# POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica (Computer Engineering)

A.a. 2020/2021

Sessione di Laurea Ottobre 2021


Politecnico di Torino

# Piattaforma WebRTC per Networked Music Performance Multi-Peer

**Relatore**

**prof. Antonio SERVETTI**

**Candidato**

**Paolo GASTALDI**

# Indice

# Elenco delle tabelle

# Elenco delle figure

# Elenco degli algoritmi

# Capitolo 1

# Introduzione

## 1.1 Trasmissione audio su internet

La larga diffusione dell'utilizzo di internet da parte di moltissime applicazioni ha coinvolto e continua a coinvolgere sempre di più, tra le altre cose, anche tante attività artistiche, ampliandone gli spunti creativi e superando tante difficoltà che si possono avere con un approccio che si limita a svolgersi in presenza. L'ambiente musicale è stato sicuramente uno dei primi a giovare di queste tecnologie, con la possibilità non solo di condividere del materiale in una forma definitiva, ma addirittura di interagire tra musicisti in diretta o in differita per crearne di nuovo. Considerando una bassa preparazione tecnico-informatica richiesta agli artisti, i programmatori e le piattaforme web si sono mossi per semplificare e rendere più accessibili le tecnologie esistenti. Per giungere a questo obiettivo i browser sono senz'altro tra le applicazioni di maggiore interesse, data la semplicità nel creare interfacce grafiche adatte a vari tipi di utilizzo e alla possibilità di eseguire codice senza il bisogno di installarlo.

## 1.2 Networked music performance

Con il termine *Networked music performance* indichiamo le attività musicali che coinvolgono la trasmissione su rete internet e che spesso che si svolgono in tempo reale. Soprattutto le situazioni realtime risultano essere molto più impegnative dal punto di vista delle latenze: si parla infatti della necessità di rimanere sotto i 30 ms affinché due musicisti riescano a coordinarsi per poter suonare insieme. Una latenza così bassa, considerando una velocità del suono di circa 340 m/s, corrisponde a due musicisti che distano circa 10 m tra loro nella stessa stanza. Superando questa

soglia si ha la percezione di un'eco che rende difficile continuare la performance per musistici non professionisti e non allenati a tecnologie del genere. Una latenza molto maggiore, di circa 150 ms, è considerata accettabile per videoconferenze e sistemi VoIP, dove l'obiettivo è l'intelligibilità del discorso parlato tra i partecipanti e spesso anche la sincronizzazione con una componente video [5]. Data la criticità del problema da affrontare, le attività di *Networked music performance* compongono una intera classe di analisi a sé stante. Oltre alla questione della latenza tra gli utenti si pongono anche domande sulla qualità della trasmissione. Per diminuire l'occupazione di banda di trasmissione, così da facilitare il problema e rendere l'applicazione realtime, si può agire sulla compressione del segnale audio. Benché esistano versioni di codificatori senza perdita di qualità (*lossless*) progettati per applicazioni realtime, così da aggiungere bassi ritardi di codifica, la maggior parte delle applicazioni utilizza codificatori che incidono sulla qualità del segnale audio (*lossy*). Uno tra i codificatore lossy più utilizzati soprattutto nell'ambiente dei browser è il codificatore *OPUS*[1], che permette una fine configurazione sulla frequenza di campionamento e della dimensione dei blocchi (*frame*) di campioni audio (*sample*). Agendo su questi parametri è possibile modellare il flusso di dati in trasmissione, utile per adattare l'applicazione a differenti contesti di utilizzo. Con l'aumentare della banda di rete disponibile, negli ultimi anni è stata anche considerata la trasmissione di dati in formato PCM (pulse code modulation) non compresso e senza codifica, ovvero una rappresentazione diretta del segnale audio senza un'infrastruttura per definire informazioni riguardo ai frame. Col formato PCM non si ha perdita di qualità, né ci sono ritardi aggiuntivi dovuti a un codificatore intermedio. Lo svantaggio è la conseguente maggiore dimensione del flusso dati. Una seconda possibilità per ridurre i byte complessivi da trasmettere è diminuire la profondità di bit di ogni campione. Per un'elaborazione dell'audio lo standard adottato dalla maggior parte di sistemi è di 32 bit per campione, mentre per la diffusione audio si riduce a 16 o 24 bit. È importante fare un'analisi di questi aspetti quando si combinano elaborazione audio, interattività e qualità del segnale per attività di *Networked music performance*.

## 1.3 Tecnologie esistenti

Sempre di più negli anni i principali browser hanno fornito interfacce e sistemi ai programmatori per la gestione audio in tempo reale. Tramite l'`AudioContext` è possibile creare thread ad alta priorità (chiamati *AudioNodeProcessor* per le API audio oppure *WebWorker* nel caso più generale) per acquisire, elaborare e riprodurre segnali audio. Combinandoli con le API per la trasmissione di rete

[1] `https://opus-codec.org/`

*WebRTC* [2] che gestiscono il protocollo *RTP* (real-time protocol), un protocollo disegnato appositamente per contesti realtime basato *UDP*, è possibile creare delle applicazioni peer-to-peer per la condivisione audio e video. Con una riduzione dei ritardi, tramite un'accurata regolazione dei parametri, si riescono a realizzare delle applicazioni specifiche per la condivisione di audio non compresso in tempo reale. Benché i browser non siano ancora in grado di raggiungere le latenze considerate allo stato dell'arte, la portabilità e la facilità di utilizzo di queste applicazioni web permettono un più vasto accesso ai musicisti rispetto a molte applicazioni native esistenti.

## 1.4 Metriche

Il principale ostacolo nell'implementare un'applicazione realtime non è il ritardo, come potrebbe sembrare da un'iniziale analisi al problema, ma che può essere ridotto con un'adeguata architettura e l'evoluzione dell'hardware e del software, bensì la sua variazione: il jitter. Il jtter (1.3) è la differenza tra il ritardo reale (1.1) e il ritardo atteso (1.2).

$$ritardo\ reale(i) = T(i) = t(i) - t_0 \tag{1.1}$$

$$ritardo\ atteso(i) = T'(i) = t_0 + \frac{i * \texttt{window size}}{\texttt{sample rate}} \tag{1.2}$$

$$jitter(i) = T(i) - T'(i) \tag{1.3}$$

Le fluttuazioni del ritardo nel tempo hanno molte cause: dispositivi intermedi che raggruppano più pacchetti, cambio di tragitto nella rete, congestioni etc. Una soluzione in grado di risolvere jitter di piccole dimensioni è l'utilizzo di un buffer intermedio (procedura indicata informalmente come "bufferizzazione"). Alla ricezione di un frame il sistema si occupa di inserirlo in un buffer prima di poterlo usare. In questo caso anche i pacchetti fuori ordine possono essere accettati, se non hanno un ritardo eccessivo tale per cui quel pacchetto sarebbe già dovuto essere processato; in quella situazione si procede scartando il pacchetto. Inoltre, un buffer intermedio permette di partizionare ulteriormente il sistema tra ricezione e processamento. Questa soluzione inserisce anche dei ritardi aggiuntivi come indicato in (1.5) e per questo è necessaria un'attenta configurazione per non aggiungere troppo ritardo nel sistema realtime. Tanto più grande è il buffer, maggiore è il jitter che l'applicazione è in grado di sopportare, ma maggiore è anche il ritardo prima che un pacchetto venga processato.

$$\textit{offset pacchetto} = [(\texttt{write index} - \texttt{read index}) + \texttt{queue size}] \mod \texttt{queue size} \quad (1.4)$$

$$\textit{ritardo del buffer(i)} = \texttt{packet offset} * \frac{\texttt{window size}}{\texttt{sample rate}} \quad (1.5)$$

Siccome il buffer non può avere dimensione infinita, consideriamo un buffer circolare. Nel sistema utilizzeremo due indici: un `write index` per l'operazione di accodamento (*enqueue*) e un `read index` per l'estrazione dalla coda (*dequeue*). Scegliere la posizione nel buffer dove posizionare il frame è un problema difficile, che può interessare anche statistiche effettuate in realtime. In questo sistema il primo pacchetto ricevuto da un client viene inserito alla posizione indicata in (1.6), così che quello sia il primo valore per il `write index`; la posizione nel buffer dei frame successivi sarà progressiva a questo valore basandosi sul numero di pacchetto.

$$\textit{indice iniziale} = (\texttt{read index} + \texttt{packet offset}) \mod \texttt{queue size} \quad (1.6)$$

Questa soluzione sottintende l'ipotesi che il primo pacchetto abbia jitter pari a zero, caso molto distante dalla realtà. Per esempio, il browser Google Chrome passa alle applicazioni Javascript due frame alla volta, quindi il primo frame arriva un ritardo di acquisizione di almeno un tempo calcolabile con (1.7). Ciò comporta che non possiamo usare completamente il buffer circolare, altrimenti il `write index` potrebbe andare a sovrapporsi o superare il `read index`; per il sistema sarebbe come ottenere frame dal futuro. Per risolvere questo problema la soluzione adottata è stata di dimensionare il buffer circolare pari al doppio della distanza `packet offset` tra gli indici (1.8) (per maggiori informazioni su questa soluzione, vedere [3]). In questo modo anche un jitter iniziale sui primi pacchetti viene sopportato dall'applicazione. Questo comporta un possibile ritardo maggiore in caso di jitter sul primo pacchetto, anche perché l'applicazione non prevede dei sistemi di recupero degli indici di scrittura e di lettura. Modificare le posizioni degli indici di scrittura `write index` comporterebbe un'analisi dei pacchetti persi, una scelta dei pacchetti da scartare nel caso di avvicinamento con il `read index`, con la conseguente generazione di discontinuità nel segnale audio che vanno gestite con tecniche di stretching o cross-fade. Data la complessità della soluzione e la non completa attinenza con gli obiettivi del progetto, questa opzione è stata solamente considerata ma non implementata.

$$\textit{ritardo acquisizione frame} = \frac{\textit{frame size}}{\textit{sample rate}} \tag{1.7}$$

$$\textit{dimensione del buffer} = 2 * \texttt{packet offset} \tag{1.8}$$

Una rapida panoramica di buffer circolari per differenti applicazioni, nonché esempi di implementazioni in pseudocodice possono essere trovate in [3].

# Capitolo 2

# Lavori correlati

## 2.1 Jacktrip e Jacktrip-WebRTC

Jacktrip[1] è un software in C++ sviluppato per la trasmissione audio in alta qualità, senza compressione, a bassa latenza che opera con i moduli JACK[2]. Nella propria tesi magistrale *JackTrip-WebRTC Networked music performance with web technologies* [6] Matteo Sacchetto ha ricreato questo software con delle innovative tecnologie del browser. L'obiettivo era rendere maggiormente fruibile un software come Jacktrip, molto complesso da configurare, da installare e da controllare da linea di comando per un utente non specializzato, e di riportarne le funzionalità nel browser dove è molto più semplice creare una GUI adeguata, oltre a non richiedere installazioni. Nel suo elaborato Sacchetto ha sperimentato le migliori prestazioni del `RTCDataChannel` rispetto a una soluzione basata solamente su `MediaStream`. L'architettura utilizzata da Sacchetto [fig. 2.1] si basa sulla creazione di due thread audio ad alta priorità, chiamati propriamente `AudioWorklet` o `WebWorker`, e di un buffer circolare per ogni nuovo peer connesso. La comunicazione tra il thread principale e i thread audio avviene tramite i `MessageChannel`, un sistema di comunicazione basato su pipe. Il progetto comprende anche un server *node.js* che gestisce la creazione di stanze e un sistema di segnalazione tra i peer che vogliono connettersi basato su protocollo *WebSocket*.

[1] https://ccrma.stanford.edu/software/jacktrip/

[2] https://jackaudio.org/

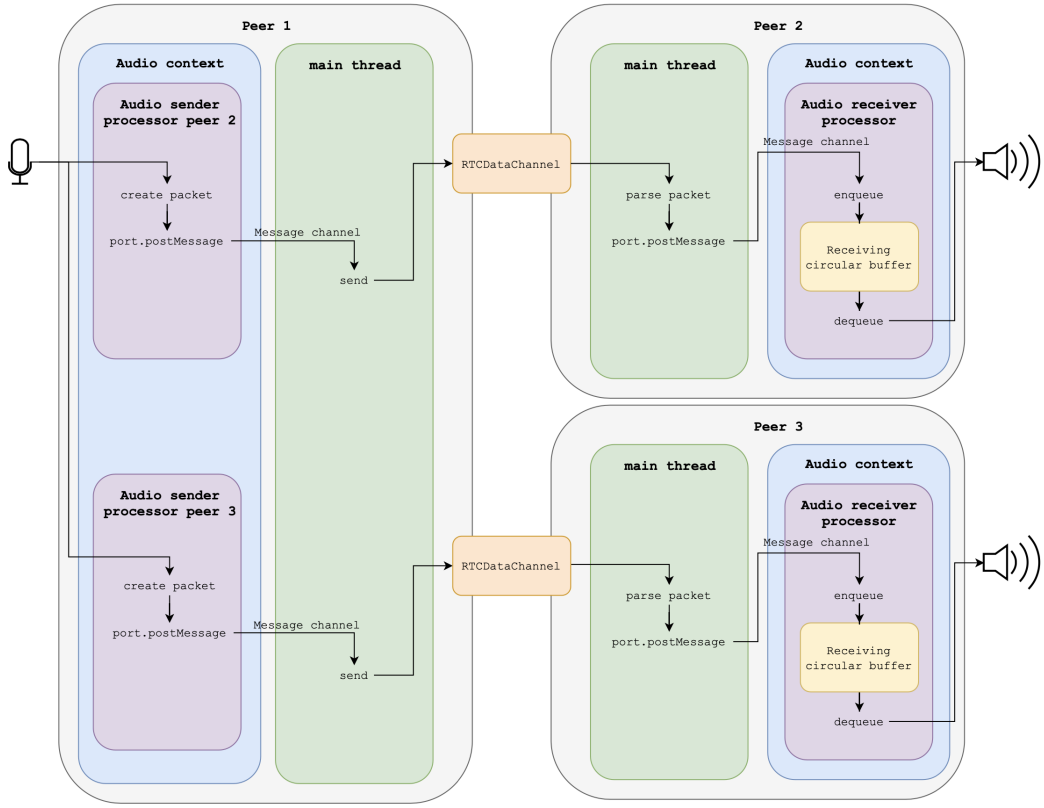


Figura 2.1. Architettura di Jacktrip-WebRTC

## 2.2 Prodotti hardware

Sono disponibili sul mercato specifiche periferiche hardware che permettono l'acquisizione e la trasmissione a bassa latenza. Un esempio è il modulo esterno creato da ELK[3] e che si connette a una scheda Raspberry Pi. Oltre alla trasmissione audio permette inoltre di controllare il guadagno in ingresso e il volume dell'audio tramite dei selettori e slider fisici sulla scheda.

