DAVIDE PISANÒ - ANTONIO SERVETTI

IDENTIFICAZIONE AUDIO RESISTENTE AL RUMORE IN WEBASSEMBLY

I wanted to make noise, not study theory.
James Hetfield

Indice

1
Introduzione

Negli ultimi anni si è notato un trend sempre crescente nell'utilizzo di JavaScript per la creazione di applicazioni desktop[1].

Ci sono diversi fattori che hanno contribuito alla popolarità di JavaScript, primo fra tutti è che rappresenta il linguaggio standard, de facto, per l'implementazione di funzionalità dinamiche su pagine web. Nello specifico, JavaScript è l'unico linguaggio di scripting supportato nativamente da tutti i browser web moderni[2].

Ci sono stati dei tentativi per introdurre alcune novità in questo ambito, seguendo principalmente due approcci:

1. l'inclusione di una nuova macchina virtuale all'interno di un browser web che supportasse un nuovo linguaggio

2. la realizzazione di un nuovo linguaggio ma eseguito sulla stessa macchina virtuale JavaScript già presente in un web browser

Ricade nella prima categoria VBScript di Microsoft, basato su Visual Basic, introdotto a metà degli anni '90, oggi non più supportato da nessun browser moderno.

Nella seconda categoria possiamo annoverare, più recentemente, TypeScript[3] (sempre di Microsoft), CoffeeScript e Dart (di Google). Questi linguaggi sono basati su un cosiddetto *transpiler*[4], ovvero un compilatore che prende in input il codice sorgente scritto ad esempio in TypeScript e lo converte in codice JavaScript, mantenendo le stesse funzionalità del codice originale.

Un altro punto di forza di JavaScript è la sua facilità di apprendimento[5], soprattutto rispetto ad altri linguaggi di programmazione più a basso livello come C o C++, permettendo agli sviluppatori di iniziare a scrivere codice più rapidamente, senza dover investire troppo tempo nell'apprendimento di una nuova tecnologia. Oggi, infatti, si assiste ad una quantità sempre crescente di librerie e framework per JavaScript, atti a semplificare lo sviluppo di applicazioni web

[1] Guillermo Rauch. *Smashing node. js: Javascript everywhere*. John Wiley & Sons, 2012

[2] Charles Severance. Javascript: Designing a language in 10 days. *Computer*, 45(2):7–8, 2012

[3] Dan Maharry. *TypeScript revealed*. Apress, 2013

[4] Thiago Nicolini, Andre Hora, and Eduardo Figueiredo. On the usage of new javascript features through transpilers: The babel case. *IEEE Software*, pages 1–3, 2023

[5] Sabah A Abdulkareem and Ali J Abboud. Evaluating python, c++, javascript and java programming languages based on software complexity calculator (halstead metrics). In *IOP Conference Series: Materials Science and Engineering*, volume 1076. IOP Publishing, 2021

complesse e a migliorarne la qualità, giusto per citarne alcuni: React, Angular, Vue.js, ma anche il più "anziano" jQuery.

Per questi ed altri motivi, durante la fine degli anni 2000, ci si è iniziati a porre una domanda: "è possibile eseguire codice JavaScript al di fuori del contesto web browser?". La risposta (affermativa) a questa domanda è stata la nascita di Node.js[6] che ha dato il via al paradigma del *JavaScript everywhere*[7]. Questo vuol dire, in estrema sintesi, poter utilizzare lo stesso linguaggio per creare applicazioni web, sia lato front-end sia lato back-end. In teoria si potrebbe così ridurre il tempo necessario per il processo di sviluppo di un'applicazione, riducendo il portfolio di tecnologie che un programmatore deve conoscere.

[6] Guillermo Rauch. *Smashing node. js: Javascript everywhere*. John Wiley & Sons, 2012

[7] Del quale non mancano i detrattori

Nei primi anni 2010, quando Node.js iniziava a prendere piede, ci si è posti un'altra domanda "è possibile scrivere e distribuire applicazioni desktop/mobile scritte in JavaScript?". La risposta, ancora una volta, è stata affermativa. Nasce Electron[8]: un framework open-source che consente agli sviluppatori di creare applicazioni desktop multi-piattaforma utilizzando tecnologie web standard come HTML, CSS e JavaScript. Dal lato mobile nascono tecnologie analoghe a Electron come Ionic, React Native e PhoneGap, tutte con obiettivi abbastanza simili. Man mano l'ecosistema JavaScript ha iniziato a diventare quello che Java era nei primi anni 2000[9] per la scrittura di applicazioni desktop consumer multipiattaforma.

[8] Adam D Scott. *JavaScript everywhere: building cross-platform applications with GraphQL, React, React Native, and Electron*. O'Reilly Media, 2020

[9] Pankaj Kamthan. Java applets in education. *Electronic Resource] Retrieved on May*, 17, 1999

Il motivo principale dietro alla popolarità di questo ecosistema basato su JavaScript è la possibilità di utilizzare un'unica codebase (in JavaScript) che può essere eseguita su piattaforme molto diverse tra loro, problematica che è molto sentita nell'ambito mobile dove si hanno due piattaforme completamente diverse: Android e iOS[10].

[10] All'epoca della prima apparizione di queste tecnologie bisognava supportare anche Windows Phone

Il trend di scrivere applicazioni in JavaScript è stato amplificato dalla crescente importanza del web come piattaforma per la distribuzione di applicazioni software. Software utilizzati quotidianamente da miliardi di utenti sono basati sul web e, per forza di cose, devono essere scritti in JavaScript.

Da qui la nascita delle cosiddette *Rich Internet Applications* (RIA)[11], ovvero applicazioni web che offrono un'esperienza utente interattiva e avanzata simile a quella di un'applicazione desktop tradizionale. Le RIA sono caratterizzate da una vasta gamma di funzionalità e interattività che le distinguono dalla semplici pagine web statiche. Oltre alla classica triade HTML + CSS + JavaScript le RIA possono fare uso di tecnologie più avanzate e recenti come WebSockets, WebAudio, WebAssembly, WebRTC, WebVR, WebGPU, Web Animations API. In sostanza il browser diventa un'interfaccia o un'astrazione della macchina sottostante, alla quale si può accedere utilizzando JavaScript.

[11] Piero Fraternali, Gustavo Rossi, and Fernando Sánchez-Figueroa. Rich internet applications. *IEEE Internet Computing*, 14(3):9–12, 2010

Nello specifico WebAudio[12] è un'API JavaScript avanzata che consente di manipolare e generare audio all'interno del browser. È stata progettata per consentire agli sviluppatori di creare RIA che includono funzionalità audio, come la registrazione, la riproduzione e l'elaborazione di suoni. L'API è basata su un'architettura a nodi, dove ogni nodo rappresenta una singola operazione di elaborazione del suono. I nodi possono essere collegati tra loro per creare una catena di elaborazione, in cui il suono viene elaborato in successione da ogni nodo che attraversa. La manipolazione del suono avviene in real time. WebAudio è oggi supportato su tutti i browser recenti basati sugli engine JavaScript V8 e SpiderMonkey.

[12] Hongchan Choi. Audioworklet: the future of web audio. In *ICMC*, 2018

La necessità di scrivere applicazioni real time ha portato la necessità di dover eseguire codice ad alta efficienza, obiettivo non realizzabile completamente con un linguaggio interpretato quale JavaScript. Alla fine degli anni 2010 nasce quindi *WebAssembly*[13] (Wasm): un formato di codice binario portabile che consente di eseguire codice di basso livello all'interno del browser web. È stato progettato per essere compatibile con i linguaggi di programmazione come C, C++ e Rust. In pratica, quindi, Wasm permette di creare applicazioni web che eseguono codice più velocemente e con maggiore efficienza rispetto a soluzioni basate su JavaScript. Wasm è stato pensato per essere altamente interoperabile con JavaScript: codice JavaScript può richiamare codice Wasm e viceversa, creando quindi soluzioni ibride che combinano il meglio di entrambi i mondi.

