POLITECNICO DI TORINO

DIPARTIMENTO DI AUTOMATICA E INFORMATICA

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea Magistrale

RENDERING AUDIO REMOTO

Analisi della latenza in ambiente web

Relatore
prof. Antonio Servetti

Candidato
Stefano Imperiale
Matricola: 241972

Anno accademico 2018-19

Sommario

L'obiettivo principale di questo lavoro di tesi è lo studio e l'analisi della realizzazione di un ambiente di rendering audio condiviso.

La caratteristica principale di quest'ambiente è lo spostamento del lavoro di rendering lato server e la conseguente semplificazione dei client. Questo tipo di concetto si sta sempre di più diffondendo nel mondo web basti pensare ai recenti servizi di cloud gaming offerti da aziende come Sony, Microsoft o recentemente dalla stessa Google che permettono di giocare a titoli che richiedono una grande potenza di calcolo ovunque e da qualunque tipo di dispositivo, sia esso un computer, una smart TV o uno smartphone. Per quanto riguarda il rendering audio ad oggi è possibile trovare e utilizzare molte soluzioni in rete che permettono di lavorare con tracce audio o crearne delle proprie, ma queste utilizzano la macchina del client per eseguire il lavoro di rendering.

Ciò si verifica, in quanto l'elaborazione dell'audio utilizzando un browser è dato dall'utilizzo di particolari librerie, le WebAudioAPI, disponibili oggi sulla maggior parte dei browser moderni; queste API però, in quanto librerie del browser, possono essere utilizzate solo sulla macchina client (trattandosi di codice Javascript), aggiungendo quindi carico computazionale a quest'ultimo e rendendo difficoltosa la realizzazione di progetti particolarmente complessi, sopratutto se il client ha a disposizione risorse hardware limitate. Il presente lavoro di testi si colloca quindi in uno scenario simile e cerca di studiare una soluzione ai problemi precedentemente descritti. Le motivazioni dietro questo lavoro possono essere sintetizzate nei seguenti punti:

1. l'interesse sempre maggiore di molte aziende verso il cloud computing e lo streaming real-time

2. la libertà di utilizzare un'applicazione in qualunque momento e da qualsiasi dispositivo "smart" e relativo risparmio dell'utente che non è legato ad acquistare hardware dedicato

3. la riservatezza per chi mette a disposizione il servizio, in quanto non è tenuto a concedere e a diffondere agli utenti finali i propri codici sorgenti

Lo sviluppo dell'ambiente di lavoro che verrà descritto è stato preceduto da un'analisi di alcune pubblicazioni che trattano il rendering audio remoto seguita da una ricerca delle possibili tecnologie da utilizzare. La realizzazione è stata poi finalizzata implementando in parallelo una soluzione client e server:

- Il server presenta due diverse soluzioni: Nw.js, una soluzione ibrida che permette di utilizzare Google Chrome insieme ai moduli Node.js, e Chrome headless, un'implementazione di Chrome che può essere eseguita senza la presenza di un motore grafico. Quest'ultima soluzione è stata poi realizzata concretamente utilizzando due librerie di controllo ad alto livello: Selenium e Puppeteer.

- Il client invece è realizzato come una semplice pagina HTML e codice Javascript il cui compito principale è quello di instaurare la connessione con il server, inviare i vari comandi dell'utente e cambiare i parametri della connessione.

Ciò che permette di collegare tra loro le componenti descritte è l'utilizzo di WebRTC, un progetto open source e libero che aggiunge le funzionalità di Real-Time Communications a browser e dispositivi mobili attraverso l'uso di semplici API. WebRTC necessità però di un canale di signaling per poter scambiare i pacchetti necessari e instaurare una connessione peer-to-peer. Per questo motivo è stato aggiunta la funzionalità di websocket al server, come si vedrà successivamente. Si è quindi proceduti alla realizzazione di suite di test da eseguire per la raccolta delle misurazioni delle varie latenze sulle diverse tratte della comunicazione per poter quindi raccogliere dati e stilare poi dei grafici. Questo ha permesso di analizzare su quali fattori è possibile o no andare ad agire per influenzare in meglio o in peggio la comunicazione e cercare di trovare la soluzione che offra la minor latenza da quando il client invia un comando a quando effettivamente riceve la traccia audio. Alla luce di quanto detto il lavoro di tesi sarà suddiviso in tre parti principali:

- La prima parte si occuperà, dopo un'analisi dell'attuale stato dell'arte, di presentare ed esporre nel dettaglio al lettore le varie tecnologie utilizzate.

- La seconda parte invece presenterà il lavoro svolto per lo sviluppo dell'architettura generale andando poi a descrivere nel dettaglio i vari componenti realizzati, in particolare il server, sia in ambito locale che remoto, e il client
- L'ultima parte, infine, illustrerà le procedure dei test svolti al termine della realizzazione della soluzione e si occuperà di presentare i dati raccolti in grafici, descrivendo quindi i risultati ottenuti e le relative conclusioni

Ringraziamenti

"Costruire il futuro e tenere vivo il passato sono la stessa cosa."

- Solid Snake, *Metal Gear Solid*

Il mio primo ringraziamento va al professore Antonio Servetti senza il quale questo lavoro non sarebbe stato possibile, dimostrandosi sempre disponibile e pronto ad aiutarmi a risolvere i problemi che si manifestavano durante tutto lo sviluppo.
Ringrazio mia madre e mia sorella Chiara per i loro sforzi e sacrifici che mi hanno permesso di intraprendere questo percorso ed essere riuscito a concluderlo nonostante tutte le prove affrontate insieme.
Infine ringrazio di cuore Silvia, la mia forza e la mia vita, che mi ha sostenuto in ogni situazione ed è sempre stata al mio fianco aiutandomi ad andare avanti anche quando questo sembrava impossibile.

Indice

Elenco delle figure 9

1 Introduzione 11
1.1 Stato dell'arte . . . 11
1.2 Workflow . . . 13

2 Le tecnologie utilizzate 15
2.1 Le WebAudioAPI . . . 15
2.2 WebRTC e WebSocket . . . 16
2.3 NW.js e Chrome Headless . . . 22
2.3.1 Puppeteer . . . 23
2.3.2 Selenium . . . 25

I Sviluppo del progetto **27**

3 L'architettura generale 29

4 Il server 31
4.1 L'architettura del server . . . 31
4.2 Server Web . . . 31
4.3 Caricamento delle tracce . . . 33
4.4 Il server di signaling . . . 34
4.4.1 Signaling in Chrome Headless . . . 34
4.4.2 Signaling in NW.js . . . 36
4.5 Il peer WebRTC . . . 38
4.5.1 Il DataChannel . . . 39
4.6 Gestione dei comandi e rendering audio . . . 41
4.7 L'ambiente remoto . . . 44
4.7.1 Il problema della sincronizzazione dei clock . . . 47

5 Il client 51
5.1 L'architettura del client 51
5.2 La connessione 53
5.2.1 Instaurazione della connessione 53
5.2.2 La gestione dei messaggi 55
5.3 La gestione di WebRTC 57
5.3.1 La gestione del DataChannel 60
5.3.2 Invio dei comandi 61
5.4 La scelta dei codec 62
5.4.1 Le opzioni di OPUS 64
5.5 Grafici e misurazioni 66

II Risultati e considerazioni finali **69**

6 Test e risultati 71
6.1 I parametri studiati 71
6.2 Struttura dei test 74
6.3 Risultati ottenuti 77
6.3.1 I grafici sulla rete 77
6.3.2 I grafici dell'audio 80

7 Considerazioni finali 87
7.1 La sicurezza nella comunicazione 88

Bibliografia 91

Elenco delle figure

1.1 Schema concettuale del progetto DSPNode 12
1.2 Schema dei vari contributi al ritardo OOSE, la tonalità più scura indica i contributi maggiori 14
2.1 Esempio di collegamento di più nodi audio; ognuno di esso può essere configurato in modo da utilizzare un preciso numero di canali 16
2.2 Struttura dell'oggetto RTCPeerConnection, che costituisce l'elemento fondamentale per instaurare una connessione WebRTC 18
2.3 Esempio minimale di implementazione di un server WebSocket per signaling 20
2.4 Schema completo di un architettura WebRTC applicata a reti reali 21
2.5 La differenza delle architetture di un'applicazione web contro un'applicazione NW.js 24
3.1 Schema generale dell'architettura sviluppata 30
4.1 Differenze architetturale delle due soluzioni sviluppate lato server 32
4.2 Schema dello scambio dei messaggi su WebSocket nella soluzione Headless 36
4.3 Rappresentazione del componente GainNode delle Web Audio API che permette un aumento del guadagno del segnale di ingresso 43
4.4 Rappresentazione del componente ChannelMerger delle Web Audio API, il cui compito è quello di unire più canali in un unica uscita audio 44
4.5 Schema logico della gestione dei messaggi del DataChannel sul server 45
4.6 Contenuto del file docker-compose.yml contentente la configurazione del container usato per eseguire NW.js 47
4.7 Esempio di calcolo dell'offset del clock su due sistemi sulla rete 48

5.1 Schema dei componenti del client . . . 51
5.2 La pagina del client . . . 54
5.3 Struttura di un array di server STUN-TURN, solitamente per i server TURN è necessario anche fornire delle credenziali di autenticazione . . . 56
5.4 Schema dei messaggi scambiati durante l'instaurazione della connessione WebRTC tra client e server . . . 58
5.5 Schema degli eventi dell'oggetto RTCPeerConnection del client 60
5.6 I comandi inviati dal client al server tramite il DataChannel, sottoforma di stringhe JSON . . . 62
5.7 Struttura di un pacchetto SDP . . . 63
5.8 Rappresentazione dell'oggetto analyser . . . 68
6.1 Confronto in termini di bitrate/qualità tra OPUS e altri codec 74
6.2 Confronto delle latenze di rete in ambiente locale in caso di utilizzo di server STUN (sopra) o di soli server TURN (sotto) 78
6.3 Confronto delle latenze di rete con server remoto in caso di utilizzo di server STUN (sopra) o di soli server TURN (sotto) 79
6.4 Analisi dei pacchetti inviati dal server al client in caso di connessione diretta (sopra) e connessione dietro un container docker (sotto) . . . 80
6.5 Grafici latenze audio in ambiente locale su ISAC, G.722, PCMU e PCMA. Nelle latenze totali dell'audio è possibile vedere anche la differenza in termini di tempo tra i due metodi di sostituzione delle tracce . . . 82
6.6 Grafici latenze audio in ambiente remoto su ISAC, G.722, PCMU e PCMA. Come nel caso locale, nel riquadro della latenza totale dell'audio è possibile notare la differenza di latenze dei due metodi di cambio traccia . . . 83
6.7 Grafici latenze audio in ambiente locale per codec OPUS. Ogni riquadro confronta le latenze nel caso di attivazione o meno di determinati parametri del codec . . . 85
6.8 Grafici latenze audio in ambiente remoto per codec OPUS . . . 86
7.1 Esempio di sviluppo sicuro di un applicazione WebRTC in base alle indicazione dell'IETF . . . 89

Capitolo 1

Introduzione

1.1 Stato dell'arte

Attualmente non sembrano esserci molte ricerche o progetti focalizzati allo sviluppo di un ambiente rendering audio in ambiente web. La gran parte della applicazioni web presenti in rete che offrono questo servizio eseguono sempre il rendering audio utilizzando le API offerte dal browser del client, lasciando quindi a quest'ultimo il compito di elaborare e manipolare le tracce audio.
Un progetto ancora in fase di sviluppo che riguarda temi simili a quelli trattati in questo lavoro di tesi è **DSPnode**.[1] Il progetto, simile in molti aspetti alla soluzione che verrà esposta in questo documento, si sviluppa in una parte client e una server; la componente client è pensata per essere il più semplice possibile in modo da non richiedere molte risorse alla macchina su cui è eseguita mentre il server si occupa del rendering audio tramite Web Audio API e di inviare poi le tracce attraverso un collegamento WebRTC; in figura 1.1 è rappresentata l'architettura del progetto.
Ciò che viene aggiunto da questo lavoro è lo studio e il confronto dei tempi di latenza presi in base a varie configurazioni (siano esse riguardanti la rete, i codec o altro), aspetto che il progetto dspNode non approfondisce ma si limita a proporre una soluzione con parametri fissi scelti dallo sviluppatore senza poter, in qualche modo, analizzare come potrebbe influire, in termini di latenza, un'opzione rispetto ad un'altra.
Nonostante quindi il lavoro che verrà esposto avrà molti punti in comune con

[1] https://github.com/dodds-cc/dspNode

DSPnode, soprattutto per quanto riguarda la scelta delle tecnologie usate e l'architettura, esso si differenzierà principalmente per quelle che saranno poi le finalità: la ricerca di quelle configurazioni e opzioni che permettono di ottenere i minori tempi di latenza possibili.

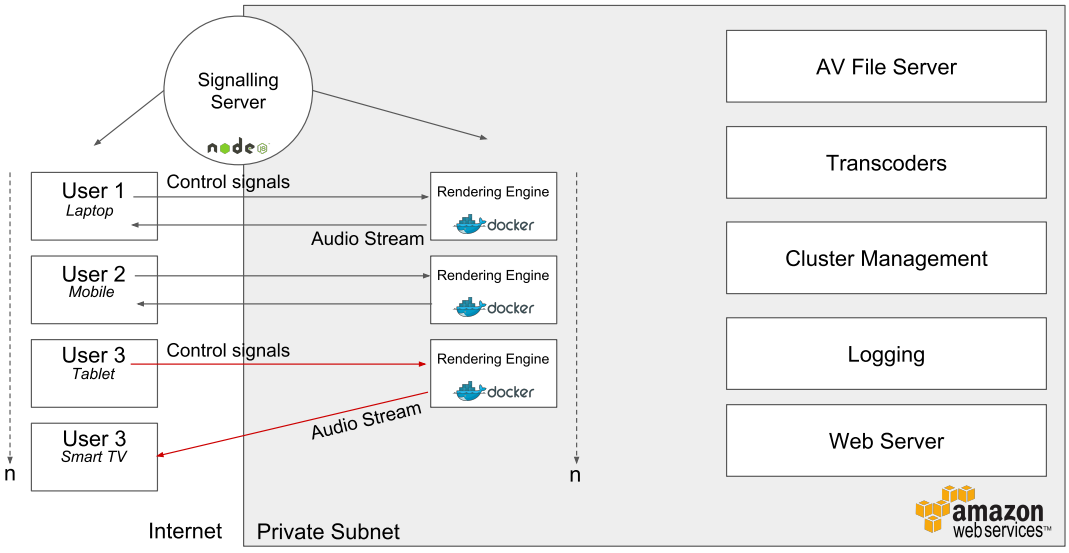

Figura 1.1: Schema concettuale del progetto DSPNode

Sempre in ambito di rendering remoto, un altro studio molto interessante riguarda l' **NPM** o **Networked Music Performance**[9], una modalità di interazione musicale che pone particolare importanza sulla latenza di rete, a cui vengono imposti severi vincoli. La finalità di questa tecnologia è permettere a musicisti situati in diverse aree geografiche di interagire e suonare insieme attraverso la rete. Il documento raccoglie gli studi psico-percettivi condotti negli ultimi anni per individuare la tolleranza sulla latenza per progetti real-time in ambito musicae e descrive quali soluzioni hardware-software possono essere utilizzate per applicazioni di questo genere.
Di particolare interesse è l'analisi effettuata su ciò che contribuisce ad introdurre ritardi nella comunicazione end-to-end. I ritardi nella comunicazione derivano principalemente da:

1. uso stesso di un sistema che deve inviare informazioni da un trasmettitore ad un ricevitore in quanto entrambi i fronti introducono latenze in fase di acquisizione, esecuzione, processing e inpacchettamento/spacchettamento dei dati.

2. il tempo necessario affinchè il segnale attraversi il mezzo fisico
3. il ritardo introdotto dai nodi intermedi della rete per processare i pacchetti lungo l'intero percorso
4. il buffering di playout che potrebbe essere richiesto per compensare l'effetto del jitter in modo tale da garantire una qualità audio sufficiente

In figura 1.2 sono invece rappresentati più dettagliatamente tutti gli elementi che introducono in un modo o nell'altro una latenza nell'Over-all One-way Source-to-Ear, o OOSE, percepito da un utente del framework NPM, cioè il ritardo totale dalla generazione del suono su un estremo della rete a quando effettivamente l'utente riesce a percepirlo a livello psicologico.

1.2 Workflow

Questo lavoro di tesi si è sviluppato seguendo le seguenti fasi:

1. Una fase di studio e ricerca delle possibili tecnologie da utilizzare; una volta capito quale sarebbero state le tecnologie utili allo scopo prefisso ha fatto seguito una fase di approfondimento su ognuna di esse in modo da poter comprendere e padroneggiare i metodi messi a disposizione da ciascuna tecnologia.
2. La fase di sviluppo e di implementazione della soluzione costituisce sicuramente la parte più importante dell'intero progetto. Data la grande correlazione tra la componente client e quella server, l'implementazione di queste due componenti è avvenuta pressoché in parallelo. Nella prima parte di questa fase ci si è focalizzati sul creare un'architettura seppur basilare ma funzionante, in cui client e server riuscivano ad instaurare una connessione, scambiarsi messaggi e ricevere le prime tracce audio. Successivamente sono state man mano aggiunte e implementate tutte le varie opzioni che hanno poi permesso di ampliare lo studio finale.
3. Concluso lo sviluppo su ambiente locale si è poi passati al rilascio dell'applicativo su una macchina server reale, in modo tale da ottenere delle misure di latenza che si avvicinassero a quelle che si potrebbero avere con una reale applicazione web.
4. Si è passati quindi alla fase di test in cui venivano automatizzate le test suite che sarebbero state poi usate per ricavare le misurazioni di diversi casi d'uso, sia in ambiente locale che remoto.

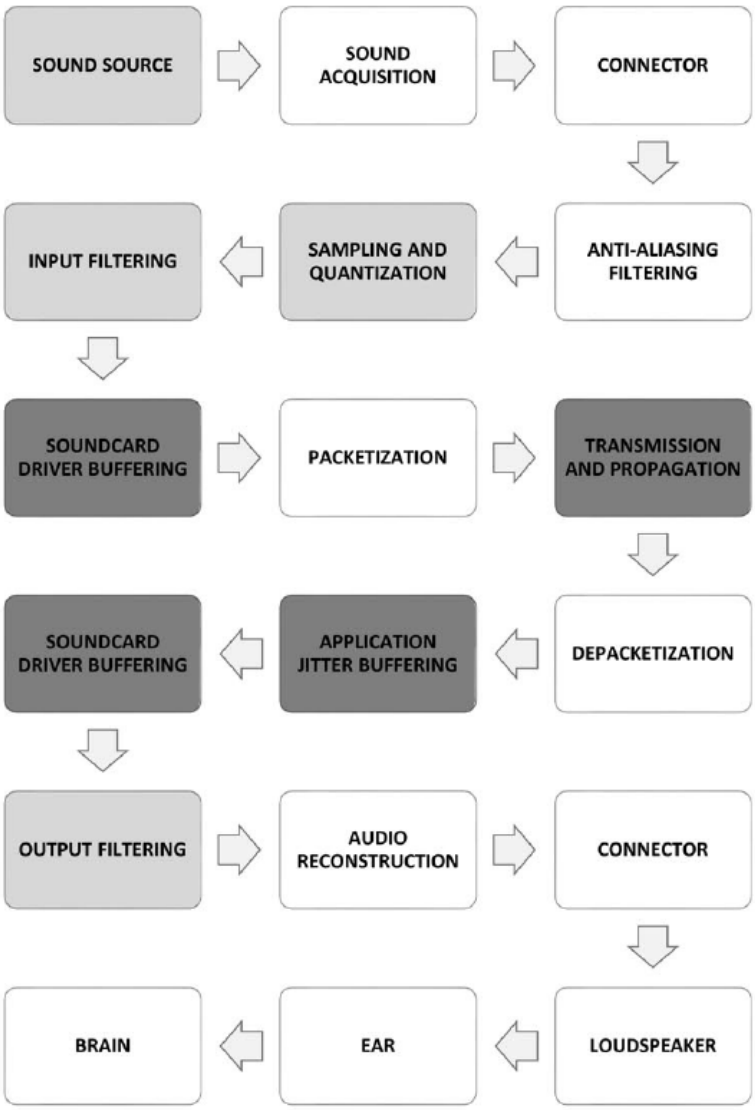

Figura 1.2: Schema dei vari contributi al ritardo OOSE, la tonalità più scura indica i contributi maggiori

5. Il lavoro si è poi concluso con la fase di raccolta dei dati in file e successiva rappresentazione con dei grafici, che permettono di riassumere e confrontare efficacemente i risultati ottenuti.