[3] `https://elk.audio/`

## 2.3 Prodotti software

Sonobus[4] è una soluzione software gratuita e open-source per lo streaming audio ad alta qualità. Funziona sui principali sistemi operativi e si basa su una comunicazione peer-to-peer. Fornisce la possibilità di trasmettere l'audio in formato PCM e con l'utilizzo di codificatori OPUS.

Tra i sistemi considerati per l'analisi e il confronto delle performance sono presenti anche dei classici servizi di videoconferenza che funzionano su tecnologia *WebRTC*, quindi con possibilità di avere connessioni peer-to-peer, come *Google Meet*[5] e *Jitsi*[6]. Come spiegato anche in [6], molti software peer-to-peer non riescono a superare specifiche tipologie di NAT. Per questo è necessario disporre dei server TURN (Traversal Using Relay NAT) intermedi tra i peer con lo scopo di fare il forward dei pacchetti. In questo modo si passa da un'architettura peer-to-peer a una client server. Mentre le applicazioni che utilizzano tecnologie *WebRTC* sono in funzione, è possibile analizzare dal browser lo stato della connessione e rilevare l'utilizzo o meno di questi server TURN. Su Firefox si accede alla pagina con i log e le informazioni sulla connessione *WebRTC* scrivendo `about:webrtc` nella barra di ricerca, mentre su Google Chrome il comando è `chrome://webrtc-internals`.

[4] `https://sonobus.net/`

[5] `https://meet.google.com/`

[6] `https://meet.jit.si/`

# Capitolo 3

# Architetture e tecnologie

Questa tesi[1] prosegue il lavoro iniziato da Matteo Sacchetto in [6] con l'obiettivo di dare maggiore enfasi alla scalabilità e all'efficienza dell'applicazione piuttosto che alla velocità di comunicazione ristretta al caso di due client. Sono stati pertanto mantenuti l'architettura peer-to-peer, il sistema di segnalazione con protocollo *WebSocket* e il server *node.js*, fatto salvo alcune modifiche agli header HTTP necessarie ad abilitare specifiche funzionalità del browser che per ragioni di sicurezza sono normalmente disabilitate. È stato conservato anche il concetto di *stanza*: tramite lo stesso url più peer possono connettersi allo stesso spazio virtuale e condividere l'audio solo tra loro. Su questa idea si basa l'organizzazione e la gestione delle connessioni peer-to-peer all'interno del codice sia del client che del server.

Le nuove architetture esposte in questo capitolo mirano a testare e comparare differenti tecnologie implementate dai browser più moderni. Dato l'obiettivo iniziale di scalabilità del software, in tutte le seguenti architetture è stato cercato di ridurre il numero di thread audio ad alta priorità del sistema. Nello specifico non c'è più una riplica dei thread di trasmissione e di ricezione per ogni peer connesso, bensì solamente una coppia di questi per tutto il sistema. Questa è la prima differenza implementativa valida per tutte le architetture di seguito presentate rispetto all'architettura presentata nella sezione 2.1.

## 3.1 MessageChannel

Il `MessageChannel` è un sistema di comunicazione basato su pipe utilizzato tra *WebWorker*, ovvero tra più thread. Tramite i metodi `.port.postmessage()` e

[1] `https://github.com/paologastaldi-polito/master-thesis`

l'event handler `.port.onmessage` è possibile inviare e ricevere dati. Come nella tesi di Sacchetto [6], l'implementazione della prima nuova architettura proposta utilizza un `MessageChannel` sia per l'invio che per la ricezione [fig. 3.1]. In questo caso, le operazioni di inserimento in coda (`enqueue`) e di estrazione dalla coda (`dequeue`) sono entrambe svolte dal `ReceiverProcessor`. L'inserimento in coda avviene tramite la funzione che gestisce l'evento di arrivo di un nuovo messaggio al thread `.port.onmessage`; per questo l'evento non dev'essere sincrono.

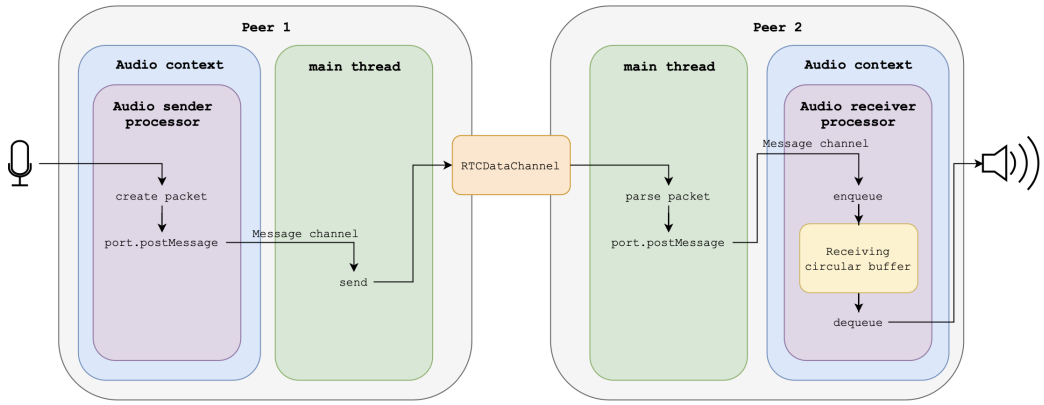


Figura 3.1. Architettura con `MessageChannel`

## 3.2 SharedArrayBuffer

Lo `SharedArrayBuffer` è una struttura che permette di creare una zona di memoria in formato binario (*raw*) di dimensione fissa condivisibile tra più *WebWorker*. L'operazione di creazione è simile a una `malloc` in C (oppure a una `calloc` dato che solitamente tutti i valori vengono settati a zero, benché non sia una specifica dello standard). Per utilizzare quest'area di memoria è necessaria una struttura che ne faccia da wrapper e definisca il formato dei dati. Una classe derivata dalla `ArrayBuffer` (es. `Int16Array`, `Float32Array`...) può essere utilizzata per effettuare un accesso diretto ai dati secondo il formato specifico della classe.

```
let n_items = 10;
let buffer = new SharedArrayBuffer(n_items * Float32Array.BYTES_PER_ELEMENT);
let shared_array = new Float32Array(buffer);
// adesso tramite 'shared_array' e' possibile effettuare l'accesso diretto alla
    ↪ memoria associata a 'buffer' in formato Float32
let shared_array[0] = 10.50;
```

Con gli `SharedArrayBuffer` è possibile modificare l'architettura precedente per rimuovere il `MessageChannel` in ricezione, come indicato in [fig. 3.2]. Viene aggiunto un buffer circolare intermedio così che ogni peer possa scrivere quando riceve dei dati dal `RTCDataChannel` e un singolo thread dell'`AudioContext` può leggere e azzerare il contenuto del buffer (con una specifica operazione atomica `Atomics.exchange`).

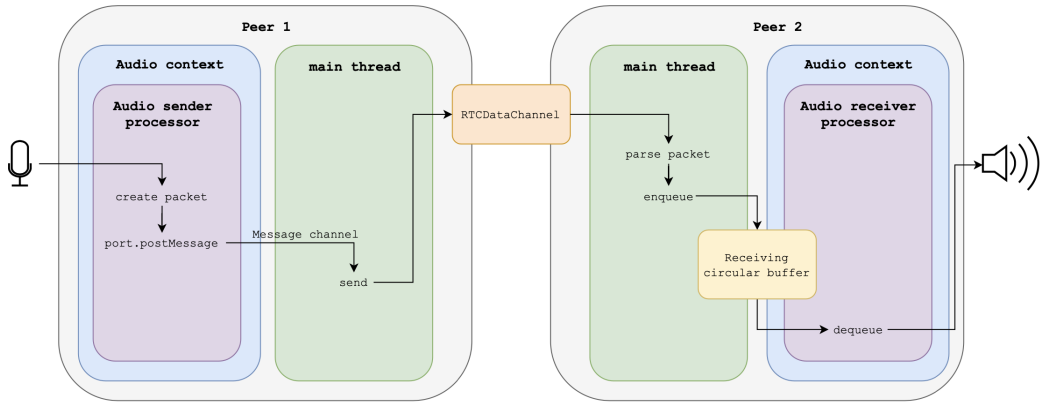


Figura 3.2. Architettura con `SharedArrayBuffer`

Lavorando con più thread e strutture dati condivise, la sincronizzazione tra tutte le parti diventa molto importante. Tramite i metodi `Atomics` possiamo operare in maniera sicura su un `SharedArrayBuffer` oppure su un `ArrayBuffer`. Queste operazioni hanno il limite di bloccare (tramite un'operazione interna di lock - unlock) tutta la struttura dati, ma di poter operare su un solo elemento della struttura alla volta. Per questo è necessario ripetere l'operazione per ogni cella della struttura dati.

Gli `SharedArrayBuffer` non sono abilitati automaticamente nei browser per ragioni di sicurezza. Il loro utilizzo è limitato a siti web che forniscono una connessione HTTPS e che, tramite specifici header, richiedono di poter usare queste tecnologie. Il server fornito da Sacchetto in [6] già includeva la creazione automatica di certificati temporanei SSL autofirmati e l'apertura di una connessione via HTTPS. I certificati autofirmati vengono rilevati dai browser, che a seguito di una informativa sui rischi permettono comunque di utilizzare il sito con tutte le funzionalità necessarie.

```
// header necessari per abilitare l'utilizzo degli SharedArrayBuffer:
Cross-Origin-Embedder-Policy: require-corp
Cross-Origin-Opener-Policy: same-origin
```

## 3.3 Atomics.waitAsync

I metodi `Atomics` sono molto importanti, ma hanno alcune limitazioni di utilizzo. La più notevole è che `Atomics.wait` non può funzionare nel thread principale. Al momento solamente Google Chrome ha implementato una versione speciale di questo metodo chiamato `Atomics.waitAsync`. Una volta richiamato, viene restituita una `Promise` che viene risolta alla chiamata di una `Atomics.notify`. Con questa tecnica è possibile rimuovere dall'architettura il `MessageChannel` anche per la parte di trasmissione [fig. 3.3]. Dato che `Atomics.notify` è solamente una funzione di notifica di un evento, per il trasferimento dei dati è necessario un altro buffer circolare intermedio. Inoltre, l'operazione di notifica tra thread non è garantito vada a buon fine; per questo la dequeue nel thread principale è in grado di leggere più posizioni dal buffer circolare, tante quante ne sono state accodate (questa operazione non è stata esplicitata nello schema per non complicarlo ulteriormente).

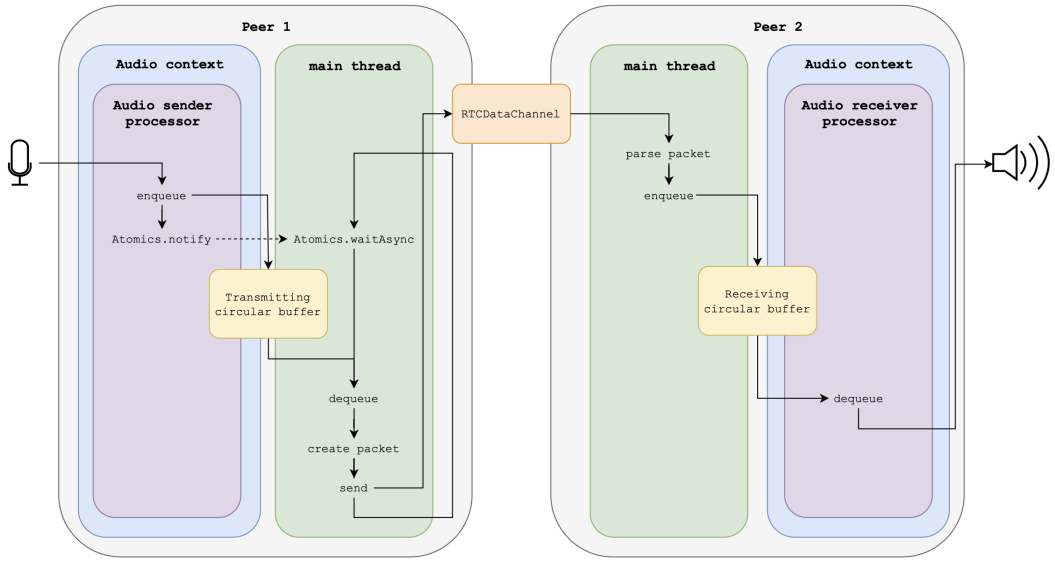


Figura 3.3. Architettura con `Atomics.waitAsync`

## 3.4 Formato dei campioni audio

I campioni audio, indicati qua col termine inglese *sample*, vengono acquisiti e riprodotti in formato `Float32`. Questa profondità di bit è utile per l'elaborazione audio, ma diventa un notevole carico aggiuntivo per la trasmissione di rete. Per questa ragione è stata mantenuta l'idea di Sacchetto [6] di ridurre i campioni a 16 bit, così come avviene per la trasmissione di segnali audio in altri contesti

web. Durante la creazione del pacchetto viene fatta una trasformazione lineare da `Float32` a `Int16`. Ci sono alcuni vantaggi pratici nell'utilizzo dei formati interi, tra cui il pieno supporto della funzioni `Atomics`, che invece non sono disponibili su tutti i formati `Float*`.

## 3.5 Buffer circolare e missaggio

Partendo nuovamente dal progetto di Sacchetto [6], è stato reimplementato il buffer circolare in modo da avere una sola istanza del buffer per ogni istanza dell'applicazione. Da qui la sostanziale differenza di non utilizzare il buffer per memorizzare i pacchetti così come arrivano dai peer, ma di unire le funzioni di inserimento in coda e missaggio dei frame in arrivo da tutti i peer, in modo da memorizzare dei frame già pronti per l'estrazione dalla coda. Dopo aver valutato differenti tecniche di missaggio [1], quella scelta per l'implementazione è stata la più semplice: vengono sommati i sample per ogni canale di ogni peer e divisi per il numero di peer, come indicato in 3.1. Con questa tecnica si è sicuri di non sforare mai l'ambito $[-1, 1]$ della gamma dinamica ammessa per ciascun sample.

$$\begin{aligned} output[c][s] &= \frac{\sum_{p=1}^{P} input_p[c][s]}{N}, c \in [1, C], s \in [1, S] \\ &dove \\ &\quad P : numero\ di\ peer, \\ &\quad C : numero\ di\ canali, \\ &\quad S : numero\ di\ sample\ in\ un\ frame \end{aligned} \tag{3.1}$$

Per ridurre al minimo il numero di divisioni totali, la divisione per il numero di peer viene svolta solamente durante l'estrazione dalla coda sulla somma di tutti i campioni. Per evitare overflow, le celle del buffer circolare sono state dimensionate in formato `Int32`. Questo comporta che, avendo sample da `Int16`, il buffer circolare è in grado di gestire teoricamente fino a $2^{16}$ peer. Siccome nella pratica la probabilità che tutti i picchi coincidano è molto bassa, il numero di peer supportati potrebbe essere anche essere maggiore. Come indicato nella sezione 3.4, il formato `Int32` è stato anche necessario perché supportato dalle funzioni `Atomics`, a differenza del formato `Float32`. Durante l'estrazione dalla coda i sample vengono anche convertiti dal formato `Int16` al formato `Float32`, a meno di specifici flag.