[13] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017

Sfruttando tutte queste tecnologie e un ecosistema ormai maturo, l'obiettivo di questa tesi è quello di discutere la realizzazione di un sistema per l'identificazione di audio: un utente sottopone uno spezzone di un brano audio di pochi secondi al sistema, il quale risponde col nome di quel brano. L'obiettivo principale è quello di eseguire l'algoritmo di identificazione su dispositivi eterogenei all'interno di un web browser, utilizzando Wasm e WebAudio. Questo presenta numerosi vantaggi, tra i più importanti si possono individuare:

- l'evitare all'utente il download di un'app addizionale, potendo sfruttare le funzionalità dell'algoritmo di riconoscimento direttamente dal suo web browser
- la notevole riduzione del carico lato server: grazie alla sua architettura distribuita, parte della complessità viene spostata sul dispositivo dell'utente finale, il quale porta a termine buona parte del processo di identificazione, rendendo possibile un'identificazione più veloce ed efficiente rispetto ad altre applicazioni simili

In definitiva, si renderà l'esperienza dell'utente ancora più piacevole e soddisfacente, mantenendo le stesse funzionalità e caratteristiche di una classica app eseguita nativamente su un dispositivo dell'utente.

1.1 *Architettura generale*

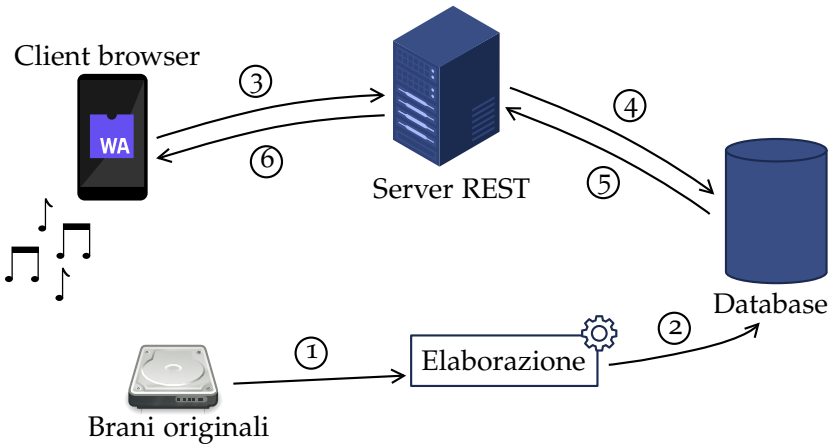

Figura 1.1: Schema architettura generale

L'architettura di base del sistema (in figura 1.1), seppur ispirata al modello client/server, si discosta dalla tradizionale asimmetria tra i due attori, in cui il primo agisce come mero terminale passivo[14], limitandosi a interagire con l'API del server. La nuova soluzione adottata, invece, si propone di spostare parte della logica di business dal server al client, in un'ottica distribuita che avvicina la computazione all'utente e alleggerisce, al contempo, il carico sul server centrale, riducendo così i costi correlati. Tale approccio innovativo sfrutta le risorse disponibili sui dispositivi dell'utente, aumentando la scalabilità del sistema e garantendo prestazioni elevate e una maggiore efficienza. Questa soluzione rappresenta un notevole passo avanti nella progettazione di applicazioni web avanzate, fornendo un'esperienza utente fluida e gradevole.

[14] In gergo tecnico è ciò che si definisce *dumb terminal*

1.1.1 *Scomposizione dell'architettura*

L'architettura (in figura 1.1) può essere scomposta come segue:

1. Si inizia dai brani originali, la canzone nella sua interezza, salvata su una memoria di massa. La canzone è sottoposta ad una algoritmo di *fingerprinting*, in cui vengono estratte alcune features[15] caratterizzanti.
2. Le features estratte vengono memorizzate all'interno di un database insieme al nome della canzone alla quale appartengono.
3. Si immagini quindi che, ad un certo punto, un client voglia avviare il processo di riconoscimento di un brano: viene registrato uno spezzone audio di pochi secondi e viene innescata la stessa procedura di *fingerprinting* al punto 1 sul client, ma in questo caso le features estratte vengono inviate ad un endpoint REST.

[15] In seguito queste features prenderanno il nome di *Links*

4. Il server REST cerca di individuare delle similarità tra le features già presenti nel database e quelle appena inviategli dal client.
5. Se la ricerca ha successo, il server REST estrae dal database il nome della corrispondenza migliore.
6. Se la ricerca ha successo, il server REST invia al client il nome della corrispondenza migliore.

Si noti, anzitutto, che la parte più intensiva dal punto di vista computazionale è l'estrazione delle features, al contrario la ricerca delle similarità, sebbene impegnativa, non lo è quanto l'estrazione delle features stesse[16]. In altre parole, il momento di maggior carico computazionale si verifica in due fasi:

[16] Ulteriori considerazioni sull'argomento verranno fatte in seguito

- *Lato server*: solo nella fase iniziale che porta al popolamento del database, durante l'analisi dei brani originali (ovvero fasi 1 e 2)
- *Lato client*: nell'estrazione delle features della registrazione del brano da riconoscere

In altre parole, l'operazione più onerosa per il server viene eseguita una sola volta: all'atto del fingerprinting dei brani originali. Sarà poi il client a farsi carico dell'operazione di fingerprinting per l'identificazione del singolo brano.

Il server REST ha una duplice funzione:

- Presentare i dati nel formato corretto sia lato client che lato database, fungendo quindi da una sorta di relay e disaccoppiando la rappresentazione interna dei dati a quella esposta al client
- Individuare similarità con le features già presenti nel database

L'intero sistema verrà descritto in modo più dettagliato nei capitoli successivi.

1.2 *Organizzazione del codice*

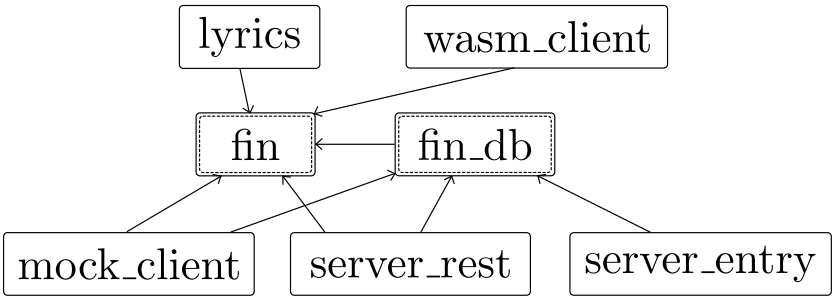

Figura 1.2: Schema organizzazione codice

Il codice sorgente del sistema è suddiviso nei seguenti componenti (figura 1.2):

- La libreria *fin*, deputata all'estrazione delle features (ovvero fare *fingerprinting*) dei brani in esame.
- La libreria *fin_db* preposta all'interazione con il database, svolgendo i compiti di inserimento e ricerca delle features. Dipende da *fin*.
- L'eseguibile *mock_client*, riservato esclusivamente ad attività di testing, il quale riceve in input un segmento noto di un brano, al fine di verificare la corretta identificazione del brano stesso. Dipende da entrambe le librerie.
- L'eseguibile *server_entry*, in grado di elaborare i brani completi per estrarne le features, per poi memorizzarle nel database insieme al nome del brano associato
- L'eseguibile *server_rest* che espone l'endpoint REST per l'individuazione dei brani: riceve le features del segmento audio estratte dal client, effettua una ricerca di un brano compatibile all'interno del database e, in caso di esito positivo, restituisce al client il nome del brano individuato. Dipende da entrambe le librerie.
- L'eseguibile[17] *wasm_client*, ovvero il client, in grado di acquisire il segmento audio tramite microfono del client, estraendone le features per poi inviarle a *server_rest*.
- L'eseguibile[18] *lyrics*, costruito sulla base di wasm_client, aggiunge la possibilità di visualizzare il testo sincronizzato (in tempo reale) del brano riconosciuto.

Successivamente, nei prossimi capitoli, verranno esaminate in dettaglio le specifiche funzionalità di ciascun componente sopracitato.

[17] In realtà l'eseguibile è la RIA, contenente HTML, il modulo Wasm e il codice JavaScript necessario al caricamento del modulo Wasm

[18] Anche in questo caso si ha a che fare con una RIA

2
La libreria fin

La libreria *fin* è il componente principale del sistema e ha il compito di estrarre le features caratterizzanti di un brano, garantendo il più possibile, che audio simili abbiano features simili. D'ora in avanti ci si riferirà alle features indicandole come *Links*.