Capitolo 2

Le tecnologie utilizzate

2.1 Le WebAudioAPI

Le Web Audio API permettono di effettuare il rendering audio in ambiente web. Esse operano in un contesto audio (offerto dal browser attraverso l'oggetto AudioContext) in cui vige il principio di routing modulare.
Le operazioni audio basilari sono effettuate in nodi audio, collegati tra loro in modo da formare un grafico. È possibile collegare tra loro molte varietà di nodi audio per creare un sistema complesso e dinamico ricco di effetti.
Questi nodi sono collegati tra loro attraverso i relativi collegamenti di input e output, formando una catena che parte da una o più sorgenti audio e termina con una destinazione (generalmente la destinazione di base è costituita dagli speaker). Un esempio di un semplice contesto audio può essere costituito dai seguenti elementi: una sorgente, quale un oscilloscopio, una traccia audio o uno stream; uno o più nodi che aggiungano un effetto a questa sorgente come riverbero o eco, oppure applicando un filtro; una destinazione per l'audio, come gli speaker del computer o uno stream di rete.
I nodi inoltre possono essere configurati per usare un determinato numero di canali, come il canale stereo o il canale 5.1, costituito dal canale destro, sinistro, centrale, subwoofer, posteriore sinistro e posteriore destro (figura 2.1). I canali vengono rappresentati con una notazione numerica puntata (ad esempio per il canale stereo la notazione è 2.0) in cui il primo numero rappresenta il numero di canali che include il segnale, il secondo il numero di canali adibiti alle basse frequenze (o LFE).
Ci sono vari modi per generare un suono da utilizzare con le Web Audio API: attraverso JavaScript creando un audio node (come un oscillatore); creando

dei dati audio PCM (attraverso i metodi messi a disposizione dall'oggetto AudioContext); tramite un elemento media HTML5 (tag <audio> o <video>); da uno stream WebRTC.[1]

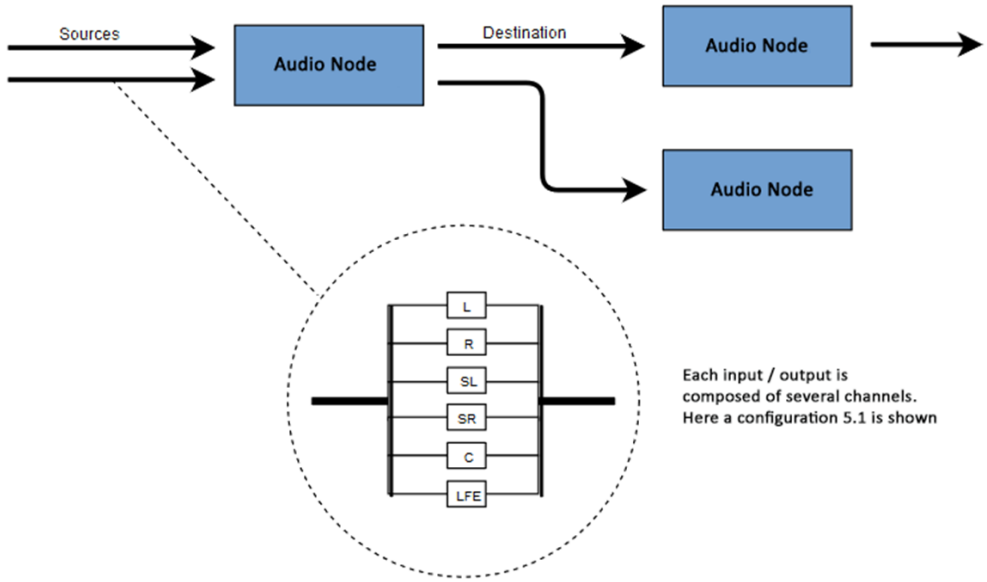

Figura 2.1: Esempio di collegamento di più nodi audio; ognuno di esso può essere configurato in modo da utilizzare un preciso numero di canali

2.2 WebRTC e WebSocket

La tecnologia che è alla base della soluzione sviluppata e che permette il collegamento tra client e server è WebRTC.
WebRTC è un progetto libero e open source che offre a browser e a dispositivi mobili la possibilità di effettuare collegamenti real-time (o RTC, Real-Time Communications) attraverso l'uso di semplici API. L'obiettivo di questo progetto, come riportato nel sito ufficiale https://webrtc.org/, *è di permettere a ricche applicazioni RTC di alta qualità di essere sviluppate su browsers, dispositivi mobili e dispositivi IoT, e permettere a tutte loro di comunicare attraverso un comune set di protocolli.*

[1] https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Basic_concepts_behind_Web_Audio_API

L'oggetto che si occupa di gestire la connessione e lo streaming dei pacchetti è **RTCPeerConnection**. Questo componente fa sì che lo scambio dei dati tra i peer sia sempre stabile ed efficente. In figura 2.2 sono riportati tutti i componenti di questo oggetto; è possibile notare come complesso sia questo oggetto ma allo sviluppatore tutto ciò è trasparente e a quest'ultimo sono esposti solo i metodi di cui ha bisogno per poter sviluppare la sua applicazione.
I codec e i protocolli usati da WebRTC si occupano di gestire tutti i meccanismi necessari a rendere possibile la comunicazione real-time, anche in presenza di una rete non affidabile, come ad esempio:

- gestione della perdita dei pacchetti
- cancellazione dell'eco
- adattamento della larghezza di banda
- jitter buffering dinamico
- controllo del guadagno automatico
- riduzione e soppressione del rumore[2]

WebRTC però non è in grado di instaurare una connessione facendo affidamento unicamente ai suoi meccanismi interni; necessita infatti di un sistema esterno per coordinare la comunicazione e inviare messaggi di controllo, sistema noto come signaling. Questo meccanismo non è parte delle API di WebRTC; lo sviluppatore infatti è libero di utilizzare il sistema di signaling che più preferisce, come ad esempio SIP, WebSocket e qualunque meccanismo di comunicazione full-duplex.
Più specificatamente il sistema di signaling è necessario per poter scambiare tre tipo di informazioni[3]:

1. messaggi sul controllo di sessione: permettono di instaurare o chiudere una connessione e segnalare gli errori
2. configurazione della rete: tutti i messaggi realativi alla topologia della rete dei due peer, gli indirizzi IP, le porte...

[2] https://www.html5rocks.com/en/tutorials/webrtc/basics/#toc-rtcpeerconnection

[3] https://www.html5rocks.com/en/tutorials/webrtc/basics#toc-signaling

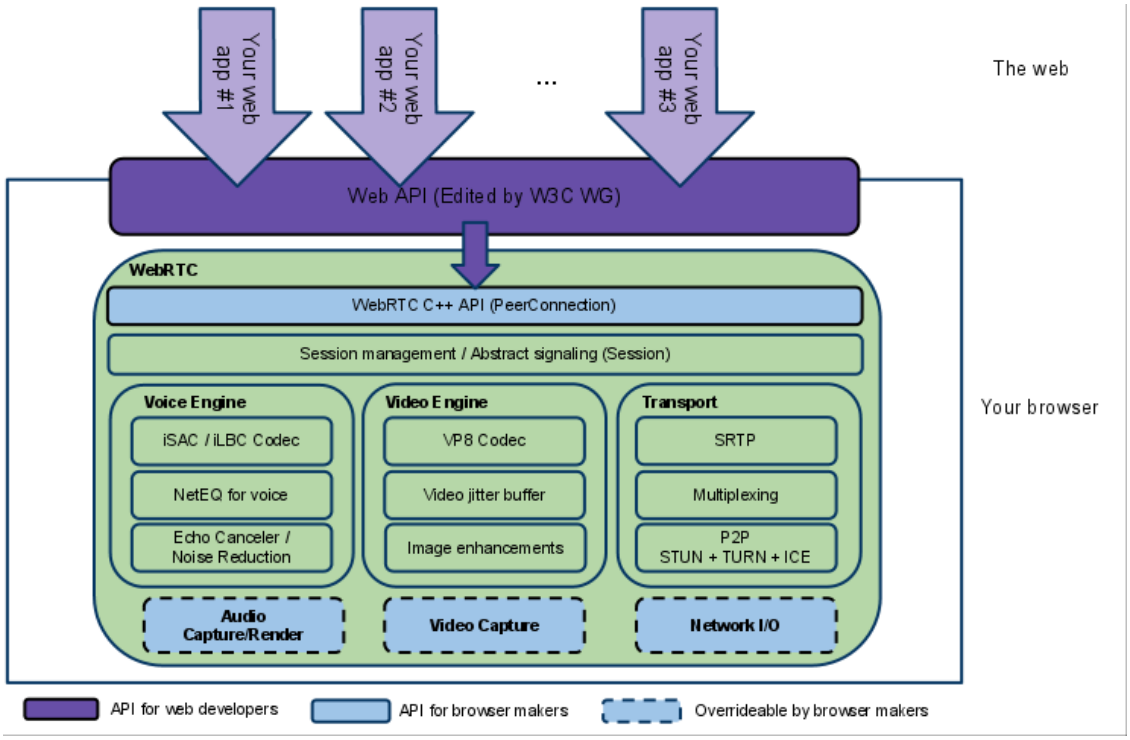

Figura 2.2: Struttura dell'oggetto RTCPeerConnection, che costituisce l'elemento fondamentale per instaurare una connessione WebRTC

Fonte: https://webrtc.org/reference/architecture

3. caratteristiche dei media: i codec da utilizzare, la risoluzione supportata dal browser...

Tutti questi messaggi devono essere stati scambiati prima di poter utilizzare la connessione WebRTC vera e propria.
Tra i vari meccanismi che è possibile utilizzare per il signaling è stato deciso di utilizzare per questo progetto la tecnologia WebSocket, una tecnologia web che permette una comunicazione full-duplex basata su eventi sfruttando una singola connessione TCP. È standardizzato dal RFC 6455.[1]
I motivi di questa scelta possono essere principalmente tre:

1. se un browser supporta WebRTC allora supporta anche WebSocket

2. WebSocket è legato ad un server web il che lo rende in grado di operare sulla porta 80 (nella sua versione non sicura) riuscendo quindi a superare la gran parte dei firewall

3. è di semplice implementazione sia su un browser, grazie alle librerie già presenti nel DOM, che su un server (le librerie WebSocket sono disponibili per una moltitudine di linguaggi di programmazione)

In figura 2.3 è possibile vedere una semplice implementazione di un server WebSocket con funzionalità di signaling per WebRTC.
Una volta inviati tutti i messaggi di configurazione, WebRTC non utilizza più il server di signaling ma cerca di stabilire una connessione diretta tra i due peer. Ciò nel mondo reale non è quasi sempre possibile in quanto i due peer non hanno un indirizzo IP unico pubblico che possono scambiarsi per la comunicazione diretta, in quanto solitamente essi si trovano dietro un NAT, a cui per motivi di sicurezza vi sono collocati solitamente dei firewall, degli antivirus che bloccano determinate porte e protocolli e dei proxy. Per superare le complessità di una rete reale WebRTC utilizza un framework denominato **ICE**(Interactive Connectivity Establishment), il cui compito è trovare il percorso migliore tra i peer.
In caso ad esempio di NAT asimmetrico il framework ICE si affida ai server STUN (Session Traversal Utilities for NAT), i quali permettono ai client di scoprire il loro indirizzo IP pubblico e il tipo di NAT dietro cui sono collocati. Il protocollo STUN è definito dal RFC 3489[8].
Nella maggior parte dei casi l'uso dei server STUN si limita unicamente alla fase di instaurazione della connessione; una volta instaurata i media sono scambiati direttamente tra i peer. Se invece i server STUN non riescono a stabilire una connessione, il framework ICE si affida ai server TURN (Traversal Using Relay NAT), i quali rappresentano un estensione dei server STUN e solitamente si usano in presenza di NAT simmetrici.
A differenza dei server STUN, i server TURN persistono anche a connessione instaurata in quanto si comportano da "ripetitori" per i media scambiati.[4]
Per quanto possibile si cerca di limitare l'utilizzo dei server TURN ma non sempre questo è possibile; per questo ogni applicazione che utilizza WebRTC dovrebbe supportare entrambe le tipologie di server per riuscire a instaurare sempre una connessione tra i peer. Uno schema della tecnologia WebRTC applicata a reti reali è raffigurato in figura 2.4.

[4] https://www.avaya.com/blogs/archives/2014/08/understanding-webrtc-media-connections-ic
html

```
// handles JSON.stringify/parse
const signaling = new SignalingChannel();
const constraints = {audio: true, video: true};
const configuration = {iceServers: [{urls:
'stuns:stun.example.org'}]};
const pc = new RTCPeerConnection(configuration);

// send any ice candidates to the other peer
pc.onicecandidate = ({candidate}) => signaling.send({candidate});

// let the "negotiationneeded" event trigger offer generation
pc.onnegotiationneeded = async () => {
  try {
    await pc.setLocalDescription(await pc.createOffer());
    // send the offer to the other peer
    signaling.send({desc: pc.localDescription});
  } catch (err) {
    console.error(err);
  }
};

// once remote track media arrives, show it in remote video element
pc.ontrack = (event) => {
  // don't set srcObject again if it is already set.
  if (remoteView.srcObject) return;
  remoteView.srcObject = event.streams[0];
};

// call start() to initiate
async function start() {
  try {
    // get local stream, show it in self-view and add it to be sent
    const stream =
      await navigator.mediaDevices.getUserMedia(constraints);
    stream.getTracks().forEach((track) =>
      pc.addTrack(track, stream));
    selfView.srcObject = stream;
  } catch (err) {
    console.error(err);
  }
}
signaling.onmessage = async ({desc, candidate}) => {
  try {
    if (desc) {
      // if we get an offer, we need to reply with an answer
      if (desc.type === 'offer') {
        await pc.setRemoteDescription(desc);
        const stream =
          await navigator.mediaDevices.getUserMedia(constraints);
        stream.getTracks().forEach((track) =>
          pc.addTrack(track, stream));
        await pc.setLocalDescription(await pc.createAnswer());
        signaling.send({desc: pc.localDescription});
      } else if (desc.type === 'answer') {
        await pc.setRemoteDescription(desc);
      } else {
        console.log('Unsupported SDP type.');
      }
    } else if (candidate) {
      await pc.addIceCandidate(candidate);
    }
  } catch (err) {
    console.error(err);
  }
};
```

Figura 2.3: Esempio minimale di implementazione di un server WebSocket per signaling

Fonte: https://www.html5rocks.com/en/tutorials/webrtc/basics/

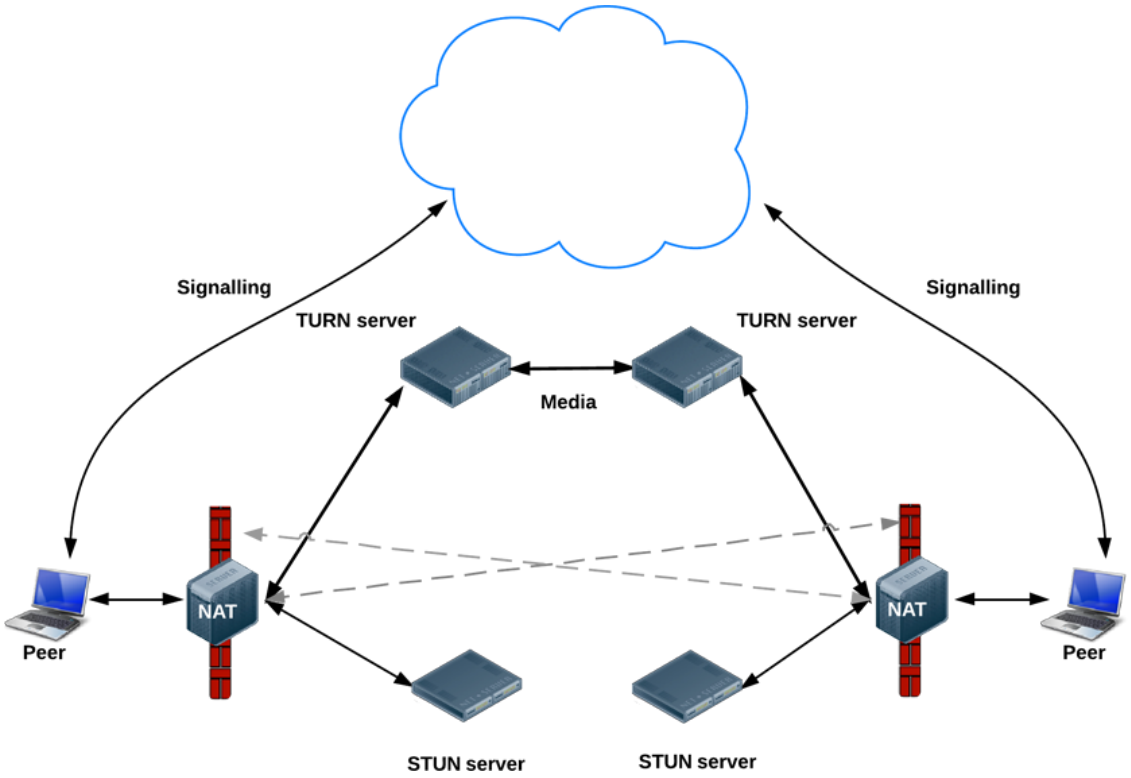

Figura 2.4: Schema completo di un architettura WebRTC applicata a reti reali

2.3 NW.js e Chrome Headless

La tecnologia WebRTC è pensata principalmente per mettere in collegamento due utenti finali, o più in generale, due applicativi destinati ad essere utilizzati da un'utenza fisica. Per riuscire quindi ad adattare un'architettura pensata per il peer-to-peer ad una più classica client-server, è necessario affidarsi a soluzioni specifiche. Per questo progetto si è deciso di sviluppare la componente server dell'applicativo usando come tecnologie NW.js e Chrome Headless.
NW.js è un framework che permette di sviluppare applicazioni desktop utilizzando HTML, CSS e JavaScript. Originariamente era conosciuto col nome di Node Webkit in quanto derivante dalla combinazione del framework Node.js con il browser engine Chromium (Webkit). Con questo applicativo è possibile non solo eseguire siti web come delle applicazioni desktop ma anche far interagire queste applicazioni con il sistema operativo stesso attraverso le API JavaScript e i moduli Node (ad esempio accedendo al file system, cosa impossibile per un browser).
Questo tipo di approccio si scontra con quello tipico delle applicazioni web, costituite da un lato backend e un lato frontend distribuiti in posti diversi e con compiti separati. Il backend è situato solitamente in un server remoto, e la pagina che invia al client è eseguita nel contesto del browser che deve rispettare le policy di sicurezza imposte dal browser stesso. Eseguendo l'applicativo con NW.js come programma controllato dal sistema operativo, il lato frontend è libero dai vincoli di sicurezza imposti da un browser permettendo quindi al codice JavaScript di interagire su entrambi i lati dell'applicazione web[5]. L'idea quindi alla base di NW.js è riunire sotto un'unica soluzione i compiti del backend e del frontend (figura 2.5).
Esistono comunque in rete altre soluzioni simili ad NW.js, come Electron o Chrome App ma rispetto a queste soluzioni NW.js presenta alcuni punti di forza[6]:

1. supporto per vecchie edizioni di Windows e Mac OS

2. è possibile impostare come avvio del programma una pagina HTML o uno script Node

[5] https://dzone.com/articles/what-is-nwjs

[6] https://hackernoon.com/why-i-prefer-nw-js-over-electron-2018-comparison-e60b7289752

3. offre due tipi di contesto per JavaScript e Node: separato e misto[7]
4. supporta le estensioni di Chrome e relative API
5. supporto e stampa per i documenti PDF
6. offre protezione del codice sorgente
7. segue gli aggiornamenti di Chromium
8. supporta tutti i flag (o Switch) della command line di Chromium [8]
9. è possibile distribuire le applicazioni con NW.js senza gli strumenti di sviluppo integrati

L'altra tecnologia utilizzata sul server è Chrome Headless, in pratica è un modo per eseguire un browser Chrome in una modalità senza componente grafica. Permette di portare la piattaforma web con le relative funzioni su linea di comando. Solitamente è utilizzato per eseguire dei test automatizzati (come viene fatto nella parte finale di questo progetto). Su una macchina server può rivelarsi utile per permettere di utilizzare le funzionalità offerte dal browser in un ambiente privo quasi sempre di una GUI.
In questo progetto non è utilizzata direttamente la modalità headless di Chrome da linea di comando ma vengono utilizzate due librerie di alto livello che permettono di controllare il browser programmaticamente: Puppeteer e Selenium.

2.3.1 Puppeteer

Puppeteer è una libreria Node che mette a disposizione delle API di alto livello per controllare le azioni del browser. Permette di eseguire il browser anche in modalità non headless, utile ad esempio durante le fasi di test dell'applicazione. Anche se ancora in fase sperimentale esiste una versione che include il suppoerto per Firefox.
Permette di eseguire quasi tutte le azioni che compirebbe normalmente un normale utente, permettendo quindi di automatizzare dei processi e di rieseguirli quante volte si vuole. Supporta la generazione degli screenshot, analizzare il contenuto di una pagina, automatizzare la sottomissione dei form,

[7] http://docs.nwjs.io/en/latest/For20Users/Advanced/JavaScript20ContextsinNW.js/

[8] https://www.chromium.org/developers/how-tos/run-chromium-with-flags

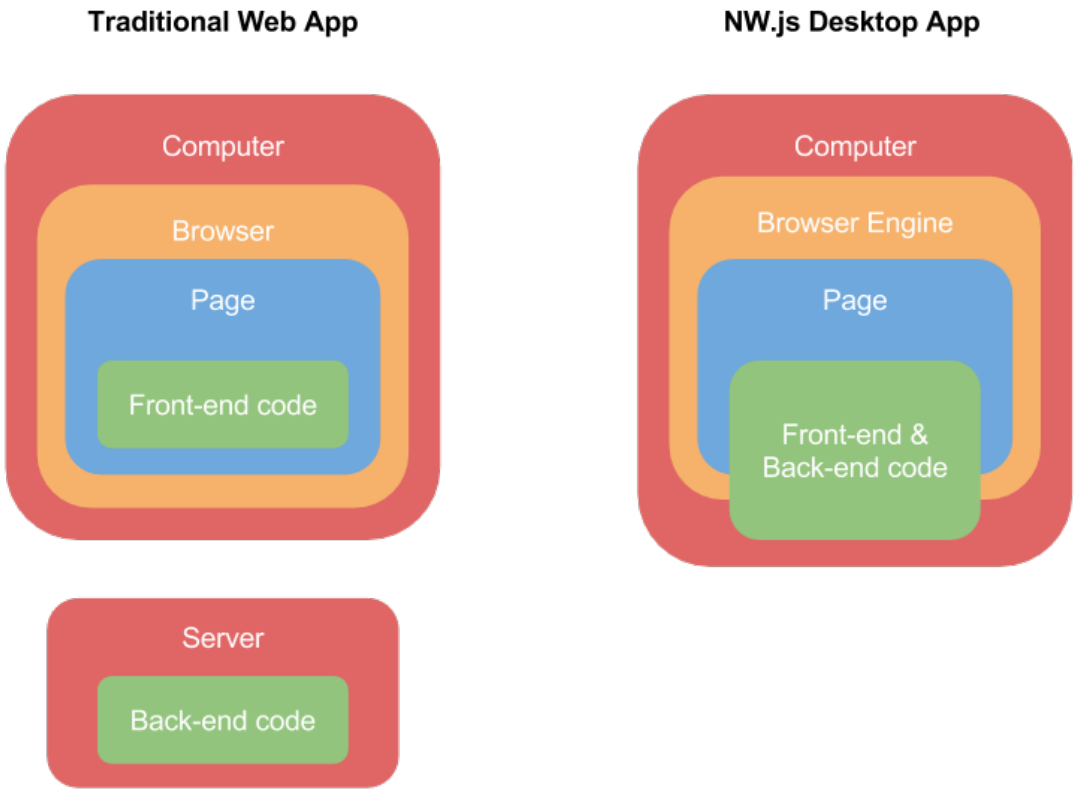

Figura 2.5: La differenza delle architetture di un'applicazione web contro un'applicazione NW.js

simulare l'input da tastiera o da mouse...[9] Il suo utilizzo è molto semplice, è sufficiente creare un'applicazione Node.js e inserire il pacchetto *puppeteer* nel file package.json; verrà inoltre scaricata insieme alla libreria la versione più recente di Chromium.
Alcune note da tenere presente: Puppeteer è rilasciato con Chromium dal quale eredita le limitazioni, ad esempio non supporta i formati AAC o H.264 a causa della mancanza delle relative licenze (questa limitazione è superabile se si utilizza una versione di Chrome installata separatamente) oppure non supporta funzioni disponibili sono nella versione mobile di Chrome come HTTP Live Stream (HLS).

[9] https://github.com/GoogleChrome/puppeteer

2.3.2 Selenium

Selenium, conosciuto anche come Selenium/WebDriver, è un progetto che incorpora numerose libreria e tool per permettere l'automazione web browser. Anche Selenium è pensato principalmente per il testing automation e offre supporto per gran parte di linguaggi di programmazione. Permette inoltre di eseguire automation test non solo su Chrome ma anche su Firefox, Safari, Opera e Internet Explorer.
Anche se Selenium e Puppeteer sono dei progetti che offrono quasi le medesime funzionalità, presentano alcune differenze.
Selenium si focalizza sull'automazione cross-browser, offrendo delle API che funzionino su tutti i principali browser. Puppeteer invece ruota intorno a Chromium offrendo delle API più ricche e performanti.
Sempre per la sua natura cross-browser Selenium necessità di un setup più complesso e file aggiuntivi prima di poter essere utilizzato, Puppeteer invece viene già distribuito come soluzione completa insieme a Chroumium facilitando in questo modo l'inizio dello sviluppo. Inoltre, a differenza di Selenium, Puppeteer è basato su un'architettura event-driver[10].

[10] https://github.com/GoogleChrome/puppeteer#q-is-puppeteer-replacing-seleniumwebdriver

Parte I

Sviluppo del progetto

Capitolo 3

L'architettura generale

L'obiettivo di questa parte sarà quello di illustrare nel dettaglio il lavoro svolto, in particolare tutto ciò che riguarda lo sviluppo e la configurazione dell'ambiente client e server. Verrà spiegato, per quanto possibile, ogni parte di cui il progetto è composto e come è stato possibile realizzarla.