Con l'implementazione dell'architettura *Atomics.waitAsync*, esposta in 3.3, è stato necesario l'utilizzo di un buffer circolare anche per la trasmissione. Per riusabilità

del codice sono state estese le funzionalità del codice della classe `AudioCircularBuffer` con la possibilità, tramite appositi flag, di inserire ed estrarre dati sia in formato `Int16` che in formato `Float32`, sia con header di pacchetto che senza. È inoltre possibile generare degli effetti di dissolvenza in entrata e in uscita molto brevi su un frame, per evitare improvvise discontinuità nell'ampiezza del segnale audio quando un peer si connette o disconnette. Ne risultano delle interfacce più complesse a livello di parametri, ma maggiormente flessibili nel loro utilizzo. Tramite parametri aggiuntivi accodati al fondo di quelli esplicitati è possibile mandare dei comandi alle istanze interne `AudioCircularBufferSingle`.

```
/**
 * add data to queue
 * @param {Int16Array|Float32Array} packet
 * @param {String} isLast
 * @param {Boolean} isInputInt16
 * @param {Boolean} hasPacketHeader
 * @param {...any} args
 * @returns {Number}
 * 0 if okay, -1 if packet discarded
 */
enqueue(packet, envelop=null, isInputInt16=true, hasPacketHeader=true, ...args)
```

```
/**
 * extract an element from the queue
 * @param {Int16Array|Float32Array} output
 * output vector, can have many channels
 * @param {Boolean} isOutputInt16
 * @param {Boolean} hasPacketHeader
 * @param {...any} args
 */
dequeue(output, isOutputInt16=false, hasPacketHeader=false, ...args)
```

Sono state implementate interfacce aggiuntive, necessarie per conoscere lo stato interno della coda e per permettere lo scorrimento degli indici senza estrapolare i dati, operazione necessaria in caso si vogliano silenziare gli speaker.

```
/**
 * erase the next position in the buffer and increment the dequeue index
 */
eraseNext()
```

**Algoritmo 1:** Inserimento di un pacchetto in coda (`enqueue`)

```
data ← contenuto del pacchetto allineato a Int16;
n ← numero pacchetto;
r ← numero dell'ultimo pacchetto letto dal buffer;
offset ← distanza nel buffer tra indice di scrittura e di lettura;
dim ← dimensione del buffer circolare;
header ← dimensione dell'header del pacchetto;
C ← numero di canali;
S ← numero di sample per frame;

if primo pacchetto da questo peer then
    shift ← −n + r + 1 + offset;                    ▷ indice di scalamento
n ← n + shift;
if n <= r then
    end;        ▷ si sta provando a inserire in coda un pacchetto troppo tardi
else
    i ← n mod dim;
    b ← buffer[i];
    c ← 0;                                          ▷ indice del canale corrente
    k ← header;                                     ▷ indice lettura pacchetto
    while c < C do
        s ← 0;                                      ▷ indice del sample corrente
        while s < S do
            b[c][s] ← b[c][s] + data[k];            ▷ missaggio dei sample ricevuti
            s ← s + 1;
            k ← k + 1;
        c ← c + 1;
end;
```

**Algoritmo 2:** Estrazione di un frame dalla coda (`dequeue`)

```
P ← numero di peer connessi;
f ← frame in output, già allocato dal codice nativo;
r ← numero dell'ultimo pacchetto letto dal buffer;
dim ← dimensione del buffer circolare;
C ← numero di canali;
S ← numero di sample per frame;

r ← r + 1;
i ← r mod dim;
b ← buffer[i];
c ← 0;                                          ▷ indice del canale corrente
while c < C do
    s ← 0;                                      ▷ indice del sample corrente
    while s < S do
        x ← b[c][s];
        b[c][s] ← 0;                                          ▷ reset della cella
        x ← x/P;                          ▷ normalizzazione sul numero di peer
        f[c][s] ← trasformaInt16InFloat32(x);
        s ← s + 1;
    c ← c + 1;
end;
```

```
/**
 * check if data can be extracted from queue
 * @returns {Number} hasData
 * index distances if has data, 0 otherwise (no negative values)
 */
hasData()
```

Per la divisione dei sample per il numero di peer indicata in 3.1 viene tenuto un contatore del numero di peer in una memoria condivisa accessibile a tutte le istanze della classe `AudioCircularBuffer`. Si precisa che le strutture dati condivise che realizzano il buffer fisicamente, ovvero degli `SharedArrayBuffer`, sono uniche in tutta l'applicazione, ma è possibile accedervi tramite diverse istanze della classe `AudioCircularBuffer`. Quando instanziata, questa classe registra un nuovo peer nel contatore, a meno di un esplicito flag (per esempio, il `AudioReceiverProcessor` deve accedere alla coda, ma non conta come un peer aggiuntivo). Alla disconnessione del peer è necessario rimuovere il valore dal contatore, così come deallocare altre strutture dati. Per questo è stata creata un'interfaccia che permette la corretta distruzione del buffer.

```
/**
 * free resources (if allocated)
 */
destroy()
```

È importante considerare quante copie degli stessi dati avvengono nel sistema. Con l'architettura implementata da Sacchetto [6] c'è una copia dei dati alla creazione del pacchetto e una alla sua parsificazione. Con le nuove architetture presentate si ha una copia dei dati nella crezione del pacchetto per la fase di invio e due copie in fase di ricezione: una per l'inserimento in coda di ogni pacchetto di ogni peer e una per l'estrazione dalla coda. Per evitare ulteriori copie intermedie, l'inserimento in coda viene fatto con sample `Int16` che vengono automaticamente convertiti in `Int32`, mentre durante l'estrazione dalla coda i sample vengono scalati e convertiti in formato `Float32`. Nell'estrazione il frame viene caricato direttamente nella struttura dati fornita dall'`AudioWorkletProcessor`, senza allocarne una ulteriore.

## 3.6 Struttura, creazione e parsificazione dei pacchetti

La struttura presentata dal progetto di Sacchetto [6] definiva un pacchetto con la struttura indicata in [fig. 3.4]. Ogni pacchetto contiene un frame da 128 sample che può essere composto da uno o più canali. L'header è composto da una cella

`BigUint64` per il numero di pacchetto e una cella `UInt8` per il numero di canali, per un totale di $8+1 = 9$ byte. Siccome i sample trasmessi sono in formato `Int16`, l'idea è quella di allineare tutto il pacchetto a 16 bit, così da evitare la parsificazione del pacchetto. Parsificare tutto il pacchetto comporterebbe una copia intermedia in più dei dati: una prima copia dal pacchetto a un'area di memoria temporanea e una seconda copia dall'area di memoria temporanea al buffer circolare. Con l'allineamento a 16 bit viene data la possibilità di leggere con accesso diretto e inserire in coda i sample copiando direttamente i frame dal pacchetto al buffer circolare. L'allineamento è stato raggiunto aumentando la dimensione della cella del numero di canali, portandola a una cella `Int16`, per un totale di $8+2 = 10$ byte. Il pacchetto appena ricevuto è in formato *raw* e quindi dev'essere inserito in una struttura wrapper tra quelle ammesse dagli `ArrayBuffer` per operare con l'accesso diretto. In questo modo, il pacchetto può essere inserito in una struttura di tipo `Int16Array` e, saltando le prime 5 celle (10 byte corrispondono a 5 celle `Int16`), è possibile leggere direttamente i sample dal pacchetto ricevuto. Questo overhead di 1 byte per ogni pacchetto comporta, in caso di una frequenza di campionamento di 48000 Hz, 1 canale e frame di 128 sample, un invio di 375 byte in più al secondo (vedi 3.2). Considerato il ridotto carico aggiuntivo imposto alla trasmissione di rete, è stata implementata questa soluzione.

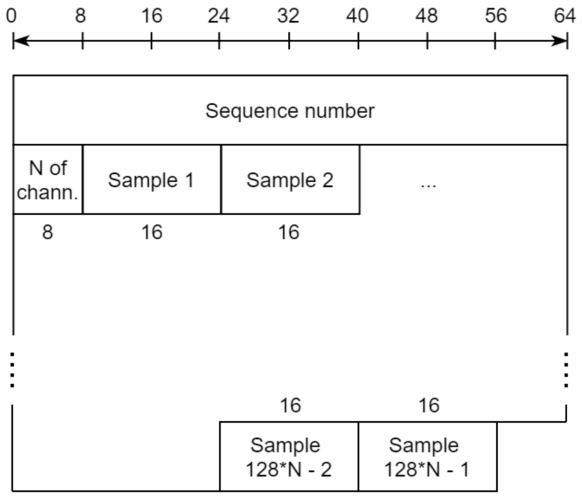


Figura 3.4. Pacchetto del progetto di Sacchetto

**Algoritmo 3:** Creazione di un pacchetto (`create`)

```
n ← numero dell'ultimo pacchetto creato;
f ← frame da inserire nel pacchetto;
C ← numero di canali;
S ← numero di sample per frame;

header ← 8 + 2;                          ▷ dimensione di un BigUInt64 e un UInt16
dim ← header + 2 * C * S;                ▷ ogni sample è in formato Int16
k ← 0;                                   ▷ indice scrittura pacchetto
p ← allocaMemoria(dim);
n ← n + 1;
scriviBigUInt64(p, k, n);                ▷ scrivi in p con offset di k il valore di n
k ← k + 8;                               ▷ aggiunge 8 byte
scriviUInt16(p, k, C);
k ← k + 2;                               ▷ aggiunge 2 byte
c ← 0;                                   ▷ indice del canale corrente
while c < C do
    s ← 0;                               ▷ indice del sample corrente
    while s < S do
        x ← trasformaFloat32InInt16(f[c][s]);
        scriviInt16(p, k, x);
        s ← s + 1;
        k ← k + 2;                       ▷ aggiunge 2 byte
    c ← c + 1;
return p;
```

**Algoritmo 4:** Parsificazione di un pacchetto (`parse`)

```
p ← pacchetto;

k ← 0;                                   ▷ indice lettura pacchetto
n ← leggiBigUInt64(p, k);                ▷ numero di pacchetto, leggi da p con offset di k
k ← k + 8;
C ← leggiUInt16(p, k);                   ▷ numero di canali
k ← k + 2;
data ← ArrayInt16(p);                    ▷ struttura per leggere allineato a Int16
struct ← {n, C, data};                   ▷ raggruppa i dati in una struttura
return struct;
```

$$
\begin{aligned}
&\textit{Dimensione del pacchetto specificato da Sacchetto} = 8 + 1 + 128 * 2 = 265 \text{ byte} \\
&\textit{Nuova dimensione del pacchetto} = 8 + 2 + 128 * 2 = 266 \text{ byte} \\
&\textit{Frame al secondo} = \textit{packets per second} = 48000/128 = 375 \\
&\textit{Byte extra al secondo} = 375 * 1 = 375 \\
&\textit{Pacchetti extra al secondo} = 375/265 = 1{,}42 \\
&\textit{Byte inviati al secondo} = 375 * 265 = 99375 \\
&\textit{Banda di rete necessaria} = 99375 * 8/1024 = 766{,}4 Kbit/s \\
&\textit{Carico aggiuntivo richiesto} = 375 * 100/99375 = 0{,}38\%
\end{aligned}
\tag{3.2}
$$

## 3.7 Struttura generale dell'applicazione

In maniera molto superficiale e parziale viene qui descritta la struttura generale del codice dell'applicazione, almeno per quelle parti comuni a tutte le architettura e configurazioni che verranno di seguito testate. Partendo dal codice di Sacchetto [6], l'applicazione è stata ristrutturata con un maggior numero di classi per semplificare le modifiche concettuali e strutturali che questa tesi si propone di testare. Una rappresentazioni delle principali classi e delle loro relazioni può essere osservata in [fig. 3.5].

La classe su cui fa riferimento l'intera applicazione è `AudioApp`. Questa classe presenta all'esterno le principali funzionalità su cui si può avere controllo. Gestisce la creazione dell'`AudioContext`, ossia dell'ambiente che permette di prendere il controllo delle periferiche di acquisizione e riproduzione, e dei thread ad alta priorità con le API *WebAudio*.

La classe `AudioClientManager` ha il compito di creare e gestire gli oggetti `AudioPeer` per ogni peer connesso alla stanza a cui si è attualmente connessi. Tramite il metodo `sendToAll()` è in grado di inoltrare i pacchetti a tutti i peer perché li inviino sulla propria connessione RTC.

La classe `RTCDataChannelWrapper` mira a gestire la connessione direttamente sul canale `RTCDataChannel`. Si occupa di inviare all'esterno i pacchetti che riceve dal thread principale e di inserire in coda i pacchetti che arrivano dai peer esterni. Per l'accodamento, nel caso di architetture `SharedArrayBuffer` e `Atomics.waitAsync`, ha necessità di un riferimento al buffer circolare condiviso, a cui può accedere con l'ausilio della classe `AudioCircularBuffer`.

I thread ad alta priorità della catena audio sono principalmente due: `AudioSenderProcessor`, che si occupa di acquisire i frame audio, creare i pacchetti con

gli appositi formati e passarli al thread principale per la trasmissione agli altri peer; e `AudioReceiverProcessor`, che invece si occupa di estrarre i frame audio dalla coda e mandarli al sistema di riproduzione. Per controllare i thread in Javascript vengono creati altri oggetti che estendono la classe `AudioNode`. Nello specifico, nell'applicazione le classi create sono `AudioSenderNode` e `AudioReceiverNode`. Tramite il metodo `.port.postMessage()` e la gestione dell'evento `.port.onmessage` è possibile scambiare dati tra il thread principale e gli *AudioWorklet* (ossia i thread audio), e viceversa. È con questo meccanismo che vengono trasmessi le configurazioni e i buffer circolari condivisi.

Il buffer circolare è stato gestito tramite una coppia di classi: `AudioCircularBufferSingle`, per gestire un singolo canale audio, e `AudioCircularBuffer`, per la gestione di tutti i canali insieme, degli indici di lettura e di scrittura più altre variabili necessarie al controllo dello stato interno della coda. Queste classi permettono l'accesso alle strutture condivise in maniera univoca all'interno dell'applicazione. Tramite la funzione `share()` è possibile creare una copia dei riferimenti a tali strutture per andare a creare una nuova istanza della classe `AudioCircularBuffer` che operi sulle stesse aree di memoria. Queste strutture condivise vengono create per la prima volta all'avvio dell'applicazione dalla classe `AudioApp` e successivamente condivise con tutte le altre istanze che si vengono a creare in fase di inizializzazione o a runtime. Durante la creazione di una nuova istanza della classe `AudioCircularBuffer` da parte dell'oggetto `RTCDataChannelWrapper` avviene anche la registrazione del peer nel buffer, utile alla normalizzazione dei sample in fase di estrazione. Nel caso dell'architettura con `Atomics.waitAsync`, viene creato un nuovo buffer circolare, quindi delle strutture dati differenti utili a formare la coda di trasmissione che viene utilizzata dal `AudioSenderProcessor` al posto del `MessageChannel`.

