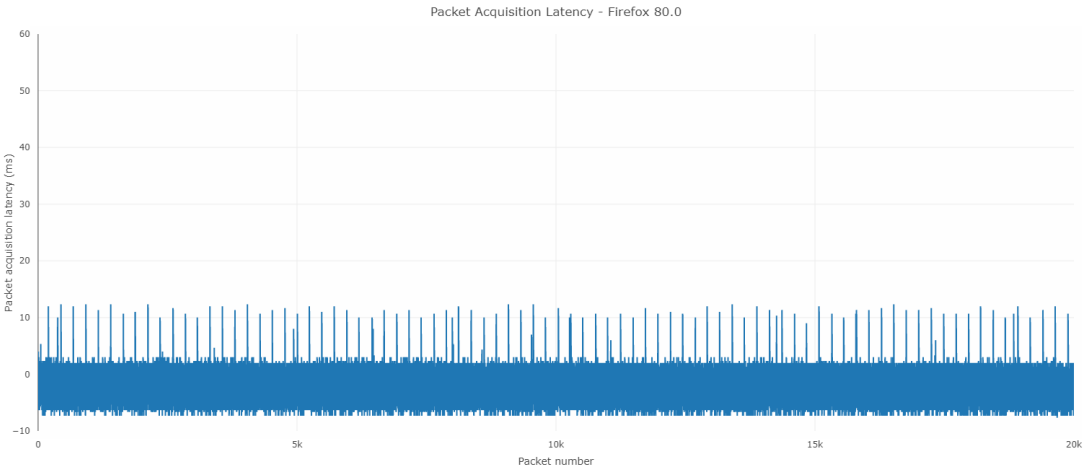

Figure 4.6. Packet Acquisition Latency - Firefox 80 - Best configuration

Due to the fact that, in particular on Firefox, we have some regular peeks of additional 10ms, in order to tolerate a variation of 20ms we need at least a playout buffer size of 8, which is the default value set for the application. In a real case scenario, where also network is involved in the communication, the amount of jitter can only increase, so in this case the playout buffer size becomes an important parameter to tune, anyway as a general rule of thumb I suggest to never go below 4 on Chrome and 8 in Firefox. One thing to keep in mind when tuning the playout buffer size is that by increasing its size by 1 you are introducing an additional 2.67ms of latency, so its size should be set as the lowest value which allows you to avoid audio glitches, in order to have the lowest possible latency.

For this reason, in the Table 4.4, I also reported the latency measures obtained with a playout buffer size of 8, to have a sort of reference of how this configuration behaves with a more significant value for the playout buffer size and to have a sort of latency reference below which we can not go in a real case scenario, once we introduce network in the communication process.

4.2 Audio chain delay measurements

Now that we found the best configuration for both the `getUserMedia` and the `AudioContext`, we decided to perform some measurements on the latency introduced by the different steps of the audio chain. In order to do so, we took the DataChannel audio chain shown in Figure 3.2 and we created a sort of short circuit at three different levels, in this way we are able to measure the latency introduced by the different elements.

The three audio chains we identified are the following:

- **A - MediaStream chain**: in this case we have no audio processing, so we are just measuring the input-to-output delay by connecting directly the input device to the output one.
- **B - MediaStream + AudioWorklets chain**: in this case we introduce the audio processing performed by the AudioWorklets.
- **C - MediaStream + AudioWorklets + DataChannel chain**: in this case we introduce also the DataChannel chain. This audio chain is actually the full audio chain shown in Figure 3.2.

The measurements were performed using the best configurations found before.

For each audio chain we considered two different configurations:

- **1**: *MediaStream* → *MediaStreamAudioSourceNode* → *AudioContext.destination*. In this case we are using as destination of our audio chain the default destination of the Web Audio API, which corresponds to the system default output device.
- **2**: *MediaStream* → *MediaStreamAudioSourceNode* → *MediaStreamAudioDestinationNode*. In this case the output device is configurable and it can be different from the system default one.

For the audio chain A (MediaStream chain), we defined an additional configuration:

- **0**: *MediaStream* → *HTML Audio TAG*. In this case we are not going through any Web Audio API's node, so we do not have any Web Audio API processing.

As for previous measurements, I am measuring latency on the localhost interface of the same machine used in previous section (4.1), so I am again keeping network out of the picture.

4.2.1 MediaStream chain

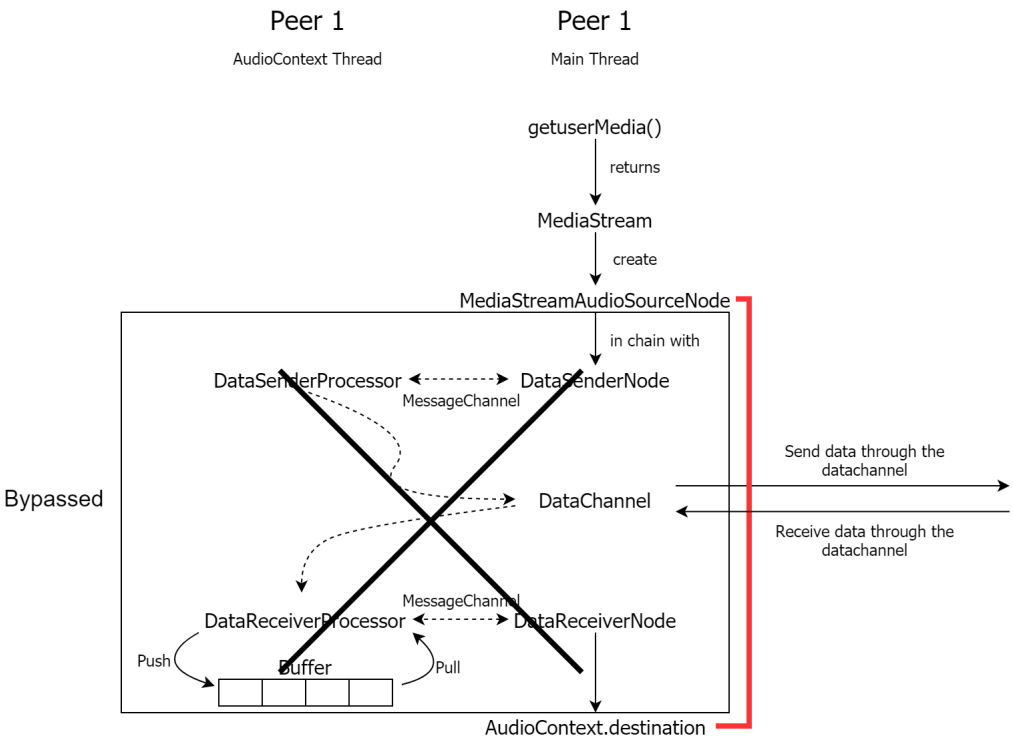

Figure 4.7. MediaStream chain

In this first scenario I am creating a short circuit between the audio source and destination, so between the MediaStreamSourceNode and the destination, bypassing all the processing performed by AudioWorklets and bypassing the transmission over the DataChannel. This first configuration involves **only one peer**, and gives us an idea of the latency which is introduced by the first and last steps of the audio chain. Over the latency introduced by those steps we do not have much control, the only way we can influence it is through the configuration of the `getUserMedia` and the `AudioContext`. This first scenario gives us an idea of the base latency we have in our chain, since we are excluding all the other latencies (processing, transmission, network, ...), and gives us a way to understand which is the best approach to playing back audio, in order to achieve the lowest possible latency.