L'architettura sviluppata può essere rappresentata in linea generale dalla figura 3.1, che illustra le due componenti principali (client e server) e le tecnologie presenti.
Il cuore dell'architettura è costituito dal collegamento WebRTC, insime ai server STUN e TURN, che mette a disposizione del server un mezzo di comunicazione real-time per la trasmissione delle tracce riducendo drasticamente le latenze rispetto ad altri tipi di comunicazione, come ad esempio HTTP.
Possiamo inoltre notare come la componente di rendering audio sia localizzata interamente lato server, lasciando quindi al client solo le componenti necessarie ad effettuare il collegamento (client WebSocket e peer WebRTC), il che rende possibile l'utilizzo di questa soluzione anche a sistemi meno performanti di un sistema desktop (come uno tablet o smartphone)

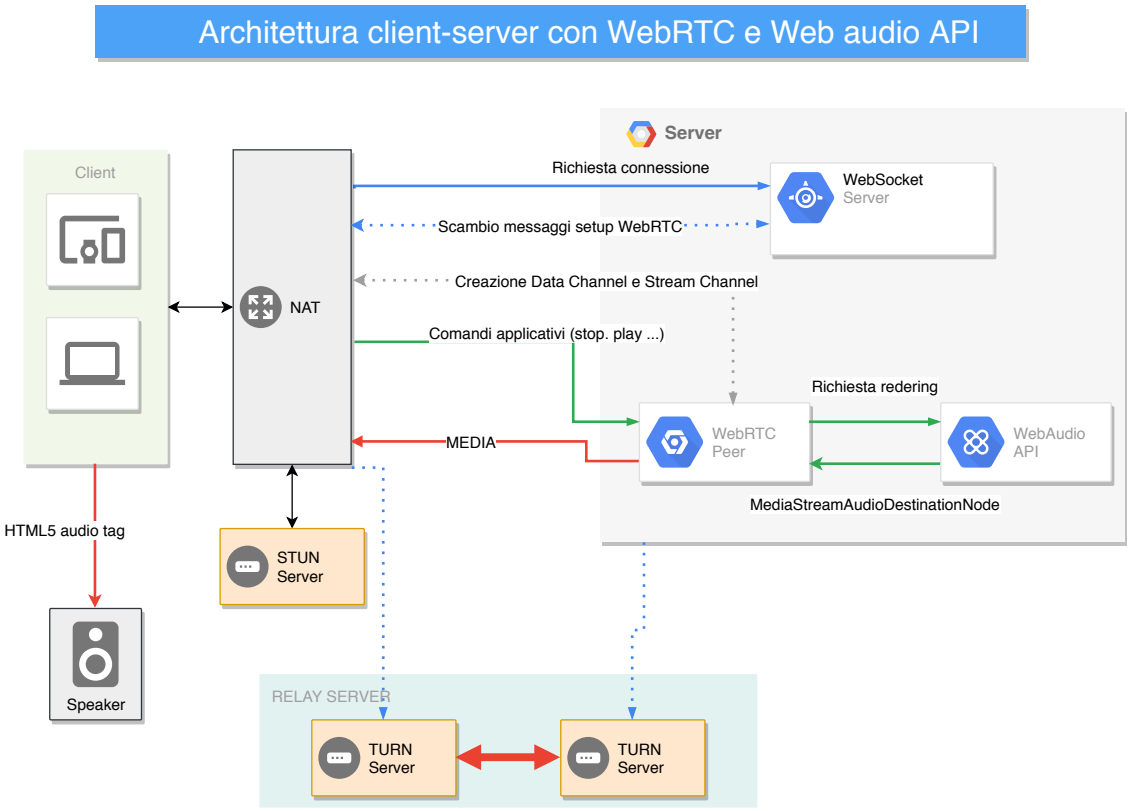

Figura 3.1: Schema generale dell'architettura sviluppata

Capitolo 4

Il server

4.1 L'architettura del server

Iniziamo l'analisi del progetto partendo dal lato server. La soluzione per il server è stata sviluppata e distribuita sia in ambiente locale che su un server remoto e utilizzando tre tecnologie diverse; i componenti di cui il server è costituito sono i seguenti (vedi 4.1):

- un server di signaling
- una componente adibita al rendering audio
- un peer WebRTC
- nel caso delle soluzioni Headless (Puppeteer, Selenium), è necessario anche un server web.

Approfondiamo ora queste componenti nell'ordine in cui vengono utilizzate, evidenziando anche le differenze di implementazione in base alla tecnologia utilizzata.

4.2 Server Web

Questo componente è necessario solo se utilizziamo una tecnologia Chrome Headless, in quanto operando praticamente su un browser, è necessario un server web che metta a disposizione le risorse della macchina, come i file audio o i file JavaScript con il codice da eseguire; NW.js, invece, può interagire direttamente con il file system della macchina server.

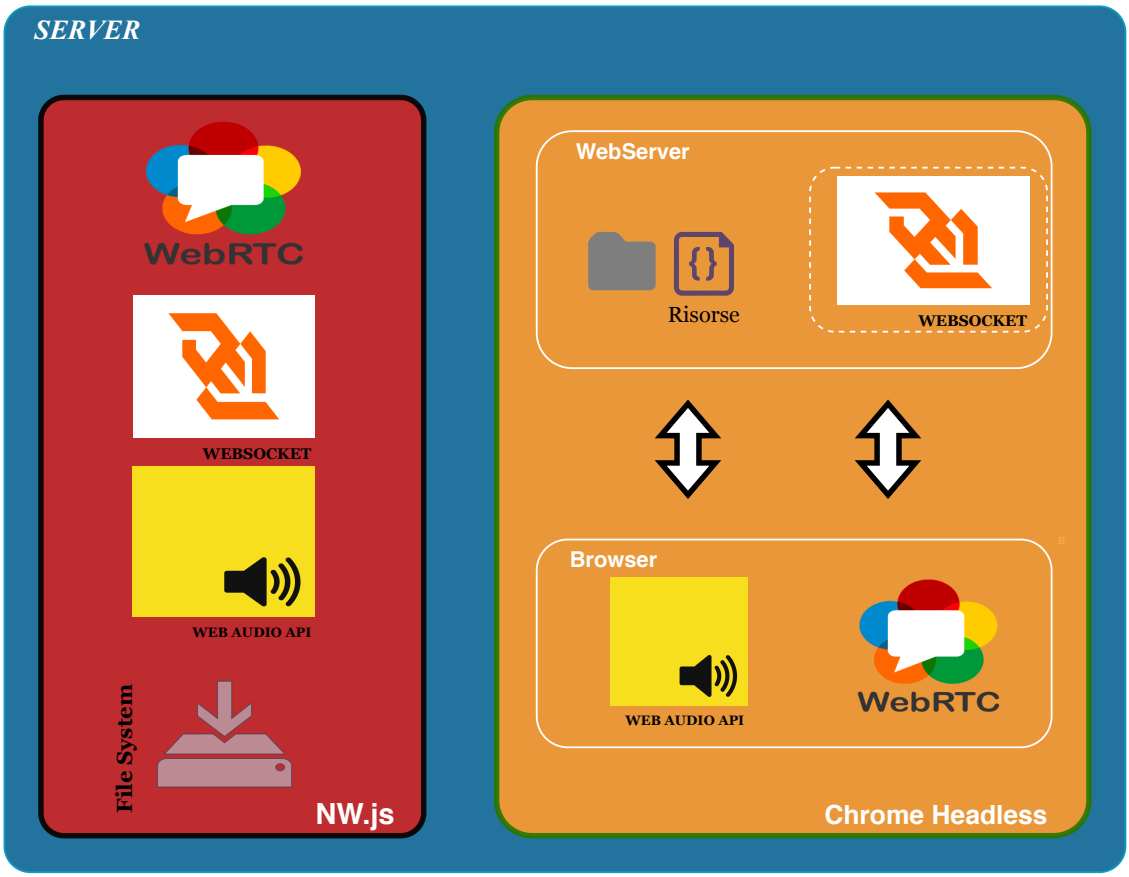

Figura 4.1: Differenze architetturale delle due soluzioni sviluppate lato server

Per creare il server Web è stato utilizzato il modulo Node **Express**, un framework open source per applicazioni web ed è ormai definito il server framework standard per Node. Oltre a questo è stata fatta questa scelta in quanto Express supporta anche le richieste WebSocket, avendo quindi in questo modo anche la componente di signaling. Il compito quindi di questo server è quello di mettere a disposizione degli endpoint per Chrome Headless da cui è possibile ottenere alcuni file presenti sulla macchina sottoforma di risorse; per farlo una volta a disposizione l'oggetto che rappresenta il server si può utilizzare il metodo *use()* passando come parametri una stringa che rappresenta l'endpoint delle risorse e l'oggetto ritornato dal metodo *express.static()*, una funzione middleware integrata che gestisce i file statici, quali immagini, file CSS e file JavaScript. È necessario inoltre abilitare la ricezione di richieste HEAD, in quanto questo metodo sarà utilizzato dal client per calcolare la latenza della connessione come spiegato nella sezione 5.5.

4.3 Caricamento delle tracce

All'avvio, la prima operazione effettuata dal server prima di essere pronto ad accettare connessioni in ingresso è quella di caricare in memoria le tracce audio disponibili. In questo modo il tempo di attesa del client per ottenere una traccia diminuisce notevolmente, anche in funzione della dimensione della traccia, in quanto il server la troverebbe già disponibile per il rendering senza ulteriori attese. Questa operazione può comunque essere eseguita ogni volta che il client richiede una traccia indicando esplicitamente al server di caricarla nuovamente in memoria e non usando quella già presente; questo potrebbe verificarsi in un applicazione finale quando, ad esempio, un client richiede l'annullamento delle modifiche effettuate sulla traccia, ricaricando quella originale.
Il processo di caricamento deve trasformare una file audio presente sul disco (può essere un *.wav* o un *.mp3*) in un oggetto JavaScript di tipo **AudioBuffer** che può essere utilizzato con le Web Audio API. In base alla soluzione usata, il caricamento delle tracce può avvenire in due modi diversi:

- con NW.js avendo a disposizione tutti i moduli Node, possiamo ad esempio utilizzare il modulo **fs**, che permette di interagire con il file system della macchina; con questo modulo è possibile caricare i file audio in memoria con il metodo *readFileSync()* o con la sua versione asincrona *readFile()*, e da ognuno di essi ottenere un AudioBuffer utilizzando il metodo *decodeAudioData()* dell'oggetto **AudioContext**; questo metodo accetta come parametro un ArrayBuffer (è sufficiente richiamare la proprietà *buffer* dell'oggetto restituito leggendo il file) e restituisce una Promise<AudioBuffer>.

- con le soluzioni Headless, come già detto, non è possibile accedere direttamente al file system, in quanto si sta eseguendo il codice da un browser; è necessario quindi effettuare una **XMLHttpRequest** sul server web locale per ottenere la traccia audio. Il metodo può essere GET ma la response type deve essere di tipo **arraybuffer**, in questo modo nella callback di successo della chiamata è possibile richiamare il metodo *decodeAudioData()* passando come parametro la risposta ottenuta, come visto nel caso precedente.

L'utilizzo del metodo *decodeAudioData()* è il metodo preferito per la creazione di una risorsa audio da passare alle Web Audio API, ma questo metodo

funziona solo su dati completi e non su frammenti di file audio[1]. Tutte le tracce vengono messe a disposizione in memoria in un array di AudioBuffer.

4.4 Il server di signaling

Per poter scambiare i messaggi necessari a instaurare la connessione tra due peer, WebRTC deve affidarsi ad un meccanismo di segnalazione esterno. Avendo libertà di scelta sul sistema da utilizzare, si è deciso di utilizzare un server WebSocket.
Anche in questo caso la scelta della tecnologia influenza l'implementazione della soluzione, in quanto WebSocket deve fare affidamento su un server HTTP per effettuare l'handshake.

4.4.1 Signaling in Chrome Headless

Nel caso di Chrome Headless la scelta di Express come server Web permette anche di avere connessioni WebSocket. Per farlo è necessario creare una nuova istanza **Router** di Express, che permette la creazione di handler di route modulari, e della libreria Node **express-ws**[2], che aggiunge le funzionalità WebSocket ad applicazioni Express. Sull'istanza creata è sufficiente quindi richiamare il metodo *ws()* passando come parametri l'endpoint di ascolto e la funzione da eseguire all'arrivo di una nuova connessione.
La funzione riceve in ingresso un parametro **ws** che rappresenta il client che richiede la connessione WebSocket; con il metodo *on()* di questo oggetto è possibile passare degli handler a vari eventi; quelli utilizzati nel progetto sono l'evento **message** e l'evento **close**.
Di seguito un esempio di creazione di un endpoint WebSocket che manda indietro al client il messaggio ricevuto:

```
1 server = express();
2 new websocket(server);
3 let router = express.Router();
4 router.ws('/echo', function(ws, req) {
5   ws.on('message', function(msg) {
6     ws.send(msg);
7   });
```

[1] https://developer.mozilla.org/en-US/docs/Web/API/BaseAudioContext/decodeAudioData

[2] https://github.com/HenningM/express-ws

```
8 });
9 server.use('/', router);
```

Il caso Headless presenta delle difficoltà maggiori rispetto a NW.js in quanto le entità dove si trova il peer WebRTC (il browser) e il server WebSocket sono completamente separate; il server WebSocket non può distinguere autonomamente i messaggi provenienti dal peer locale. La gestione dei messaggi è quindi un passaggio cruciale del server WebSocket in quanto esso deve riconoscere se il messaggio in arrivo sia destinato al peer WebRTC presente sulla macchina server (quindi è un messaggio di un client remoto verso il peer locale) o ad uno dei client connessi in quel momento (quindi dal peer locale ad un client remoto).
Quando il browser sul server termina la fase di caricamento delle tracce audio, apre un collegamento con il server WebSocket presente sulla macchina locale; appena il collegamento è instaurato il codice JavaScript del browser invia un messaggio con un contenuto particolare per indicare che il client che ha spedito il messaggio è quello localizzato sulla macchina server. Vediamo l'intero processo nel dettaglio.
All'arrivo di una nuova connessione, l'oggetto **ws**, che racchiude le informazioni del client, è inserito in un array; alla scatenazione di un evento **message** il WebSocket controllerà che esso contenga un particolare campo (quello inserito solo dal browser client sul server) e riconoscerà così se quel client è quello presente sul server. A questo punto i successivi messaggi ricevuti possono essere di due tipo:

- messaggi del client locale verso uno dei tanti client remoti; per riconoscere questo messaggio il client locale inserisce un campo *boolean* nel messaggio per indicare che è partito dal server stesso e un campo *clientID* che rappresenta l'indice dell'array dei client connessi al WebSocket, in modo da richiamare il metodo *send()* sull'instanza client corretta.

- messaggi da uno dei client remoti verso il client locale; rinconoscibile dall'assenza del campo *boolean* menzionato nel punto precedente; viene quindi inserito nel messaggio, che arriverà poi al client locale, l'indice corrispondente dell'array di client. Questo indice sarà quello che permetterà di distinguere le varie istanza WebRTC presenti sul server.

L'evento **close** è utilizzato per eliminare l'istanza del client che si è disconnesso dall'array dei client; viene inoltre inviato un messaggio WebSocket al client locale indicando l'indice del client disconnesso per la liberazione delle risorse del relativo peer WebRTC (*RTCPeerConnection.close()*).

Uno schema del funzionamento del WebSocket per la soluzione Chrome Headless è rappresentato in figura 4.2

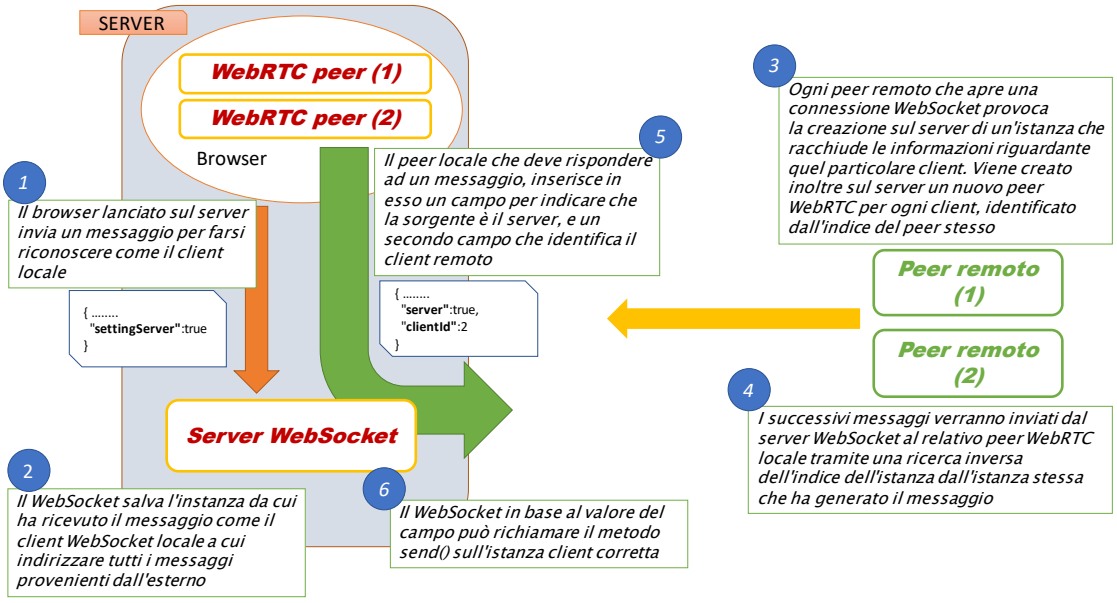

Figura 4.2: Schema dello scambio dei messaggi su WebSocket nella soluzione Headless

4.4.2 Signaling in NW.js

In NW.js l'implementazione del WebSocket è più semplice in quanto il componente WebRTC è integrato nello stesso ambiente in cui è possibile sviluppare il server WebSocket. Anche se, come già detto, il server web non è necessario per NW.js in quanto è possibile accedere direttamente a tutte le risorse della macchina, è richiesto comunque un meccanismo capace di trasmettere richieste HTTP per poter utilizzare WebSocket.
In questo caso possiamo evitare di utilizzare Express in favore di un modulo più leggero e incluso in Node.js stesso, il modulo **http**. Per avviare il server http è sufficiente richiamare il metodo *http.createServer()*, abilitando comunque la ricezione di richieste HEAD (vedi sezione 5.5), e successivamente il metodo *listen()* sull'oggetto restituito precedentemente indicando la porta su cui il server (e di conseguenza il WebSocket) starà in ascolto.

Utilizzando poi un secondo modulo Node, il modulo **websocket**[3], è possibile legare le richieste WebSocket al server creato. In questo caso la richiesta di nuove connessioni è gestita con un handler sull'evento **request**; come nel caso Headless anche qui viene gestito un array di client remoti collegati al server ma con la differenza che non vi è un client locale da contrassegnare in quanto i messaggi WebSocket vengono passati direttamente alla componete WebRTC. L'evento **message** in questo caso aggiunge solamente l'indice del client remoto che ha inviato il messaggio per permettere al server di richiamare il peer WebRTC relativo a quello specifico client.
Se invece è la componente WebRTC del server a dover inviare un messaggio, può accedere direttamente all'array dei client e richiamare il metodo *send()* sull'istanza corretta tramite l'indice passato precedentemente.
Di seguito un esempio di configurazione di WebSocket con il modulo http:

```
let wsServer:server = new webSocketServer.server({
        // WebSocket server is tied to a HTTP server.
        httpServer: server});
    // This callback function is called every time someone
       tries to connect to the WebSocket server
    wsServer.on('request', function (request: request) {
        console.log((new Date())+' Connection from origin '+
           request.origin);
        // accept connection
        let connection=request.accept(null, request.origin);

        // we need to know client index to remove them on '
           close' event
        let index = clients.push(connection) - 1;
        console.log((new Date()) + ' Connection accepted.');

        // user sent some message
        connection.on('message', function (message: IMessage)
            {
            onMessageReceived.next({
                clientID:       index,
                message:        message
            });
        });

        // user disconnected
        connection.on('close', () => {
```

[3] https://github.com/theturtle32/WebSocket-Node

```
24              console.log((new Date()) + " Peer "+ connection.
                    remoteAddress + " disconnected.");
25              // remove user from the list of connected clients
26              clients.splice(index, 1);
27              onClientDisconnected.next(index);
28          });
29      });
```

4.5 Il peer WebRTC

Come già detto nelle sezioni precedenti, in base alla soluzione utilizzata, il peer WebRTC può trovarsi direttamente integrato nell'architettura server o essere un peer separato lanciato in un browser.
A prescindere dalla tecnologia, l'implementazione di questo componente è quasi del tutto simile per tutti i casi sviluppati eccezion fatta per il modo in cui esso contatta il componente WebSocket per inviare un messaggio di configurazione al peer remoto. Se infatti si utilizza NW.js possiamo richiamare direttamente il metodo *send()* dell'istanza del peer remoto dal modulo WebSocket, se invece siamo su una soluzione Headless è necessario inviare il messaggio sul canale WebSocket come se si stesse contattando un server remoto in modo tale che questo messaggio (opportunamente contraddistinto come proveniente dal peer WebRTC locale) sia poi recapitato al corrispondente client remoto dal server WebSocket presente sulla macchina.

La creazione di un peer WebRTC vera e propria viene effettuata quando viene ricevuto un particolare messaggio dal canale WebSocket; la distinzione dei vari tipo di messaggi avviene tramite il controllo di un particolare campo settato dall'applicativo, il campo *type*.
La creazione del nuovo oggetto *RTCPeerConnection*, quindi, avviene alla ricezione di un messaggio di tipo **connect**; esso indica una richiesta da parte di un client remoto di creazione di una nuova connessione WebRTC. Essendo una connessione WebRTC, come già visto, di tipo peer-to-peer è necessario creare una nuova istanza di questo oggetto per ogni client che si connetta al server, se si vuole permettere la presenza di connessioni multiple.
Anche in questo caso, come già fatto per il server WebSocket, si utilizza un array di oggetti RTCPeerConnection che verranno indicizzati dal clientID passato dal layer WebSocket (che rappresentava l'indice dell'istanza client remota connessa in quel momento). Si è cercato di usare una classe Typescript che inglobasse tutte le funzioni necessarie al corretto settaggio e configurazione di un peer WebRTC. Al momento di creazione dell'istanza vengono anche

passati degli event-handler a dei particolari eventi che vedremo nel dettaglio nella sezione 5.3; per ora ci limiteremo a descrivere i seguenti eventi:

- **onnegotiationneeded**: si verifica quando sono presenti delle modifiche tali da richiedere una rinegoziazione della sessione; l'event-handler utilizzato in questo caso creerà una nuova offerta da mandare al client remoto tramite WebSocket (metodo *createOffer()*), il metodo può accettare delle informazioni sul tipo di media che vogliamo inviare; per il nostro progetto possiamo indicare come limitazione il fatto che utilizzeremo come media solo audio; fatto questo il peer locale setterà l'offerta appena creata come descrizione locale tramite il metodo *setLocalDescription()* e provvederà quindi a inviare l'offerta stessa sul canale di signaling (con un messaggio di tipo **offer**).
  Segnalerà inoltre al client, sempre con WebSocket, il timestamp di avvenuta creazione dell'offerta.

- **onicecandidate**: si verifica quando è disponibile un nuovo ICE Candidate da inviare al client. L'handler infatti riceve come parametro un oggetto di tipo *RTCIceCandidate* che dovrà essere inviato con WebSocket impostando il tipo di messaggio come **ice-candidate**, in modo da essere riconosciuto come tale dal client remoto. Se l'oggetto passato è undifiened o null significa allora che tutti i candidati sono stati inviati.