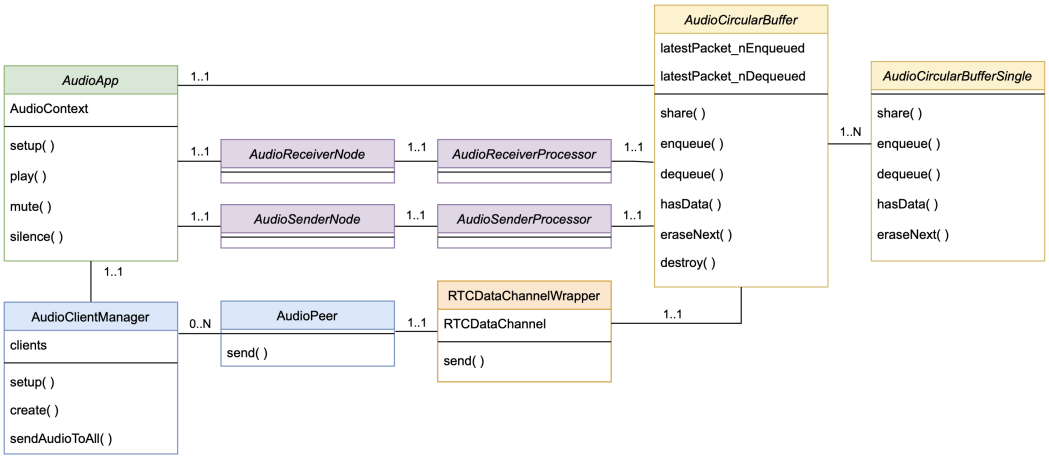


Figura 3.5. Diagramma delle principali classi dell'applicazione e delle loro relazioni. Il diagramma è una versione semplificata, per maggiori dettagli si invita a consultare il codice

# Capitolo 4

# Misure e performance

Le misurazioni sono state fatte sui due principali browser utilizzati a livello globale: *Firefox* e *Google Chrome*, nelle loro versioni rilasciate ad oggi più recenti. La configurazione utilizzata nei test mirava a ricreare una situazione quanto più simile alla realtà. Per questo tutte le misure effettuate sono mouth-to-ear, o più precisamente, in questo caso speaker-to-microphone.

L'applicazione è inoltre stata arricchita con alcune funzionalità per la misurazione del jitter. È possibile misurare il jitter in tre istanti: in acquisizione, in ricezione e durante l'inserimento nel buffer circolare. In acquisizione si cerca di misurare quanto oscilla l'acquisizione dei valori della scheda audio, in ricezione invece le oscillazioni tra un pacchetto e l'altro, quindi considerando le oscillazioni di acquisizione, di trasmissione, del canale e di ricezione, e infine nell'inserimento dei frame nel buffer circolare le oscillazioni dei ritardi necessari all'accodamento. Quest'ultima misurazione può risultare difficile con un sistema a pipe, in quanto non è sempre possibile sapere dall'altro lato quando il valore viene effettivamente ricevuto dalla pipe e inserito nel buffer circolare. Inoltre, dato l'elevato flusso di dati, la raccolta e l'elaborazione del valori istantanei del jitter è un peso aggiuntivo percepibile nelle performance complessive dell'applicazione.

Le misurazioni dei tempi all'interno delle applicazioni possono avvenire in differenti maniere. Per un clock ad alta precisione esiste l'API `performance.now()`. Essendo questa una proprietà del DOM, non è accessibile dai *WebWorker* per ragioni di sicurezza. Lì le misurazioni possono basarsi sulla creazione di oggetti `Date`, che però non garantiscono una alta precisione dei valori, oppure tramite lettura della proprietà `currentTime` degli oggetti `AudioContext`. Il tempo restituito da `AudioContext.currentTime` è relativo alla configurazione audio con cui è stato inizializzato l'`AudioContext` alla creazione.

## 4.1 Configurazione e architettura di test

Tutti i test sono stati eseguiti con la seguente configurazione [tab. 4.1] e coi seguenti dispositivi [tab. 4.2].

| | |
|---|---|
| `sample rate` | 48000 Hz |
| `frame size` | 128 |
| `queue size` | 16 |
| `queue offset` | 8 |
| versione Google Chrome | 94.0.4606 x64 |
| versione Firefox | 92.0.1 x64 |

Tabella 4.1. Configurazione test

| | |
|---|---|
| PC1 | MacBookPro 13" 2020<br>OS Catalina, CPU Intel i5, RAM 8GB |
| PC2 | Asus N551JX<br>OS Windows 10, CPU Intel i7, RAM 16GB |
| PC3 | assemblato<br>Linux PopOs 20.04, CPU Intel i9, RAM 32GB |
| schede audio esterne | Focusrite 2i2 v.2 |
| mixer esterno | Yamaha MG06 |
| registratore | Sony IC recorder |
| switch | Netgear GS308 Gigabit Switch |
| router | Technicolor TG588v |

Tabella 4.2. Dispositivi di test

Per la disposizione dei dispositivi durante i test è stato preso questo documento del NIST come guida [4] [fig. 4.1]. Vi sono due dispositivi distinti, uno sorgente *PC1* e uno destinazione *PC2*. Le analisi e le registrazioni dei risultati vengono effettuati da dispositivi esterni. Un terzo dispositivo *PC3* si occupa mano a mano di aggiungere istanze dell'applicazione, così da aumentare il carico di dati per misurarne la scalabilità. L'accesso all'applicazione e la creazione delle stanze dove si connetto i peer vengono gestiti dal *server*.

La situazione è stata adattata alle tecnologie e alle condizioni che si possono avere in una rete domestica. In particolare, il *server* è in esecuzione sullo stesso computer dell'istanza sorgente *PC1*. Questo è dovuto ad alcune limitazioni nell'utilizzo di HTTPS (necessario per poter attivare gli `SharedArrayBuffer`) in Google Chrome su dispositivi Apple. Il *server* si occupa solamente delle operazioni di notifica

tra i peer (con protocollo WebSocket) e creazione delle stanze, quindi il suo peso computazione è molto limitato.

Tutti i collegamenti tra i computer sono via cavo ethernet con uno switch intermedio. Allo switch è collegato anche un router per la creazione di una rete locale e la gestione degli indirizzi IP. È stato preferito inserire questo switch intermedio nella connessione diretta dei computer siccome è stato considerato il possibile uso di buffer intermedi da parte del router.

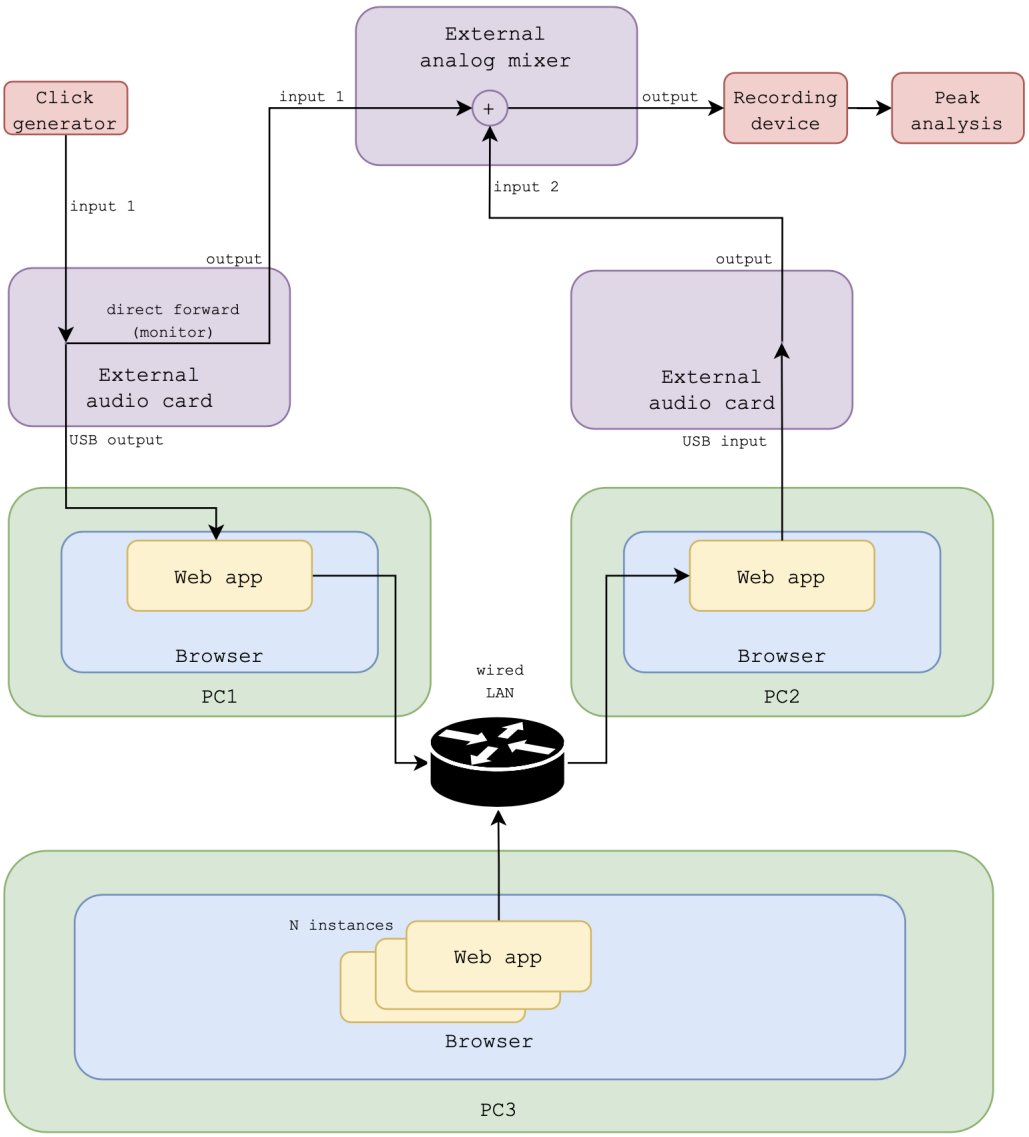


Figura 4.1. Architettura utilizzata per i test

Per ogni architettura o configurazione di seguito presentata sono stati svolti tre test. Dove viene indicato un singolo valore, quello è la media. Questa scelta è dovuta a un'alta stabilità dei valori anche in test consecutivi. I test preliminari che sfruttano un segnale che viene generato e reimmesso nel sistema (*loopback*) sono stati nominati con delle lettere progressive per semplificare la lettura dei dati e dei grafici. Ogni caso rappresenta una configurazione utilizzata per il test. Il test di varie architetture viene identificato da un numero progressivo. I casi con lievi variazioni nella configurazione sono stati differenziati con degli apici a fianco della lettera.

## 4.2 Looback sullo stesso dispositivo

I primi test sono stati effettuati utilizzando un solo dispositivo, *PC1* nello specifico, andando a misurare il ritardo in maniera incrementale aggiungendo parti dell'applicazione. In questi casi l'obiettivo è uno studio preliminare del comportamento del browser con le varie architetture, senza un reale carico e senza ancora porsi nelle condizioni di analizzare la scalabilità e l'efficienza dell'applicazione.

I valori raccolti sono la media di tre test consecutivi con la stessa configurazione, aprendo e chiudendo il browser ogni volta. Per ogni test il valore selezionato è un valore a cui tendenzialmente il sistema si stabilizza entro pochi secondi dall'avvio dell'esecuzione.

Per svolgere questi test sono stati aggiunti due moduli *AudioNode* all'applicazione nella catena audio dell'`AudioContext`: un `BeepGeneratorProcessor` che genera una breve sinusoide a istanti regolari (indicata per semplicita come un *beep*), e un `BeepAnalyzerProcessor` in grado di rilevare i picchi. Questi due *AudioNode* hanno una memoria condivisa per memorizzare i tempi di generazione e rilevazione di un beep, con l'ausilio degli oggetti `Date`. Quando viene rilevato un nuovo picco, viene calcolato il ritardo come differenza dei due tempi e visualizzato nella console del browser. Nel generare il beep, il `BeepGeneratorProcessor` scarta completamente il segnale originale che potrebbe arrivargli in input e lo sostituisce con una breve sinusoide oppure silenzio. I beep sono distanziati sufficientemente in modo che il loro periodo di riproduzione sia molto maggiore dei ritardi misurabili.

### 4.2.1 Loopback diretto sullo stesso dispositivo (caso A)

Il primo test consiste nella misurazione del ritardo Mouth-to-ear senza moduli aggiunti nella catena audio. L'architettura risultante dell'applicazione è rappresentata in [fig. 4.2]. Questo caso è stato definito come un looback diretto tra il generatore `BeepGeneratorProcessor` e il rilevatore `BeepAnalyzerProcessor`, così da avere il minimo ritardo che il sistema può raggiungere. Sostanzialmente, il test

mira a rilevare il ritardo imposto dal dispositivo e dal browser al di sotto del quale non possiamo andare. I ritardi rilevati sono presentati nella tabella [tab. 4.3].