As we can see from Table 4.5, the solution based on using the Web Audio API's default output destination (Configuration **1**) is the winning solution in order to achieve the lowest possible latency. This is particularly visible in Chrome on Windows 10, where I was able to reduce the latency introduced by this first step by

Windows	Chrome	Configuration 0	114 ms
		Configuration 1	27 ms
		Configuration 2	98 ms
	Firefox	Configuration 0	83 ms
		Configuration 1	53 ms
		Configuration 2	93 ms
Linux	Chrome	Configuration 0	131 ms
		Configuration 1	98 ms
		Configuration 2	155 ms
	Firefox	Configuration 0	59 ms
		Configuration 1	73 ms
		Configuration 2	77 ms

Table 4.5. Mouth-to-ear latency measurements - MediaStream chain

almost 90 ms with respect to the configuration based on using only the MediaStream. This is true also for the other (OS, web browser) couples, where I was able to reduce it by 30-40 ms. The only exception to this is Firefox on Linux, where I obtained the lowest latency by not going through the Web Audio API at all (Configration **0**). Another thing that is visible in Table 4.5 is that in general the configuration where we can select the audio output device is the one which gives the worst performance, even worse than the MediaStream only configuration (Configration **0**). This is probably due to the fact that in this configuration the MediaStream is converted into a Web Audio API's node and then it is converted back into a MediaStream, so, we end up with the same MediaStream we had in the beginning, but in order to obtain it we introduced some additional latency due to the unnecessary conversion process. Unfortunately, in the Web Audio API, the last configuration is the only one where we can select the output device without needing to change the system setting, so that is why we considered relevant also this configuration.

4.2.2 MediaStream + AudioWorklets chain

In this second scenario I am creating a short circuit at the AudioWorklet's level, so I am including audio processing in the audio chain but I am bypassing the transmission mechanism (the DataChannel). So, also this configuration involves **only one peer**, and by comparing the result of this scenario with the one obtained in the previous case we have an idea of which is the latency introduced by the processing mechanism (the AudioWorklets). One thing to note, is that, since now I am introducing the AudioWorklets, in particular the DataReceiver AudioWorklet,

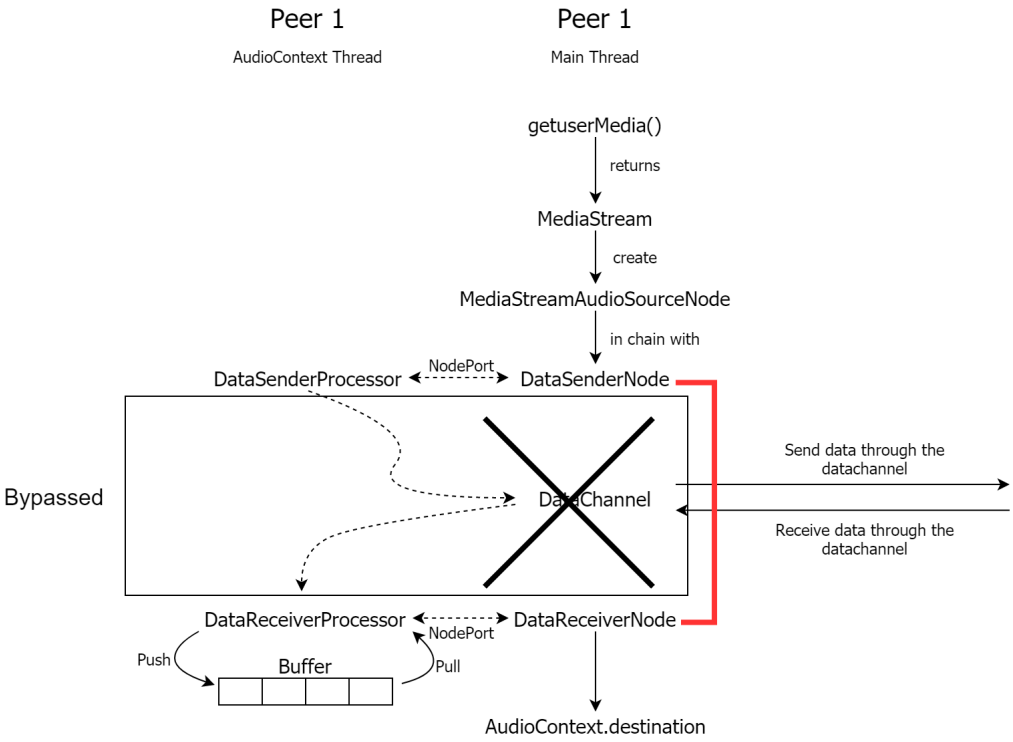

Figure 4.8. MediaStream + AudioWorklets chain

in the audio chain, I am also introducing the play back queue, so the size of the playout buffer becomes another parameter which influences the measured latency. Given what we discovered previously, during the mouth-to-ear measurements, and in order to be able to compare the different latency measures obtained, I set the playout buffer size to 8 and kept it fixed at that value for all the measurements performed from here on. Since I am now introducing audio porcessing, the configuration **0** can no longer be used, so from now on we will only perform measurements of configuration **1** and **2**.

As we can see, I obtained again the best results in Chrome on Windows 10 with the configuration **1**. In particular, if we subtract from the 51 ms I obtained now, the 27 ms measured in the previous scenario, we obtain a latency of 24 ms, which is the total latency introduced by the processing step of the audio chain. Given that the playout buffer size is set to 8, which correspond to around 21 ms of latency due to the queue mechanism, it means that we have more or less 3 ms of actual processing latency caused by the AudioWorklets, the MessageChannels and also measurements errors, which is something really impressive, since it means

Windows	Chrome	Configuration 1	51 ms
		Configuration 2	122 ms
	Firefox	Configuration 1	79 ms
		Configuration 2	95 ms
Linux	Chrome	Configuration 1	98 ms
		Configuration 2	145 ms
	Firefox	Configuration 1	80 ms
		Configuration 2	80 ms

Table 4.6. Mouth-to-ear latency measurements - MediaStream + AudioWorklets chain

that in this case AudioWorklets introduce around 128 samples of latency, which is really low. I obtained a similar result also with Firefox on Windows 10, with a total processing latency of 26 ms. This shows really well how powerful and revolutionary AudioWorklets are in the context of low latency audio processing with a web browser. On Linux the results didn't change much with respect to the one obtained in the previous scenario, and this is probably due to the size of the internal PulseAudio's buffer, as explained in the section 4.1.