4.5.1 Il DataChannel

Operazione molto importante effettuata a questo punto è la creazione del DataChannel; per questo progetto infatti è stato deciso che fosse il server a procedere con la sua creazione.
Questo processo avviene subito dopo aver settato gli handler prima discussi; sempre dall'oggetto RTCPeerConnection si richiama il metodo *createDataChannel()*. Questo metodo accetta due parametri, una stringa che funge da label per il DataChannel e un oggetto opzionale di tipo **RTCDataChannelInit** che permette di dare delle configurazioni al canale.
Il DataChannel può usare come protocollo lo *Stream Control Transmission Protocolo* o SCTP che permette di definire il tipo di trasmissione da utilizzare e configurarla[12].
Il DataChannel è supportato in Chrome, Firefox, Opera sia in versione Desktop che su Android. Permette la trasmissione di vari tipi di dati come Blob, ArrayBuffer e ArrayBufferView oltre ovviamente alle stringhe di testo.

Questo si può rivelare molto utile in caso di trasmissione di file e giochi multiplayer ma in questo progetto è sufficiente inviare e ricevere comandi sotto forma di stringhe JSON (usando quindi *JSON.stringify()* e *JSON.parse()*). Le varie configurazioni del DataChannel sono raffigurate in tabella 4.1.

	TCP	**UDP**	**SCTP**
Affidabilità	affidabile	non affidabile	configurabile
Trasporto	ordered	unordered	configurabile
Trasmissione	byte-oriented	message-oriented	message-oriented
Controllo del flusso	SI	NO	SI
Controllo della congestione	SI	NO	SI

Tabella 4.1: Configurazioni e proprietà del WebRTC DataChannel

Fonte: https://www.html5rocks.com/en/tutorials/webrtc/datachannels/

La modalità affidabile e ordinata garantisce la consegna dei messaggi e il loro ordine; ciò comporta dell'overhead che potrebbe rendere questo metodo più lento. D'altra parte la modalità non affidabile e non ordinata rimuove questi overhead a discapito della non garanzia di consegna e arrivo dei pacchetti in ordine, risultando quindi quella più veloce. La terza modalità garantisce l'affidabilità e l'ordine sotto specifiche condizioni (come ad esempio specificando un timeout di ritrasmissione). Il vantaggio di questa modalità intermedia consiste nel fatto che se non vi è perdita di pacchetti la latenza è simile a quella della modalità UDP.
Per poter indicare che modalità utilizzare è necessario valorizzare alcuni parametri dell'oggetto RTCDataChannelInit[4]

- **ordered**: Indica se garantire o meno che i messaggi arrivino a destinazione nello stesso ordine in cui sono stati inviati. Di defualt è true.
- **maxPacketLifeTime** Tempo massimo in millisecondi di attesa per trasferire o ritrasferire un messaggio prima di rinunciare. Di defualt è null.

[4] https://www.html5rocks.com/en/tutorials/webrtc/datachannels/

- **protocol**: nome di un sub-protocollo da usare sul DataChannel, definito a livello applicativo (ad esempio "json", "raw"...), di default è una stringa vuota.

- **negotiated**: se true rimuove tutte le configurazioni automatiche del DataChannel sul peer remoto; ciò significa che è necessario dare una propria configurazione sul DataChannel che ha lo stesso id sul peer remoto. Di defualt è false.

- **id**: Permette di dare il proprio ID al DataChannel, usato di solito insieme a **negotiated=true**.

Se ad esempio si vuole utilizzare la configurazione UDP bisogna settare **maxRetransmits** a 0 e **ordered** a false. Nel progetto sviluppato si è deciso di garantire l'ordine e l'affidabiità dei comandi inviati, in quanto la latenza introdotta dall'overhead necessario a garantire queste caratteristiche può essere trascurato rispetto alla latenza introdotta da altri fattori presenti. Per questo in fase di creazione del DataChannel il parametro RTCDataChannelInit è passato a null, in modo tale da utilizzare i valori di defualt precedentemente esposti.

4.6 Gestione dei comandi e rendering audio

Una volta che il DataChannel è configurato ed è passato nello stato **open**, è possibile cominciare a ricevere comandi dal client. Questi comandi, come già detto, sono delle semplici stringhe JSON che verrano poi convertite in oggetti con il metodo *JSON.parse()*.
Nei comandi inviati dal client è presente un campo, di nome **command**, che permette di comprendere il tipo di richiesta del client e poterla così soddisfare.

Se il comando arrivato è di tipo **play_techno** o **play_song** allora il client ha richiesto l'esecuzione di una traccia; in base ad un secondo campo indicato sempre dal client, la traccia verrà prelevata dall'array in memoria caricato in fase di avvio del server o ricaricata nuovamente dal file system (che sia in modo diretto o tramite chiamata HTTP).
Una volta ottenuta la traccia è necessario collegarla alla connessione WebRTC. Si utilizzano di nuovo le Web Audio API per creare un oggetto di tipo **AudioBufferSourceNode** dall'AudioContext tramite il metodo *createBufferSource()* a cui assegnare come proprietà buffer la traccia scelta. Si

collega quindi questo nodo ad un'istanza di un oggetto di tipo **MediaStreamAudioDestinationNode**; questa interfaccia rappresenta una destinazione audio consistente in un MediaStream WebRTC necessario per trasportare la traccia audio al client.[5]
Effettuato il collegamento è necessario dare il comando di inzio per la traccia che verrà poi riprodotta sul peer remoto, per far questo si richiama il metodo *start()* dall'AudioNode relativo alla traccia scelta; il metodo permette inoltre di riprodurre la traccia da un determinato momento e per un determinato numero di secondi passando questi valori come parametri. È possibile inoltre riprodurre la traccia in loop valorizzando a true la proprietà **loop** dell'oggetto AudioBufferSourceNode.
Ora che la traccia è collegata al MediaStreamAudioDestinationNode è necessario aggiungere lo stream contenuto in questo oggetto ad un sender della connessione WebRTC, cioè il componente responsabile dell'encoding e del trasporto dei dati sullo stream di trasmissione. È possibile avere un solo sender attivo per connessione, motivo per il quale è necessario procedere, prima di effettuare il collegamento della traccia, ad una ricerca del sender attivo, attravero il metodo *getSenders()*; in caso di successo è possibile agire in due modi, in base alla preferenza indicata dal client nel comando inviato: rimuovere la traccia dal sender con il metodo *removeTrack()* e quindi aggiungere la nuova traccia con il metodo *addTrack()* oppure si può utilizzare il metodo *replaceTrack()* che provvede automaticamente ad effettuare questo cambio. La preferenza del client per la modalità di cambio traccia è usata come caso di studio per valutare le diverse latenze nei due casi.
Appena effettuato un add o un replace della traccia il client inizierà a ricevere i primi dati audio.

Se il comando ricevuto è di tipo **stop** il client vuole interrompere una riproduzione in corso. Il compito del server in questo caso si limita alla ricerca del sender attivo e ad effettuare una chiamata al metodo *removeTrack()* sull'istanza trovata.

Se il comando è di tipo **mixer** il client ha richiesto la creazione di un mixer che faccia l'unione di due tracce. Questo componente non è presente nelle WebAudio API ma è possibile combinare delle API per ottenere un risultato simile, cioè l'unione di due canali audio con un proprio peso regolabile.
Da due tracce presenti in memoria si creano i rispettivi oggetti AudioBufferSourceNode con il metodo *createBufferSource()*. Si creano quindi due nodi

[5] https://developer.mozilla.org/en-US/docs/Web/API/MediaStreamAudioDestinationNode

audio di tipo **GainNode**[figura 4.3].

Questa interfaccia permette l'applicazione di un guadagno, che si traduce in un aumento di volume, alla traccia in ingresso prima che venga propagata in uscita. Un GainNode ha un solo ingresso e una sola uscita, entrambi con lo stesso numero di canali.

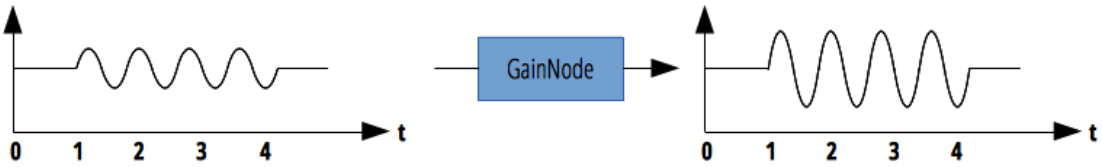

Figura 4.3: Rappresentazione del componente GainNode delle Web Audio API che permette un aumento del guadagno del segnale di ingresso

Fonte: https://developer.mozilla.org/it/docs/Web/API/GainNode

Settato il livello di volume per entrambi i nodi (inizialmente entrambi ad un valore pari a 0.5) si utilizza un secondo componente delle WebAudio, il ChannelMergerNode.[figura 4.4]
Questa interfaccia permette l'unione di differenti input mono in una singola uscita. Ogni input è usato come canale dell'output; nel nostro caso avendo un output stereo è possibile riunire due tracce; il numero di canali in input è passato come argomento del costruttore.

Una volta collegati i due GainNode al MergerNode, si provvede a collegare l'output al solito oggetto MediaStreamAudioDestinationNode che verrà poi usato da un sender del peer WebRTC. Il client in questo modo riceverà come audio una singola traccia stereo come unione di due tracce diverse, una per ogni canale.

L'ultimo comando che il client può inviare è di tipo **firstChUp**; esso richiede al server di aumentare il guadagno al massimo per uno dei due GainNode presenti nel mixer. Per farlo è sufficiente richiamare il metodo *gain()* del primo dei GainNode passando come parametro il valore 1.0. L'aumento del volume verrà quindi eseguito sul primo canale della traccia in esecuzione sul client.

Uno schema della gestione dei comandi da parte del server è dato dalla figura 4.5.

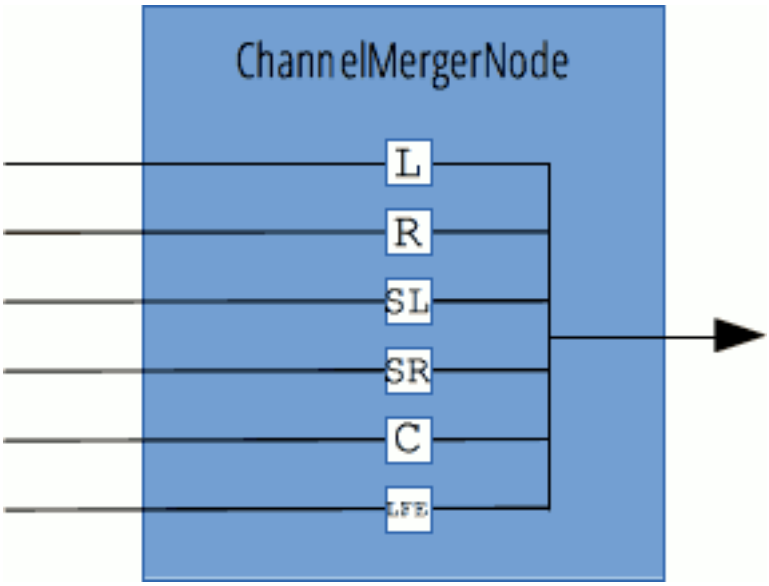

Figura 4.4: Rappresentazione del componente ChannelMerger delle Web Audio API, il cui compito è quello di unire più canali in un unica uscita audio

Fonte: https://developer.mozilla.org/en-US/docs/Web/API/ChannelMergerNode

4.7 L'ambiente remoto

Lo sviluppo del server è stato dapprima eseguito sulla stessa macchina desktop del client, una volta completato il lavoro è stato possibile portare il progetto su una macchina server remota messa a disposizione dal Politecnico di Torino.
Per poter eseguire però il progetto del server su un sistema remoto è necessario un'adeguata configurazione sia a livello di pacchetti software che di impostazioni del sistema stesso.
Per poter utilizzare le soluzioni Headless è necessaria l'installazione dei pacchetti Node.js e Npm; dopo è necessario lanciare il comando **npm install** nelle cartelle delle tre implementazioni dove è situato il file **packege.json** per poter scaricare tutte le librerie richieste dal progetto.

La soluzione Headless che utilizza Selenium necessità inoltre la presenza sul sistema di Google Chrome (quindi se non presente è necessario installarlo) e di ChromeDriver. Quest ultimo è un eseguibile esterno a Selenium che il WebDriver utilizza per poter controllare il browser.

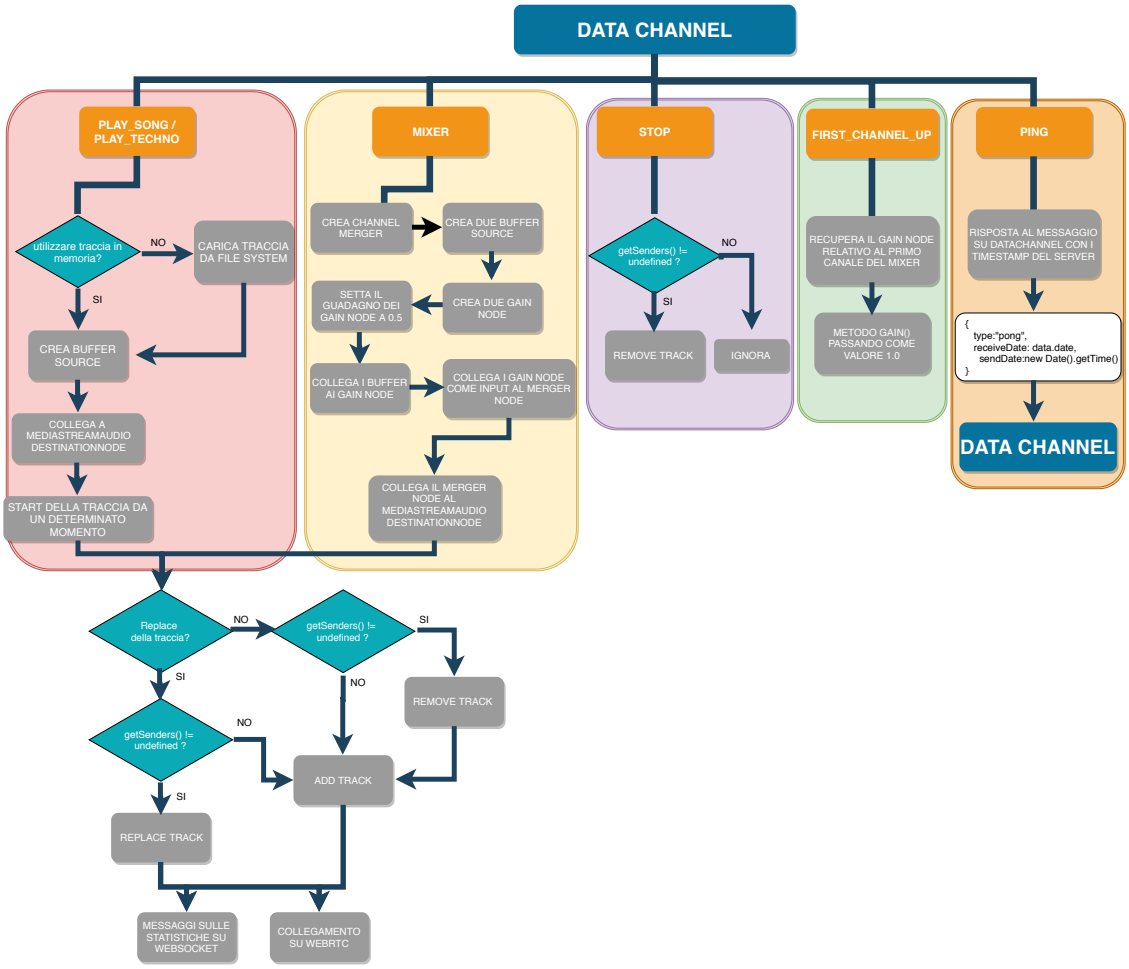

Figura 4.5: Schema logico della gestione dei messaggi del DataChannel sul server

La soluzione NW.js richiede invece una configurazione particolare, in quanto per poter essere eseguita necessita di un ambiente desktop, il che potrebbe essere un problema in un server privo di interfaccia grafica. Le soluzioni possibili a questo problema sono state principalemente due: l'utilizzo di un desktop virtuale (Xvfb) e docker.
Xvfb (acronimo per X Virtual FrameBuffer) è un server display per piattaforme UNIX che utilizza il protocollo X11. Permette di eseguire qualsiasi applicazione che richieda un interfaccia grafica senza la presenza di un display, permettendo anche, ad esempio, di poter acquisire degli screenshot. A differenza di altri display server può eseguire tutte le operazione in una memoria virtuale senza alcun output a schermo. È richiesta solo la presenza

del layer di network.
Per utilizzarlo è necessario installare il programma e una volta fatto è sufficiente far partire Xvfb su una display port specifica ed lanciare il processo in background; si indica quindi alla shell di utilizzare la display port in questione ed, in questo modo, rendere possibile l'esecuzione di NW.js anche in assenza di un display reale. Di seguito un esempio dei comandi necessari per eseguire un progetto NW.js posizionato nella cartella di lavoro corrente:

```
$: Xvfb :99 &
$: export DISPLAY=:99
$: nw .\
```

Docker è un software open source che permette di eseguire dei container Linux. Per questo progetto è stata utilizzata un'immagine docker basata su Ubuntu 16:04 con xrdp[6].
Xrdp è un Server RDP Open-Source che incapsula il protocollo VNC sotto RDP in maniera molto veloce; **RDP** (Remote Desktop Protocol) è un protocollo proprietario di Microsoft che permette una connessione desktop remota; per questo infatti una volta avviata l'immagine docker è possibile collegarsi alla macchina server tramite Desktop Remoto di Windows senza necessità di un client VNC esterno.
Per configurare il container è stato creato un file **docker-compose.yml** che permette di specificare i parametri di configurazione e l'immagine da utilizzare invece di passare queste informazioni volta per volta da linea di comando; in figura 4.6 è riportato il contenuto del file docker-compose. Le porte esposte sono la **5900** per permettere il collegamento con l'applicazione di Desktop Remoto, la **222** per permettere il collegamento tramite protocollo ssh e la porta **7001** utilizzata dal progetto per esporre l'endpoint WebSocket e permettere il collegamento del client.
Portati i sorgenti sul volume del container (l'utilizzo di un volume permette di avere la persistenza dei dati) e recuperate le dipendenze necessarie, è possibile eseguire la parte del progetto sviluppata con NW.js come se fosse su un comune ambiente desktop. Il vantaggio di utilizzare docker è sicuramente quello di poter avere un'applicazione capace di scalare molto bene

[6] https://hub.docker.com/r/danielguerra/ubuntu-xrdp/

orizzontalmente, che riesce quindi a creare una nuova istanza dell'immagine e allocare le giuste risorse solo quando vi è effettivamente richiesta di una nuova connessione. Lo svantaggio invece, come vedremo nell'analisi delle latenze, si traduce in un significativo ritardo causato dal layer di astrazione di Docker e dal ritrovarsi in una rete chiusa che necessita di affidarsi ai relay server (TURN) per riuscire ad effettuare il collegamento con WebRTC.

```
version: '3'
services:
    nwjserver:
        container_name: nwjs
        image: danielguerra/ubuntu-xrdp
        restart: always
        hostname: terminalserver
        shm_size: 2g

        ports:
            - 5900:3389
            - 222:22
            - 7001:7001
        volumes:
            - ./data/ssh/:/etc/ssh/
            - ./home/:/home/
            - ./Desktop/:/home/ubuntu/Desktop/

#download the nwjs sdk and create a lancher with command
#"nwjs-sdk/nw NwjsPeer" with "NwjsPeer" the name of the project directory
```

Figura 4.6: Contenuto del file docker-compose.yml contentente la configurazione del container usato per eseguire NW.js

4.7.1 Il problema della sincronizzazione dei clock

Un problema sicuramente importante del deploy dell'applicativo su un server remoto reale riguarda lo sfasamento dei clock delle due diverse macchine dove l'applicativo del client e quello server sono collocati. L'offset di sfasamento tra la macchina client e quella server non è assolutamente trascurabile in quanto questo progetto è finalizzato allo studio delle latenze delle varie soluzioni proposte, latenze dell'ordine della decina di millisecondi; è chiaro quindi come anche solo un offset di 100 ms non sia tollerabile.
Per eliminare, o quantomeno ridurre quanto più possibile, questo sfasamento, è stata utilizzata una libreria che sfrutta l'algoritmo di sincronizzazione dei clock proprio del protocollo NTP, chiamata **GoTime**[7].

[7] https://github.com/nicksardo/GoTime

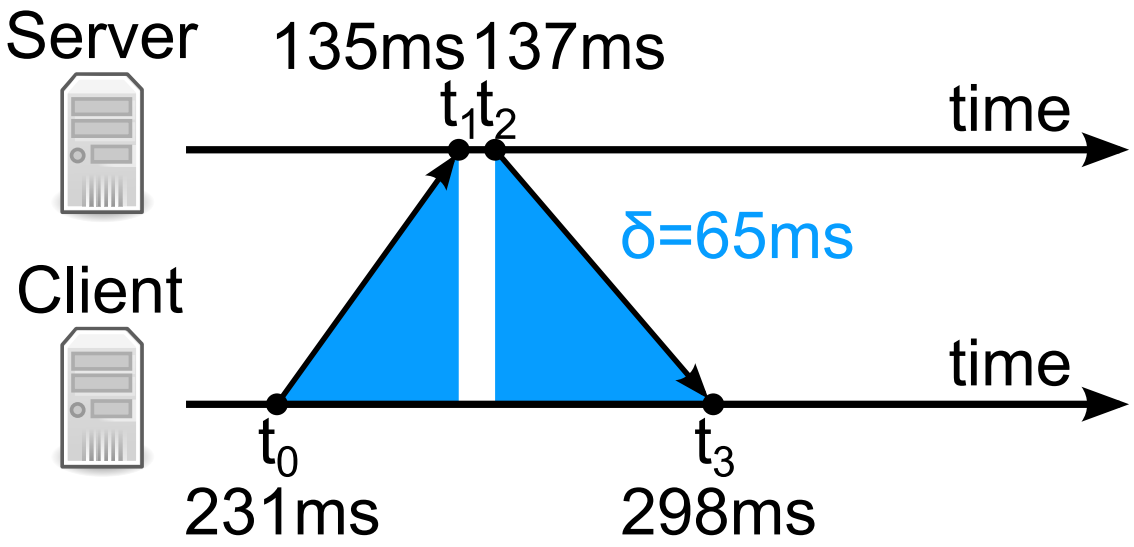

Figura 4.7: Esempio di calcolo dell'offset del clock su due sistemi sulla rete

Fonte: https://en.wikipedia.org/wiki/Network_Time_Protocol#Clock_synchronization_algorithm

NTP(Network Time Protocol) è il protocollo di rete utilizzato per sincronizzare i clock di sistemi diversi ed è uno dei protocolli di rete più vecchi esistenti. L'algoritmo, utilizzato anche dalla libreria, è basato sul fatto che il client, per sincronizzare il proprio orologio, deve calcolare l'offset dei due clock e un ritardo di "andata-ritorno"(round-trip) del pacchetto. Questo offset θ è definito come:

$$\theta = \frac{(t_1 - t_0) + (t_2 - t_3)}{2}$$

mentre il ritardo di round-trip δ come:

$$\delta = (t_3 - t_0) - (t_2 - t_1)$$

dove:

- **t0** è il timestamp di trasmissione del pacchetto sul client
- **t1** è il timestamp di ricezione del pacchetto sul server
- **t2** è il timestamp di invio del pacchetto di risposta sul server
- **t3** è il timestamp di ricezione del pacchetto di risposta sul client [figura 4.7]

θ e δ sono utilizzati poi per eseguire delle analisi matematiche che, al netto degli outliers, permettono di ridurre sempre di più l'offset tra i due clock[4].