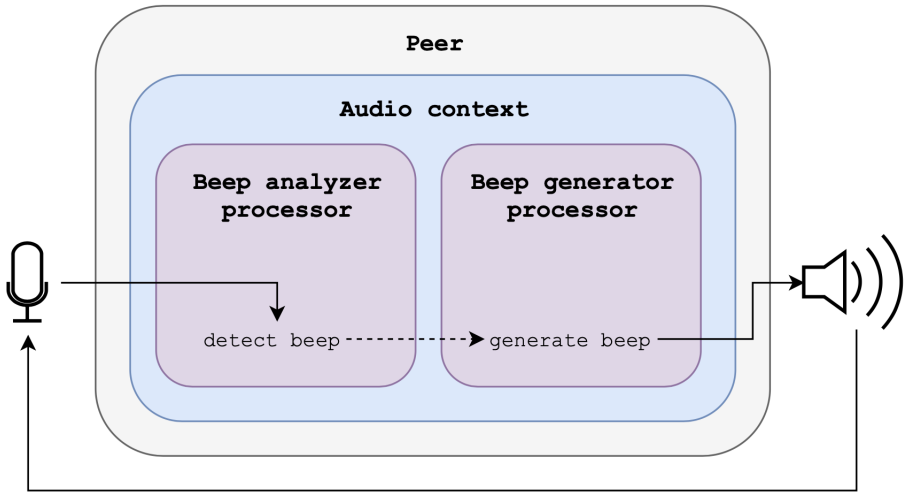


Figura 4.2. Architettura looback diretto sullo stesso dispositivo (caso A)

| Browser | Latenza (ms) |
|---|---|
| Firefox | 49,3 |
| Google Chrome | 62,7 |

Tabella 4.3. Latenza con architettura looback diretto (caso A)

Come consigliato da alcuni programmatori conosciuti durante la conferenza *WAC2021*, su molti dispositivi i ritardi sono sensibilmente differenti a seconda se si utilizzano microfono e speaker integrati oppure la connessione minijack. Il test è stato ripetuto usando un caso ancora più conveniente, ossia con una scheda audio esterna di supporto collegata via USB e cortocircuitata tra l'uscita cuffie e un ingresso tramite un cavo jack fisico. La scheda audio è in grado di garantire una maggiore stabilità nel flusso di acquisizione e di riproduzione. L'architettura modificata è rappresentata in [fig. 4.3], mentre i risultati ottenuti e la loro differenza rispetto a [tab. 4.3] sono riportati in [tab. 4.4]. La selezione della scheda audio esterna come dispositivo di input/output è stata fatta a livello di sistema operativo e non internamente al browser, perché, come indicato nell'elaborato [6], comporterebbe maggiori ritardi. Da qui in poi, per differenziarsi dai casi con microfono e speaker integrati, verrà posto un apice a fianco della lettera che identifica il caso con l'utilizzo della scheda audio esterna.

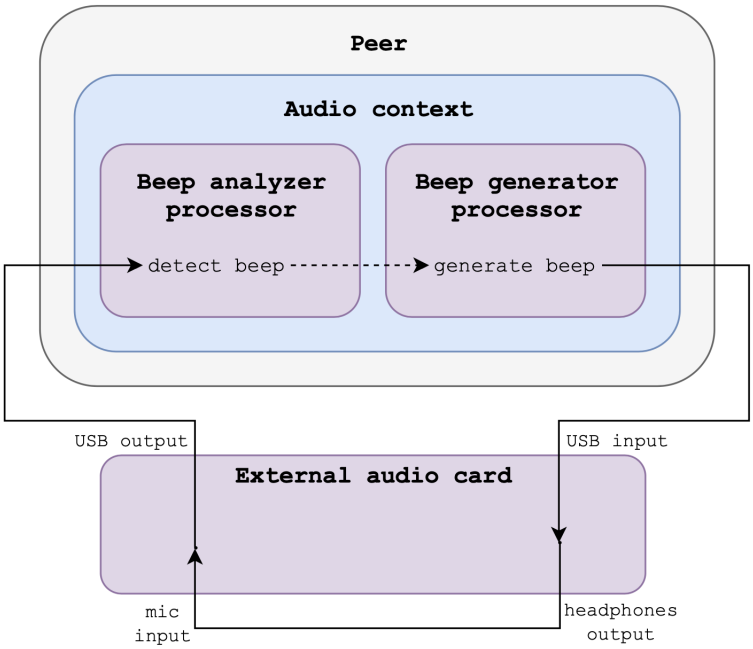


Figura 4.3. Architettura looback diretto sullo stesso dispositivo con scheda audio esterna (caso A')

| Browser | Latenza (ms) | Differenza (ms) |
|---|---|---|
| Firefox | 10,2 | 39,1 |
| Google Chrome | 20,9 | 41,8 |

Tabella 4.4. Latenza con architettura looback diretto con scheda audio esterna (caso A') e differenza rispetto i dati presentati in [tab. 4.3]

Con questa prima analisi dei ritardi si vuole dimostrare come i suggerimenti dati riguardo all'utilizzo o meno di microfono e speaker integrati possano avere importanti differenze sulle prestazioni a partire dall'acquisizione e dalla riproduzione audio. Abbassando il limite sulle latenze imposto dal dispositivo e dal browser al di sotto dei 30 ms, è possibile ipotizzare un reale impiego dell'applicazione web per contesti live di *Networked music performance*.

### 4.2.2 Loopback indiretto sullo stesso dispositivo (caso B)

Dopo le misurazioni dirette, sono stati inseriti i primi moduli dell'applicazione finale, quindi il `AudioSenderProcessor` utile a raccogliere i frame dal microfono,

convertire i campioni da `Float32` a `Int16` e creare i pacchetti, e il `AudioReceiverProcessor` utile a estrarre i frame dalla coda e mandarli ai dispositivi di playout. Per effettuare questo test è stata fatta una modifica al codice che permette all'`AudioSenderProcessor` di creare il pacchetto e di inserirlo direttamente nella coda di ricezione. In questo caso non c'è passaggio di dati al *main thread*. Una semplificazione dello schema risultante è rappresentato in [fig. 4.4]. Ci sono piccole differenze implementative relative all'architettura usata: con `MessageChannel` (caso B1) e `SharedArrayBuffer` (caso B2) non vi è differenza nelle operazioni svolte dal `AudioSenderProcessor` di creazione del pacchetto e inserimento nella coda di ricezione; con `Atomics.waitAsync` (caso B3), siccome il pacchetto verrebbe creato dal *main thread*, i dati vengono direttamente inseriti nella coda di ricezione in formato `Int16`. Il differente formato viene gestito tramite appositi flag dalla classe che implementa il buffer circolare `AudioCircularBuffer`. Le misurazioni effettuate sono riportate in [tab. 4.5], associate alla corrispondente differenza misurata nel caso del loopback diretto, ovvero al ritardo aggiunto con questi nuovi moduli. In questa situazione si usa un buffer circolare di dimensione 16 frame e con offset di 8 frame, che con una frequenza di campionamento di 48000 Hz e frame di 128 campioni corrispondono a circa 21,3 ms di ritardo.

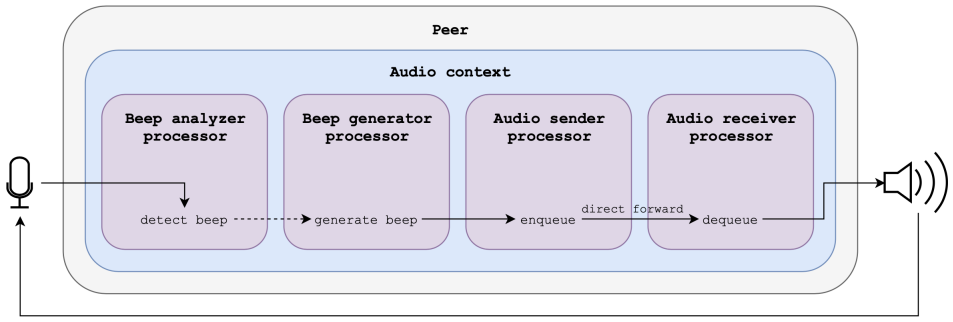


Figura 4.4. Architettura looback indiretto su un solo dispositivo (caso B)

| **Browser** | **Architettura** | **Latenza (ms)** | **Differenza (ms)** |
|---|---|---|---|
| Firefox | MessageChannel | 72,4 | 23,1 |
| Firefox | SharedArrayBuffer | 72,4 | 23,1 |
| Google Chrome | MessageChannel | 86,7 | 24,0 |
| Google Chrome | SharedArrayBuffer | 87,2 | 24,5 |
| Google Chrome | Atomics.waitAsync | 86,7 | 24,0 |

Tabella 4.5. Latenza con architettura looback locale (caso B) e differenza rispetto i dati presentati in [tab. 4.3]

Lo stesso esperimento è stato nuovamente ripetuto con l'ausilio della scheda audio esterna, come indicato in [fig. 4.5]. I risultati presentati in [tab. 4.6] indicano anche la differenza rispetto il medesimo caso con loopback diretto e con l'impiego della scheda audio esterna.

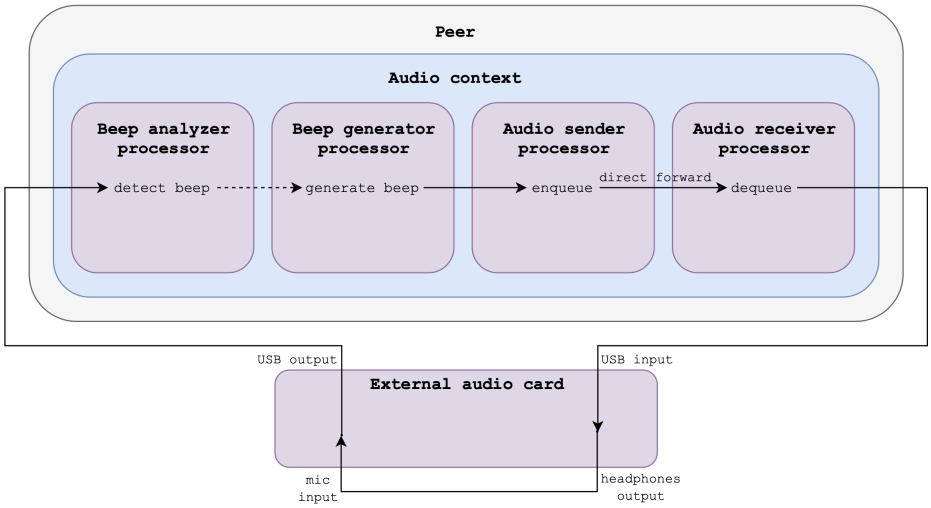


Figura 4.5. Latenza con looback indiretto sullo stesso dispositivo con scheda audio esterna (caso B')

| **Browser** | **Architettura** | **Latenza (ms)** | **Differenza (ms)** |
|---|---|---|---|
| Firefox | MessageChannel | 31,6 | 21,4 |
| Firefox | SharedArrayBuffer | 31,7 | 21,5 |
| Google Chrome | MessageChannel | 42,2 | 21,3 |
| Google Chrome | SharedArrayBuffer | 42,3 | 21,4 |
| Google Chrome | Atomics.waitAsync | 42,2 | 21,3 |

Tabella 4.6. Latenza con looback indiretto sullo stesso dispositivo con scheda audio esterna (caso B') e differenza rispetto i dati presentati in [tab. 4.4]

Mentre nei casi che utilizzano una scheda audio esterna i ritardi sono limitati al ritardo imposto dalla presenza di un buffer intermedio, nel caso di utilizzo di microfono e speaker integrati si vedono lievi peggioramenti, probabilmente dovuti al maggiore jitter del flusso dei dati audio in acquisizione e in riproduzione con l'utilizzo delle periferiche fornite direttamente dal computer e a una conseguente maggiore difficoltà di elaborazione da parte del browser.

### 4.2.3 Loopback indiretto con interfaccia di rete sullo stesso dispositivo (caso C)

Infine, è stato considerato il caso con due istanze dell'applicazione aperte in due finestre dello stesso browser. I due peer si trovano quindi sullo stesso dispositivo, utilizzando l'indirizzo di localhost per scambiarsi i dati. Mentre la prima istanza si occupa di generare e rilevare i beep, la seconda effettua il forward di tutti i dati ricevuti senza inserirli nel buffer. L'architettura risultante è indicata in [fig. 4.6], mentre i dati raccolti sono in [tab. 4.7]. Nuovamente sono state testate tutte e tre le architettura precedentemente discusse: `MessageChannel` (caso C1), `SharedArrayBuffer` (caso C2) e `Atomics.waitAsync` (caso C3).

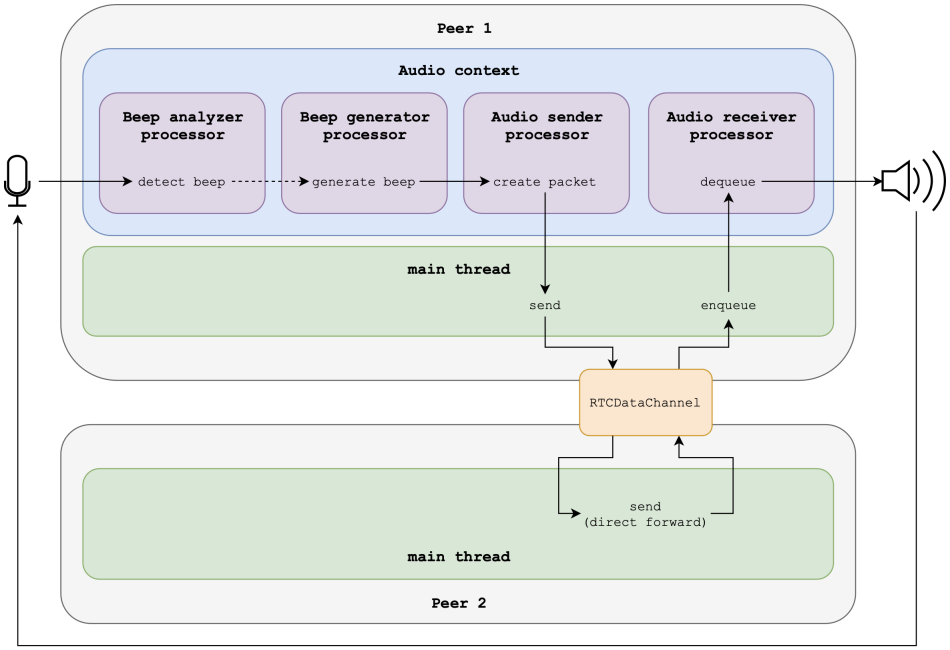


Figura 4.6. Architettura looback indiretto con interfaccia di rete (caso C)

Analogamente a quanto fatto prima, anche in questo caso sono stati ripetuti i test con una scheda audio esterna. L'architettura è indicata in [fig. 4.7] e i dati raccolti sono esposti in [tab. 4.8].