4.2.3 MediaStream + AudioWorklets + DataChannel chain

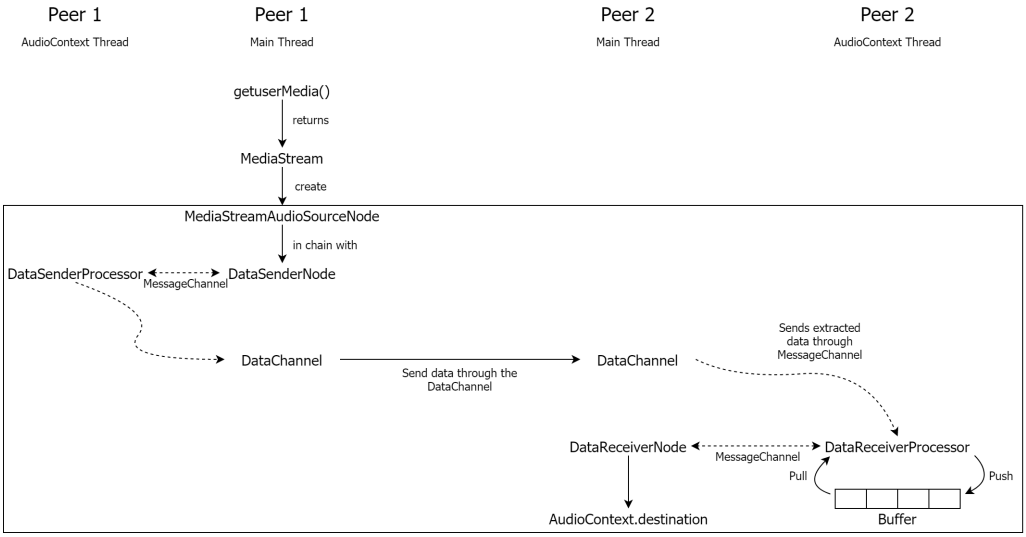

Figure 4.9. MediaStream + AudioWorklets + DataChannel chain

In this last scenario I am actually measuring the latency introduced by the whole audio chain, which includes a source connected to a DataSender AudioWorklet on one peer, a DataReceiver AudioWorklet connected to a destination on the other peer and a DataChannel to allow peers to exchange data, so, it goes without saying that in this scenario we have **two peers** involved. If we then compare the results obtained with this audio chain with the results obtained through the previous audio chain we are able to have an idea of the delay introduced by the transmission mechanism, so by the DataChannel.

Windows	Chrome	Configuration 1	54 ms
		Configuration 2	165 ms
	Firefox	Configuration 1	107 ms
		Configuration 2	143 ms
Linux	Chrome	Configuration 1	102 ms
		Configuration 2	170 ms
	Firefox	Configuration 1	88 ms
		Configuration 2	112 ms

Table 4.7. Mouth-to-ear latency measurements - MediaStream + AudioWorklets + DataChannel chain

In this case the results are a bit different across browsers. As we can see in Table 4.7, on Chrome the DataChannel introduced between 3 and 5 ms of latency, while on Firefox it introduced between 8 and 30 ms with respect to the results obtained in the previous scenario. The difference may be caused by actual differences between browser or simply by measurements imperfections. What is clearly visible at the end of all of those measurements is that in general on Windows 10 Chrome performs better than Firefox, while on Linux is the opposite. Moreover we can also see that the configuration **2** is the one which consistently gives the worst results, this is caused by the conversion from Web Audio API's node back to MediaStream. For this reason we ended up including in the web application the possibility to select the audio output device directly from the GUI, if, for some reason, the user can not or does not want to modify the system audio settings, but, since the selection of the audio output device through the Web Audio API can only be performed as we did it in the configuration **2** (at least for now), this functionality is disabled by default. If the user wants they can activate this functionality, but we are informing them of the fact that by activating it some additional latency may be introduced. Given that, we highly recommend to change the system default settings, in case you need or want to change the audio output device, since, by doing so, the application keeps using the Web Audio API's default audio destination, which is the one used in configuration **1** and, as already seen, is the one which gives best results.

4.3 Latency variation

In this section we will focus on analyzing how the measured latency varies from one execution to the other one and which is the range of values in which we expect the actual latency to be. This measurements help us understand which is the average latency we have with the different web browsers and OSes, allowing us to compare them more precisely. In order to have a better understanding of how the latency varies I measured the local latency (no network latency involved) of the full DataChannel audio chain, represented in Figure 3.2. The measurement was performed by using the best configuration of the `getUserMedia()` and the `AudioContext`, found in section 4.1 and by setting the playout buffer size to 8. For each pair (OS, web browser) defined in the section mentioned above I recorded ten times an abrupt sound (which in practice was a finger snap) then I restarted the application and recorded other ten finger snaps, repeating this process ten times for each pair, obtaining 100 measures for each pair (OS, web browser).

I then proceeded to plot all those measurements and to evaluate the mean and standard deviation for each of the pair. The graph I obtained is a box plot and is shown in Figure 4.10. For reference in Table 4.8 I reported all the different values with the addition of the 3σ value, which is useful to identify the range where the 99.7% of the measures should land.

OS	**Web browser**	**Median**	**Mean**	σ	$\mathbf{3\sigma}$
Windows	Chrome	55 ms	55.81 ms	3.30 ms	9.90 ms
	Firefox	93.50 ms	92.20 ms	6.55 ms	19.65 ms
Linux	Chrome	88 ms	84.21 ms	6.76 ms	20.28 ms
	Firefox	68.50 ms	70.90 ms	10.98 ms	32.94 ms

Table 4.8. Latency variation

As we can see from both the graph and the table, Chrome on Windows is the pair which gives us more consistent results, in fact not only we have the lowest average latency, of around 56 ms, but we also have the lowest variation (3σ) of just around 10 ms, which means that (almost) all measures should fall in the range [46 ms, 66 ms], with the majority of them being in the range [53 ms, 59 ms]. Firefox on Windows and Chrome on Linux showed a similar variation, of around 20 ms, but with a slightly lower mean latency of Chrome on Linux with respect to Firefox on Windows. With Firefox on Linux we have a kind of strange behavior, in fact it is true that Firefox on Linux is giving on average a lower latency than Chrome on Linux, as we already saw in previous measurements, but it is also true that its the less consistent among the four pairs, with a 3σ variation of more than 30 ms,

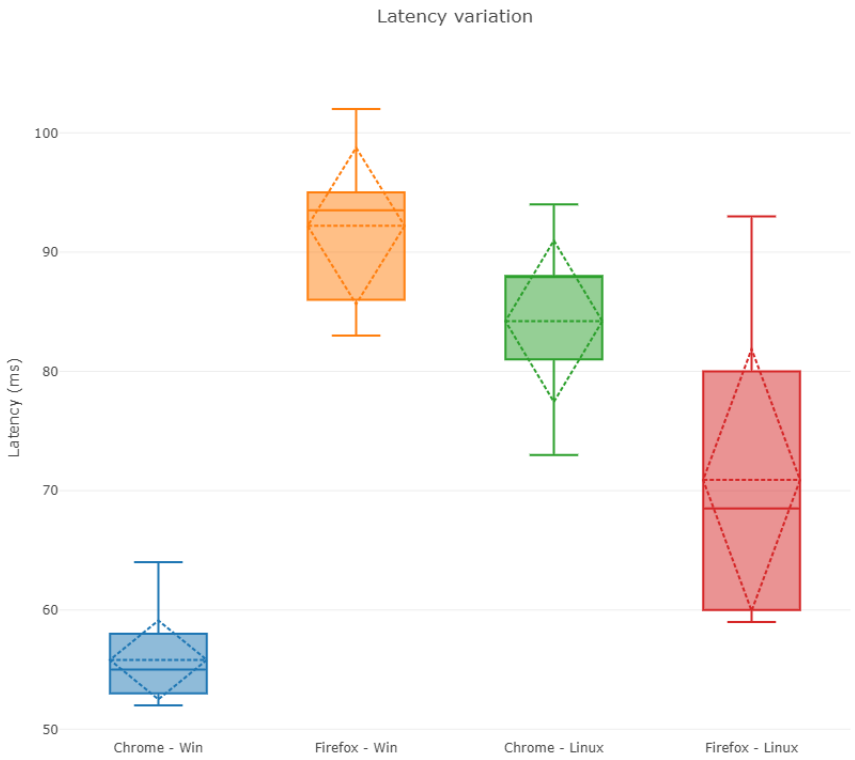

Figure 4.10. Latency measurements. For each configuration, the box is drawn from Q1 to Q3 with a horizontal line drawn in the middle to denote the median, then the two lines at the extremes show the minimum and maximum value measured, the dotted line within the box shows the mean and the dotted rhombus shows the mean ± the standard deviation.

which means that in theory sometimes we could measure a latency as low as 40 ms other times we could measure one greater than 100ms.