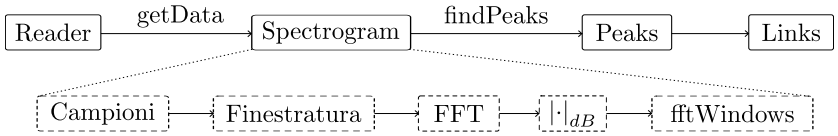

Figura 2.1: Schema architettura libreria fin

Il punto di ingresso della libreria è un *Reader*, ovvero un contenitore dei campioni che compongono un audio; in uscita si hanno i *Links* che caratterizzano quell'audio. Analizzando più dettagliatamente lo schema in figura 2.1:

1. **Reader** è una classe astratta del namespace `readers`, la rappresentazione di un audio nel dominio del tempo. La classe definisce un metodo virtuale puro `getData()` che restituisce i campioni dell'audio.

2. **Spectrogram** è una classe del namespace `math`, rappresenta lo spettrogramma di un audio. Riceve i campioni dal *Reader* e procede come segue:

   (a) Finestra il segnale ottenendone un segmento

   (b) Calcola la DFT per ogni segmento

   (c) Calcola il modulo dell'output della DFT per ogni segmento

   (d) Salva il risultato del punto precedente in una struttura chiamata `fftWindow`

   (e) Le varie `fftWindow` compongono lo spettrogramma, in altre parole una rappresentazione in frequenza dell'audio

3. **findPeaks** è una funziona nel namespace `core` che estrae i picchi più alti[1] dallo spettrogramma. Ogni picco è rappresentato da un

[1] Nonché i più significativi per il sistema in analisi

oggetto `Peak`.

4. i **Links** sono il risultato dell'operazione di *fingerprinting*, ovvero le features che caratterizzano l'audio. Vengono creati a partire dai vari `Peak` estratti da `findPeaks` e fanno parte del namespace `core`.

Tutte le classi e le funzioni della libreria *fin* sono contenute nel namespace `fin`.

2.1 La classe Reader e le sue sottoclassi

La classe `Reader` generica rappresenta un contenitore di campioni audio. Definisce due metodi puri virtuali:

- `getData()` che ritorna un `std::vector<float>` contenente i campioni
- `dropSamples()` che svuota il `vector` dei campioni

La classe `Reader` viene estesa da due classi *reader* concrete (vedi figura 2.2):

- `WavReader`, in grado di leggere i file Wave.
- `DummyReader`, un mero wrapper attorno al contenitore `std::vector`, con un metodo `addSamples` per inserire nuovi samples nel vettore.

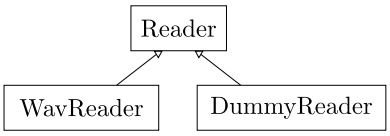

Figura 2.2: Schema ereditarietà tra readers

2.2 Lo spettrogramma

Lo *spettrogramma* è una rappresentazione tridimensionale del contenuto in frequenza di un segnale nel tempo. Questa rappresentazione di un segnale audio viene calcolata e memorizzata all'interno della classe `Spectrogram` del namespace `math`.

Il costruttore riceve in input come parametro un `std::vector<float>` di campioni nel dominio del tempo dell'audio da analizzare: il primo passo da compiere è *finestrare il segnale*.

2.2.1 La finestratura

Il primo passo per ottenere uno spettrogramma è finestrare il segnale[2]: il caso più semplice consiste nel utilizzare una finestra rettangolare come indicato in figura 2.3.

L'utilizzo di una *funzione finestra*, tuttavia, porta al fenomeno dello *spectral leakage*, ovvero la comparsa nello spettro di nuove frequenze che non esistono realmente nello spettro del segnale audio originale. Nello specifico, l'energia di un picco di frequenza confluisce parzialmente nelle frequenze vicine (da cui il termine *leakage*), *sporcando* la rappresentazione dello spettrogramma.

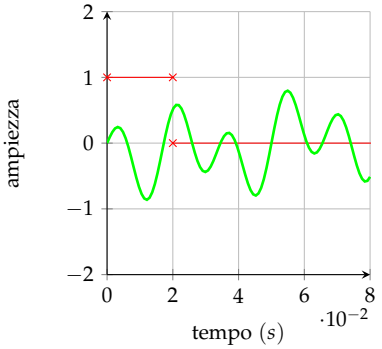

Figura 2.3: Finestratura con finestra rettangolare

[2] Prajoy Podder, Tanvir Zaman Khan, Mamdudul Haque Khan, and M Muktadir Rahman. Comparative performance analysis of hamming, hanning and blackman window. *International Journal of Computer Applications*, 96(18):1–7, 2014

Lo *spectral leakage* non può essere del tutto evitato, ma può essere tenuto sotto controllo e ridotto utilizzando una funzione finestra più complessa rispetto a quella rettangolare.

La scelta della finestra implica un trade-off tra la *risoluzione spettrale* e il *range dinamico* del sistema. Infatti, l'utilizzo di una finestra con banda passante stretta consente di ottenere una maggiore risoluzione spettrale, ovvero di distinguere frequenze molto vicine tra loro, ma allo stesso tempo comporta una riduzione del range dinamico del sistema, ovvero della capacità di distinguere segnali di ampiezza molto differente tra loro. Al contrario, l'utilizzo di una finestra con banda passante ampia, comporta una maggiore sensibilità ai segnali con un ampio range dinamico, ma al costo di una riduzione della risoluzione spettrale.

Inoltre, è importante considerare che la scelta della finestra deve essere fatta in base alle specifiche caratteristiche del segnale che si intende analizzare, come ad esempio la presenza di rumore o la distribuzione di energia spettrale. Pertanto, la scelta dev'essere attentamente valutata in base alle esigenze specifiche dell'applicazione.

Il vantaggio principale della *finestra rettangolare* è la sua risposta in frequenza piatta, risultando quindi la migliore in termini di risoluzione spettrale. Tuttavia, la finestra rettangolare ha una bassa attenuazione laterale, ovvero non è in grado di attenuare il rumore presente nelle frequenze circostanti, il che limita il suo range dinamico.

Per ovviare a questo problema, sono state sviluppate altre finestre, come quelle di *Blackman* e *Hann*. La finestra di *Hann* è una scelta intermedia tra la finestra *rettangolare* e la finestra di *Blackman*. Infatti, la finestra di *Hann* presenta una risoluzione spettrale inferiore rispetto alla finestra *rettangolare* ma un range dinamico migliore. La finestra di *Blackman*, invece, è la scelta migliore in termini di range dinamico, poiché, tra quelle citate, è quella che riduce maggiormente la diffusione dell'energia in altre frequenze vicine. Tuttavia, la sua risoluzione spettrale è la peggiore.

La scelta della finestra da utilizzare dipende dalle caratteristiche del segnale e degli obiettivi dell'analisi. In questo caso di studio si avrà a che fare potenzialmente con un segnale rumoroso, quindi si è portati ad utilizzare la finestra di *Blackman*. D'altro canto, però, si deve considerare che, anticipando il contenuto dei prossimi paragrafi, l'estrazione dei Links risulta migliore tanto migliore è l'analisi in frequenza del segnale e, quindi, si dovrebbe prediligere la massima risoluzione spettrale, individuando con precisione le componenti di frequenza presenti nel segnale, a discapito del range dinamico.