Nuovamente, è visibile come le versioni dell'applicazione che utilizzano microfono e speaker integrati siano molto più soggette a ritardi variabili in base al carico del dispositivo utilizzato rispetto a quelle che utilizzano una scheda audio esterna. Una causa che può spiegare questi ampi ritardi introdotti con l'aggiunta di un secondo peer può essere ricercata nell'utilizzo dei dispositivi di acquisizione e riproduzione da parte di più schede del browser in contemporanea. Se si prosegue con questa

| Browser | Architettura | Latenza (ms) | Differenza (ms) |
|---|---|---|---|
| Firefox | MessageChannel | 86,0 | 13,6 |
| Firefox | SharedArrayBuffer | 86,9 | 14,5 |
| Google Chrome | MessageChannel | 120,0 | 33,3 |
| Google Chrome | SharedArrayBuffer | 119,4 | 32,2 |
| Google Chrome | Atomics.waitAsync | 118,4 | 31,7 |

Tabella 4.7. Latenza con architettura loopback indiretto con interfaccia di rete con un solo device (caso C) e differenza rispetto i dati presentati in [tab. 4.5]

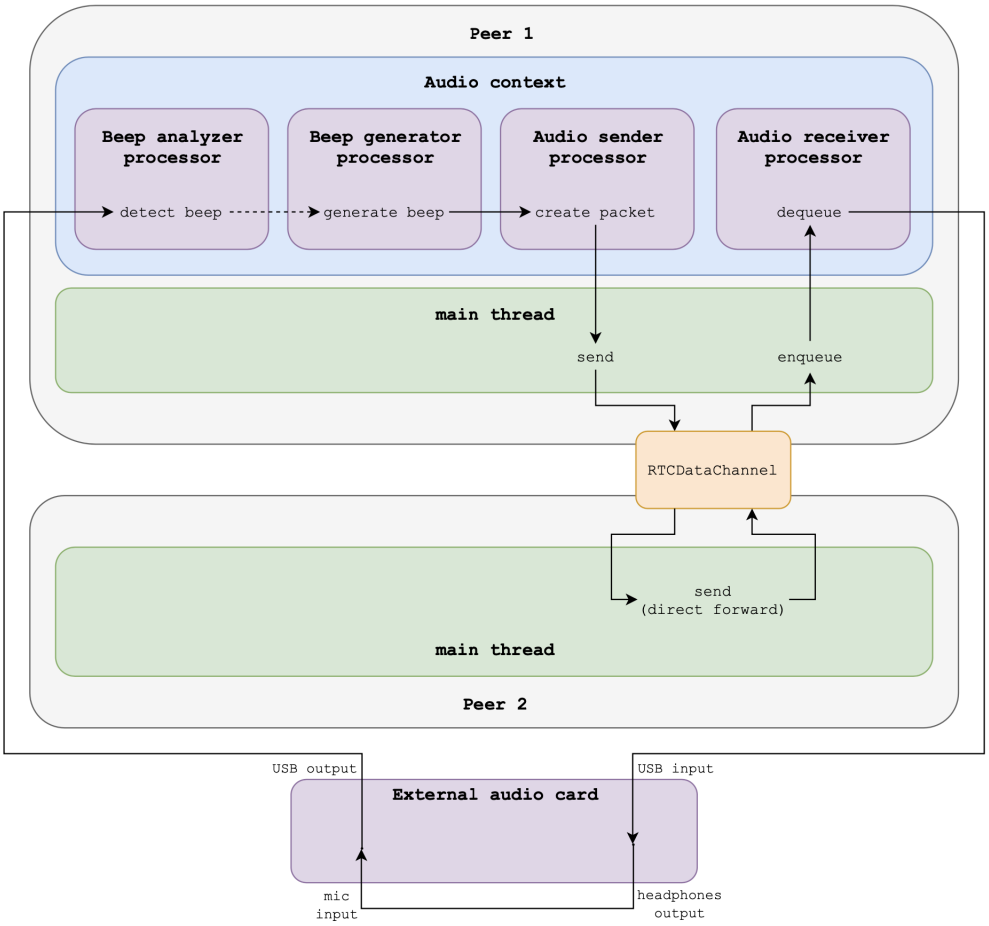


Figura 4.7. Latenze con looback indiretto con interfaccia di rete con scheda audio esterna (caso C')

ipotesi, le differenti prestazioni tra con o senza scheda audio esterna potrebbero quindi ricondursi a una differente ottimizzazione dei driver utilizzati per controllare

| Browser | Architettura | Latenza (ms) | Differenza (ms) |
|---|---|---|---|
| Firefox | MessageChannel | 35,6 | 4,0 |
| Firefox | SharedArrayBuffer | 35,1 | 3,4 |
| Google Chrome | MessageChannel | 47,6 | 5,4 |
| Google Chrome | SharedArrayBuffer | 49,3 | 7,0 |
| Google Chrome | Atomics.waitAsync | 53,8 | 11,6 |

Tabella 4.8. Latenza con architettura loopback indiretto con interfaccia di rete con un solo dispositivo con scheda audio esterna (caso C') e differenza rispetto i dati presentati in [tab. 4.6]

tali periferiche.

Osservando in generale i test sinora svolti su un solo dispositivo, non è identificabile una netta differenza di performance delle varie architetture testate in questa analisi preliminare. I valori ottenuti in questo paragrafo saranno considerati di riferimento per i successivi test. Inoltre, mentre questi dati possono essere letti come valori assoluti perché ottenuti in condizioni standard sullo stesso dispositivo e con al più l'ausilio di una scheda audio esterna, non sarà possibile considerare il valore assoluto dei dati ottenuti con più dispositivi perché quelli utilizzati sono molto eterogenei tra loro, non riuscendo a ricreare una situazione di test ideale. Le differenze sono a livello sia hardware, che di sistema operativo e software. Per questo i valori con più dispositivi sono da leggere in maniera relativa per comparare le diverse architettura o identificare dei trend.

Infine, è possibile osservare e confrontare i risultati dei test sinora considerati divisi per browser nei grafici [fig. 4.8] e [fig. 4.9] per i casi con microfono e speaker integrati, mentre nei grafici [fig. 4.10] e [fig. 4.11] per i casi con l'utilizzo della scheda audio esterna.

## 4.3 Loopback indiretto su due dispositivi (caso D)

Questa sezione di test è stata effettuata mettendo in loobpack indiretto due istanze dell'applicazione su due dispositivi distinti ma connessi alla stessa rete. Nello specifico, mentre il *PC1* si occupa di generare e rilevare i beep, il *PC2* è stato configurato in modo che invii al *PC1* i dati appena ne riceve, senza inserirli nel buffer. Lo schema risultante è simile a quello indicato in [fig. 4.6], con la differenza che i due peer sono su due dispositivi distinti, con l'accortezza che operino entrambi nello stesso tipo di browser (Firefox o Google Chrome) per uniformità nei test. I

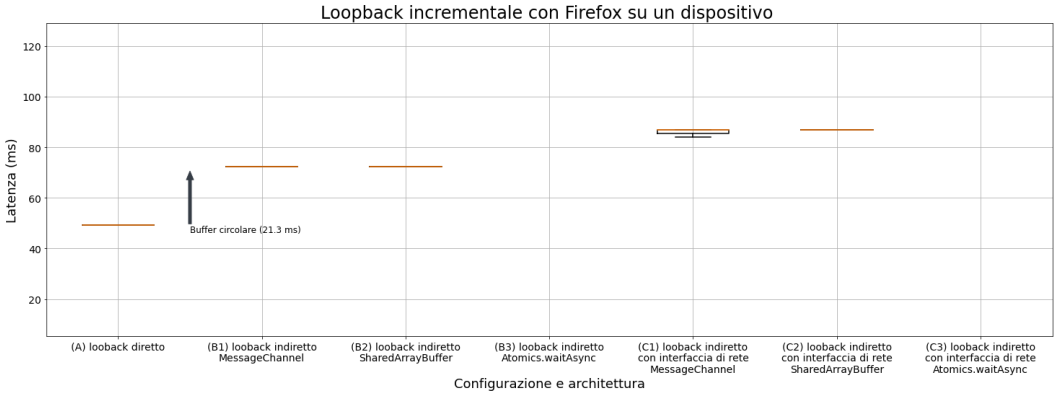


Figura 4.8. Riassunto delle latenze di loopback in maniera incrementale con Firefox con un dispositivo

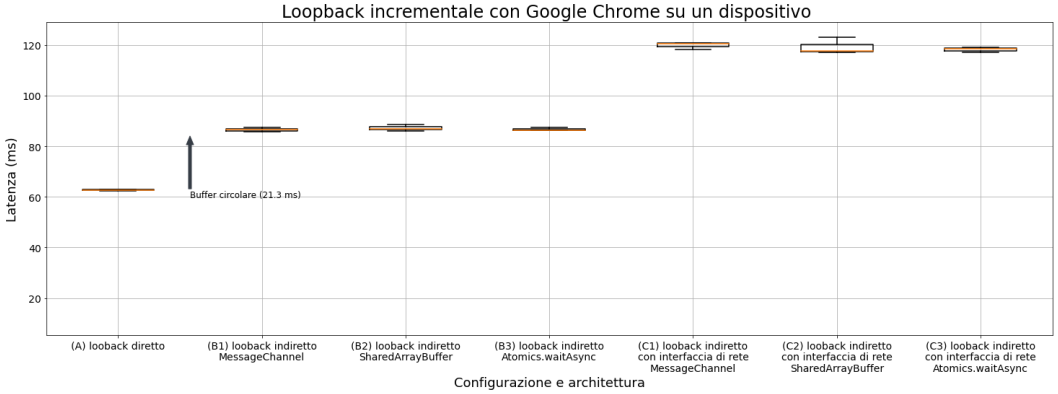


Figura 4.9. Riassunto delle latenze loopback in maniera incrementale con Google Chrome con un dispositivo

test sono stati ripetuti tre volte per tutte le architettura precedentemente testate: `MessageChannel`, `SharedArrayBuffer` e `Atomics.waitAsync` (quest'ultima solo nel caso di Google Chrome). I grafici [fig. 4.12] e [fig. 4.13] mostrano i dati raccolti, divisi per browser.

Nuovamente il test è stato ripetuto con una scheda audio esterna [fig. 4.7], producendo i risultati riportati in [fig. 4.14] e in [fig. 4.15].

Inserendo elementi di rete tra i due dispositivi si iniziano a notare risultati più sparsi, ma con valori che rimangono molto stabili per ogni singolo test. In generale non possiamo ancora notare sostanziali differenze tra le architetture prese in esame.

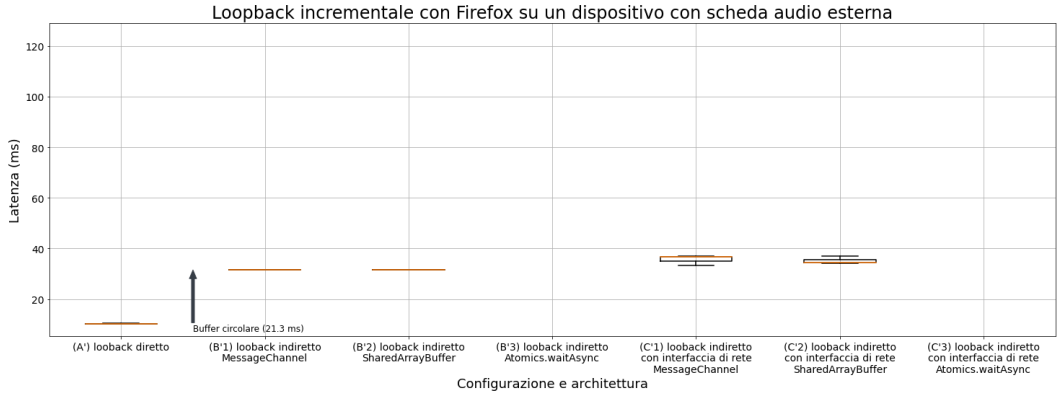


Figura 4.10. Latenze loopback in maniera incrementale con Firefox con un dispositivo con scheda audio esterna

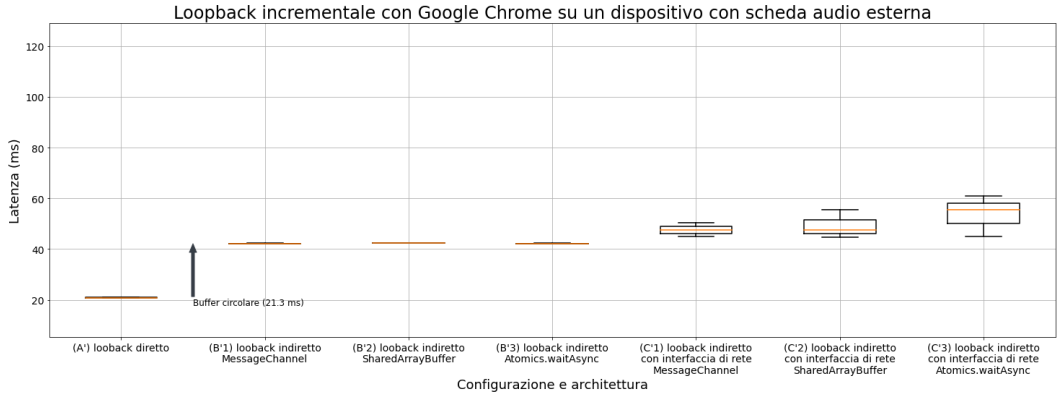


Figura 4.11. Latenze loopback in maniera incrementale con Google Chrome con un dispositivo con scheda audio esterna

## 4.4 Scalabilità su multipeer

Per i test di scalabilità è stato impiegato il *PC3*, dove sono state mano a mano aperte più istanze dell'applicazione. Si è partiti da zero istanze extra (solamente sorgente e destinazione) e si è arrivati fino a quattro istanze extra (sei in totale).

Tutti i test sono stati realizzati con due schede audio, connesse ai computer *PC1* e *PC2*. Per la scheda audio collegata al *PC1* è stato abitilitato il monitoring fisico, ovvero viene riprodotto sull'uscita l'audio acquisito all'ingresso. A questa scheda audio è stato collegato in ingresso un cellulare con una registrazione di sedici *click* emessi a istanti regolari di un secondo (indicato con *click generator* nei grafici). Con *click* si intende in questo caso un segnale audio di durata molto ridotta e con

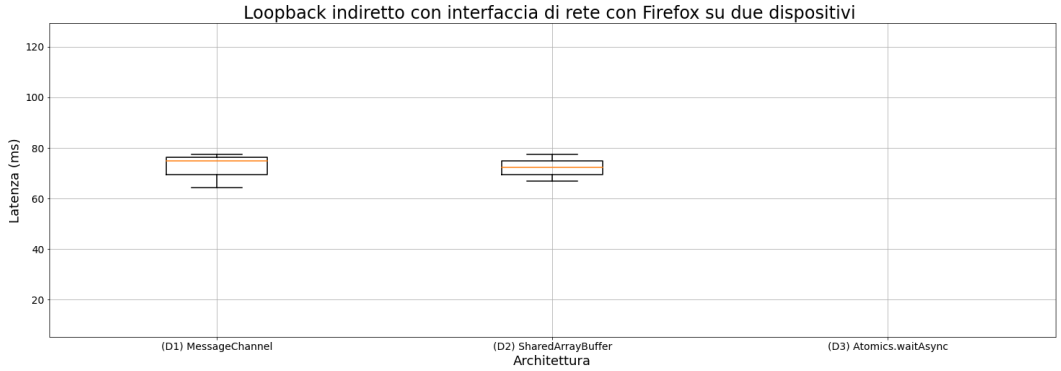


Figura 4.12. Latenze loopback indiretto con interfaccia di rete con Firefox con due dispositivi (caso D)

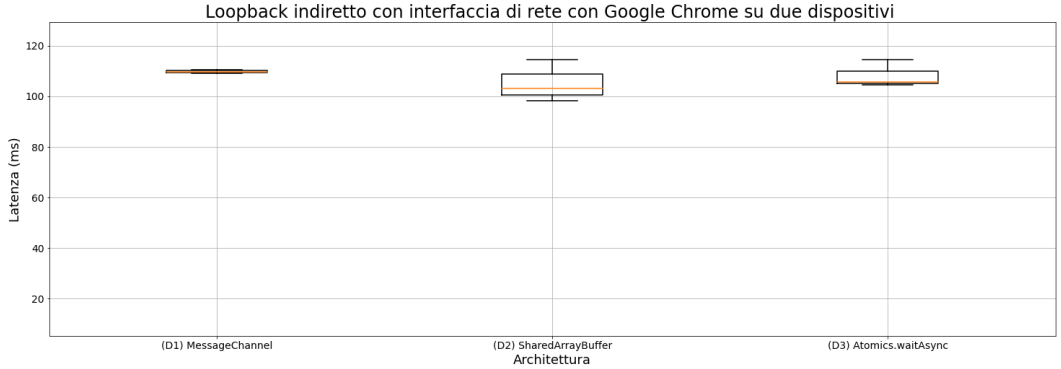


Figura 4.13. Latenze loopback indiretto con interfaccia di rete con Google Chrome con due dispositivi (caso D)

un attacco molto ripido. La durata non è unitaria come potrebbe essere inteso un impulso in un sistema digitale, ma di pochi millisecondi (20 ms circa). Così facendo diventa un segnale maggiormente ascoltabile dall'orecchio umano e più facilmente individuabile dai rilevatori di picco integrati nelle schede audio e nel mixer che, facendo illuminare dei led, permettono di avere un feedback del corretto funzionamento del test. I segnali uscenti dalle uscite cuffie delle schede audio sono stati missati con un mixer esterno e raccolti con un registratore (*recording device*). In una seconda fase è stata fatta tramite *Audacity* la misurazione delle distanze picco-picco tra il click generato e il click ricevuto dal *PC2*, così da valutare la latenza del sistema. Per ogni architettura testata sono stati fatti tre test consecutivi, ciascuno con da due a sei peer connessi progressivamente.