One thing to note is that, while these measurements give us a clearer idea of how the different web browsers behave, on average, on the different OSes, a lot of parameters can influence their behavior. So, in order to have a better representation of this, a lot more measures should be collected and it would be better to collect them in groups of four/five consecutive measurements with a break between each group of measurements, to keep in consideration also the differences in the execution environment. Due to time limitations and mainly due to the fact that the collection and evaluation process was executed manually, I decided that 100 measures where enough for our purpose.

4.4 Acquisition and reception latency

Let's begin with the packet acquisition and reception delays. In the first scenario we are analyzing a normal situation where there are no issues, and the communication just works. The test was performed on Chrome 86, by collecting the timings of 20 000 packets (375 packets = 1 s), on the localhost interface, so network is not involved in the communication, and the playout buffer size is set to 8. The graphs show the difference between when the packet has been created/received and the ideal timing when it should have been created/received. As we can see in Figures 4.11 and 4.12, since network is not involved in the communication process, the reception jitter is similar to the acquisition jitter, and it is around 2ms. One additional thing we can notice especially from the acquisition latency (Figure 4.11) is that, thanks to the optimization of the `AudioContext` configuration, the acquisition jitter reduced a lot with respect to Figure 4.5, where the `AudioContext` configuration was not optimized. In the last Figure 4.13 we can see the packetization of the DataChannel, where with packetization I mean the number of packets in the DataChannel buffer each time a packet is sent. The value of 1 means that the packet is delivered before the next one is sent, while a value higher than 1 means that the packet has been buffered and it will be delivered together with other packets. This graph can be helpful in order to understand if during the communication everything worked correctly or not. In this case we can see that the highest value reached during the test was 4, and since this value is lower than the the playout buffer size, it means that the circular buffer was able to keep the playback going, without any issue. We can also see that during almost all the communication the packets were delivered as soon as they were sent, so it means that the peer should not have experienced any audio glitch or issue.

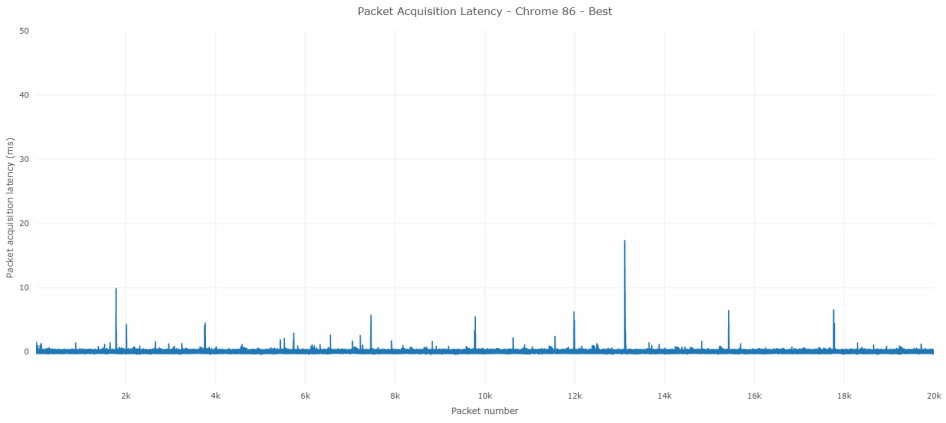

Figure 4.11. Packet Acquisition Latency - Chrome 86 - 1

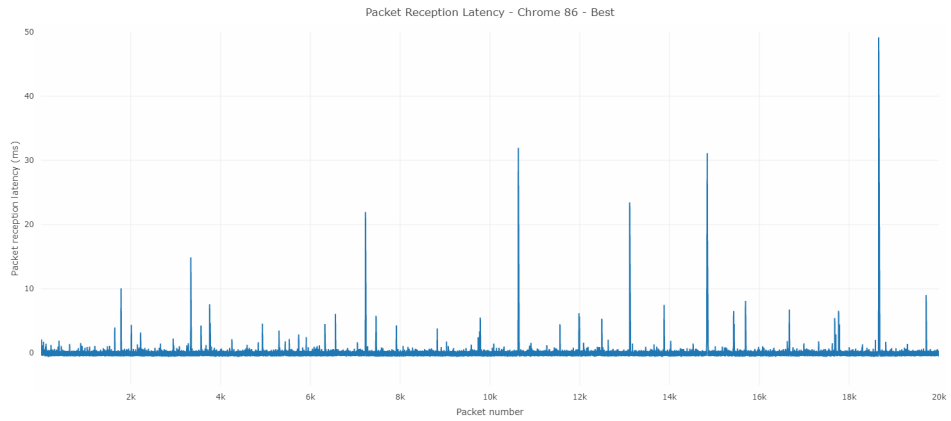

Figure 4.12. Packet Reception Latency - Chrome 86 - 1

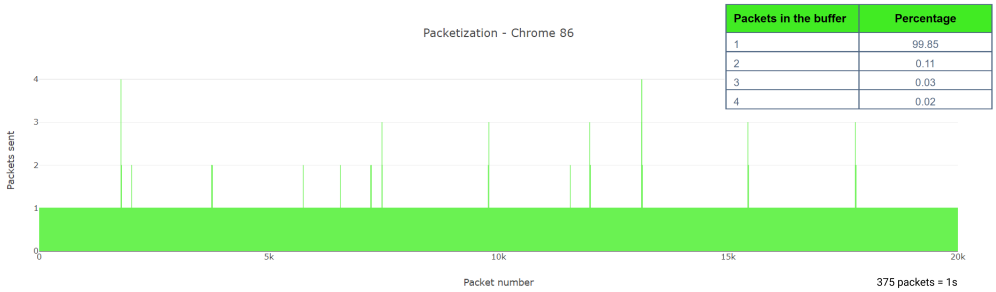

Figure 4.13. Packetization - Chrome 86 - 1

In the next scenario I tried a connection between the same machine used previously, which is the same machine used during all the tests, and a quite recent Android smartphone (CPU: Qualcomm Snapdragon 855). In this case the connection involved a wireless local network connection between the two clients.