A questo punto risulta necessario valutare *sul campo* le performance di queste finestre. Per questo motivo è stato predisposto un

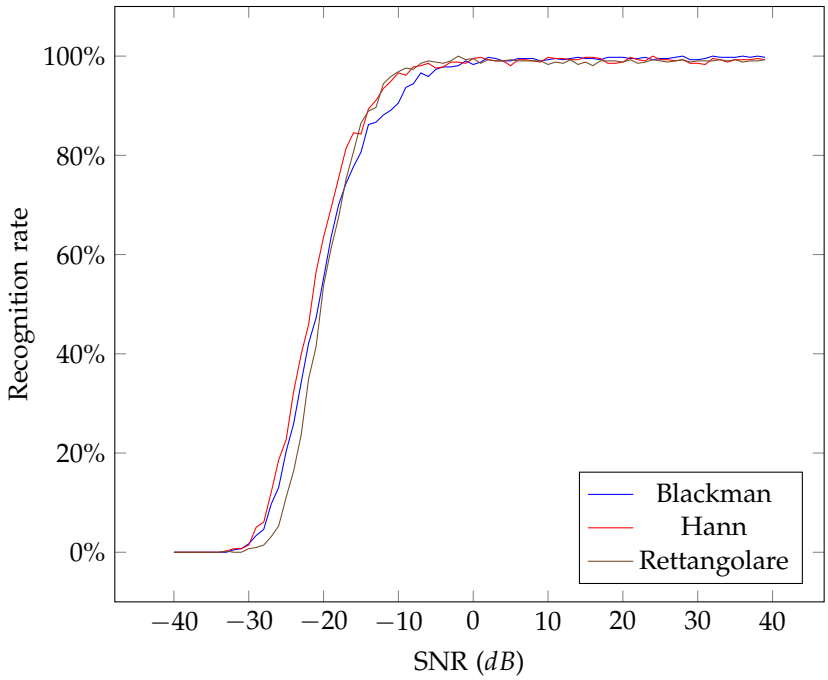

Figura 2.4: Recognition rate in funzione dell'SNR per le varie finestre

ambiente di test composto da circa 400 brani di generi molto diversi tra loro[3] e per ognuno di questi brani:

1. è stato estratto un segmento di pochi secondi.
2. il segmento è stato distorto, sommando del rumore non bianco e introducendo del clipping.
3. nota l'energia del segmento del segnale, è stato sommato un rumore per ottenere un SNR target.

Si è quindi fatto variare, per ogni brano, l'SNR tra $-40dB$ e $+40dB$ con passo di $1dB$, per le tre finestre considerate, contando quante volte l'algoritmo fosse in grado di individuare il brano correttamente: si è ottenuto il grafico in figura 2.4.

Analizzando dal grafico le prestazioni delle finestre di Hann, rettangolare e di Blackman, si può notare che:

- per un SNR fino a $-15dB$ la finestra con le performances migliori è quella di Hann
- nel tratto tra $-15dB$ in poi la finestra rettangolare e quella di Hann sono comparabili
- la finestra di Blackman fa ottenere un recognition rate sempre minore o di quello della finestra di Hann o di quella rettangolare

[3] I generi inclusi sono stati il metal, il rock, il blues e la musica classica

L'efficacia della finestra di Hann rispetto alle altre due è quindi dimostrata. Tuttavia, non esiste una finestra *migliore* in assoluto e la scelta della finestra più adatta dipende dalle specifiche del problema in esame. Nel caso in analisi la finestra di Hann è stata selezionata poiché ha ottenuto complessivamente le performance migliori, raggiungendo mediamente il *recognition rate* più alto.

La libreria fin finestra il segnale audio con un overlap[4] del 50% (vedi figura 2.5). Questa scelta è dovuta a due motivi:

1. Sopperire all'attenuazione agli estremi della finestra introdotti dalla funzione finestra di Hann
2. Analizzare meglio il contenuto in frequenza a cavallo tra due finestre senza overlap

[4] MW Trethewey. Window and overlap processing effects on power estimates from spectra. *Mechanical Systems and Signal Processing*, 14(2):267–278, 2000

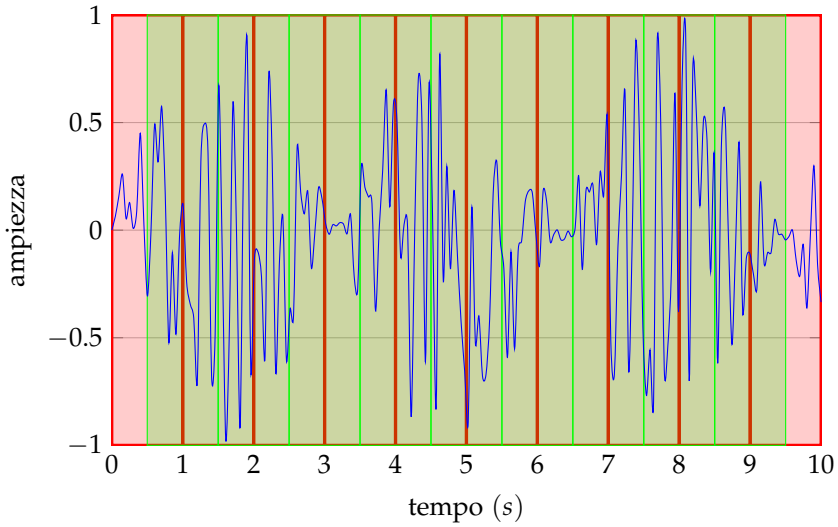

Figure 2.5: Finestratura audio con overlap

2.2.2 *La DFT*

La *trasformata di Fourier discreta* viene calcolata facendo ricorso alla libreria **fftw**[5].

In prima analisi è necessario introdurre il concetto di *risoluzione spettrale*[6], ossia la capacità di un'analisi spettrale di distinguere due componenti di frequenza vicine nel dominio della frequenza. Dipende dal numero di campioni utilizzati nella finestra e dalla finestra utilizzata[7]. La *risoluzione in frequenza* della DFT può essere calcolata come segue:

$$\Delta f = \frac{F_S}{N}$$

Dove:

[5] Reperibile all'indirizzo https://www.fftw.org/

[6] William L Briggs and Van Emden Henson. *The DFT: an owner's manual for the discrete Fourier transform*. SIAM, 1995

[7] Vedere paragrafo 2.2.1 sulla risoluzione spettrale delle finestre

- Δf è la *risoluzione spettrale*, anche detta *dimensione del bin di frequenze*
- F_S è la *frequenza di campionamento*
- N sono il numero di campioni in una finestra

La libreria fin è configurata per trattare segnali audio campionati a $8000Hz$, utilizzando finestre di dimensione pari a 512 campioni[8], quindi:

$$\Delta f = \frac{8000Hz}{512} = 15.625Hz$$

[8] Successivamente verranno fatte delle considerazioni su queste scelte

Inoltre, è necessario fare alcune osservazioni aggiuntive:

1. Il primo bin non contiene informazioni rilevanti riguardo la rappresentazione in frequenza del segnale[9]
2. L'output della DFT su segnali reali è simmetrico[10]

[9] William L Briggs and Van Emden Henson. *The DFT: an owner's manual for the discrete Fourier transform*. SIAM, 1995

[10] William L Briggs and Van Emden Henson. *The DFT: an owner's manual for the discrete Fourier transform*. SIAM, 1995

A dimostrazione del primo fatto si può partire dalla definizione della DFT:

$$X[n] := \sum_{t=0}^{N-1} x(t) e^{-i\frac{2\pi tn}{N}} \tag{2.1}$$

e valutarla in $n = 0$:

$$\begin{aligned} X[n]|_{n=0} &:= \sum_{t=0}^{N-1} x(t) e^{-i\frac{2\pi tn}{N}}|_{n=0} \\ &= \sum_{t=0}^{N-1} x(t) e^{0} \\ &= \sum_{t=0}^{N-1} x(t) \end{aligned}$$

In altre parole, il primo bin dell'output della DFT corrisponde alla *componente DC* del segnale in ingresso che può essere assunto pari a 0 e quindi ignorato nel caso di segnali audio.

Per quanto riguarda il secondo fatto, si deve dimostrare che se $x(t)$ è un segnale a valori reali allora:

$$X[N-n] = X^*[n] \tag{2.2}$$

dove:

- $X(\odot)$ è l'output della DFT applicata a $x(t)$
- $(\odot)^*$ denota il coniugato di $\odot$

Si sostituisce n con $N - n$ nella definizione della DFT 2.1, proveniente dalla 2.2:

$$\begin{aligned} X[N-n] &:= \sum_{t=0}^{N-1} x(t) e^{-i\frac{2\pi t(N-n)}{N}} \\ &= \sum_{t=0}^{N-1} x(t) \underbrace{e^{-i2\pi t}}_{1 \forall t} e^{i\frac{2\pi tn}{N}} \\ &= \sum_{t=0}^{N-1} x(t) e^{i\frac{2\pi tn}{N}} \\ &= \left(\sum_{t=0}^{N-1} x(t) e^{-i\frac{2\pi tn}{N}} \right)^* \\ &= X^*[n] \end{aligned}$$

Dove il passaggio al coniugato è invariante per $x(t)$ dato che è a valori reali.