Per poter utilizzare l'algoritmo, la libreria usata necessita di un canale per trasmettere il pacchetto su cui eseguirà i calcoli; è possibile esporre un endpoint con un web server ma per ottenere maggiore precisione nel calcolo dell'offset è possibile utilizzare il canale WebSocket già presente nel progetto. La libreria deve inviare un messaggio sul canale WebSocket ed essere notificata alla ricezione del pacchetto di ritorno. Avendo deciso che i calcoli delle latenze saranno effettuati lato client, sarà quest ultimo a configurare la libreria e a tener conto dell'offset dei clock nelle misurazioni rilevate.
Quando il client e il server, quindi, aprono il canale WebSocket il client invia un particolare messaggio al server (di tipo **date**) nel seguente modo:

```
1 GoTime.wsSend(function(){
2         sendToServer({type:webSocketMessageType.date});
3         return true // tell GoTime that websocket message was
              sent
4         });
```

Il server quindi alla ricezione di questo messaggio dovrà semplicemente replicare con un nuovo messaggio di tipo **date** contenente il timestamp di ricezione del messaggio stesso.

Quando il client riceverà a sua volta sul canale WebSocket un messaggio di tipo **date** significherà che ha ricevuto la risposta del server; il contenuto quindi del messaggio (il timestamp del server) viene passato nuovamente alla libreria che eseguirà i calcoli descritti prima; questa operazione è implementata nel seguente modo:

```
1 case webSocketMessageType.date:
2
3         // When processing websocket messages client-side
4         // If response is the server telling the time, notify
              GoTime immediately with the data
5         GoTime.wsReceived(content.data);
6         break;
```

Dopo qualche secondo in cui questo processo viene ripetuto per aumentare la precisione dei calcoli, il client può conoscere l'offset in millisecondi tra i due clock semplicemente richiamando la funzione *GoTime.getOffset()*.

Capitolo 5

Il client

5.1 L'architettura del client

Andiamo ora a vedere nel dettaglio ora le componenti del client e la relativa realizzazione all'interno del lavoro svolto. Nella figura 5.1 vengono illustrate più dettagliatamente le parti che costituiscono il client che non sono state raffigurate nella 3.1.

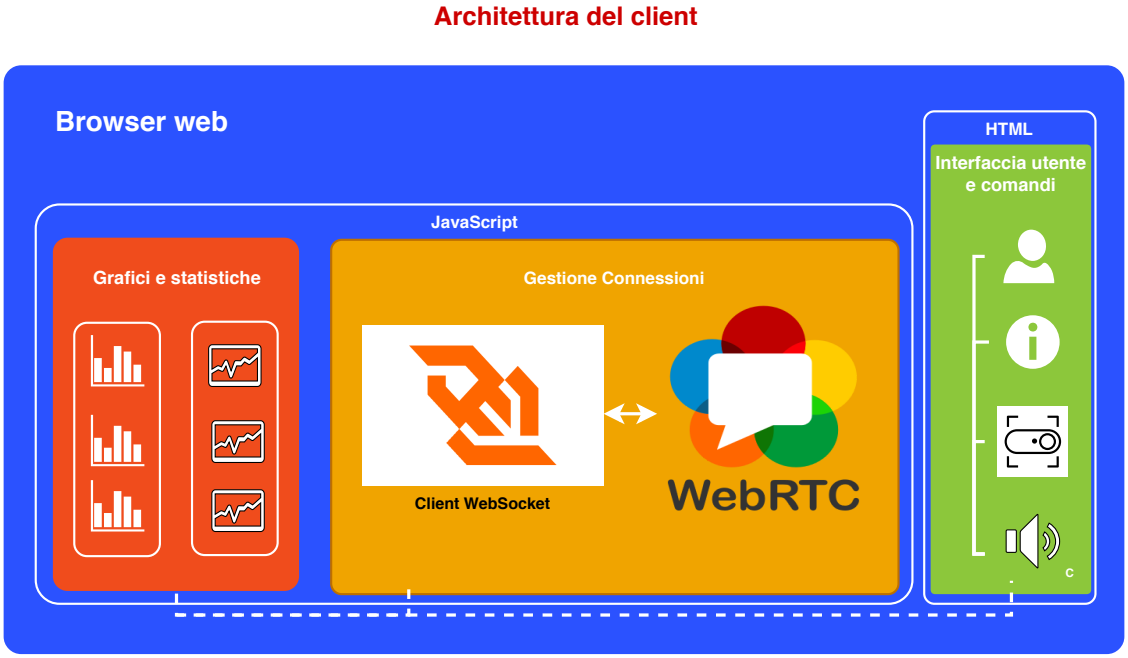

Figura 5.1: Schema dei componenti del client

Come illustrato il client è principalmente costituito da un browser web che

ha il compito di visualizzare una pagina html. Questa pagina, oltre a gestire l'input dell'utente e le varie opzioni deve caricare i file Javascript contenenti il codice che si occupa di stabilire la connessione WebSocket (il server di signaling) e tramite questo instaurare la connessione peer-to-peer WebRTC per permettere l'invio dei comandi sul data channel e di ricevere l'audio in real-time. Si occupa inoltre nel visualizzare i vari dati e grafici relativi alla connessione.

Nella figura 5.2 è mostrato come si presenta la pagina HTML all'interno del browser. La pagina è composta da:

- I comandi di connessione permettono di scegliere a che tipo di server connetterci (se locale o remoto), la tecnologia del server (NW.js, Puppeteer, Selenium)

- I comandi da dare al server per richiedere la riproduzione di una traccia o fermarne una in riproduzione. E' possibile inoltre cambiare il modo in cui il server cambia una traccia in riproduzione con un'altra, se eliminando dallo stream la vecchia traccia e poi riaggiungendola oppure utilizzare la funzione di replace della traccia offerta dall'oggetto WebRTCManager.

- Un'area di test che mostra le latenze dei vari messaggi mandati sul websocket per instaurare una connessione WebRTC (Ice candidate, Offer...) insieme alla misura di latenza totale dell'audio da quando abbiamo premuto il comando a quando effettivamente possiamo ascoltare la traccia.

- Il tag HTML5 <audio> che permette di riprodurre lo stream ricevuto e dei campi di testo che possiamo usare per decidere quale punto della traccia (minuto e secondo) andare a riprodurre.

- Il grafico sia in frequenza che temporale dell'audio ricevuto

- I campi che ci permettono di impostare varie parametri sulla connessione (se usare server STUN-TURN o solo TURN relay), se usare UDP o TCP e il codec preferito (OPUS, ISAAC, G722, PCMU, PCMA). Solo nel caso di OPUS è possibile settare dei parametri aggiuntivi come audio mono/stereo, CBR, INBAND FEC e DTX.

- Una tabella che raccoglie alcune statistiche della connessione attuale quali velocità di banda, il tipo di server utilizzato dal client e dal server, il codec, la cifratura della connessione e gli indirizzi IP dei peer.

- Un grafico che rappresenta il bitrate e i pacchetti ricevuti al secondo

5.2 La connessione

Verrà illustrato ora il meccanismo usato per instaurare una connessione con il server e avere un collegamento peer-to-peer con WebRTC.

Come già spiegato, WebRTC necessità di un sistema di signaling per poter configurare la connessione e scambiare i vari parametri di connessione; lo standard non prevede un meccanismo in particolare in quanto non è parte delle API del RTCPeerConnection; la scelta è ricaduta quindi su WebSocket.

5.2.1 Instaurazione della connessione

Cosa succede quindi quando l'utente preme il tasto "Connetti"? La prima cosa che viene fatta è instaurare una connessione WebSocket sul server, si recupera perciò l'endpoint corretto in base alla configurazione scelta dall'utente (server remoto-locale, NW.js/Puppeteer/Selenium). L'endpoint deve indicare come protocollo di connessione "ws://" o "wss://" in caso di WebSocket su TLS. In questo lavoro di tesi la sicurezza non è trattata quindi verrà utilizzato il normale protocollo "ws://".

Naturalmente questo implica che la connessione instaurata è del tutto in chiaro ed esposta ad attacchi esterni, non adatta quindi ad un eventuale prodotto finale. Si crea quindi un oggetto WebSocket (offerto dal DOM), questo oggetto offre alcuni eventi che possiamo utilizzare; in particolare:

1. Il metodo "onopen", che verrà invocato non appena il tentativo di connessione si concluderà con successo e potremmo quindi inviare messaggi

2. Il metodo "onmessage" che si attiverà ogni qualvolta riceveremo un messaggio dal server

Una volta quindi che il metodo onopen sarà richiamato possiamo procedere con la configurazione della connessione WebRTC. Per farlo creiamo un oggetto RTCPeerConnection (anche questo offerto dal DOM) a cui passeremo una lista di server STUN e TURN (vedi struttura in figura 5.3), la policy per l'ICE transport ("all" per usare i server STUN+TURN, cioè tutti gli ICE candidates verranno considerati, "relay" per considerare solo gli ICE candidates i cui indirizzi IP sono stati ritrasmessi cioè, che sono passati attraverso

Figura 5.2: La pagina del client

i server TURN). [1] Si aggiungono le implementazioni agli eventi WebRTC. Questi eventi e la loro implementazione verranno trattati in seguito.

Settato il nostro oggetto RTCPeerConnection possiamo inviare sul canale WebSocket un messaggio per avvertire il server di creare il suo oggetto RTCPeerConnection che crea quindi il data channel come illustrato precedentemente e può quindi iniziare il meccanismo di scambio dei messaggi WebRTC.

Viene inoltre mandato il messaggio contentente la data del client, richiesto della libreria GoTime, come esposto nella sottosezione 4.7.1

5.2.2 La gestione dei messaggi

Analizziamo ora cosa accade quando l'evento "onmessage" viene richiamato, quando si riceve cioè un nuovo messaggio dal server.

Nel messaggio WebSocket è stato inserito un campo "type" che permette di distinguere il tipo di messaggio ricevuto; in base al tipo di messaggio quindi si procede in questo modo:

- Se di tipo **"date"**, è stato ricevuto un messaggio contenente la data settata sul server, questa viene passata alla libreria GoTime che imposterà l'offset tra i due clock.

- Se di tipo **"updateType"** è stato ricevuto un messaggio che informa il client della latenza di un ice-candidate o di un offerta

- Se di tipo **"offer"** è un messaggio di configurazione del WebRTC: è stata ricevuta una nuova offerta di negoziazione (una session description o SDP), il contenuto del messaggio viene passato al metodo *setRemoteDescription()* dell'oggetto RTCPeerConnection che cambia la descrizione remota associata alla connessione. La descrizione indica anche le proprietà del nodo remoto, incluso il codec. Il metodo ritorna una Promise che si completa quando la descrizione è stata cambiata in modo asincrono.

  Il messaggio può anche essere ricevuto dopo aver già instaurato una connessione, ciò significa che sono magari cambiate le condizioni della rete. I peer continuano a scambiare descrizioni finchè non si accordano

[1] https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection#RTCIceTransportPolicy_enum

```
[
  {
    "urls": [
      "stun:webrtcweb.com:7788",
      "stun:webrtcweb.com:7788?transport=udp"
    ],
    "username": "muazkh",
    "credential": "muazkh"
  },
  {
    "urls": [
      "turn:webrtcweb.com:7788",
      "turn:webrtcweb.com:4455?transport=udp",
      "turn:webrtcweb.com:8877?transport=udp",
      "turn:webrtcweb.com:8877?transport=tcp"
    ],
    "username": "muazkh",
    "credential": "muazkh"
  },
  {
    "urls": [
      "turn:numb.viagenie.ca"
    ],
    "username": "Alace1957@dayrep.com",
    "credential": "passw123"
  },
  {
    "urls": [
      "stun:stun.l.google.com:19302",
      "stun:stun.l.google.com:19302?transport=udp"
    ]
  }
]
```

Figura 5.3: Struttura di un array di server STUN-TURN, solitamente per i server TURN è necessario anche fornire delle credenziali di autenticazione

su una configurazione, ciò significa che il metodo non ha effetto immediato.[2] Appena la Promise è completata con successo viene quindi creata una risposta SDP all'offerta ricevuta con il metodo *createAnswer()*. La risposta contiene informazioni su qualsiasi media già collegato alla connessione, codec, opzioni supportate dal browser e gli ICE candidates

[2] https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/setRemoteDescription

raccolti.[3] L'SDP quindi restituita dalla promise viene modificato in base al codec scelto dall'utente, e nel caso di OPUS, delle opzioni attivate.

Viene quindi inviato al server con un messaggio WebSocket di tipo "answer" e usato per settare la descrizione locale con il metodo *setLocalDescription()*. Per questo metodo valgono le stesse considerazioni del metodo *setRemoteDescription()*, con la differenza che questi permette di settare le properità del nodo locale.

- se di tipo **"ice-candidate"**, significa che abbiamo ricevuto un nuovo ice-candidate, dobbiamo quindi passare il contenuto del messaggio al metodo *onIncomingIceCandidate()* dell'oggetto RTCPeerConnection. Viene aggiunta in questo modo una nuova descrizione remote che rappresenta lo stato del nodo remoto. Se si riceve una stringa vuota come contenuto allora sono stati ricevuti tutti i candidati.

- In teoria si potrebbe ricevere un ulteriore tipo di messaggio, il tipo **"answer"** il cui contenuto è passato come argomento al metodo *setRemoteDescription()* ma nel nostro caso ciò non accade in quanto è sempre il server a creare il data channel o a richiedere l'invio di uno stream audio, inviando quindi sempre per primo il messaggio di offerta e mai uno di risposta.

Il meccanismo appena esposto è raffigurato dalla figura 5.4

5.3 La gestione di WebRTC

Analizziamo ora gli eventi che vengono configurati quando viene richiesta la creazione di un nuovo oggetto RTCPeerConnection.

Appena l'oggetto viene creato settati i seguenti eventi:

- **onicecandidateerror**: è chiamato quando si verifica un errore durante il processo di scambio degli ICE candidates. Nel nostro progetto viene gestito loggando sulla *console.error()* l'errore verificato e rilasciando le risorse dell'oggetto *RTCPeerConnection* chiamando il metodo *close()*.

- **onicecandidate**: è scatenato quando l'agente ICE locale deve mandare un messaggio al nodo remoto tramite il signaling server. Permette quindi

[3] https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/createAnswer

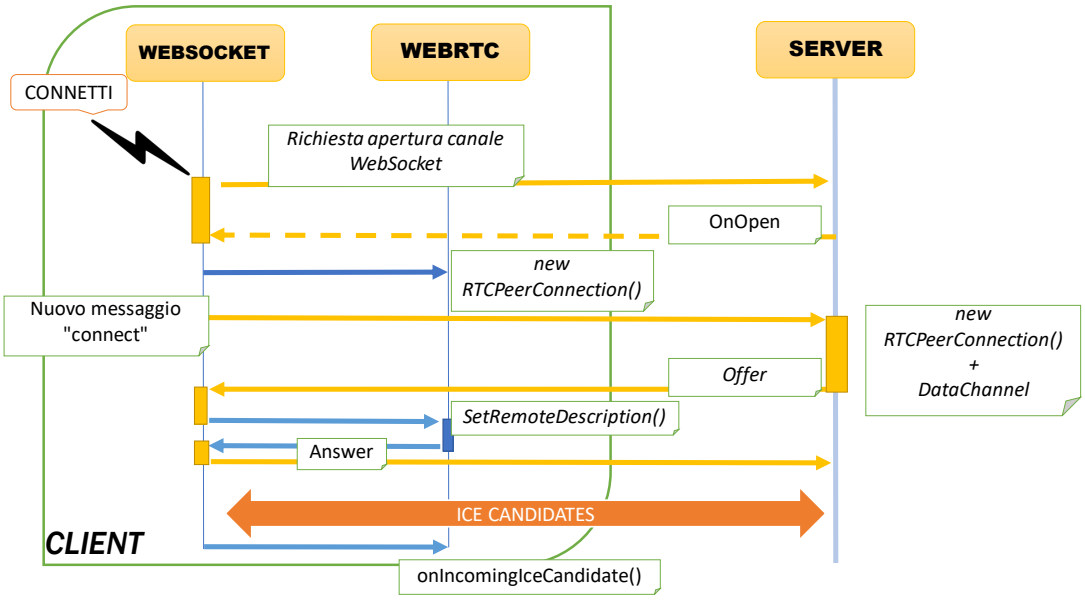

Figura 5.4: Schema dei messaggi scambiati durante l'instaurazione della connessione WebRTC tra client e server

di poter scambiare il messaggio senza specificare al browser la tecnologia usata per spedirlo.[4] Per questo è stato gestito con una funzione che invia l'ICE candidate attraverso il canale WebSocket (etichettandolo come un messaggio di tipo **"ice-candidate"**).
Se nell'evento generato non è presente il campo *candidate* allora tutti i candidati sono stati spediti. È possibile controllare lo stato dell' ICE agent associato all'oggetto RTCPeerConnection attraverso il campo *RTCPeerConnection.iceConnectionState* che fornisce un enum. Se dovesse essere uguale a **'failed'** il collegamento con il peer remoto non è riuscito.

- **onnegotiationneeded**: è richiamato quando si verifica un cambiamento tale da richiedere una rinegoziazione della sessione.

  Solitamente ciò accade quando si aggiunge una traccia alla connessione. Se è già in corso una negoziazione l'evento non verrà generato fino al suo

[4] https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/onicecandidate

termine e solo se sarà ancora necessario.[5] Quando l'evento è scatenato viene prima controllato il campo *RTCPeerConnection.signalingState* che descrive lo stato del processo di signaling sul lato locale della connessione. Solo nel caso che questo campo sia uguale a **'stable'** (cioè non ci sono scambi in uscita di offerte e risposte in quel momento) si procede alla creazione di un'offerta e al relativo invio tramite WebSocket.

- **ondatachannel**: Questo evento si verifica quando un RTCDataChannel viene aggiunto alla connessione quando il nostro server usa il metodo *createDataChannel()*. Ciò però non implica che il DataChannel sia effettivamente aperto e pronto per inviare messaggi (garantinto invece dall'evento **onopen** del DataChannel). La gestione di questo evento sarà esposta dettagliatamente nella sottosezione 5.3.1.

- **ontrack**: rappresenta l'evento principale del nostro progetto, esso notifica il client che una traccia è stata aggiunta alla connessione. In particolare l'evento generato è di tipo *RTCTrackEvent*; questo evento è generato quando una nuova *MediaStreamTrack* è creata e associata ad un oggetto *RTCRtpReceiver* che si aggiunge alla lista di ricevitori della connessione.[6].

  Se il browser utilizzato è Firefox l'evento **ontrack** è generato una sola volta nel corso della connessione; è necessario quindi gestire questo caso configurando un nuovo event handler per l'evento **onaddstream** che verrà invece richiamato ogni qualvolta un ulteriore traccia sarà aggiunta alla connessione. Sia nel primo che nel secondo evento viene gestiscono il modo per riprodurre lo stream audio ricevuto.

  Per farlo viene utilizzato l'oggetto MediaStream contenuto nell'evento; è necessario settare questo oggetto nel campo *srcObject* di un tag audio HTML5. Se nel tag è stata indicata la proprietà **autoplay**, allora l'audio verrà riprodotto automaticamente appena disponibile. Sull'oggetto stream è possibile aggiungere un event handler per l'evento **onremovetrack**, necessario se si vuole sapere quando la traccia è terminata.

La gestione degli eventi di WebRTC è riportata in figura 5.5

[5] https://developer.mozilla.org/en-US/docs/Web/API/RTCPeerConnection/onnegotiationneeded

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

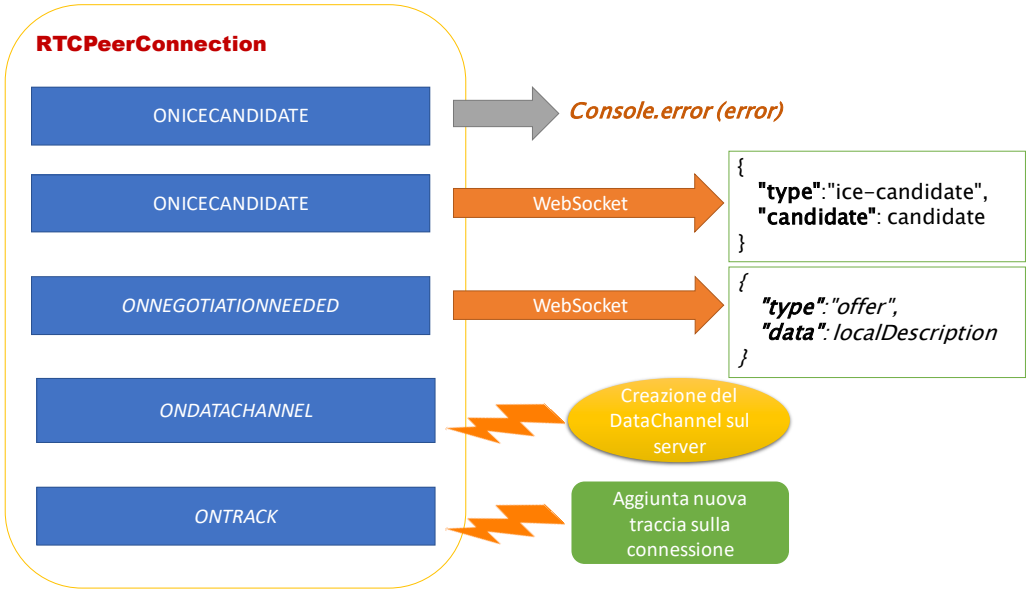

Figura 5.5: Schema degli eventi dell'oggetto RTCPeerConnection del client

5.3.1 La gestione del DataChannel

Come visto per l'oggetto RTCPeerConnection, anche l'oggetto **RTCDataChannel** basa le sue principali funzioni tramite il meccanismo degli eventi. In questo progetto è il server a creare il DataChannel, il client può quindi recuperare questo oggetto tramite l'evento *RTCPeerConnection.ondatachannel*, all'interno del campo *Event.channel*. Sono stati fornite le funzioni di handler ai seguenti eventi del DataChannel:

1. **onopen**: L'evento si scatena quando il DataChannel è effetivamente disponibile e pronto a spedire messaggi. Nel progetto è questo evento che scatena l'aggiornamento dell'interfaccia grafica del client, attivando quindi i bottoni che permettono l'invio dei comandi al server. Il verificarsi dell'evento conferma quindi che la connessione con il peer remoto è avvenuta correttamente.

2. **onerror**: viene lanciato quando si verifica un errore sul DataChannel; la funzione passata riceve in input un oggetto ErrorEvent contenente la descrizione dell'errore scatenato. Anche in questo caso si procede con il log dell'errore sulla console e al relativo rilascio delle risorse con il metodo *close()* dell'oggetto *RTCPeerConnection*.

3. **onclose**: Si verifica quando il peer remoto chiude il DataChannel o la connessione WebRTC. Anche qui si procede con il rilascio delle risorse acquisite.

4. **onmessage**: è l'evento principale, si verifica quando viene ricevuto un nuovo messaggio sul DataChannel.

Ai fini del progetto, è sufficiente che il server invii dei messaggi come stringa JSON. Per poter distinguere i messaggi, è presente un campo *type*. I messaggi ricevuti sul DataChannel sono utilizzati quasi esclusivamente per la rilevazione delle misure di latenza e la tracciatuara dei grafici. Si considereranno nel dettaglio nella sezione 5.5.

5.3.2 Invio dei comandi

L'invio dei comandi al server è affidata al DataChannel. I motivi di tale scelta sono principalemente prestazionali, avendo a disposizione un canale di comunicazione peer-to-peer real time (WebRTC) già configurato si possono ottenere minori latenze nella comunicazione rispetto all'utilizzo di un canale differente come HTTP o WebSocket (non progettati per un tipo di comunicazione così diretta).