As we can see from Figure 4.14, the Android smartphone is less powerful than the PC and this leads to a higher acquisition jitter, of around 10ms. Moreover, from Figure 4.15 we can see that, due to the fact that a wireless network connection is involved in the process, the acquisition and reception jitter differ this time around, with a reception jitter up to 30ms. The fact that the reception jitter goes above the amount of jitter that the playout buffer is able to compensate for, means the some packets arrived late, from the point of view of the receiver, and thus they were discarded. This is also confirmed by the statistics showed in Figure 4.17, where we can see that around 5% of the first 20 000 packets were discarded.

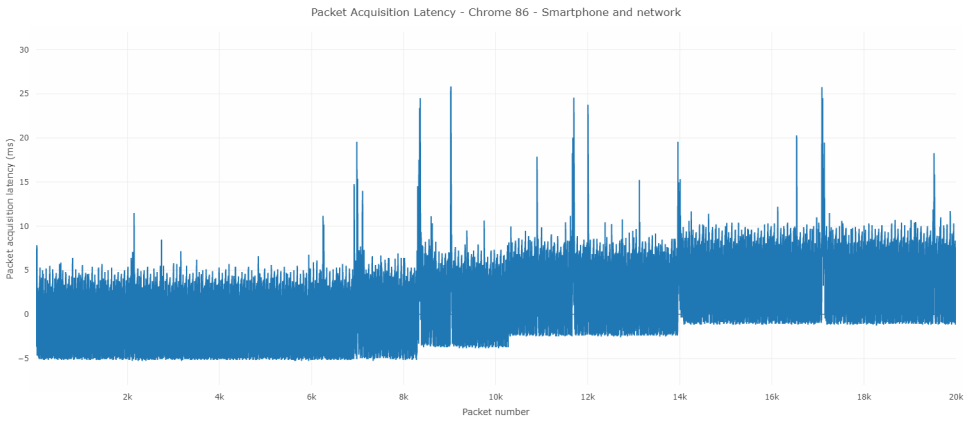

Figure 4.14. Packet Acquisition Latency - Chrome 86 - Smartphone and network

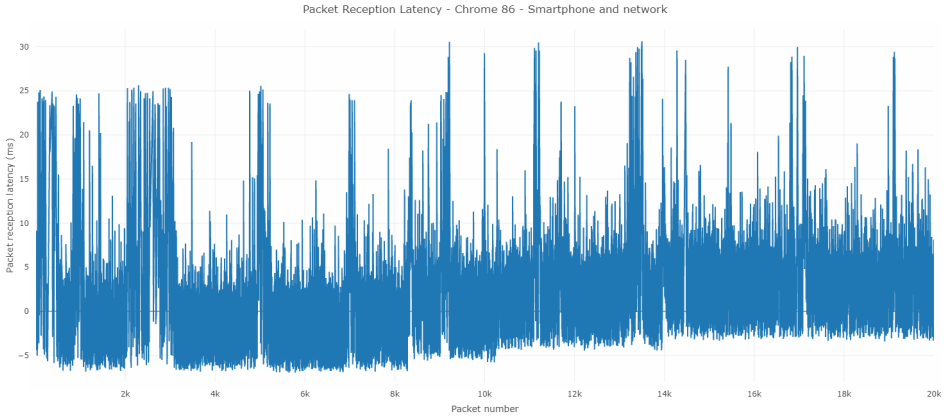

Figure 4.15. Packet Reception Latency - Chrome 86 - Smartphone and network

The last thing we analyze is the packetization of the DataChannel. As we can see in Figure 4.16, this time around the graph is less linear. In fact, we can see that in this scenario the DataChannel sent on average between one and three packets at a time, which means that at the receiver packets were delivered less regularly. Moreover we can see that there are peeks which are above the value of 8, value which we set for the playout buffer size, which means that also the buffering mechanism contributed to the percentage of discarded packets. Luckily the number of times that it went above the value of 8 (8 included) represents only the 0.4% of the overall communication, which means that the main responsible to the discarded packet percentage is network. This make sense since, all the elements involved in the network transmission may not be able to keep up with the frequency at which packets are generated, so they may be buffered in router

queues, and then delivered, which explains why we have a less regular graph.

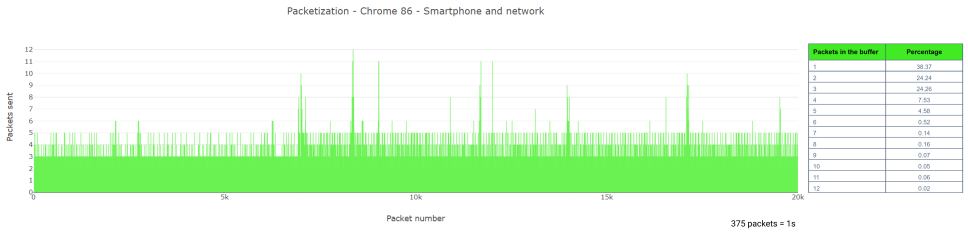

Figure 4.16. Packetization - Chrome 86 - Smartphone and network

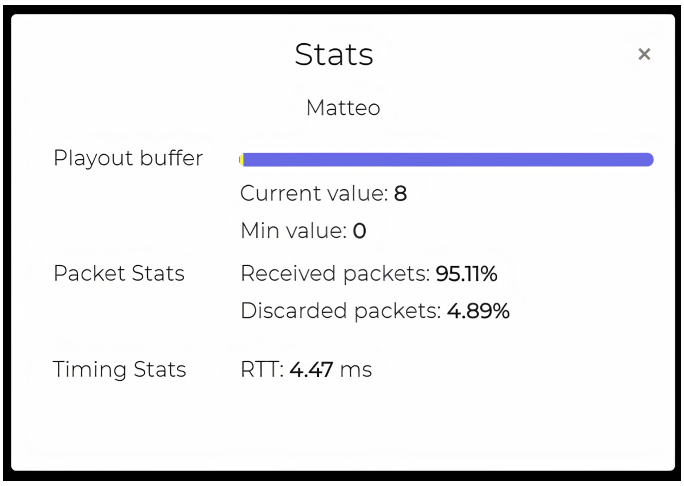

Figure 4.17. Statistics - Chrome 86 - Smartphone and network

One thing I want to address is that, as we can see from Figures 4.14 and 4.15, there are again those incremental delays I was experiencing in previous sections, but with smaller steps than previously. This incremental delay may be caused by the fact that the Android smartphone is less powerful than the PC, but the fact that those delay steps are also present on the PC side leads me to exclude that hypothesis. For reference, the graph shown in Figure 4.18 represents the packet acquisition latency measured on the PC. As we can see, we have an additional delay of around 5 ms introduced around the 9 000th packet.