Le due precedenti considerazioni permettono quindi all'algoritmo di lavorare in maniera più efficiente, escludendo la componente DC dai calcoli ed impiegando una versione ottimizzata della DFT per input reali, con soli $\frac{N}{2} - 1$ coefficienti utili in output.

2.2.3 *Il modulo dello spettrogramma e le* `fftWindows`

L'output della DFT è un vettore di $\frac{N}{2} - 1$ numeri complessi, ma l'algoritmo necessita solo del loro modulo in dB, calcolato come:

$$20 \log_{10} \sqrt{a^2 + b^2} = 10 \log_{10} \left(a^2 + b^2 \right)$$

Dove a e b sono rispettivamente la parte reale e immaginaria del numero complesso.

I vari moduli, per ogni finestra considerata, vengono memorizzati all'interno di un oggetto `fftWindow`. Ogni `fftWindow` viene inserita in un `std::vector` che verrà utilizzato dalla funzione `findPeaks`.

2.3 *I Peaks*

Pronto lo spettrogramma, lo si deve processare per ottenere i picchi di frequenze più significativi e scartare tutto il resto: questo permette di avere una prima rappresentazione più compatta del segnale audio.

Lo spettrogramma è diviso in una sorta di griglia, in cui ogni cella ha le seguenti dimensioni:

- Larghezza pari a C[11]
- Altezza pari ad un range di frequenze, chiamato *banda*

Si prenda la suddivisione semplificata della figura 2.6 dove:

[11] Definita nel file `consts.h`

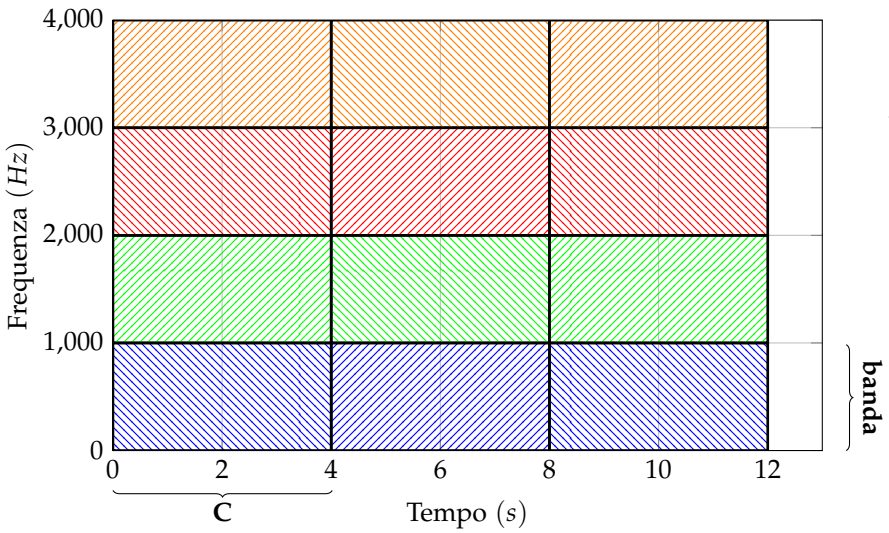

Figure 2.6: Suddivisione dello spettro (ogni colore è una banda)

- $C = 4$
- La banda ha dimensione fissa pari a $100Hz$

Nell'algoritmo questi parametri sono scelti diversamente: C è uguale a 32 e la suddivisione in bande segue una scala logaritmica (vedere 2.3.1).

Per ogni cella, attraverso la funzione `findPeaksInWindow` definita nel file `peaks_finder.cpp`, l'algoritmo individua e memorizza le 3 frequenze[12] più prominenti.

[12] Costante `N_PEAKS` in `consts.h`

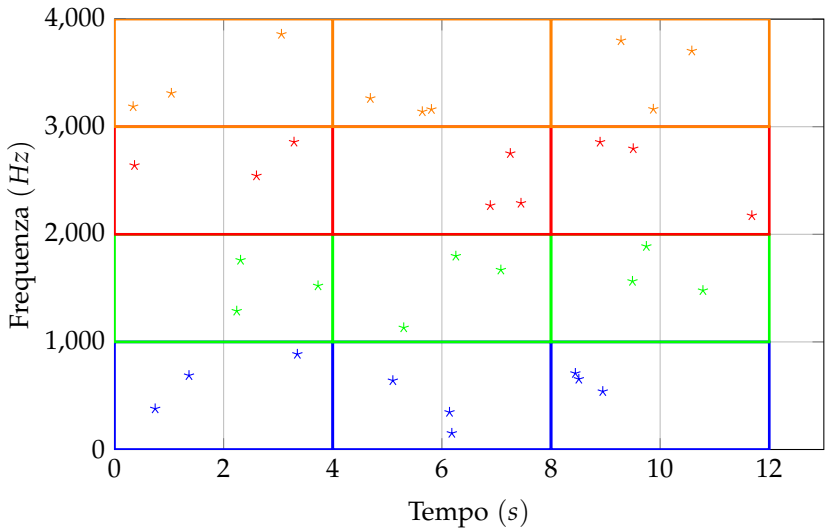

Figure 2.7: Spettrogramma con 3 picchi per cella

La funzione `findPeaks`[13] si occupa di dividere lo spettro in celle, per ogni cella richiama `findPeaksInWindow` e ne memorizza il

[13] Sempre in `peaks_finder.cpp`

risultato, ottenendo qualcosa di simile a quanto visibile in figura 2.7. Alla fine dell'esecuzione di `findPeaks` sarà disponibile un `std::vector<Peak>` ordinato per intensità del picco. Questi picchi verranno utilizzati in seguito per creare i `Links`.

2.3.1 *Scelta della dimensione delle bande*

Esistono diversi modi per suddividere uno spettrogramma in range di frequenze significativi, ma quello che la libreria fin utilizza è la *scala Mel*[14].

[14] Paul Pedersen. The mel scale. *Journal of Music Theory*, 9(2):295–308, 1965

La scala Mel è stata sviluppata per emulare il modo in cui l'orecchio umano percepisce le frequenze dei suoni: è basata sulla scoperta sperimentale che più alta è la frequenza di due suoni più è difficile per l'orecchio umano discriminarli. Questa scala è, quindi, stata creata per mappare le frequenze del suono in modo che la loro rappresentazione sia più in linea con la percezione dell'orecchio umano.

È definita come segue:

$$f(m) := 700\left(10^{\frac{m}{2595}} - 1\right) \tag{2.3}$$

dove:

- m è la *frequenza Mel*
- f è la frequenza in Hertz

Si può anche ricavare la duale:

$$m(f) = 2595\log_{10}\left(\frac{f}{700} + 1\right) \tag{2.4}$$

rappresentata in figura 2.8.

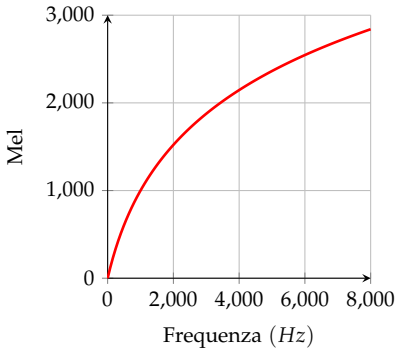

Figure 2.8: Mapping tra Hertz e scala Mel

La libreria fin crea le varie bande seguendo l'algoritmo descritto di seguito:

1. Parte dalla frequenza Mel $\alpha_M = 250$[15] che se valutata nella 2.3 corrisponde a $f(\alpha_M) = \alpha_F \cong 174Hz$: questo vuol dire che tutte le frequenze inferiori a α_F vengono scartate.
2. Si è definito un $\delta_M = 200$[16], in modo tale che gli estremi delle bande sulla scala Mel saranno a:

$$\{\alpha_M, \alpha_M + \delta_M, \alpha_M + 2\delta_M, ..., \alpha_M + k\delta_M\}$$

3. Si convertono gli estremi in frequenze in Hertz usando la 2.3, in modo tale che gli estremi delle bande siano a:

$$\{f(\alpha_M), f(\alpha_M + \delta_M), f(\alpha_M + 2\delta_M), ..., f(\alpha_M + k\delta_M)\}$$

[15] Costante `MEL_START` in `consts.h`

[16] Costante `MEL_STEP` in `consts.h`

Si è scelto di scartare le frequenze inferiori a $174Hz$ poiché rappresentano frequenze troppo basse, spesso molto rumorose, che non fanno altro che sprecare complessità computazionale, dato che non contengono informazioni rilevanti.