Una volta avuto accesso al DataChannel (in seguito all'evento *onopen*), vengono attivati i bottoni che gesticono l'invio dei comandi al server permettendo all'utente, ad esempio, di ricevere una traccia. I comandi al server sono inviati sottoforma di stringa JSON richiamando il metodo *send()* dell'oggetto DataChannel (vedi figura 5.6). Ogni messaggio ha un campo **command** che viene utilizzato dal server per comprendere il tipo di richiesta del client.

Il bottone **"PLAY TECHNO"** e **"PLAY SONG"** richiedono entrambi una traccia al server, indicando come campo *command* "play_techno" o "play_song". Inoltre è possibile indicare (tramite i relativi input utente) se si vuole che la traccia venga caricata in memoria al momento della richiesta o utilizzarne una già bufferizzata, minuto e secondo di inizio traccia e come debba essere sostituita la nuova traccia nel caso sia ne sia già presente una sulla connessione (rimuovere la traccia precedente e poi aggiungere la nuova o utilizzando la funzione di *replace()* offerta da WebRTC).

Il bottone **"STOP TRACK"** richiede l'interruzione della traccia in riproduzione; invia sul DataChannel un messaggio di tipo **"stop_track"**.

Il bottone **"MIXER"** e **"MIXER 1° CH_UP"** gestiscono invece i comandi per il mixer: il primo invia al server un comando di tipo **"mixer"** che ha come effetto la riproduzione di una traccia ottenuta come fusione di due

tracce avente ognuna il proprio volume (o peso), il secondo invia invece un comando **"firstChUp"** che richiede al server l'aumento del peso della prima traccia che produce quindi un aumento del volume della stessa.

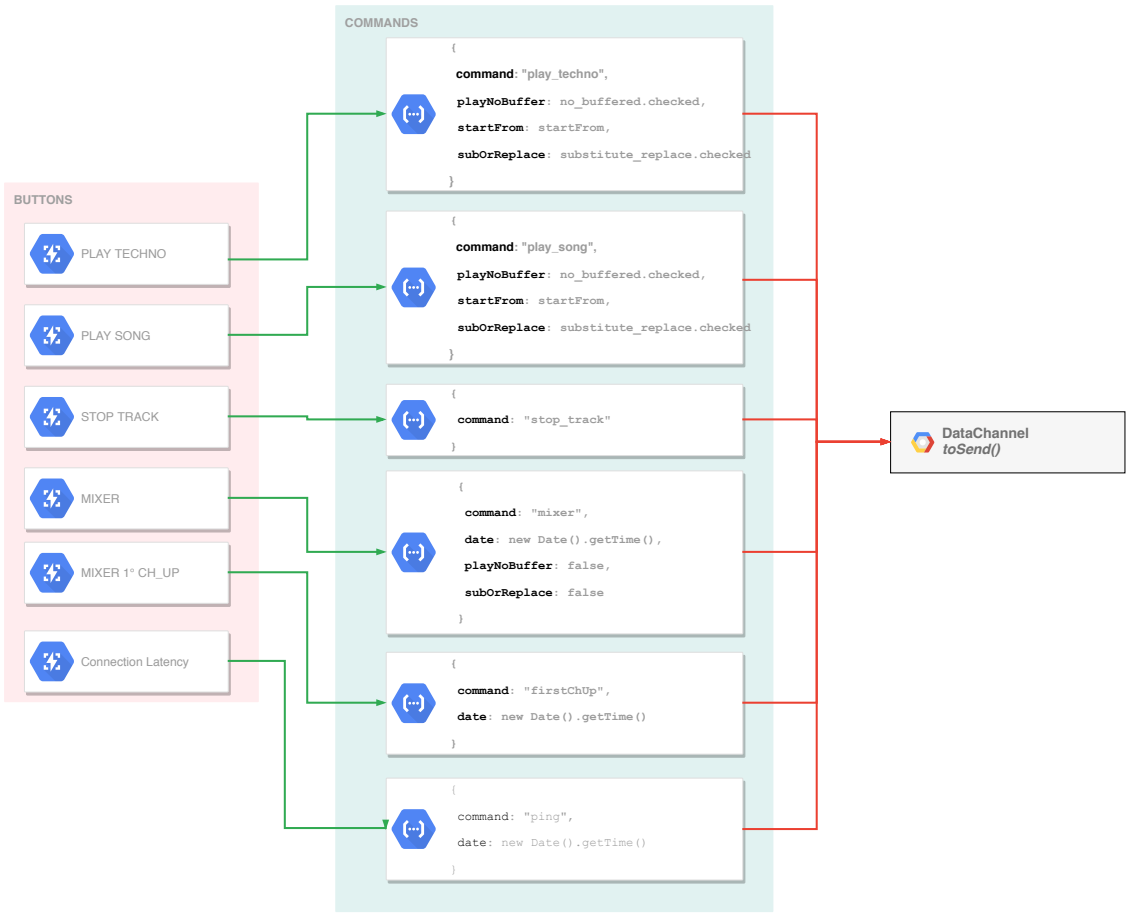

Figura 5.6: I comandi inviati dal client al server tramite il DataChannel, sottoforma di stringhe JSON

5.4 La scelta dei codec

Tra varie opzioni messe a disposizione del client, una delle più importanti è sicuramente la scelta del codec audio utilizzato nella ricezione di una determinata traccia. La scelta è possibile prima di instaurare la connessione con il server in quanto per forzare la scelta di un particolare tipo di codec è necessario modificare il pacchetto SDP durante lo scambio di offerta/risposta al momento dell'instaurazione della connessione.

In particolare, la modifica deve avvenire quando il client riceve dal WebSocket un messaggio di tipo **offer**. Come già visto, alla ricezione di questo messaggio il client provvede a settare la descrizione del peer remoto e a creare una risposta da spedire nuovamente sul canale. Questa risposta è costituita da un SDP (Session Description Protocol); un esempio di tale pacchetto è riportato in figura 5.7 [5].

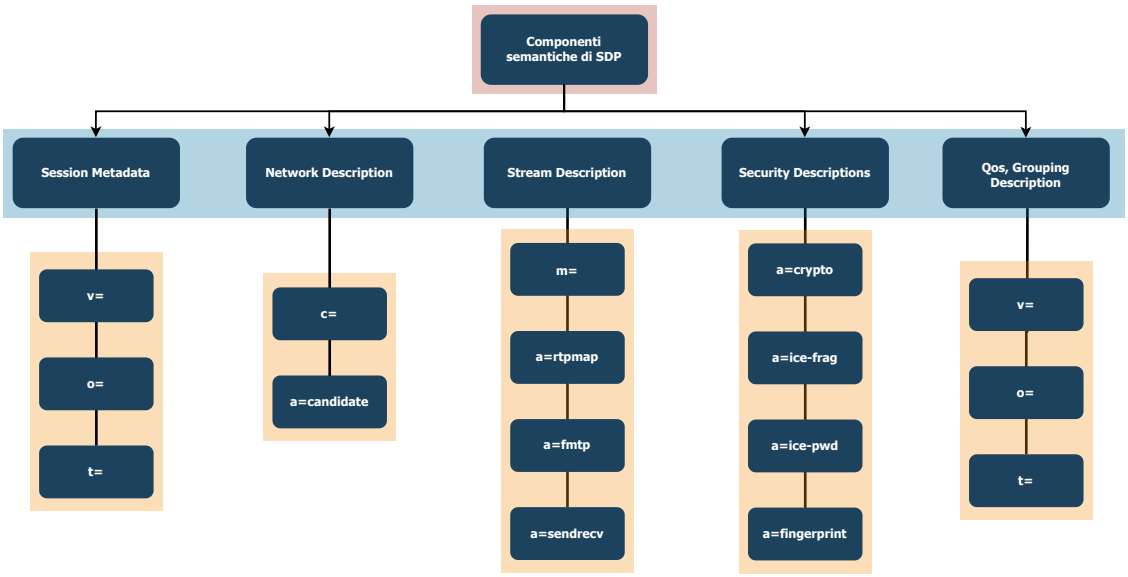

Figura 5.7: Struttura di un pacchetto SDP

Se prima della connessione quindi il client ha selezionato un particolare codec si procede alla modifica del SDP nel seguente modo: si divide il contenuto del messaggio in un array di righe (split del SDP sui caratteri "\n\r") e si ricerca la linea che descriva il media di interesse (nel nostro caso l'audio); essa è riconoscibile in quanto presenta all'inizio la stringa **"m=audio"**, ad esempio

```
m=audio 58779 UDP/TLS/RTP/SAVPF 111 103 104 9 0 8 106 105 13 126
```

Essa racchiude una serie di informazioni sugli attributi del media dello stream. In particolare i valori *111 103 104 9 0 8 106 105 13 126* rappresentano ognuno il payload di un determinato codec e il loro ordine determina

quale abbia la maggior priorità di utilizzo. Accertata perciò l'esistenza di questa linea è necessario conoscere il valore del payload del codec scelto. Per farlo si procede a ricercarlo sulle linee adibite alla descrizione dei codec; esse sono riconoscibili in quanto aventi stringa iniziale uguale a **"a=rtpmap"**, seguite dal nome del relativo codec, i cui possibili valori sono i seguenti:

- **opus**
- **ISAC**
- **G722**
- **PCMU**
- **PCMA**

Di seguito un esempio di questa linea nel caso del codec OPUS:

```
a=rtpmap:111 opus/48000/2
```

Trovata quindi la linea corretta e il relativo valore di payload si procede con il cambiare l'ordine dei valori della *m line*, ordinandoli in modo tale che il primo sia quello del payload trovato. Questo cambiamento avrà come effetto quello di forzare il peer remoto a utilizzare il codec scelto dal client[7].

5.4.1 Le opzioni di OPUS

Nel caso specifico del codec OPUS è possibile indicare ulteriori opzioni. Nella soluzione sviluppata il client può attivare o modificare le seguenti opzioni[13]:

- audio su canale **stereo** o **mono**
- **sprop-stereo**, cioè abilitare o meno il Two-way full band stereo
- **cbr** o *Constant Bit Rate*

[7] https://github.com/webrtc/apprtc/blob/master/src/web_app/js/sdputils.js#L286-L327

- **use inband fec**, questa opzione permette all'encoder di usare il FEC (o *Forward Error Correction*, cioè il meccanismo di rilevazione e correzione degli errori) anche se la decisione finale di usarlo o meno è dato dal bitrate e dalla percentuale di paccetti persi (e altri fattori minori).
- **use dtx** o *Discontinuous Transmission* riduce il bitrate durante i silenzi o i rumori di sottofondo. Se abilitato solo 1 frame è codificato ogni 400 millisecondi
- **max average bitrate** o valore massimo di bitrate medio
- **max playback rate** che dà un indicazione del valore di sampling in output che il ricevitore è capacace di renderizzare, misurato in Hz.
- **max p-time** indica il tempo massimo di decodifica in millisecondi rappresentato dall'audio in un pacchetto che può essere incampusalto in un pacchetto ricevuto come indicato nella sezione 6 del RFC4566 [2], di default il valore è 120 mentre la sezione 4 e 5 del documento indica che è possibile usare i valori 3, 5, 10, 20, 40 e 60 o un multiplo del frame size di Opus.

Il valore di queste opzioni deve essere considerato dopo la modifica del pacchetto SDP vista nella sezione 5.4. Se è stato scelto OPUS come codec principale allora è necessario modificare ulteriormente il pacchetto. Si procede nuovamente alla ricerca della linea **a=rtpmap** del codec OPUS e si recupera il payload di quest ultimo. Il payload è necessario per la ricerca di un'ulteriore linea, la **a=fmtp**. Questa linea è quella che include i parametri opzionali del codec.
Si crea quindi una stringa, concatenando tutte le varie opzioni che sono state indicate (stereo=; sprop-stereo=; maxaveragebitrate=; maxaveragebitrate; cbr=; useinbandfec=; usedtx=;), nel caso si sia indicato anche un valore per il max p-time è necessario aggiungere una nuova riga **a=maxptime:** seguita dal valore indicato. La stringa quindi ottenuta è poi concatenata alla *a=fmtp line* esistente.

Di seguito un esempio di *a=fmtp line* mentre nel RFC7587 è possibile trovare altri esempi di modifica del SDP.[11]

```
a=fmtp:111 stereo=1; useinbandfec=1; cbr=1
```

5.5 Grafici e misurazioni

La rilevazione delle varie misurazioni è fatta lato client. Analizziamo ora quali misure vengono eseguite e come esse sono rilevate.
La prima misura rilevata è relativa al tempo totale di connessione al server. Esso è misurato da quando l'utente preme il tasto **CONNECT** a quando il client è abilitato a inviare comandi al server, cioè all'apertura del DataChannel dopo scatenazione dell'evento *onopen*.

Una seconda misurazione è data quando il client preme il bottone **"Connection Latency"**, il quale restituisce le latenze della connessione col server a livello HTTP (http ping), di invio e ricezione dei comandi del DataChannel. Per ottenere la latenza di connessione il client rileva il tempo di pressione del click con il metodo *window.performance.now()*. Il metodo performance.now() è più preciso di diversi ordini di grandezza rispetto a Date.now() ed è relativo a quando la pagina è stata caricata; è utile quando è necessaria un'alta risoluzione del tempo ma può essere utilizzato solo se i tempi sono stati tutti rilevati sul client.
Effettua quindi un http ping al server, cioè una **XMLHttpRequest** con metodo HEAD; nella callback di successo si effettua un nuovo *performance.now()* e sottratto al valore precedentemente preso ricavando in questo modo la latenza della risposta.
Sempre nella callback viene inviato sul DataChannel, per richiedere la latenza dei comandi, un messaggio di tipo **ping** avente la data attuale in millisecondi (metodo *new Date().getTime()*). Il server quindi alla ricezione di questo messaggio risponde con uno di tipo **pong** contentente il tempo in millisecondi di quando ha ricevuto il comando e il tempo, sempre in millisecondi, di quando ha inviato il messaggio di risposta. Il client quindi, alla ricezione di questo messaggio effettua una differenza dei tempi contenuti nel messaggio per ricavare la latenza di upload e download dei comandi.

Tramite il DataChannel il client riceve anche altri messaggi che utilizza per rilevare ulteriori misurazioni; i base al tipo di messaggio ricevuto è possibile rilevare:

1. il tempo che il server ha impiegato per renderizzare una traccia richiesta dal client e aggiungerla alla connessione, attraverso il messaggio di tipo **audioComputation**

2. il tempo di quanto il server ha impiegato per aumentare il peso di un canale del mixer dopo la pressione del bottone *"MIXER 1° CH_UP"*

attravero il messaggio di tipo **mixer_up**

3. messaggi di report; questi report permettono al client di visualizzare i dati relativi alla bandwidth e a disegnare i grafici relativi ai pacchetti ricevuti e bitrate in funzione del tempo.

Le misurazioni sicuramente più importanti del progetto riguardano la latenza totale di una traccia audio e la latenza del comando di stop. Per l'ultimo caso la rilevazione è abbastanza semplice, in quanto viene effettuata usando come tempo di inizio quello di pressione del relativo bottone, mentre come tempo di fine il tempo rilevato al verificarsi dell'evento di tipo **onremovetrack** dell'oggetto stream in riproduzione in quel momento.
La latenza totale della traccia invece richiede l'utilizzo di un particolare oggetto delle WebAudio API, l'**analyser**. L'analyser rappresenta un nodo capace di provvedere dati sia nel dominio della frequenza che nel dominio del tempo. Non cambia l'audio stream in input ma permette di raccogliere i dati generati per processarli e utilizzarli nella visualizzazione di grafici; il suo funzionamento è schematizzato in figura 5.8. Ha un punto di input e uno di output e funziona anche se il nodo di output non viene collegato come nella soluzione sviluppata, in quanto la traccia audio di ingresso è collegata direttamente agli speaker del client.
Quando sul client si verifica l'evento **ontrack** viene creato un nuovo oggetto di tipo *MediaStreamSourceAudioNode* dallo stream ricevuto; questo nodo è collegato in input all'analyser. In base alla scelta effettuata dal client sul tipo di grafico da visualizzare della traccia (se nel dominio del tempo o della frequenza) il metodo da richiamare dell'analyzer sarà il *getByteTimeDomainData()* o il *getByteFrequencyData()*. Entrambi i metodi ricevono come parametro un unsigned byte array dove salveranno i dati rilevati.

Questo array oltre ad essere utilizzato nel disegnare un grafico ad oscilloscopio o a bande di frequenza è necessario per rilevare quando il client inizia a ricevere l'audio della traccia.
L'algoritmo utilizzato consiste nel calcolare il valor medio di questo array (come media di tutti i valori contenuti), se questo valore è diverso da 0 (nel dominio della frequenza) o 128 (nel dominio del tempo, in quanto essendo un byte di questo array un'informazione relativa alla forma d'onda dello stream, il valore medio tra il minimo, cioè 0, e il massimo, cioè 256, rappresenta una forma d'onda piatta) allora il client ha iniziato a ricevere i primi pacchetti audio dal server ed è possibile utilizzare il tempo rilevato per calcolare la latenza della traccia audio.

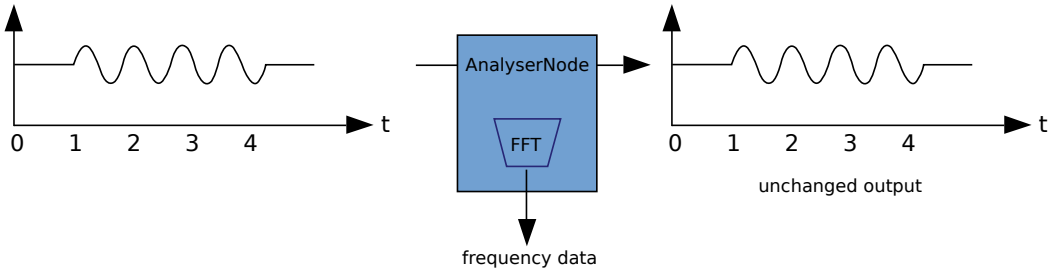

Figura 5.8: Rappresentazione dell'oggetto analyser

Fonte: https://developer.mozilla.org/en-US/docs/Web/API/AnalyserNode

Il client riceve attraverso il canale WebSocket anche misurazioni secondarie relative alla latenza dei messaggi scambiati durante il setup del canale WebRTC, come la latenza dell'offer e dell'answer o degli ICE candidate, ma che nella soluzione sviluppata sono utilizzati unicamente a scopo informativo.

Durante la riproduzione di una traccia (dopo l'evento **ontrack**), tramite l'ausilio della libreria **getStats.js**[8] è stato possibile rilevare maggiori informazioni sullo stato della connessione WebRTC come utilizzo di banda, indirizzi IP locale e remoto con relative porte, tipo di connessione ecc... Questi dati sono poi visualizzati nella relativa tabella presente nella pagina del client.

[8] https://github.com/muaz-khan/getStats

Parte II

Risultati e considerazioni finali

Capitolo 6

Test e risultati

Lo scopo di questo capitolo sarà quello di descrivere le diverse condizioni e le configurazioni utilizzate per raccogliere le misurazioni della latenza e la successiva rappresentazione di questi dati in grafici per poter avere un confronto diretto tra le varie prove effettuate.
Verranno descritti i parametri coinvolti nei vari casi di test, come essi sono stati effettuati e infine illustreremo i risultati ottenuti.

6.1 I parametri studiati

La rilevazione delle latenze non è stata effettuata solamente per un particolare caso; nell'implementare la soluzione descritta in questo lavoro di tesi si è cercato di inserire e utilizzare vari parametri che potessero in qualche modo influire sulla latenza nelle varie fasi della comunicazione client-server.
Il primo e il più importante parametro su cui è stata posta attenzione è stata la tecnologia utilizzata nell'implementazione del server. Utilizzare infatti NW.js o Selenium/Puppeteer ha differenziato non solo il codice per sviluppare l'una o l'altra soluzione ma anche come il collegamento tra il client e il server debba essere effettuato:

- nel primo caso ci troviamo in presenza di un collegamento quasi diretto tra client e server in quanto, su quest'ultimo, sia la componente WebSocket che quella WebRTC si trovano nello stesso ambiente e capaci di comunicare quasi immediatamente tra di loro. Questo potrebbe rivelarsi vantaggioso quando la componente WebRTC del server è ancora in fase di configurazione e deve mandare i suoi messaggi di setup sul canale

WebSocket; una volta però finita questa fase il peer WebRTC non utilizza più il canale WebSocket avendo instaurato un collegamento diretto con il client. L'utilizzo quindi di un framework che integra al suo interno dei meccanismi per utilizzare moduli Node e librerie del DOM può introdurre dei ritardi durante la comunicazione WebRTC.

- nel caso Headless invece ci ritroviamo con un ambiente diviso, dove la componente WebSocket è offerta da un WebServer e la componente WebRTC è localizzata in un browser (seppur senza interfaccia grafica). Ciò implica che i messaggi WebSocket di configurazione dei peer WebRTC debbano passare da due collegamenti distinti, il primo tra il client remoto e il server WebSocket e il secondo tra il server WebSocket e il peer WebRTC presente sul server. Ciò potrebbe causare dei ritardi durante l'instaurazione del collegamento e l'apertura del DataChannel rispetto ad NW.js ma una volta che tutti i messaggi di configurazione sono stati inviati il collegamento è diretto tra i due browser, senza passare da nessun altro componente intermedio.

Il resto delle opzioni invece sono configurabili e controllate sul client. Le più semplici indicano come la traccia debba essere caricata sul server e come debba eventualmente sostituita. In altre parole il client può indicare al server se la traccia che sta per richiedere può essere caricata dalla memoria, e quindi con notevole riduzione dell'attesa di ricezione della stessa oppure se debba essere ricaricata in RAM dal file system, il che, in base alla tenologia del server usata, può dilatare anche significativamente i tempi di ricezione; con la seconda opzione invece indica al server, se sta già trasmettendo una traccia, come debba essere sostituita, se con una rimozione della vecchia traccia e successiva aggiunta della nuova o con il metodo *replace()* di WebRTC; è stato quindi scelto di utilizzare questo parametro proprio per analizzare se dovesse esserci una differenza in termini di latenza tra i due metodi e quindi preferire una soluzione all'altra.
Un altro parametro ritenuto molto importante in questo studio è il tipo di codec utilizzato nella comunicazione; le possibilità offerte all'utente sono **OPUS, ISAC 16K, G722, PCMU e PCMA**. La scelta del codec infatti può influenzare la latenza audio, in quanto un codec implica il tipo di protocollo utilizzato a livello di rete per la trasmissione e quanti campioni vengono inviati; il loro confronto quindi si rivela fondamentale per la scelta della configurazione di rete migliore. Elenchiamo alcune caratteristiche di questi codec:

- ISAC è un codec sviluppato dalla **Global IP Solutions** per applicazioni VOIP e streaming audio. I blocchi codificati sono incapsulati in pacchetti RTF (Real-time Transport Protocol). La frequenza di campionamento è 16 KHz (quella usata nel progetto) e 32 KHz, ha un bitrate variabile tra 10 e 32 kbit/s o 10 e 52 kbit/s nel caso di campionamento a 32 KHz. I pacchetti possono contenere dati audio variabili da 30 ms a 60 ms.[3]

- G722 è un codec approvato dalla ITU-T (International Telecommunication Union – Telecommunication Standardization Bureau) a 7 KHz, con larghezza di banda di 48, 56 e 64 kbit/s. Utilizza una frequenza di campionamento a 16 KHz, rendendolo migliore come qualità a quella telefonica. Nonostante le diverse larghezze di banda possibili, in pratica si utilizza sempre la codifica a 64 kbit/s.