I do not know exactly why those incremental delays are present, again, but from my experience there could be many reasons. Since the AudioWorklet element is rather new to the WebAudio API, it is still under development, which means that from one version of the browser to the next one things may change. During the course of

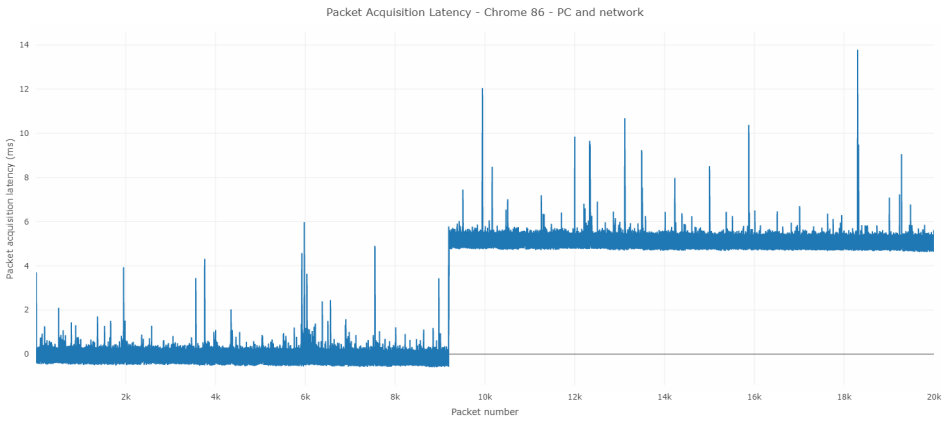

Figure 4.18. Packet Acquisition Latency - Chrome 86 - PC and network

this master thesis research it happened multiple times that from a certain version of the browser things related to the AudioWorklet changed behavior, for example before Chrome 82, if I remember correctly, when an AudioWorkletProcessor didn't set any value in the output buffer, it was automatically set to all zeros, which meant silence, while from Chrome 82 on, it stopped doing that, which means that if you forgot to set the output buffer to all zeros then you obtained an annoying noise as a consequence. So it may be possible that there is currently some sort of bug directly in Chrome itself. Another possible cause may be some application bug/misconfiguration which I'm not aware of. The last reasons which comes to my mind and which is most likely the main cause of this strange phenomenon we are observing is that the AudioWorklet threads are executed with normal priority instead of real-time priority, which means that it is possible that the thread is set to sleep for a certain amount of time during the execution of the web application, which means that when it is resumed it accumulated some delay, which is probably the delay we are seeing. All of those are just hypotesis, since I do not know which is the actual cause of this behavior. Further work needs to be done in order to assess the actual cause of this issue.

The last scenario we will examine is the one where there are more people connected to the same room than the amount the machine is able to handle. In order to be able to simulate this scenario we must remember that, since we implemented a full mesh P2P network topology, this means that for each new client we are connecting with, two additional logical threads are created, one for the DataSender AudioWorklet and one for the DataReceiver one. If we have all the clients on the same machine, then we need just to open multiple clients to exceed the amount of threads our machine is able to handle. In the case of the machine I used during

the tests, in order to reach the point where the number of threads created exceed the amount the machine is able to handle I just needed to open four clients on the same machine. Since all clients are on the same machine, network is not involved in the process.

All the graphs and stats reported below are relative to only one of the four clients involved, and considering the communication between two of the four clients, since things were pretty similar across clients.

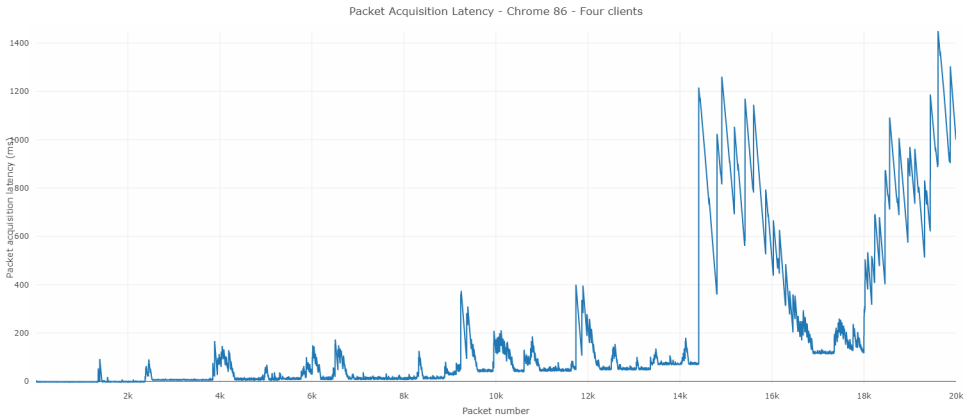

Figure 4.19. Packet Acquisition Latency - Chrome 86 - Four clients

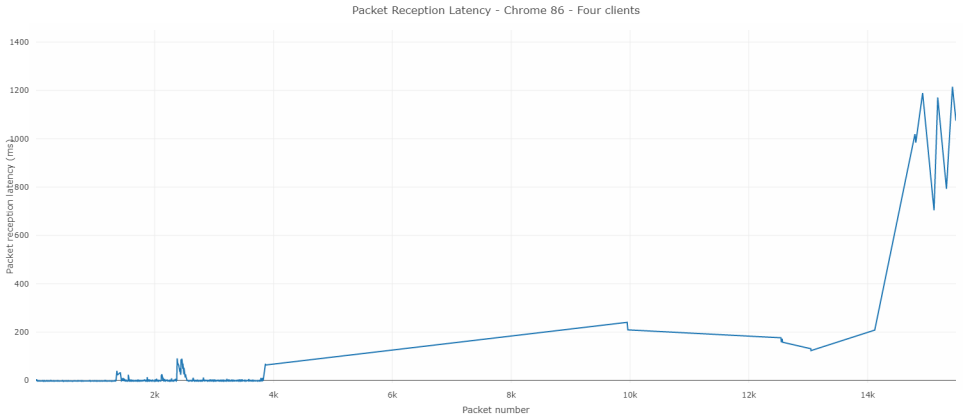

Figure 4.20. Packet Reception Latency - Chrome 86 - Four clients

As it is clearly visible from Figure 4.19, the machine was not able to keep up with all the threads created, so it had to switch between threads which lead to huge delays introduced already during the packet acquisition step, delays which reached really high values (up to 1.4 s). As a confirmation of the fact that the machine

was not able to keep up with all threads, is that delays are quite irregular.

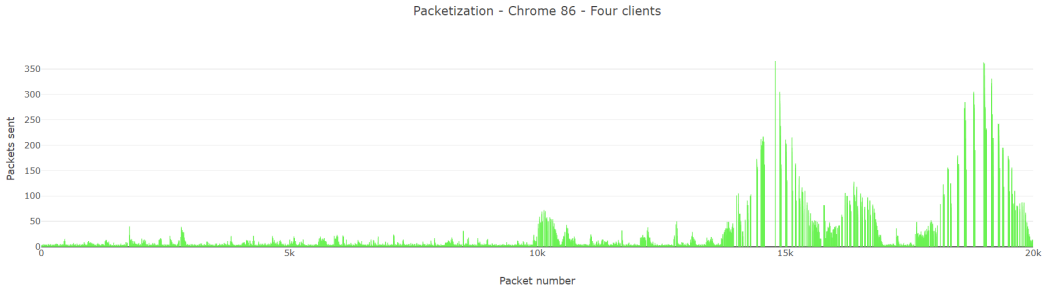

Figure 4.21. Packetization - Chrome 86 - Four clients

Moreover, we can see from Figure 4.20, that, again we have some high delays, but furthermore we can see that some packets are missing. This may be due to the fact that, during the transmission over the DataChannel some packets got lost probably due to low computing power, and so they never arrived to the receiving peer. Due to the lack of resources, the buffering mechanism of the DataChannel was heavily used, in fact, as we can see from Figure 4.21, we reached peeks of over 350 packets in the buffer, which means that those packets were delivered with almost 1 second of delay with respect to when they were generated. The last thing we will analyze are the statistics. In this case, again, we will consider only the statistics of the communication between these two clients.