2.4 *I Links*

A questo punto si dispone di alcune features (i *picchi*) che, in teoria, potrebbero essere utilizzate per identificare un brano. Ogni picco del segmento registrato dal client dovrebbe essere confrontato con i picchi del brano completo. Tuttavia, questo approccio, anche se semplice e probabilmente funzionante, risulterebbe troppo lento. Infatti, in questo caso, la complessità sarebbe dell'ordine della lunghezza totale di tutti i brani presenti nel database: paradossalmente, più brani l'algoritmo riuscirebbe a individuare, più lento diventerebbe.

Per questo motivo, la libreria fin utilizza un altro algoritmo:

1. In primo luogo, viene individuato tra i picchi un *anchor point* o un *indirizzo*.
2. Ogni anchor point definisce una *target zone*, ovvero una porzione della costellazione dei picchi: sia α un anchor point, A_α l'insieme di picchi[17] associato ad α, β un generico picco, allora:

   $$\beta \in A_\alpha \iff (1 \leq \beta.\text{window} - \alpha.\text{window} < 3) \wedge (\beta.\text{band} = \alpha.\text{band})$$

   A_α sarà quindi l'insieme dei punti che fanno parte della target zone.

[17] Inizialmente vuoto

La bande sono le stesse definite in base della scala Mel (vedi 2.3.1). Si ottiene qualcosa di simile a quanto illustrato in figura 2.9.

La coppia *anchor point* e *picco* prende il nome di *Link*. Il Link, quindi, viene costruito a partire dalla coppia (α, β) in questo modo:

$$Link \begin{cases} hash = h(\delta_w, \delta_f, \alpha.frequency) \\ window = \alpha.window \end{cases}$$

Dove:

- $\delta_w = \beta.window - \alpha.window$
- $\delta_f = \beta.frequency - \alpha.frequency$
- $h(\odot)$ è una funzione di hash

Nella 2.10 è mostrato un esempio di Link. Notare che viene memorizzata anche la finestra dell'anchor point: questo dettaglio ritornerà utile in seguito[18].

[18] Vedi paragrafo 3.2.1

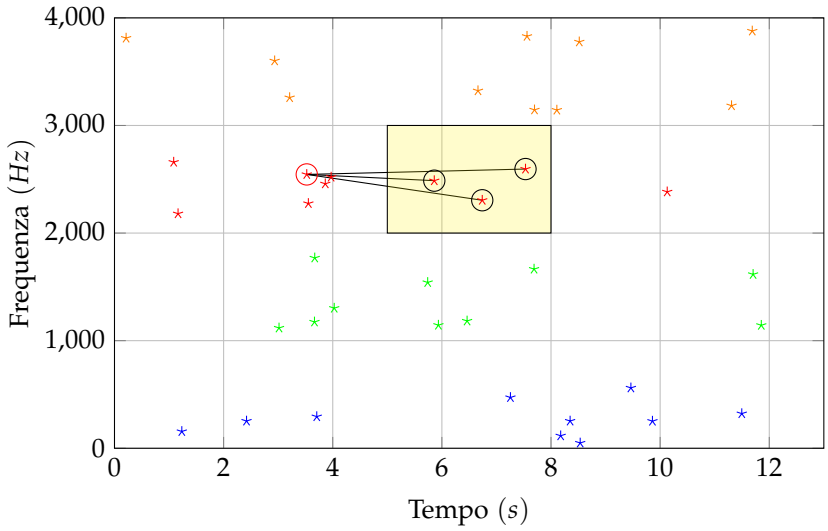

Figure 2.9: Selezione dell'anchor point e della target zone

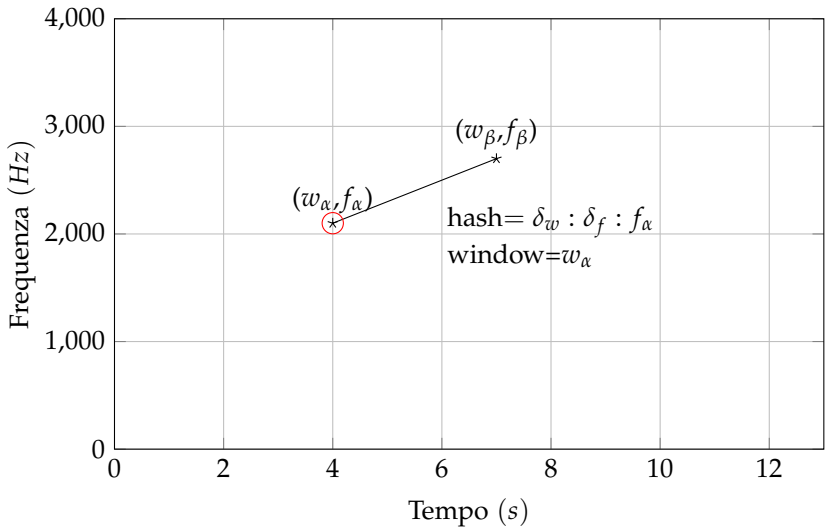

Figure 2.10: Struttura dei link

I Link generati in questo modo sono molto riproducibili, soprattutto in presenza di rumore o artefatti di codifica dell'audio.

Un altro grande vantaggio di questa rappresentazione risiede nel fatto che tutti i tempi[19] sono relativi, in quanto espressi rispetto all'anchor point: in altre parole, non importa quando l'utente inizia a registrare il segmento audio, l'algoritmo riuscirà comunque ad individuare il match.

[19] Gli indici delle finestre ad essere precisi

3
La libreria fin_db

La libreria *fin_db* è il secondo dei componenti più importanti del sistema. Il suo compito principale è quello di interfacciarsi con il database per effettuare le seguenti operazioni:

- *Inserimento*: inserire i Links di un nuovo brano nel database insieme al suo nome
- *Ricerca*: gestire il processo di individuazione del brano a partire dal Links che lo caratterizzano

In secondo luogo, fin_db può essere vista come una sorta di livello di astrazione del database: la libreria è basata sul connector *mariadb++*[1] e il database utilizzato è *MariaDB*[2], ma può essere facilmente riconfigurata per utilizzare altri database e altri connector. In particolare, `DB` è la classe[3] che si occupa di dialogare con l'istanza del database.

[1] `https://github.com/viaduck/mariadbpp`

[2] `https://mariadb.org/`

[3] Vedere il file `db.cpp`

3.1 *L'inserimento di un brano*

Il primo passo per rendere un brano individuabile è quello di memorizzare i Links che lo caratterizzano e il suo nome all'interno database. A questo scopo sono state predisposte due entità rappresentate in figura 3.1:

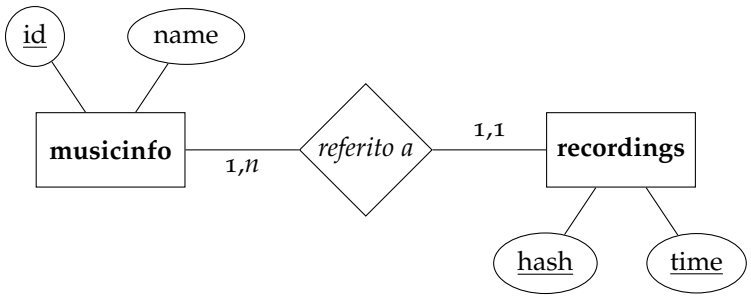

Figura 3.1: Modello ER database, inserimento

- *musicinfo* contiene le informazioni del brano, nello specifico il suo nome e un id che lo identifica univocamente.

- *recordings* contiene i Links. Utilizzando il modello relazionale, oltre ai Links, verrà memorizzata anche la foreign key *songId* che punta ad un record di musicinfo. La tupla

$$\{\underbrace{hash, time}_{Link}, songId\}$$

  fungerà da chiave primaria.