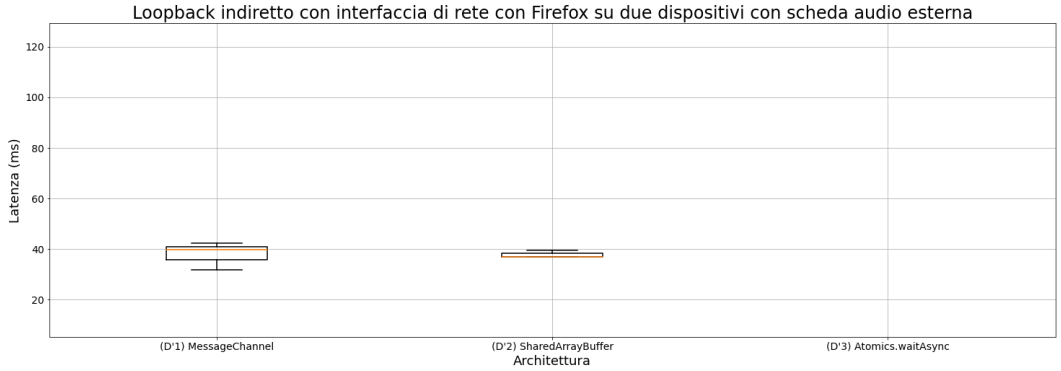


Figura 4.14. Latenze loopback indiretto con interfaccia di rete con Firefox con due dispositivi con scheda audio esterna (caso D')

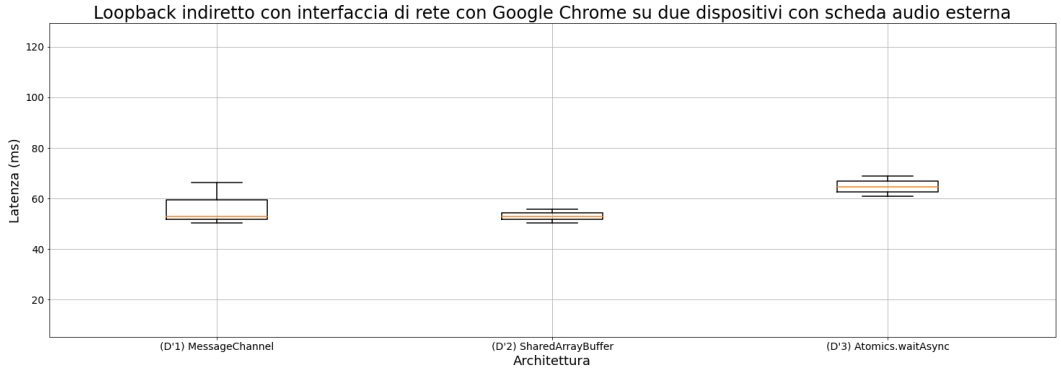


Figura 4.15. Latenze loopback indiretto con interfaccia di rete con Google Chrome con due dispositivi con scheda audio esterna (caso D')

L'altro indice della qualità del sistema si basa sul tenere traccia dei pacchetti correttamente processati. Non è stato utilizzato un meccanismo di logging interno all'applicazione in quanto, come già discusso per il tracciamento del jitter, queste operazioni inficiano sulle prestazioni che si vogliono misurare. Per semplificare quindi l'analisi dei pacchetti scartati dall'applicazione, perché arrivati all'inserimento in coda troppo tardi, è stata aggiunta la possibilità di sommare un suono continuo con un'ampiezza ridotta (una sinusoide a 200 Hz) al segnale ricevuto in input dal microfono integrato oppure dalla scheda audio. Successivamente all'esecuzione dei test è stata fatta un'analisi dei silenzi con *Matlab*. Nello specifico, tramite un rilevatore di picco sono state create delle finestre di 17 secondi attorno alle serie di click. In questo modo, ogni finestra parte un secondo prima del primo click e termina un secondo dopo il sedicesimo circa (il click ha una durata molto

ridotta, ma non nulla). Per ogni finestra è stata fatta un'analisi della percentuale di silenzio contenuto. Con un sistema con sample rate di 48000 Hz e frame da 128 si hanno 6375 frame per ogni finestra di 17 secondi. Così proseguendo è possibile conoscere il numero di pacchetti che è stato scartato dall'applicazione in ciascuna finestra all'aumentare del numero di peer connessi e che ha causato il silenzio individuato nelle registrazioni. Nelle registrazioni effettuate sono presenti i click ricevuti, il suono di fondo ricevuto e anche i click emessi. Questi ultimi, di durata ridotta ma non nulla, possono portare a lievi miglioramenti nei valori ottenuti, in quanto non contano come silenzio nella finestra. Con i dispositivi utilizzati non era possibile separare i segnali su canali differenti. In generale, data l'alta prestazione dello switch intermedio e delle connessioni di rete nella rete locale, è stato considerato molto improbabile, e successivamente anche verificato sperimentalmente durante i test, l'evento della perdita di un pacchetto dovuto a cause esterne all'applicazione; per questa ragione si parla di pacchetti scartati e non di pacchetti persi. Questo secondo valore appena ottenuto si va a combinare alle latenze picco-picco misurate in precedenza per valutare le performance, la scalabilità e la qualità complessiva del sistema.

Le medie del numero di frame scartati al crescere del numero di peer connessi alla stessa stanza per ciascuna architettura testata sono riportate in [tab. 4.9]. Le latenze misurate e la media percentuale dei frame scartati non rilevati durante l'analisi sono stati esposti nei grafici [fig. 4.16] e [fig. 4.17].

| **Browser** | **Architettura** | **2 peer** | **3 peer** | **4 peer** | **5 peer** | **6 peer** |
|---|---|---|---|---|---|---|
| Firefox | MessageChannel | 0,0 | 1905,8 | 3813,6 | 4928,8 | 3878,0 |
| Firefox | SharedArrayBuffer | 0,0 | 0,0 | 0,0 | 0,0 | 0,0 |
| Google Chrome | MessageChannel | 0,0 | 0 | 0 | 374,6 | 249.6 |
| Google Chrome | SharedArrayBuffer | 0,0 | 0,0 | 0,0 | 0,0 | 0,0 |
| Google Chrome | Atomics.waitAsync | 0,0 | 0,0 | 0,0 | 0,0 | 0,0 |

Tabella 4.9. Media su tre test consecutivi del numero di frame scartati all'interno di finestre di analisi da 6375 frame (17 secondi) per numero crescente di peer connessi alla stessa stanza

## 4.5 Confronto con applicazioni esistenti

Per confrontare i risultati precedentemente ottenuti, sono state effettuate delle misurazioni di altri sistemi quanto più simili a livello di tecnologie utilizzate. Sono state pertanto scelte delle altre applicazioni che operano nel browser e che sfruttano le API *WebRTC* per la comunicazione tra gli utenti. Per queste misurazioni la procedura di test è stata analoga a quella adottata per i test sulla scalabilità della

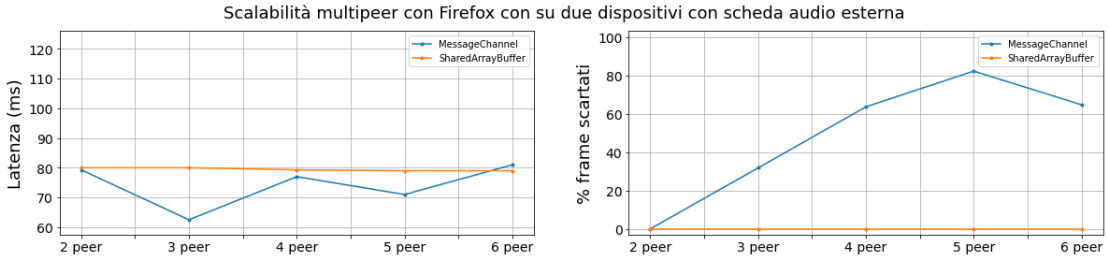


Figura 4.16. Latenze e media percentuale su tre test consecutivi del numero di frame scartati all'internodi finestre da 6375 frame (17 secondi) per numero crescente di peer connessi alla stessa stanza ottenuti con Firefox su due dispositivi con scheda audio esterna

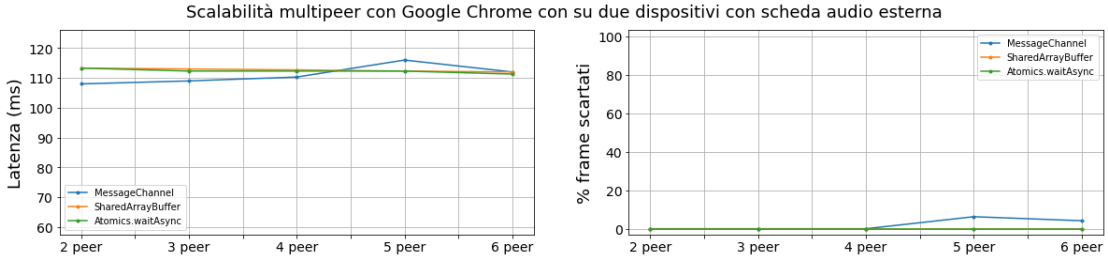


Figura 4.17. Latenze e media percentuale su tre test consecutivi del numero di frame scartati all'internodi finestre da 6375 frame (17 secondi) per numero crescente di peer connessi alla stessa stanza ottenuti con Google Chrome su due dispositivi con scheda audio esterna

sezione 4.4, con i collegamenti tra computer e dispositivi esterni come indicato in [fig. 4.1]. Nei grafici è stata cambiata la scala di riferimento per l'asse delle latenze per rendere i valori e le loro oscillazioni maggiormente leggibili al lettore.

### 4.5.1 Confronto con l'applicazione di Sacchetto

Dato che il codice sviluppato in questa tesi vuole essere una continuazione di quello iniziato da Sacchetto in [6], il primo confronto di seguito presentato è stato svolto con la versione ad oggi più recente pubblicata sull'omonimo repository *GitHub* del progetto. Avendo l'obiettivo di migliorarne la scalabilità, sono stati effettuati dei test con numero di peer crescente (da due a sei) usando la stessa procedura utilizzata nella sezione 4.4 per valutare la scalabilità delle architetture. Siccome l'applicazione di Sacchetto supportava anche la trasmissione video, per un confronto più oggettivo è stata spenta la videocamera tramite l'interfaccia grafica dell'applicazione stessa.

Per non apportare modifiche all'applicazione di Sacchetto, non è stato possibile avere il suono continuo di fondo (la sinusoide a 200 Hz) e quindi avere delle statistiche sul numero di pacchetti scartati dall'applicazione. Ne segue che nei grafici è stato valutato la media percentuale del numero di click rilevati e non la media percentuale del numero di frame scartati, com'era invece avvenuto nella sezione 4.4. Questa constatazione, combinata con la difficoltà nell'analizza precisamente il numero di pacchetti persi durante le prove svolte, porta a precisare che andrebbero svolti test maggiormente approfonditi per comparare al meglio le prestazioni e la qualità di questa applicazione con quelle presentate nel capitolo 3.

Dai dati presentati in [fig. 4.18] e [fig. 4.19], possiamo osservare come su Firefox il sistema comporti così tante perdite da non poter completare le misurazioni. Al contrario, su Google Chrome c'è una minore latenza in valore assoluto rispetto ai dati della sezione 4.4 e una accettabile invariabilità delle latenze all'aumentare del numero di peer.

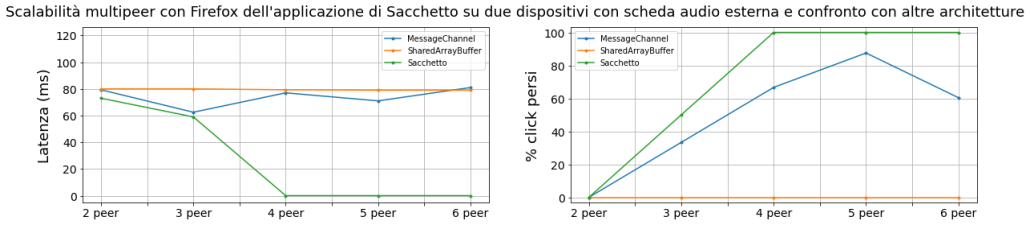


Figura 4.18. Scalabilità con Firefox dell'applicazione di Sacchetto con scheda audio esterna e confronto per numero crescente di peer connessi alla stessa stanza con le architetture precedentemente discusse

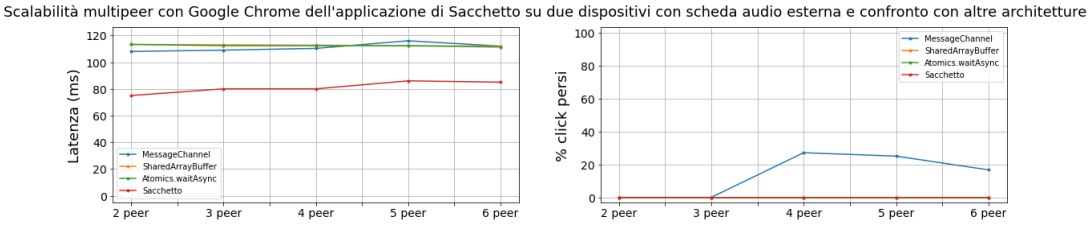


Figura 4.19. Scalabilità con Google Chrome dell'applicazione di Sacchetto con scheda audio esterna e confronto per numero crescente di peer connessi alla stessa stanza con le architetture precedentemente discusse

### 4.5.2 Confronto con Google Meet

Come ultimi test si è passati alle misurazioni di alcune applicazioni tra le più popolari che utilizzano le tecnologie *WebRTC*. La prima a essere stata considerata

per un confronto con le misurazioni precedentemente ottenute è stata *Google Meet*, la piattaforma di videoconferenze utilizzabile direttamente nel browser più in voga al momento. L'obiettivo di questo test era verificare le latenze in termini assoluti e non la scalabilità di *Google Meet*. Per questo, sono stati utilizzati solamente due client per tre prove consecutive. Anche per svolgere questi test è stata spenta la condivisione della videocamera.