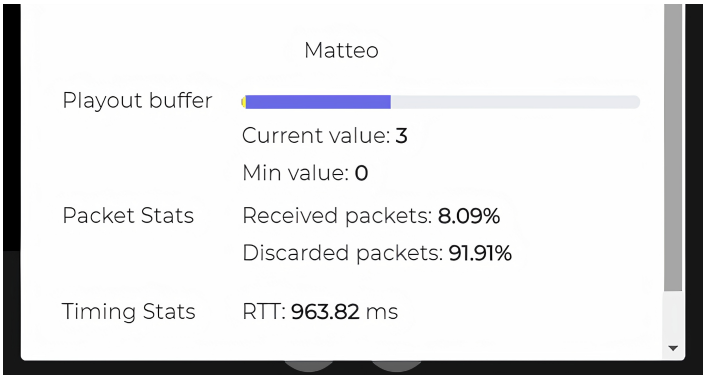

Figure 4.22. Statistics - Chrome 86 - Four clients

As we can see from Figure 4.22, communication was really bad, with more than 90% of the received packets discarded (keep in mind that the 90% refers only to received packets and does not consider packets which were never received. If we

consider also those packets probably this percentage would increase). Another interesting statistic is the round trip time, which reached really high values of up to 1s. We need to keep in mind that we are running the four clients locally on the same machine, which means that the standard round trip time should be below 1ms. Those really high values are a consequence of the way we evaluate the round trip time, since, as explained in chapter 3, the evaluation of the round trip time is based on receiving the audio packet and transmitting an ACK message to the source peer. Since during this test packets were heavily buffered, those packets were received by the peer with a high delay, which means that the value measured by the round trip time is equivalently big. The RTT, together with the statistics on the packets received and discarded, are the main indicators of issues during the communication (low computing resources, low network bandwidth, etc.).

While this test is not directly representing a real case scenario, since it is unlikely that we open more than one client on a single machine, it is important since it gives us an idea of the scalability of this application. In the current implementation the main bottleneck to scalability is the number of threads which are created, in fact each time a new client creates a connection with us, we are creating two new AudioWorklets, a DataSender and a DataReceiver, which means that we are also creating two additional logical threads. Given the fact that this application manages real time audio, if we exceed the physical number of threads our machine is able to handle, then we may experience some unexpected behavior. So the important concept to extract from this test is that there is a number of peer which the web application is able to handle which, depending on the machine which runs the we application, may be lower or higher, and if we exceed this number then we may experience some unexpected behavior (if we experience it or not depends on the ability of the browser/OS to optimize the execution of those threads). Scalability is one of the problems which this web application currently has and which I'm planning to address in a future release.

Chapter 5

Conclusions

This master thesis aimed to illustrate an alternative solution to peer-to-peer high quality low latency audio streaming, than WebRTC's MediaStreams, based on the exploitation of WebRTC's DataChannels and of Web Audio API's AudioWorklets to extract uncompressed audio, send it to the other peer and play it back, in order to create a web application focused on networked music performance. This approach gave us some real benefits, in terms of latency, over the traditional MediaStreams based solution, thanks to the fact that with this solution we are able to avoid unnecessary audio processing (like encoding, decoding, filtering, echo cancellation, etc.) and to have more control over the playback buffer, allowing us to reach a local end-to-end latency as low as 37 ms, which is 65 ms lower than what we were able to achieve with the solution based only on MediaStreams. While this is impressive, considering that we are within a web browser, it is not up to the point of what is able to achieve a native solution, like JackTrip. In order to allow musicians to have a real-time natural interaction, we would need to keep the end-to-end latency under 30 ms. While this requirement is difficult to satisfy, since we do not have any control over the delay introduced by the network, it would be a success to satisfy it at least locally, without involving network in the process. The main problem we currently have is that 30ms is the latency introduced by just accessing the input device and play back the audio it captures, without performing any processing on it and without involving any transmission among peers. So, if we want to reach a total processing latency of around 30 ms we need to reduce the latency introduced by the first and last step of the audio chain. The main issue we have in a web browser is the lack of a fine grained control over all the elements involved in the audio chain, which means that we have some limits over what we can do, limits that we do not have when developing a native solution. In this case we have no direct control over the latency introduced by the first and last step of

the audio chain, the only way we can influence that latency is by appropriately configuring the `getuserMedia()` function call and the `AudioContext` creation, which we already did, so at least for now I would not consider this as a valid alternative to tools like JackTrip for real-time networked music performance. But, this web application still has some value, since it can work as a preparation phase for tools like the one mentioned above, as it can be used to introduce more people to the networked music performance context, and it represents a valid alternative to standard video conferencing application for networked music performance, since this web application is focused towards full range audio, instead of voice only audio, and it allows the user to use either a mono or a stereo source, selection which is not available in standard videoconferencing applications. So, this may be a valid solution for any musician who wants to share ideas, any music teacher, or any other scenario which involves musicians, network and a low latency transmission. Moreover, thanks to the fact that web application are really easy to use and they do not require the user to install anything on their machine, it may attract more people than JackTrip, people which will learn all the cavets related to networked music performance and will probably then switch to something like JackTrip for performance reasons.

In conclusion, I really think that AudioWorklets and DataChannels are the way to go to achieve low latency audio streaming with a web browser, thanks to amount of control they give to developers and to the end users, but in order to obtain some more stable solutions we need to wait that AudioWorklets become more of a consolidate solution.

All the code developed during the course of this master thesis is available here [16].

5.1 Future work

Currently each peer-to-peer connection has associated to it two AudioWorklets: a DataSender and a DataReceiver. This means that each time we create a new peer-to-peer connection we create two new AudioWorklets, which leads to a scalability issue. An interesting future work would be to completely change the way AudioWorklets are created, in order to solve the scalability problem, by creating only one DataSender and one DataReceiver, which will be used by all peer-to-peer connections, in this way we decouple the number of AudioWorklets created from the number of peer-to-peer connection, which should allow more people to connect together and moreover it should reduce the resource consumption. In order to do so, it is necessary to implement, in the DataReceiver AudioWorklet, a mechanism to manage the audio samples of the different peers, which should also mix the

different audio samples together and which should manage the different queues by keeping track of the number of packets received by each peer, in order to manage the individual playout buffer size for each peer. Additionally it is necessary, in the DataSender AudioWorklet, to keep track of the generated packet number for each of the peer involved. A possible solution to this problem would be to move the assignment of the packet number outside of the DataSender AudioWorklet, in this way the AudioWorklet generates the same packet for each peer, will be then the data structure which manages the DataChannels in charge of assigning the right sequence number according to the peer the packet will be delivered to.