L'inserimento di un nuovo brano nel database avviene richiamando la funzione

```
void insertSong(
    const std::string &filename,
    DB &db
)
```

definita nel file `fin_db.h`.

La funzione prende in input il nome del file audio da leggere e un oggetto `DB`[4] e:

1. Crea un oggetto `WavReader` per leggere il file passatogli come parametro
2. Estrae i Links utilizzando la funzione `computeLinks`
3. Richiama il metodo `db.insertSong` passando come parametri il nome del brano e i Links del brano stesso

A sua volta il metodo `db.insertSong` inserisce nella tabella **musicinfo** il nome del brano e nella tabella **recordings** i Links che lo caratterizzano (vedi figura 3.1).

Uno stralcio delle due tabelle è visibile nelle tabelle a margine 3.1 e 3.2.

[4] Già istanziato dal caller

id	name
1	Arabella
2	Polly
3	The Jeweller's Hand
4	Fight Fire with Fire
5	Ride the Lightning

Tabella 3.1: Stralcio **musicinfo**

hash	songId	time
93141319434145	4	625
154709161072387	1	3180
411179850164819	5	7399
454176498247512	3	9637
770548217553156	2	8544

Tabella 3.2: Stralcio **recordings**

3.2 *L'identificazione di un brano*

Si assuma ora che:

- i Links e i nomi dei brani completi siano già nel database
- i Links siano già stati estratti dal segmento da identificare

Allo scopo di facilitare il processo di identificazione, viene introdotta una nuova tabella, rappresentata in figura 3.2, per contenere i Links del segmento registrato. Si noti come questa tabella sia molto simile alla tabella **recordings**, ma con alcune differenze:

- Non esiste alcuna relazione con la tabella **musicinfo**, in quanto **_recording** conterrà i Links di un solo audio.

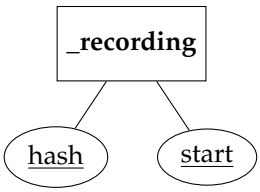

Figura 3.2: Modello ER database, ricerca

- Il campo time è stato rinominato in start, questo è dovuto al fatto che nel caso dei brani completi time rappresenta un valore assoluto[5], mentre start è un riferimento relativo[6].

In più, la tabella **_recording** viene creata con alcune particolari accortezze[7]:

1. È una come *temporary table*, ovvero una tabella che esiste solo per la durata della connessione al database.
2. Viene specificato `ENGINE=MEMORY`, ovvero la tabella viene mantenuta interamente in memoria RAM e non viene scritta su disco.

In generale, l'utilizzo di una temporary table con `ENGINE=MEMORY` è quindi molto utile in situazioni in cui è necessario mantenere temporaneamente i dati in memoria per migliorare le prestazioni, ma non è necessario preservarli per un lungo periodo di tempo, come nel caso in analisi.

La funzione che si occupa di identificare un segmento registrato dati i suoi Links è:

```
fin::DB::SearchResult searchFromLinks(
    const fin::core::Links &links,
    DB &db
)
```

definita in `fin_db.h`. Prende in input i Links della registrazione e un oggetto `DB`. Richiama il metodo `db.searchSongGivenLinks` che ritorna una struct `SearchResult`, composta come segue:

```
struct SearchResult {
    bool found;
    std::uint32_t id;
    float timeDelta;
    std::uint64_t commonLinks;
    std::string name;
};
```

Dove:

- `found` indica se il segmento è stato identificato correttamente, ovvero se è stato trovato un match nel database

[5] L'inizio del brano
[6] Rispetto all'inizio del segmento registrato

[7] Xiaolong Pan, Weiming Wu, and Yonghao Gu. Study and optimization based on mysql storage engine. In *Advances in Multimedia, Software Engineering and Computing Vol. 2: Proceedings of the 2011 MSEC International Conference on Multimedia, Software Engineering and Computing, November 26–27, Wuhan, China*, pages 185–189. Springer, 2012

- `id` è l'identificativo del brano originale identificato
- `timeDelta` è la differenza temporale in secondi tra il segmento registrato e la sua posizione nel brano originale
- `commonLinks` è il numero di Link[8] in comune tra il segmento e il brano originale
- `name` è il nome del brano originale

[8] Vedi 2.4

Quindi, se il brano viene identificato, vengono popolati di conseguenza i primi 4 campi della struct. Per reperire il nome del brano viene richiamato un ulteriore metodo: `db.getSongNameById`.

Nello specifico `db.getSongNameById` consiste, in una query su **musicinfo** per ottenere il nome associato all'id.

Il metodo `db.searchSongGivenLinks` richiede una trattazione specifica nel paragrafo 3.2.1.

3.2.1 *Il metodo db.searchSongGivenLinks*

Il metodo `db.searchSongGivenLinks` è il cuore della parte di identificazione di un brano. Si basa su tre idee di base:

1. Tra i Links del brano originale $Links_O$ e quelli del segmento registrato $Links_R$ dev'esserci una differenza temporale ΔT costante se si riferiscono allo stesso brano.[9]
2. La stessa differenza ΔT dev'essere non negativa[10]
3. Gli hash di $Links_O$ e di $Links_R$ devono corrispondere

[9] Se l'utente ha iniziato a registrare il brano dopo 15s è ragionevole pensare che la differenza temporale tra **recordings**.time e **_recording**.start sia costante e pari a 15s

[10] **recordings**.time $-$ **_recording**.start ≥ 0

A questo punto basterebbe quindi:

1. Raggruppare per ΔT e per id del brano originale, definendo con n il numero di elementi che ricadono nello stesso gruppo
2. Ordinare per n decrescente
3. Estrarre il primo gruppo che rappresenta il match migliore

Queste operazioni sono state realizzate in SQL con la query riportata di seguito:

```
SELECT
    recordings.songId,
    COUNT(*) AS n,
    recordings.time-_recording.start AS delta
FROM
    recordings INNER JOIN _recording
ON
```

```
    recordings.hash=_recording.hash
WHERE
    recordings.time>=_recording.start
GROUP BY
    delta,
    recordings.songId
ORDER BY n DESC
```

Una volta eseguita la query, il metodo `db.searchSongGivenLinks` estrae la prima prima riga del recordset e se $n > 15$[11], ovvero se il numero di Links in comune è maggiore di 15, popola la struct `SearchResult`, come riportato nel paragrafo 3.2.

In figura 3.3 è stato riportato un grafico in cui sull'asse delle ascisse è presente il numero di finestra nella quale è contenuto un Link della registrazione originale, analogamente accade per l'asse delle ordinate ma per il segmento registrato. Si nota facilmente la diagonale a densità maggiore intorno al valore 2000 sulle x, questo vuol dire che:

- Un match è stato trovato
- La registrazione è iniziata circa 2000 finestre dopo l'inizio del brano originale

[11] Costante `consts::links::MIN_HINT` in `consts.h`

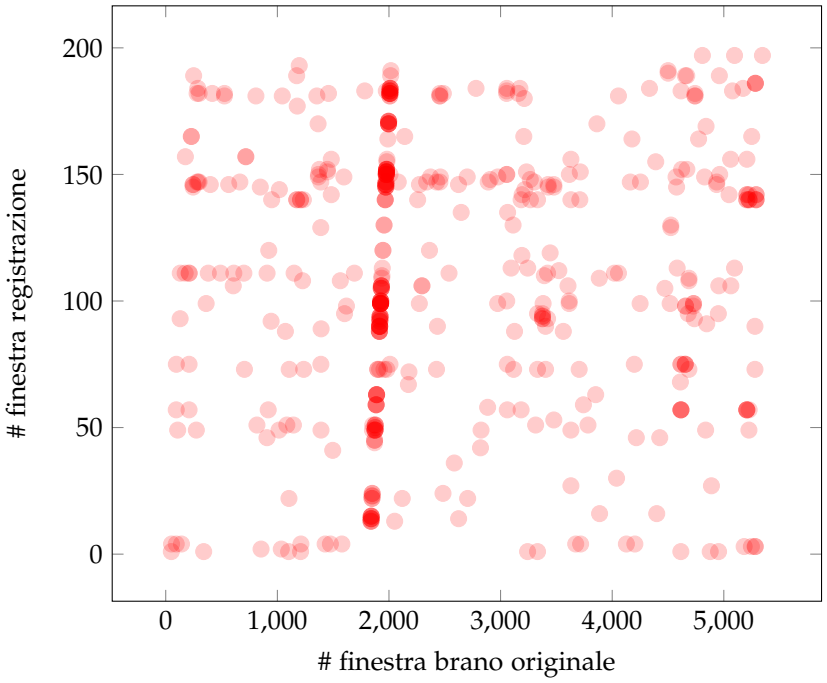

Figura 3.3: Scatterplot di un match

4
L'eseguibile server_entry

Riprendendo lo schema già presentato in figura 1.1 ed evidenziando lo scope di server_entry, si ottiene quando riportato in figura 4.1.