- PCMU e PCMA sono entrambi implementazioni del codec G.711, un codec basato su PCM (pulse-code modulation o modulazione a codice di impulsi) molto utilizzato nella telefonia. L'audio è codificato a 8 bit per campione, con frequenza di campionamento a 8 KHz. I livelli di quantizzazione non sono equidistanti ma seguono uno scaling logaritmico che migliora il rapporto segnale/rumore. La differenza nei due è proprio nello scaling, PCMU utilizza mu-law scaling (utilizzato in America e Giappone), PCMA A-law scaling (resto del mondo).[10] Utilizzano anche loro RTP come protocollo dei pacchetti.

- OPUS è un formato aperto e senza licenza commerciale, di recente sviluppo; è stato sviluppato per essere adatto alle trasmissioni in tempo reale, supporta velocità di trasmissione che vanno da 6 kbit/s a 510 kbit/s e frequenze di campionamento da 8 a 48 KHz (quest'ultima è quella utilizzata di defualt da WebRTC). Presenta un ritardo di soli 20 ms; in figura 6.1 è possibile vedere come OPUS offra il miglior rapporto tra latenza e qualità. Inoltre nel progetto sviluppato è l'unico codec che permette di specificare in fase di instaurazione del collegamento WebRTC ulteriore opzioni di trasmissione, come già visto nella sottosezione 5.4.1

L'ultima opzione che è stata considerata nello studio effettuato è la scelta della tipologia di trasporto per WebRTC: se utilizzare server STUN o utilizzare esclusivamente come modalità di collegamento quella relay dei server TURN.
Ovviamente l'utilizzo dei server TURN fa aumentare in modo significativo la latenza della connessione, ma è stato scelto di studiare anche questo parametro in quanto potrebbero esserci configurazioni per cui il server e il client

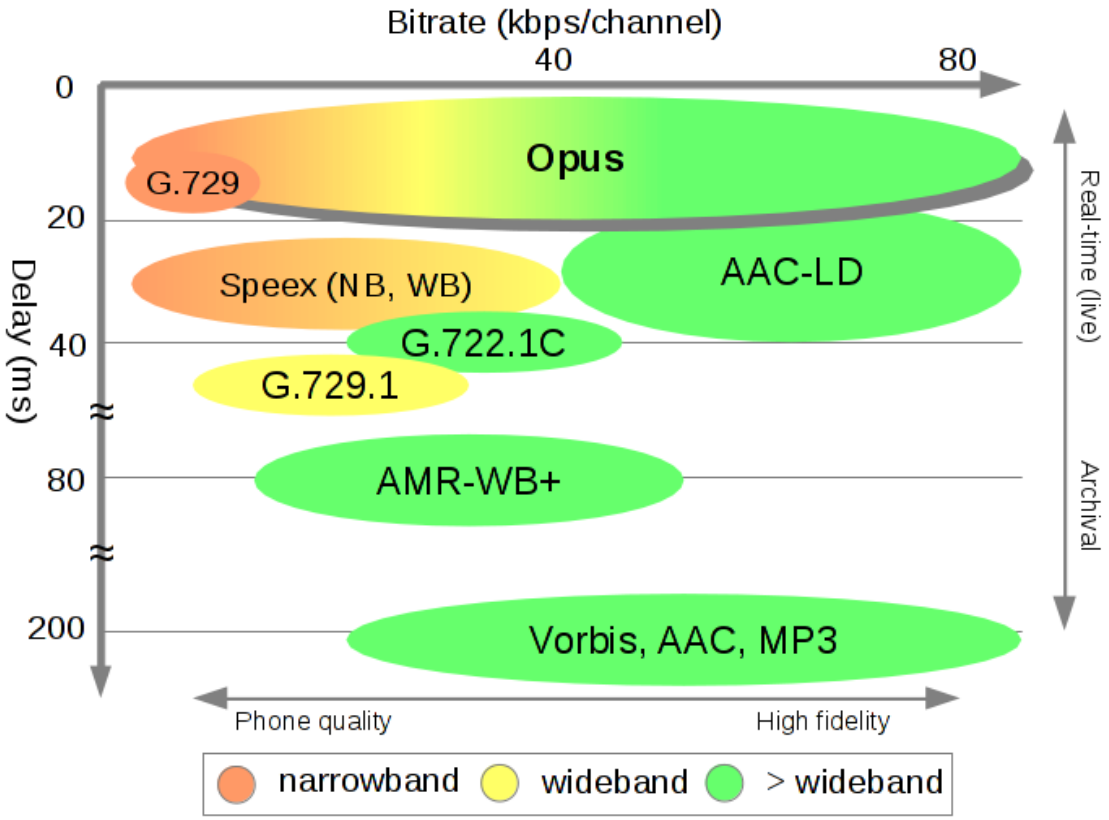

Figura 6.1: Confronto in termini di bitrate/qualità tra OPUS e altri codec

Fonte: http://opus-codec.org/static/comparison

siano impossibilitati a comunicare in maniera diretta tra loto, ad esempio a causa di un firewall oppure, come fatto in questo progetto, se il server utilizza dei container docker collocati in una rete interna inaccessibile dall'esterno a meno delle poche porte esposte in fase di creazione del container stesso.

6.2 Struttura dei test

Per raccogliere misurazioni provenienti da un numero sufficiente di combinazioni dei vari parametri descritti, è stato utilizzato ancora un WebDriver (in questo caso Selenium ma la scelta è indifferente) per automatizzare le azioni nel browser del client e permettere quindi la stesura di vari casi di test, raccogliendo poi le relative misurazioni.
È stato utilzzato anche in questo caso un server Express che mettesse a disposizione un endpoint da cui scaricare la pagina HTML del client e i relativi

sorgenti JavaScript. In questo modo è stato possibile utilizzare il WebDriver per collegare il browser all'endpoint del server e poter quindi compiere azioni sulla pagina HTML ricevuta.
Le latenze che è possibile misurare sul client sono le seguenti:

- **connection**, il tempo impiegato da client e server per configurare la connessione WebRTC e poter quindi iniziare ad inviare comandi al server
- **command**, suddivise in latenza di HTTP ping (la latenza di connessione a livello HTTP), latenza di upload (il tempo impiegato da un comando per arrivare dal client al server) e latenza di download (latenza del comando dal server al client); questi comandi sono i messaggi scambiati dai peer tramite il DataChannel.
- **song**, il tempo cioè impiegato da quando il client preme il comando di ricezione di una traccia a quando effettivamente inizia a ricevere i primi dati audio non nulli.
- **stop**, la latenza tra la pressione del bottone di STOP sul client e l'effettiva assenza del suono.
- **mixer**, il tempo trascorso da quando si preme il bottone dell'aumento di guadagno sul primo canale del mixer a quando il server effettua questa operazione.

Tutti i test effettuati hanno una test suite comune di base, che verrà poi richiamata di volta in volta cambiando le diverse opzioni disponibili al client. Questa test suite è così strutturata:

1. Si inizializza un array che conterrà le diverse latenze che il client può calcolare (latenza totale audio, latenza connessione, latenza invio comandi...) e si decide quante volte questa suite debba essere ripetuta (che determinerà poi la lunghezza di questo array); nel nostro progetto è stato scelto di ripetere la suite 3 volte.
2. Ad ogni ripetizione si effettua il collegamento col server e si attende che il DataChannel passi in stato **open** per poter inviare un comando (questo controllo può essere fatto aspettando che uno dei bottoni relativo alla richiesta di una traccia torni attivo); si recupera quindi dal codice JavaScript le misurazioni relative alla connessione (Selenium permette di accedere al codice JavaScript del browser e ai relativi oggetti utilizzando la funzione asincrona *Page.evaluate()* che può ritornare un oggetto JavaScript, a patto che questo sia serializzabile).

3. Si richiede quindi la prima traccia al server, si attende circa 4 secondi per essere sicuri che sia stata ricevuta e sia in esecuzione e si richiede quindi la seconda traccia, attendendo qualche secondo per essere sicuri di averla ricevuta. In questo modo le misure delle latenze ottenute dopo queste operazioni metteranno in evidenza la differenza in termini di tempo delle diverse modalità di sostituzione della traccia, come visto nella sottosezione 5.3.2.

4. Si preme quindi il tasto STOP e si attende un secondo per far in modo che il client calcoli il tempo impegato dall'operazione e si recupera quindi questa misurazione.

5. Si preme il tasto MIXER e si attende qualche secondo per la ricezione della traccia combinata e poi si preme il bottone per aumentare il guadagno del primo canale. Dopo un secondo possiamo recuperare il tempo impegato da quando abbiamo premuto questo bottone a quando effettivamente il server ha completato l'operazione di rendering.

6. Concluse tutte le ripetizioni indicate, si calcola quindi la media di tutte le misurazioni rilevate ad ogni ciclo. L'oggetto quindi risultate verrà poi passato alla test suite superiore.

Questa suite base viene poi richiamata di volta in volta cambiando le varie opzioni e per poter quindi raccogliere i dati sulle latenze ottenute nei diversi casi.
Tutte le varie combinazioni sono strutturate in una gerarchia, la quale permette di costruire un oggetto JavaScript che, ad ogni livello, identifica le opzioni utilizzate che hanno portato ad una determinata serie di misure. La prima opzione che si può modificare è quale server contattare, se quello situato sulla macchina locale o il server remoto. Per ognuna di queste scelte si decide poi il tipo di architettura del server, se NW.js, Selenium o Puppeteer. Si procede quindi con la suite che si occupa di eseguire i test base utilizzando i server STUN o forzando la connessione ad utilizzare i server TURN. Per entrambi i casi si procede con la scelta del codec, seguito dalla raccolta delle misurazioni per ISAC, G.722, PCMU, PCMA; OPUS ha una suite differente che descriveremo in seguito.
Per ogni codec si indica se il server debba utilizzare le tracce contenute nel suo buffer in memoria o ricaricarle tutte dal file system. Per entrambi i casi si sceglie il tipo di sostituzione delle tracce. Solo a questo punto è possibile quindi procedere con l'esecuzione della suite base che raccoglierà quindi le varie misurazioni e le restituirà alla prima funzione chiamante risalendo man

mano la gerarchia di chiamate.
OPUS ha a disposizione una suite test diversa dagli altri codec, in quanto in questo caso è possibile combinare dei parametri aggiuntivi. Si è deciso quindi, tra tutte le combinazioni possibili delle varie opzioni queste configurazioni:

1. attivare le opzioni **STEREO, CBR, INBAND FEC, DTX**
2. attivare solo **STEREO, DTX**
3. attivare solo **CBR, DTX**
4. non attivarne nessuna

L'oggetto finale, risultato della combinazione di tutte le opzioni descritte prima e delle relative misurazioni è poi salvato su un file JSON. In questo modo è possibile successivamente utilizzare questo file per la stesura dei grafici.

6.3 Risultati ottenuti

Per avere un quadro completo dei risultati ottenuti sono stati creati dei grafici che mettessero a confronto tutte le varie combinazioni create, utilizzando la libreria Python **matplotlib**[1]. Le misurazioni descritte precedentemente sono state raggruppate in due famigie di grafici: quelli relativi alle latenze di rete e quelli relativi alle latenze dell'audio. Vediamo ora nel dettaglio queste famiglie e i relativi risultati.

6.3.1 I grafici sulla rete

La famiglia dei grafici che riguardano le latenze di rete raggruppano e mettono a confronto le misure relative alla latenza totale della connessione, la latenza dei comandi e la latenza di connessione a livello HTTP. L'unico fattore che può influenzare la latenza in questo caso è il tipo di connessione utilizzata, se attraverso server STUN o forzandola ad usare server TURN. In figura 6.2 sono messi a confronto i grafici delle misurazioni in caso di server locale, mentre in figura 6.3 abbiamo i grafici delle latenze di rete in caso di server remoto.

[1] https://matplotlib.org/

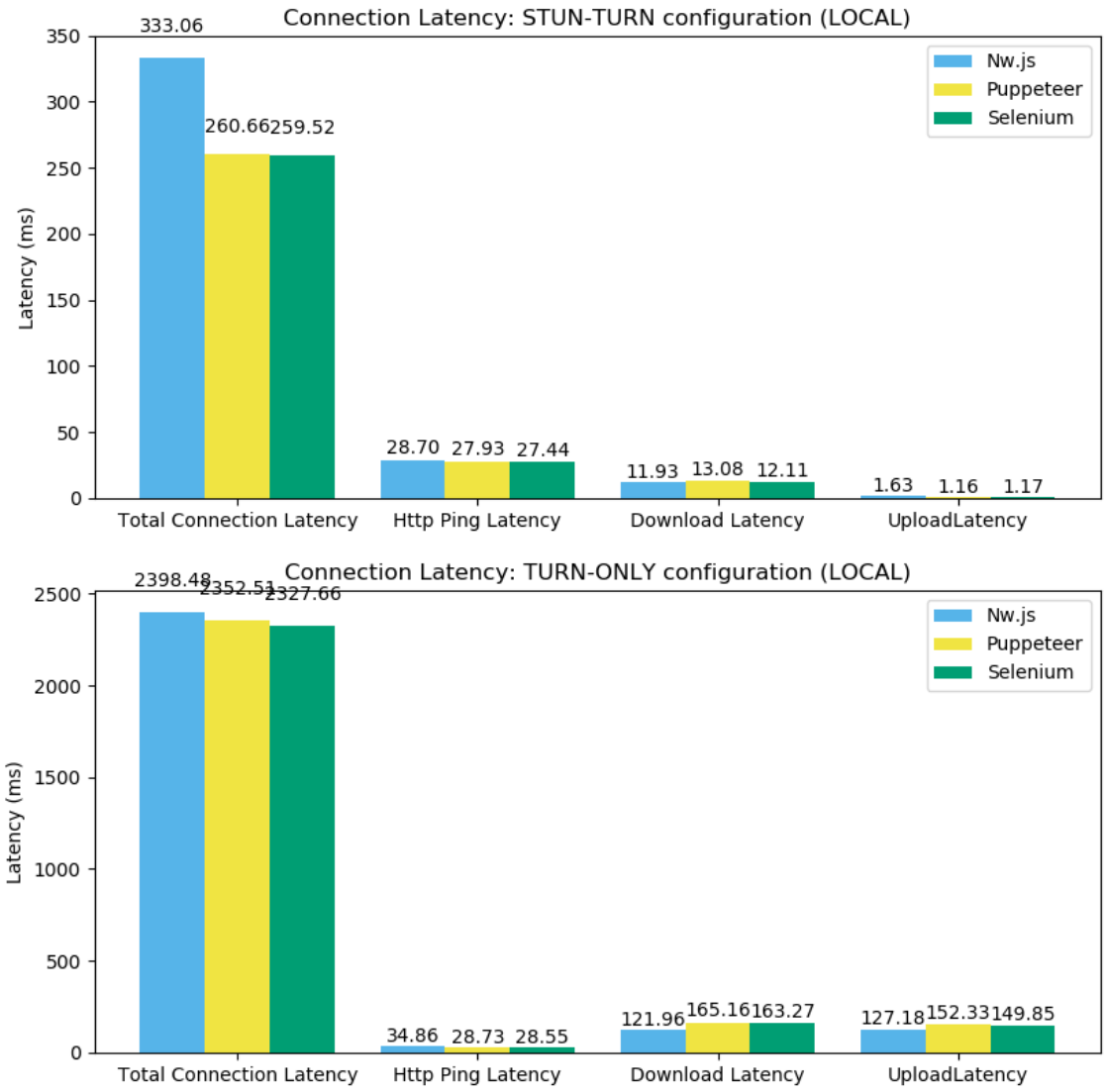

Figura 6.2: Confronto delle latenze di rete in ambiente locale in caso di utilizzo di server STUN (sopra) o di soli server TURN (sotto)

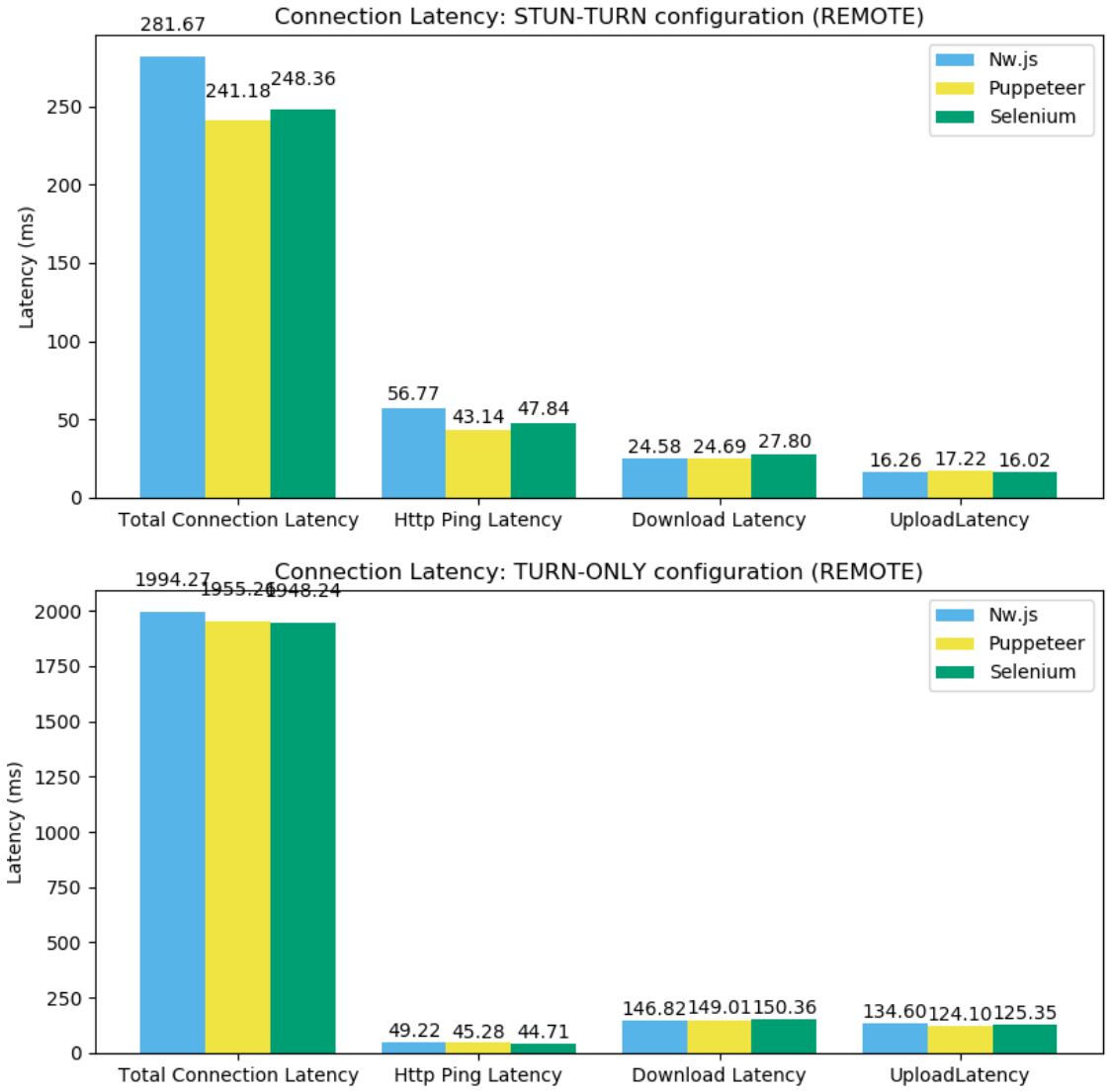

Figura 6.3: Confronto delle latenze di rete con server remoto in caso di utilizzo di server STUN (sopra) o di soli server TURN (sotto)

Dai grafici possiamo evincere chiaramente che a parità di latenza di rete (HTTP ping), i tempi delle latenze che riguardano WebRTC crescono considerevolmente in caso di affidamento ai server di relay, in quanto non si avrebbe più una connessione diretta tra client e server. Per quanto possibile quindi, se si vuole ridurre al minimo la latenza a livello di rete bisogna assicurarsi di non avere impedimenti tali da rendere necessaria l'azione dei server TURN (ad esempio un firewall che blocca la comunicazione WebRTC).
Se utilizziamo docker, come già detto, è molto probabile invece che ci troveremmo costretti ad affidarci ai server TURN in quanto i container sono lanciati in una rete interna non accessibile dall'esterno. Quanto detto è dimostrato se analizziamo ad esempio la cattura di alcuni pacchetti scambiati tra server e client durante la trasmissione di una traccia audio (figura 6.4): nel primo caso sono riportati i pacchetti inviati dal server in caso di connessione diretta e si può notare che l'audio codificato è inviato direttamente al client sottoforma di pacchetti UDP (utilizzando come subprotocollo RTP); nel secondo caso invece (utilizzando docker) i pacchetti non sono più diretti tra client e server ma devono essere incapsulati in pacchetti STUN scambiati tra i server TURN che contengono i dati della traccia audio codificata e le informazioni relative ai peer WebRTC remoti.

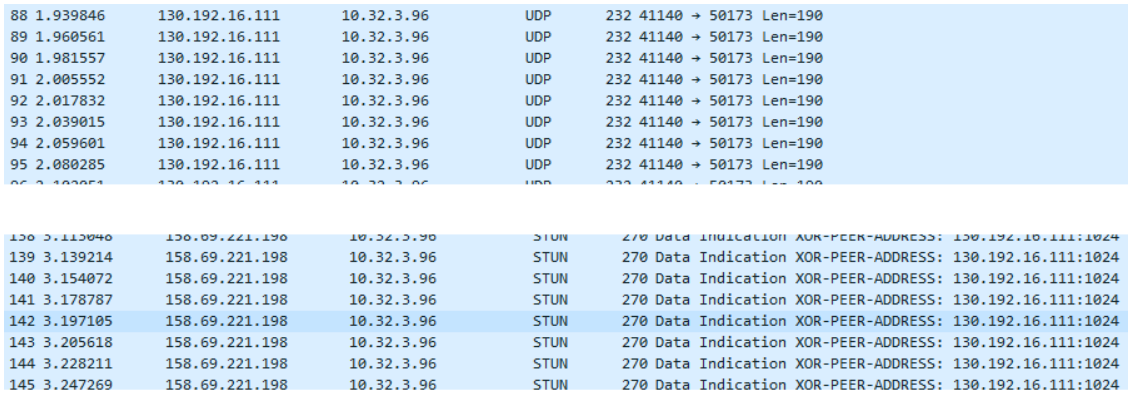

Figura 6.4: Analisi dei pacchetti inviati dal server al client in caso di connessione diretta (sopra) e connessione dietro un container docker (sotto)

6.3.2 I grafici dell'audio

Sicuramente i grafici più importanti sono quelli riguardanti le latenze delle tracce audio, in quanto obiettivo principale di questo lavoro di tesi. In questo caso si è proceduto a separare i grafici con le misure prese utilizzando il codec

OPUS da quelle prese con il resto dei codec, in modo da poter vedere in modo chiaro quanto cambi l'utilizzo di un codec rispetto ad un altro, e nel caso di OPUS quanto influenzino le opzioni di questo codec.
Iniziamo analizzando i grafici relativi ai codec ISAC, G.722, PCMU e PCMA; anche in questo caso sono riportati quelli riferiti in ambiente locale (figura 6.5) che in ambiente remoto (figura 6.6). Non verranno riportati in questo documento tutti i grafici ottenuti; si è deciso di riportare solo quelli le cui misure si riferiscono a tracce già caricate in memoria e non si faccia ricorso ai server TURN. Ogni figura riporta quattro settori, in base alle diverse tipologie di latenze studiate: il primo settore si riferisce alla latenza di stop, il secondo alla latenza totale dell'audio, il terzo alla latenza di rendering audio e il quarto alla latenza del mixer. Inoltre nel riquadro della latenza totale dell'audio è possibile avere anche un confronto tra le due modalità di cambio della traccia disponibili.
Possiamo notare come i codec della famiglia G.722 e G.711 siano quelli che presentano minore latenza audio sia per quanto riguarda la latenza totale che quella riguardante il rendering dell'audio o di modifica del mixer.
Possiamo anche notare, come già enunciato in precedenza, che la soluzione NW.js comporta quasi sempre un ritardo aggiuntivo soprattutto nelle latenze che rigurdano il rendering audio in quanto esso non viene eseguito in un vero browser ma in un framework che ne ingloba le funzionalità.
Un altro particolare lo possiamo vedere nella differenza tra i tempi di latenza totale dell'audio nel caso di sostituzione traccia vista come operazione di rimozione della vecchia e aggiunta della nuova oppure vista come effetto del metodo *replaceTrack()* di WebRTC (le barre tratteggiate), dimostrando proprio come l'utilizzo di quest'ultimo metodo riduca di molto la latenza se si richiede una nuova traccia.