Another interesting and challenging future work would be to try integrate this application and JackTrip. Here we have two alternative solutions. The first solution would be to allow people using this application to communicate with people using JackTrip. This would require some changes in the data structure of the packet and would require to integrate, in this web application, the way JackTrip creates peer-to-peer connection. In this way we could have people using the web application connect with people using Jacktrip without needing to install anything, which would be nice for people without administrator privileges. The second solution, which is even more challenging, would be to develop a standalone application, using the framework Electron [44], and integrate JackTrip directly within the application. In this way we should have JackTrip for managing audio streaming and WebRTC for managing video streaming, having the benefits of both world, so the high performance of native applications together with the ease of use of web applications. This should also help with reducing the latency introduced by the first and last steps of the audio chain, since we are not managing audio through the web browser API's but we are doing it with JackTrip.

Regarding the statistics section, it would be interesting to add some additional basic statistics, like the percentage of packets which got lost, and would be even more interesting to add a statistic about the end-to-end mouth-to-ear latency. It could be a one time measure, performed at the beginning of the communication, or something which is updated every now and then. It's something challenging and which I haven't figured out how to implement, but it would be really useful for end users.

Lastly, regarding the testing process, it would be interesting to check the impact of different audio drivers on the total local latency. In this case we have to analyze the latency we have with different audio drivers and different configurations, looking for the one, on each OS, which gives the best performance.

Bibliography

[1] WebRTC. WebRTC Documentation and Examples, 2020. URL https://webrtc.org/. [Last accessed 10-10-2020].

[2] MDN. WebRTC MediaStream, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/MediaStream. [Last accessed 10-10-2020].

[3] MDN. WebRTC - RTCDataChannel, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/RTCDataChannel. [Last accessed 10-10-2020].

[4] Calvin Nguyen. WebRTC - The technology that powers Google Meet/Hangout, Facebook Messenger and Discord, 2019. URL https://medium.com/swlh/webrtc-the-technology-that-powers-google-meet-hangout-facebook-messenger-and-discord-cb926973d786. [Last accessed 15-10-2020].

[5] Philipp Hancke. webrtch4cks: how zoom's web client avoids using webrtc (datachannel update), 2019. URL https://webrtchacks.com/zoom-avoids-using-webrtc/. [Last accessed 15-10-2020].

[6] Knowledge Hub Media. WebRTC Integration with Skype, 2020. URL https://knowledgehubmedia.com/webrtc-integration-skype/. [Last accessed 15-10-2020].

[7] BigBlueButton Inc. BigBlueButton, 2020. URL https://docs.bigbluebutton.org/. [Last accessed 15-10-2020].

[8] Technology Marketing Corp. WebRTC World, 2020. URL https://www.webrtcworld.com/webrtc-list.aspx. [Last accessed 15-10-2020].

[9] Wikipedia. Networked music performance, 2020. URL https://en.wikipedia.org/wiki/Networked_music_performance. [Last accessed 10-10-2020].

[10] MDN. Web Audio API, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API. [Last accessed 13-10-2020].

[11] Wikipedia. Digital Audio Workstation, 2020. URL https://en.wikipedia.org/wiki/Digital_audio_workstation. [Last accessed 13-10-2020].

[12] MDN. Web Audio API - MediaStreamAudioSourceNode, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/MediaStreamAudioSourceNode. [Last accessed 13-10-2020].

[13] MDN. Web Audio API - MediaStreamAudioDestinationNode, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/MediaStreamAudioDestinationNode. [Last accessed 13-10-2020].

[14] Chris Chafe, Juan-Pablo Caceres. CCRMA - Stanford University. JackTrip, 2020. URL https://ccrma.stanford.edu/software/jacktrip/. [Last accessed 13-10-2020].

[15] JACK Audio Connection Kit. JACK Audio Connection Kit, 2020. URL https://jackaudio.org/. [Last accessed 13-10-2020].

[16] Matteo Sacchetto. JackTrip-WebRTC - GitHub Repository, 2020. URL https://github.com/jacktrip-webrtc/jacktrip-webrtc. [Last accessed 15-10-2020].

[17] MDN. Web Audio API - AudioWorkelt, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/AudioWorklet. [Last accessed 15-10-2020].

[18] MDN. Web Audio API - ScriptProcessorNode, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/ScriptProcessorNode. [Last accessed 15-10-2020].

[19] MDN. Web Audio API - AudioWorkeltNode, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkletNode. [Last accessed 15-10-2020].

[20] MDN. Web Audio API - AudioWorkeltProcessor, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/AudioWorkletProcessor. [Last accessed 15-10-2020].

[21] Internet Engineering Task Force (IETF). Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal, 2018. URL https://tools.ietf.org/html/rfc8445. [Last accessed 17-10-2020].

[22] OpenJS Foundation. Node.js, 2020. URL https://nodejs.org/en/. [Last accessed 16-10-2020].

[23] MDN. The websocket api (websockets), 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API. [Last accessed 16-10-2020].

[24] OpenJS Foundation. Express.js, 2020. URL https://expressjs.com/. [Last accessed 16-10-2020].

[25] Automattic. Socket.io, 2020. URL https://socket.io/. [Last accessed 16-10-2020].

[26] Internet Engineering Task Force (IETF). Session Traversal Utilities for NAT (STUN), 2008. URL https://tools.ietf.org/html/rfc5389. [Last accessed 17-10-2020].

[27] Wikipedia. Session Traversal Utilities for NAT (STUN), 2020. URL https://en.wikipedia.org/wiki/STUN. [Last accessed 17-10-2020].

[28] Kurento. NAT Types and NAT Traversal, 2018. URL https://doc-kurento.readthedocs.io/en/stable/knowledge/nat.html. [Last accessed 17-10-2020].

[29] Wikipedia. Network Address Translation - Methods of translation, 2020. URL https://en.wikipedia.org/wiki/Network_address_translation#Methods_of_translation. [Last accessed 17-10-2020].

[30] Internet Engineering Task Force (IETF). Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN), 2010. URL https://tools.ietf.org/html/rfc5766. [Last accessed 21-10-2020].

[31] The JQuery Foundation. JQuery, 2020. URL https://jquery.com/. [Last accessed 21-10-2020].

[32] The Bootstrap Team. Bootstrap, 2020. URL https://getbootstrap.com/. [Last accessed 21-10-2020].

[33] MDN. Channel Messaging API - MessageChannel, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/MessageChannel. [Last accessed 27-10-2020].

[34] MDN. Float32Array, 2020. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Float32Array. [Last accessed 28-10-2020].

[35] MDN. BigInt, 2020. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt. [Last accessed 29-10-2020].

[36] MDN. ArrayBuffer, 2020. URL https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/ArrayBuffer. [Last accessed 29-10-2020].

[37] MDN. Transferable, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/Transferable. [Last accessed 31-10-2020].

[38] MDN. MediaDevices.getUserMedia(), 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia. [Last accessed 31-10-2020].

[39] MDN. MediaTrackConstraints, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/MediaTrackConstraints. [Last accessed 31-10-2020].

[40] MDN. AudioContext, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/AudioContext/AudioContext. [Last accessed 04-11-2020].

[41] MDN. AudioContextLatencyCategory, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/AudioContextLatencyCategory. [Last accessed 04-11-2020].

[42] Jim. OBS, 2020. URL https://obsproject.com/. [Last accessed 04-11-2020].

[43] MDN. User Timing API, 2020. URL https://developer.mozilla.org/en-US/docs/Web/API/User_Timing_API. [Last accessed 07-11-2020].

[44] OpenJS Foundation. Electron, 2020. URL https://www.electronjs.org/. [Last accessed 17-11-2020].