Osservando il grafico [fig. 4.20] si possono vedere delle alte latenze, anche superiori ai 200 ms, e una loro maggiore variabilità. L'applicazione ha una configurazione che punta a fornire buoni risultati per le videochiamate, quindi utilizza anche funzionalità di *autogain* sul segnale acquisito e di riduzione del rumore di fondo, funzionalità fornite dal browser ma che comportano un peggioramento ulteriore dei ritardi. Analizzando i log del browser, come indicato in 2.3, è stato notato che l'applicazione opera in modalità client-server tramite dei server esterni. Questo comporta ritardi maggiori, dovuto al segnale che entra ed esce dalla rete locale.

### 4.5.3 Confronto con Jitsi

Un'altra soluzione software nota che è stata valutata è *Jitsi*. In questo caso per i test è stato utilizzato un server interno alla rete dell'università, anziché uno pubblico, ma comunque esterno alla LAN creata per questi test. Il vantaggio di avere un server privato è nel poter considerare il carico a cui è sottoposto il server stesso, che in questo caso era di una sola stanza attiva per questo test. Analizzando i log del browser come indicato nella sezione 2.3, è stato notato come l'applicazione funzioni in modalità peer-to-peer, quindi è in grado di creare una connessione diretta tra i client nella rete interna. Questo è il caso più simile a quanto si è voluto ricreare nei capitoli precedenti. Per motivazioni analoghe a quelle usate nel caso di *Google Meet*, sono stati utilizzati solamente due client per tre prove consecutive. Anche in questo caso è stata spenta la condivisione della videocamera.

Come per *Google Meet*, dal grafico [fig. 4.20] si possono notare dei valori delle latenze che si avvicinano a quelli considerati accettabili per le videoconferenze, ma troppo alti per la una performance musicale in diretta. Con la trasmissione peer-to-peer interna alla rete LAN creata per i test si riescono a recuperare alcune decine di millisecondi che nel caso di *Google Meet* venivano spesi per entrare e uscire dalla rete per raggiungere il server esterno.

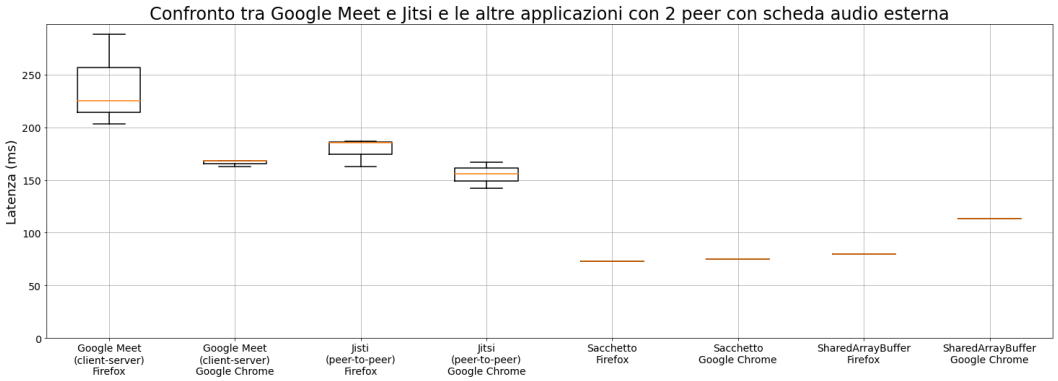


Figura 4.20. Latenze misurate con Google Chrome in modalità client-server e con Jitsi in modalità peer-to-peer con scheda audio esterna. A fianco, i risultati ottenuti con i test precedenti nel caso di 2 peer per farne una comparazione

# Capitolo 5

# Conclusioni

Analizzati i dati raccolti nella sezione 4.4, possiamo sin da subito verificare come il limite dei 30 ms sia difficilmente raggiungibile ad oggi per le applicazioni che operano nel browser, nonostante i vantaggi dell'utilizzo di schede audio esterne. Avendo però utilizzato delle strategie di test comuni nelle prove effettuate, è possibile adesso pensare a un confronto tra le applicazioni e le architetture presentate.

Inizialmente possiamo considerare i risultati delle singole architettura e tecnologie adottate. Come è visibile dalle misurazioni esposte nel capitolo 4, date delle performance simili tra le architetture che usano `MessageChannel` e quelle che usano `SharedArrayBuffer` durante le analisi preliminari, si vede una consistente migliore scalabilità del sistema andando a connettere più peer alla stessa stanza. Questo sembra essere principalmente dovuto a una scarsa scalabilità del `MessageChannel`, fattore principalmente visibile con i dati misurati su Firefox. Si ha infatti un peggioramento della qualità della trasmissione, visibile dall'innalzamento del numero di perdite e dalle alte percentuali di silenzio nella registrazione complessiva. A questa conclusione si giunge anche tramite la comparazione dell'architettura `SharedArrayBuffer`, quindi impiegando un solo `MessageChannel` per la trasmissione, con l'architettura `Atomics.waitAsync`, senza `MessageChannel`: sebbene le differenti latenze dei due sistemi, si vede una simile stabilità delle latenze e delle percentuali di pacchetti scartati. Ciò porta a pensare che sia proprio il `MessageChannel` l'elemento che si comporta da collo di bottiglia nella scalabilità dell'applicazione.

Successivamente possiamo considerare con un confronto tra le nuove architetture presentate e l'architettura di Sacchetto [6]. Possiamo notare in Firefox la diminuzione delle perdite che comporta l'adozione di un minore numero di thread e di un solo buffer circolare condiviso all'interno dell'applicazione. Se si osserva

poi solamente la soluzione con architettura `SharedArrayBuffer`, è evidente come questa tecnologia conferisca all'applicazione una maggiore stabilità e scalabilità in contesti multipeer. In Google Chrome si ha un effetto quasi opposto: viene mantenuta la stabilità delle latenza all'aumentare del numero di peer, ma si hanno più alti valori delle latenze assolute. Siccome questo comportamento avviene anche con solo due peer connessi, una possibile causa di ciò potrebbe essere una scarsa prestazione nell'impiego dei metodi `Atomics` e della sincronizzazione tra thread contro una migliore efficienza e scalabilità dei `MessageChannel` e dei *WebWorker*. Inoltre, l'utilizzo dall'architettura `Atomics.waitAsync` per rimuovere completamente l'impiego dei `MessageChannel` nell'applicazione sembra avere solamente un lieve costo aggiuntivo, ma non reali vantaggi rispetto all'architettura `SharedArrayBuffer`.

Per ultimo, diamo spazio alle misurazione rilevate con le applicazioni *Google Meet* e *Jisti*. Gli alti ritardi ottenuti confermano quanto esposto da Sacchetto in [6]: l'utilizzo dei moduli `MediaStream` connessi direttamente tra loro tramite API *WebRTC* comportano peggiori performance rispetto l'utilizzo a più basso livello del `RTCDataChannel`. Le latenze misurate con queste applicazioni si possono definire discretamente sufficienti per un contesto di videochiamate, ma si discostano di molto da quelle ricercate per poter svolgere *Networked music performance* in un contesto in diretta. Inoltre, l'utilizzo di funzionalità come *autogain* o riduzione del rumore di fondo possono essere elementi che vanno in contrasto con quanto ricercato dall'artista durante la performance, oltre che essere fonte di un carico supplementare per il sistema.

Per quanto riguarda le latenze in termini assoluti, non c'è stato miglioramento con l'introduzione delle nuove tecnologie. Per i test con Google Chrome, anzi, sembra che l'impiego di queste nuove tecnologie comporti dei ritardi maggiori per l'applicazione. Per i valori che sono stati presentati in questa tesi si vuole far presente che i test svolti da Sacchetto sono stati effettuati su un solo dispositivo. Le configurazioni di test utilizzate avevano l'obiettivo di avvicinarsi quanto più possibile a una situazione reale. Introdurre ulteriori dispositivi intermedi (oltre a quelli utilizzati) come router e switch, sistemi di bufferizzazione o frammentazione dei pacchetti, sistemi di traffic shaping, così come la connessione via WiFi, non sono stati contesti studiati in questa tesi, ma che, come è facile immaginare, inficiano notevolmente in negativo sui ritardi misurabili tra due client.

Non sono stati esposti grafici e dati precisi sul jitter, in quanto il suo tracciamento all'interno dell'applicazione è molto complesso, sia tramite log che tramite memorizzazione in vettori interni ed esposizione in un secondo momento rispetto allo svolgimento del test. Entrambe queste operazioni, visto l'elevato numero di pacchetti e la rapidità a cui devono essere elaborati, comportavano un'aggiunta

dei ritardi e del carico a cui l'applicazione era sottoposta. Questo sforzo maggiore richiesto produceva rallentamenti interni col conseguente effetto di far arrivare più in ritardo i pacchetti alla fase di inserimento in coda, quindi con maggiore probabilità che venissero scartati perché troppo in ritardo. Inoltre, fasi come la connessione e la disconnessione di un peer interessano pesantemente il thread principale, per cui in questi frangenti i dati misurati presentavano comportamenti molto imprevedibili.

Concludendo le considerazioni sui risultati raccolti, si intende sottolineare come sia importante che i test vengano svolti su computer differenti perché misurare le performance di applicazioni onerose in termini di risorse, come in questo caso, con molte istanze su un solo computer porta a dei risultati che indicano principalmente le prestazioni del browser per allocare di risorse, gestire i thread etc., piuttosto che le prestazioni effettive dell'applicazione. Questa procedura di test è a oggi difficilmente automatizzabile, il che rende molto dispendiosa in termini temporali la possibilità di raccogliere un alto numero di dati. Inoltre, non è possibile considerare purtroppo questi dati per uno studio approfondito sul funzionamento delle applicazioni sui diversi computer, sistemi operativi e browser. Data l'eterogeneità dei dispositivi utilizzati, è complesso ricreare una situazione analoga per espandere gli esperimenti ad altri casi di studio. Il rapido e continuo aggiornamento dei browser peggiora ulteriormente la possibilità di comparazione dei risultati.

# Capitolo 6

# Lavori futuri

## 6.1 Utilizzo dei lock invece degli Atomics

Una implementazione dei lock è stata fatta da Lars T. Hansen con l'utilizzo degli `Atomics`[1]. Al momento gli `Atomics` operano solamente su strutture dati `Int*Array` oppure `Float32Array`. Tramite questa implementazione è possibile utilizzare le operazione di lock anche su `Float64Array` e `BigInt64Array`. Inoltre, sono state implementate delle operazioni con funzionalità di mutex e guard-lock. Infine, con piccole variazioni al codice sarebbe possibile fare in modo che le operazioni sull'intera struttura dati acquisiscano e rilascino il lock una volta sola.

## 6.2 Buffer circolare dimensionato dinamicamente a runtime

Il passo ulteriore che potrebbe permettere di sfruttare al massimo la connessione di rete, riducendo effettivamente le latenze, consiste nel calcolo in tempo reale della distanza `packet offset` tra gli indici di inserimento in coda `write index` e di estrazione `read index`. In questo modo si agisce non sul ritardo della rete che, per quanto possa essere modellato, rimane un valore istantaneo incognito, ma sul ritardo imposto dal buffer, che invece è noto e controllabile dal programmatore. Idee per un buffer dimensionato dinamicamente si possono trovare in [3].

[1] https://github.com/lars-t-hansen/parlib-simple

## 6.3 HTTP v.3

Una nuova versione di HTTP sta per essere diventare uno standard[2], per il momento ne è stata pubblicata solamente una bozza. Un grande cambiamento proposto è la possibilità di utilizzare HTTP su UDP. In questo modo sarebbe possibile ricreare un'architettura client-server che non sia suscettibile ai problemi di NAT, che invece interferiscono con architetture peer-to-peer.

## 6.4 Oggetti DOM accessibili dai WebWorker

Venendo a contatto con alcuni programmatori di browser nell'ambito della conferenza *WAC2021*, è stato dichiarato come Firefox e Google Chrome si stiano muovendo per abilitare il controllo degli oggetti del DOM anche dal codice all'interno ai *WebWorker*. Il principale vantaggio applicato a questo progetto sarebbe una gestione diretta dei canali dati `RTCDataChannel` da parte degli *AudioWorklet*, evitando così le comunicazioni tra i thread audio e il thread principale. Questi cambiamenti comportano molti dubbi sulle caratteristiche di sicurezza delle applicazioni web che i browser devono considerare. L'accesso a troppe risorse da parte di thread audio ad alta priorità potrebbe essere sfruttata per profilazione dei dispositivi utilizzati oppure per operazioni che necessitano di importanti processi di calcolo da parte di siti malevoli.

[2]https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34

# Bibliografia


[1] Sekhar Chandra, Senthil Kumar, and Bala Prasad. Audio mixer for multi-party conferencing in voip. pages 1 – 6, 01 2010.

[2] Cullen Jennings (Cisco), Henrik Boström (Google), and Jan-Ivar Bruaroey (Mozilla). Webrtc 1.0: Real-time communication between browsers. `https://www.w3.org/TR/webrtc/`.

[3] TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU. *Multimedia Quality of Service and performance – Generic and user-related aspects.* 2012.

[4] Tim Thompson (Communications Technology Laboratory) Jesse Frey (Communications Technology Laboratory), Jaden Pieper (Communications Technology Laboratory). Mission critical voice qoe mouth-to-ear latency measurement methods. `https://doi.org/10.6028/NIST.IR.8206`.

[5] Cristina Rottondi, Chris Chafe, Claudio Allocchio, and Augusto Sarti. An overview on networked music performance technologies. *IEEE Access*, 4:8823–8843, 2016.

[6] Chris Cafe (CCRMA Stanford University) Sacchetto Matteo (Politecnico di Torino), Servetti Antonio (Politecnico di Torino). Jacktrip-webrtc networked music performance with web technologies. `https://github.com/jacktrip-webrtc/jacktrip-webrtc`.