I grafici audio di OPUS

Il codec OPUS è stato analizzato con grafici dedicati data la possibilità di settare parametri aggiuntivi; è stato possibile (scelti solo una piccola combinazioni di questi) poterli confrontare tra di loro e vedere quindi la miglior configurazione possibile. Anche in questo caso ci limiteremo ad illustrare solo alcuni grafici, in particolare quelli riguardanti l'ambiente locale (figura 6.7) e l'ambiente remoto (figura 6.8) solo per tracce già caricate in memoria e in configurazione STUN (in modo da poter anche avere un confronto con i grafici precedenti redatti nelle medesime condizioni). I grafici sono divisi nei quattro settori già descritti nella precedente sezione, con la differenza che il

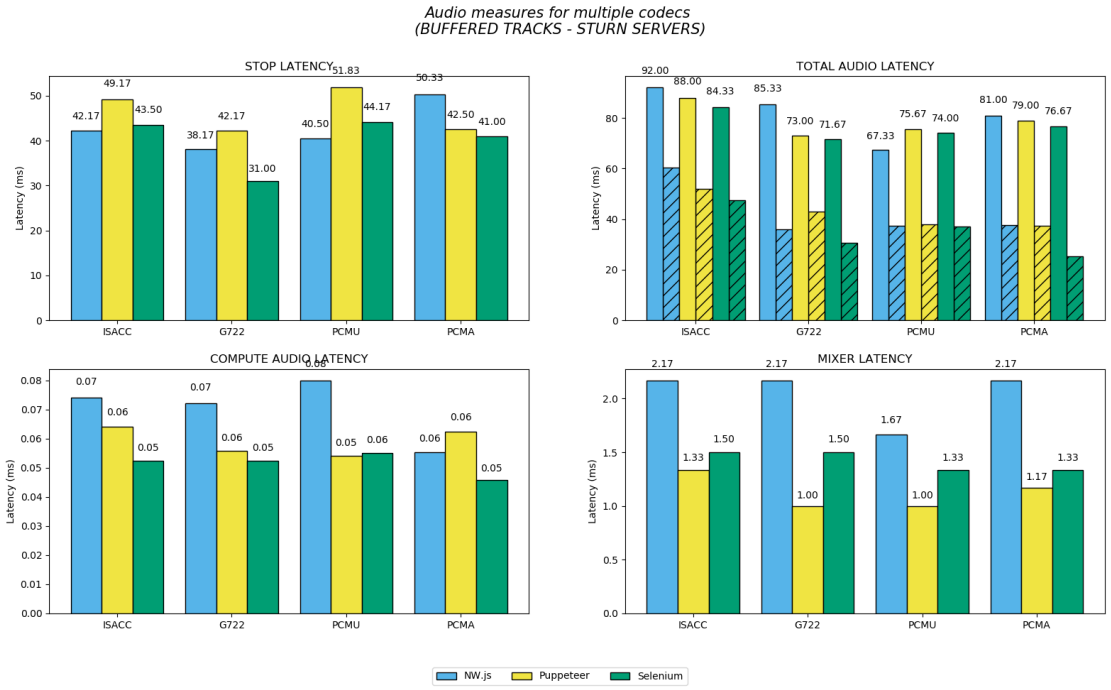

Figura 6.5: Grafici latenze audio in ambiente locale su ISAC, G.722, PCMU e PCMA. Nelle latenze totali dell'audio è possibile vedere anche la differenza in termini di tempo tra i due metodi di sostituzione delle tracce

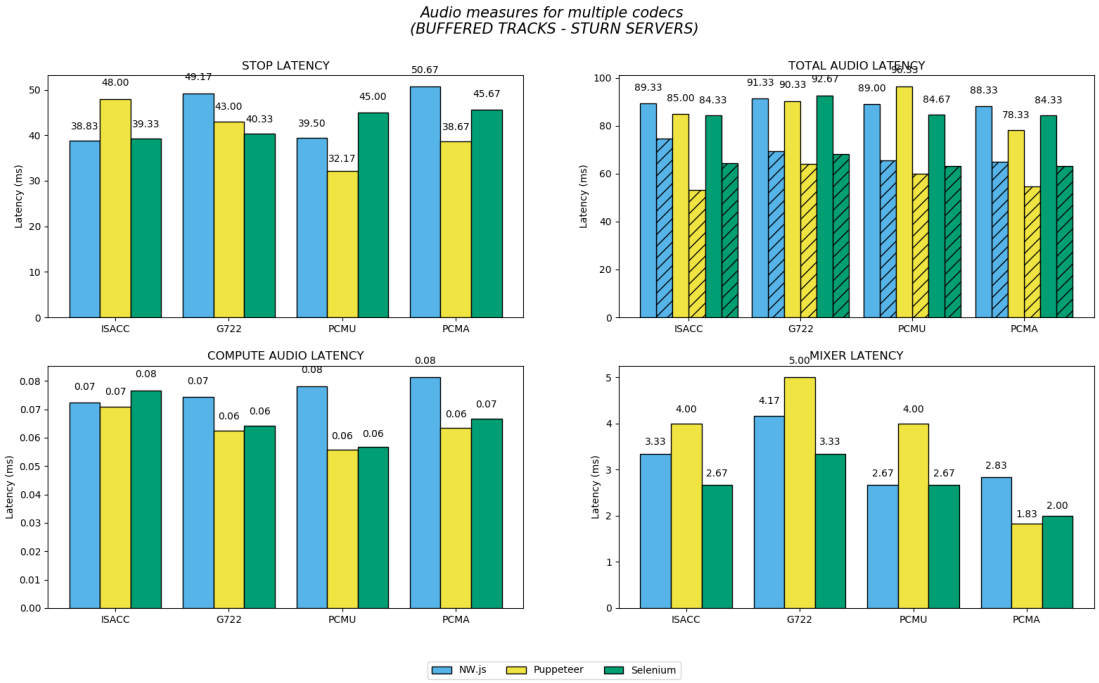

Figura 6.6: Grafici latenze audio in ambiente remoto su ISAC, G.722, PCMU e PCMA. Come nel caso locale, nel riquadro della latenza totale dell'audio è possibile notare la differenza di latenze dei due metodi di cambio traccia

confronto è fatto sulla base delle opzioni di OPUS attivate.
Ciò che possiamo notare è che OPUS dà le migliori prestazioni attivando tutte le opzioni: STEREO, DTX, CBR e INBAND FEC, come dimostrato dai riquadri delle latenze di STOP e di quella totale. Per quanto riguarda invece il rendering vero e proprio la differenza con gli altri codec è abbastanza trascurabile. In più, se utilizziamo il metodo di replace della traccia attraverso il metodo *replaceTrack()* di WebRTC , OPUS raggiunge latenze di molto inferiori a quelle ottenute con tutti gli altri codec, rendendolo quindi la miglior scelta per WebRTC (essendo anche concepito per questo scopo) e , di conseguenza, per la soluzione da noi sviluppata; viceversa si ottengono misure di latenza simili ai codec G.722 e G.711 visti prima.

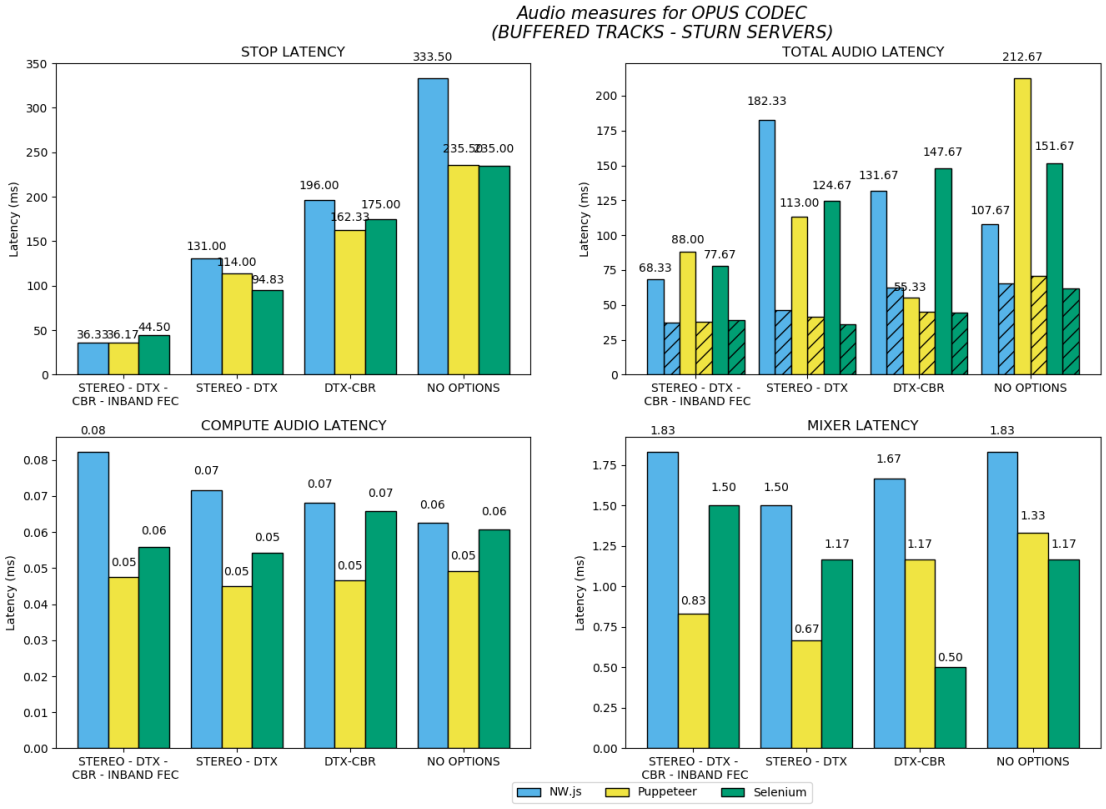

Figura 6.7: Grafici latenze audio in ambiente locale per codec OPUS. Ogni riquadro confronta le latenze nel caso di attivazione o meno di determinati parametri del codec

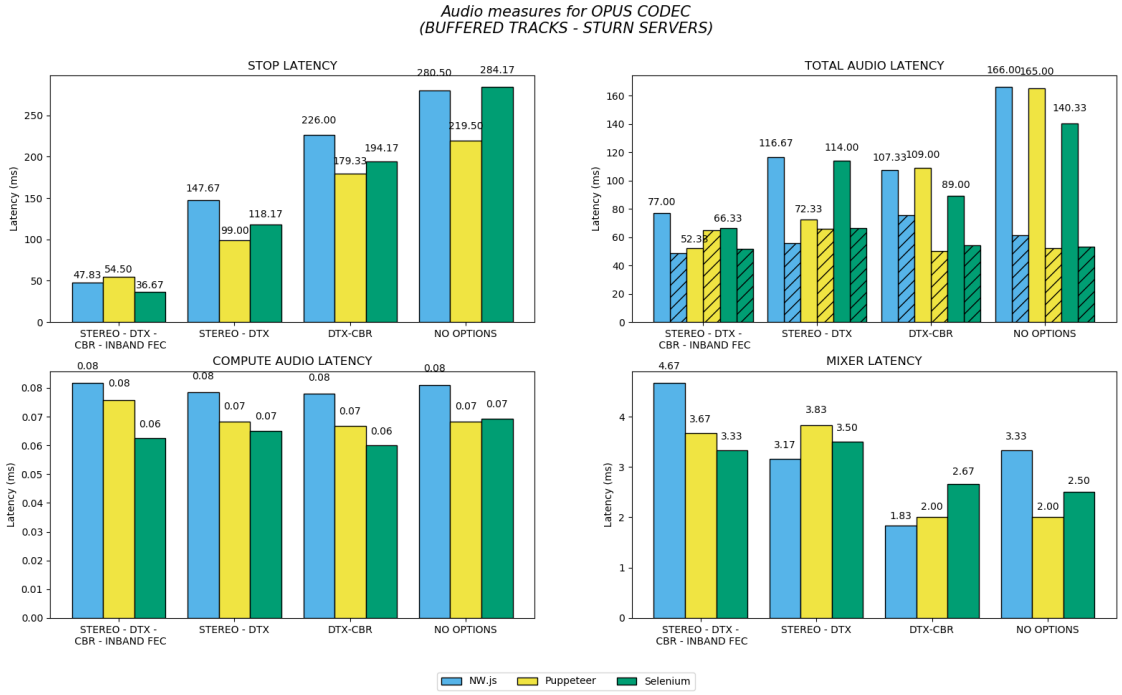

Figura 6.8: Grafici latenze audio in ambiente remoto per codec OPUS

Capitolo 7

Considerazioni finali

Dopo aver confrontato, anche con l'ausilio dei grafici, i tempi delle varie configurazioni che è possibile ottenere nel progetto possiamo ora stilare le giuste conclusioni.
Abbiamo visto quindi che la miglior combinazione per poter ottenere la latenza minima nella connessione client-server si basa sui seguenti punti:

1. Anche se più difficile da implementare, è consigliabile utilizzare una soluzione Headless (in questo progetto sviluppata con Selenium/Puppeteer) che permette di utilizzare le librerie e i metodi del DOM direttamente da un browser reale, anche senza la sua componente grafica (il che lo rende di facile distribuzione su una macchina server). Se invece si vuole utilizzare NW.js per poter sviluppare tutte le componenti necessarie in un unica soluzione ne consegue un guadagno in termini di semplicità ma a costo di una maggiore latenza.

2. Utilizzare come codec OPUS con le sue opzioni di defualt, con le quali offre le migliori performance in termini di qualità/latenza. Gli altri codec visti rasentano latenze simili ma non offrono sicuramente la stessa qualità audio data da OPUS; anche perché quest'ultimo è di più recente creazione e sviluppato appositamente per connessioni real-time.

3. Si possono ottenere latenze minime nel cambio di una traccia ad una successiva utilizzando il metodo *replaceTrack()* offerto da WebRTC.

4. Evitare per quanto possibile l'affidamento della connessione ai server di relay (TURN) in quanto il loro impiego introduce una significativa latenza nella connessione WebRTC. È consigliabile quindi configurare la rete in modo da non avere dei meccanismi che ostacolino in qualche

modo la connessione WebRTC (ad esempio una corretta configurazione del firewall).

In conclusione se si sviluppa un'architettura client-server seguendo queste indicazioni è possibile ottenere tempi di latenza tali da rendere possibile lo sviluppo di applicazioni, anche commerciali, che offrano un ambiente di rendering audio remoto (anche collaborativo) in cui è possibile concentrare le risorse in una sola macchina server per permettere qualunque tipo di elaborazione audio, anche le più dispendiose in termini computazionali. I client possono quindi essere sistemi anche molto semplici in termini hardware in quanto sarebbe richiesta unicamente l'esecuzione di un browser, che non andrà a richiedere ulteriori risorse al sistema, e la capacità di riprodurre dei suoni da un uscita audio. Non di minor importanza è quel che riguarda anche il fattore economico, in quanto è possibile ridurre i costi per i diritti d'autore acquistandoli unicamente per la macchina server.

7.1 La sicurezza nella comunicazione

Un argomento che non è stato trattato in questo lavoro di tesi ma che potrebbe incidere sulle latenze studiate è la sicurezza della connessione client-server, soprattutto per la componente WebRTC.
Non affronteremo il discorso nel dettaglio ma ci limiteremo a menzionare alcuni problemi e possibili soluzioni da utilizzare quando si sviluppa un'applicazione WebRTC. Ci possono essere molti motivi per cui la privacy e la sicurezza di una connessione real-time potrebbero essere compromessi. Ad esempio un grave problema potrebbe riguardare la non cifratura dei media scambiati nella connessione, in quanto essi potrebbero essere intercettati e distribuiti senza che l'utente ne sia a conoscenza oppure si potrebbero rischiare attacchi di tipo Man-in-the-middle.
Per questo motivo WebRTC utilizza dei protocolli sicuri come DTLS (Datagram Transport Layer Security) definito nel RFC 6347 [6] e utilizzato di default da tutti i browser che supportano WebRTC. Per quanto concerne lo sviluppatore invece è necessario seguire le linee guida date dall'IETF le quali indicano che tutte le componenti di WebRTC devono essere crittografate, anche il server di signaling (non applicato in questo lavoro di tesi); se utilizziamo ad esempio WebSocket è necessario utilizzare la versione sicura del protocollo che si appoggia su TLS, cioè il protocollo **WSS** e non il **WS**. Inoltre deve essere richiesto esplicitamente il permesso per l'utilizzo di microfono e webcam se l'applicazione ne richiede l'utilizzo.

Altro aspetto molto importante è l'autenticazione dei peer; il server di signaling dovrebbe provvedere questa funzione attraverso meccanismi di autenticazione e di autorizzazione volti a determinare l'identità dell'utente. Uno schema di implementazione sicura di un applicazione WebRTC è dato dalla figura 7.1, mentre è possibile trovare ulteriori informaizoni a riguardo nel seguente documento dell'IETF [7].

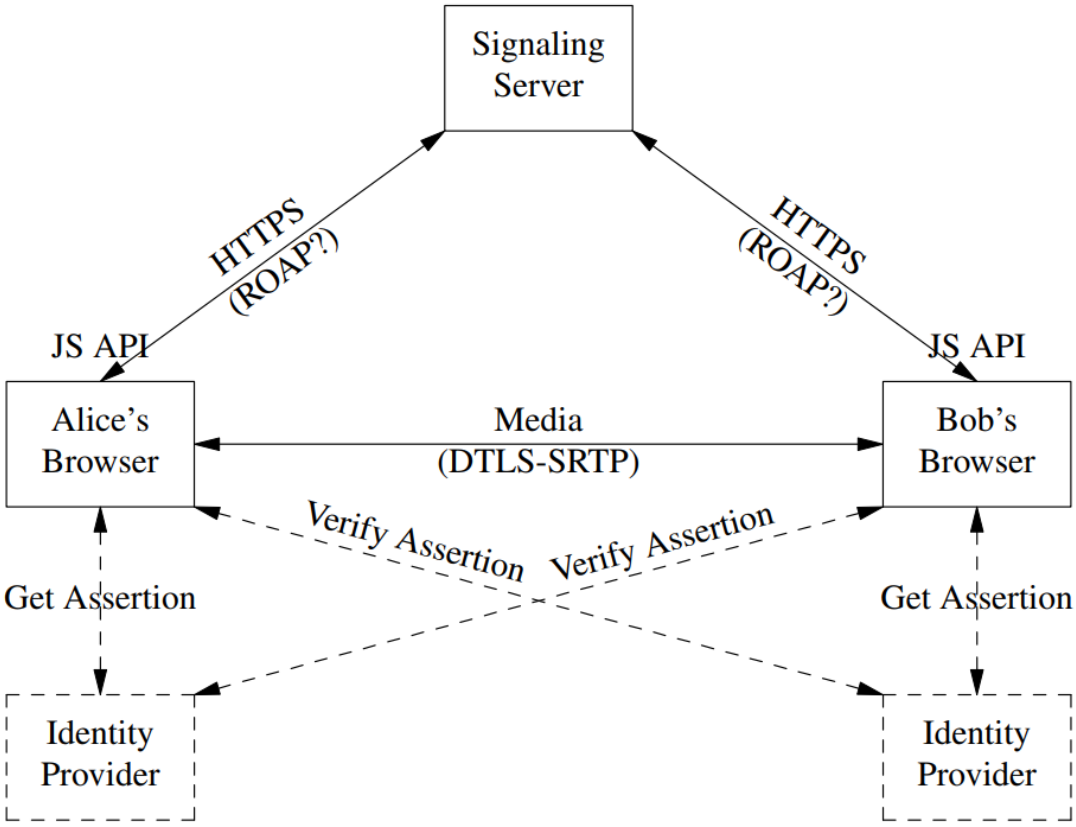

Figura 7.1: Esempio di sviluppo sicuro di un applicazione WebRTC in base alle indicazione dell'IETF

Fonte: https://www.ietf.org/proceedings/82/slides/rtcweb-13.pdf

Bibliografia

[1] I. Fette and A. Melnikov. The websocket protocol. RFC 6455, RFC Editor, December 2011. http://www.rfc-editor.org/rfc/rfc6455.txt.

[2] M. Handley, V. Jacobson, and C. Perkins. Sdp: Session description protocol. RFC 4566, RFC Editor, July 2006. http://www.rfc-editor.org/rfc/rfc4566.txt.

[3] Tina le Grand, Paul Jones, Pascal Huart, and Harald Alvestrand. Rtp payload format for the isac codec. Internet-Draft draft-ietf-avt-rtp-isac-01, IETF Secretariat, April 2012. http://www.ietf.org/internet-drafts/draft-ietf-avt-rtp-isac-01.txt.

[4] D.L. Mills. *Computer Network Time Synchronization: The Network Time Protocol*. CRC Press, 2006.

[5] Suhas Nandakumar and Cullen Jennings. Annotated example sdp for webrtc. Internet-Draft draft-ietf-rtcweb-sdp-11, IETF Secretariat, October 2018. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-sdp-11.txt.

[6] E. Rescorla and N. Modadugu. Datagram transport layer security version 1.2. RFC 6347, RFC Editor, January 2012. http://www.rfc-editor.org/rfc/rfc6347.txt.

[7] Eric Rescorla. Webrtc security architecture. Internet-Draft draft-ietf-rtcweb-security-arch-17, IETF Secretariat, November 2018. http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-security-arch-17.txt.

[8] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. Stun - simple traversal of user datagram protocol (udp) through network address translators (nats). RFC 3489, RFC Editor, March 2003. http://www.rfc-editor.org/rfc/rfc3489.txt.

[9] Cristina Rottondi, Chris Chafe, Claudio Allocchio, and Augusto Sarti. An overview on networked music performance technologies. *IEEE*

Access, 4:8823–8843, 2016.

[10] H. Schulzrinne and S. Casner. Rtp profile for audio and video conferences with minimal control. STD 65, RFC Editor, July 2003. https://tools.ietf.org/html/rfc3551#page-28.

[11] J. Spittka, K. Vos, and JM. Valin. Rtp payload format for the opus speech and audio codec. RFC 7587, RFC Editor, June 2015. http://www.rfc-editor.org/rfc/rfc7587.txt.

[12] R. Stewart. Stream control transmission protocol. RFC 4960, RFC Editor, September 2007. http://www.rfc-editor.org/rfc/rfc4960.txt.

[13] JM. Valin, K. Vos, and T. Terriberry. Definition of the opus audio codec. RFC 6716, RFC Editor, September 2012. http://www.rfc-editor.org/rfc/rfc6716.txt.