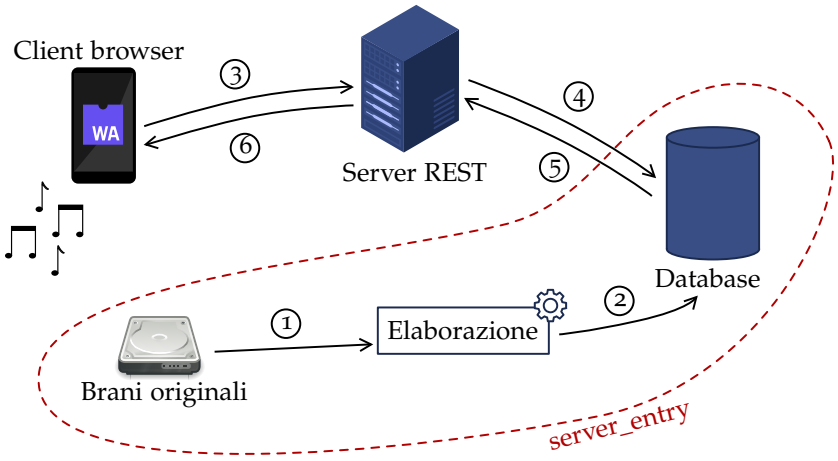

Figura 4.1: Schema architettura generale, dettaglio *server_entry*

Più nello specifico, l'eseguibile *server_entry* è la parte del sistema che si occupa di:

1. leggere i file Wave dei brani originali da dalla memoria di massa
2. estrarne i Links
3. memorizzare i Links e nomi dei brani all'interno del database

Server_entry non è altro che un wrapper costruito attorno alle funzionalità offerte dalla libreria fin_db (vedere figura 1.2).

Più tecnicamente, server_entry riceve attraverso la riga di comando un path che punta ad una cartella contenente dei Wave e per ogni Wave nella cartella richiama una singola funzione di fin_db:

```
void fin::insertSong(
    const std::string &filename,
    DB &db
)
```

Passando il nome del brano da processare e l'oggetto `DB` per accedere al database[1].

[1] Per maggiori dettagli vedere 3.1

5
L'eseguibile server_rest

Riprendendo lo schema già presentato in figura 1.1 ed evidenziando lo scope di *server_rest*, si ottiene quando riportato in figura 5.1.

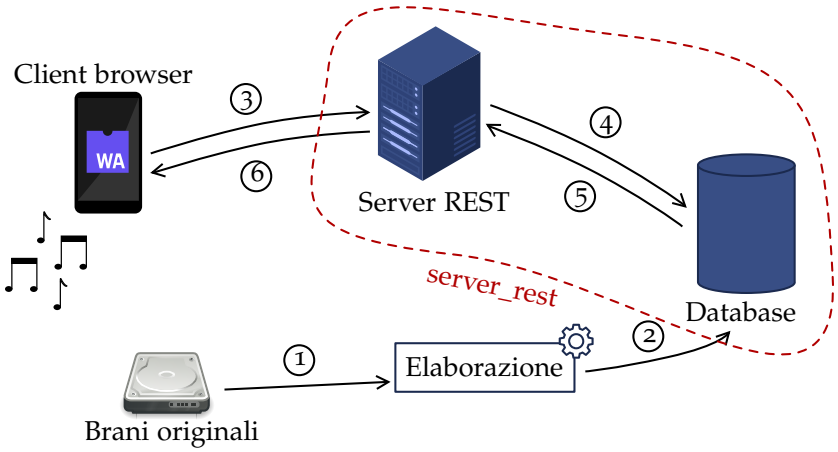

Figura 5.1: Schema architettura generale, dettaglio *server_rest*

Il compito dell'eseguibile server_rest è, quindi, quello di mettere a disposizione un endpoint REST per l'identificazione delle registrazioni audio.

Questo componente è basato sulla libreria open-source *cpp-httplib*[1] che implementa un semplice server HTTP bloccante. Server_rest sfrutta funzionalità sia della libreria fin, sia della libreria fin_db.

[1] `https://github.com/yhirose/cpp-httplib`

5.1 *Il problema del CORS*

Il protocollo *CORS*, ovvero Cross-Origin Resource Sharing[2], è stato introdotto per prevenire che un sito web malintenzionato possa accedere ai dati degli utenti di un altro sito web legittimo. In altre parole si vuole evitare, o quantomeno regolare, il resource sharing tra entità differenti per questioni di sicurezza.

I browser moderni che supportano CORS, prima di inviare la vera e propria richiesta[3] verso un'origin esterna, inviano una *preflight*

[2] Jianjun Chen, Jian Jiang, Hai-Xin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We still don't have secure cross-domain requests: an empirical study of cors. In *USENIX Security Symposium*, pages 1079–1093, 2018

[3] Una classica GET, POST, PUT o DELETE

request. La preflight request è una richiesta OPTIONS inviata dal browser al server esterno per verificare se le richieste cross-domain sono ammissibili e entro che limiti lo sono. Se il server risponde positivamente alla richiesta OPTIONS, il browser effettuerà la richiesta effettiva.

In questo modo, il protocollo CORS garantisce che solo le richieste provenienti da origine sicure e autorizzate possano accedere alle risorse del server esterno, garantendo la sicurezza nello scambio di informazioni tra origin diverse.

È stato necessario tener conto delle policy di sicurezza imposte dal protocollo CORS nella comunicazione tra il client wasm e il server_-rest (figura 5.2).

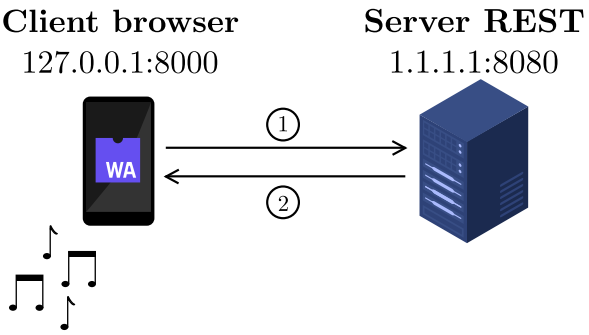

Figura 5.2: Schema richieste cross-origin

Anticipando quanto verrà trattato nel capitolo 7, nello scenario presentato, il client Wasm invia al server REST i Links che ha estratto dalla registrazione. Si noti, però, che la RIA Wasm viene servita da un server HTTP eseguito localmente al client. Di fatto, quindi, il server HTTP sul client (*origin 1*) vuole dialogare con il server HTTP REST (*origin 2*): si ha una richiesta cross-origin.

A questo punto, nel server_rest, si è reso necessario introdurre un *pre routing handler* che gestisca le richieste preflight OPTIONS, insieme ad un *post routing handler* che aggiunga alcuni header alla risposta del server, previsti dalle specifiche del protocollo CORS.

Tra gli header aggiunti, vale la pena citare `Access-Control-Allow-Origin`, che indica da quali domini un server può ricevere richieste cross-origin. Mentre in passato era possibile specificare un asterisco (*) come valore di questo header per consentire l'accesso a tutte le origin, oggi questa pratica è stata limitata per ragioni di sicurezza. Infatti, l'uso dell'asterisco potrebbe aprire la porta a attacchi di tipo *cross-site request forgery* (CSRF) e a violazioni della politica di sicurezza del browser.

Invece di utilizzare l'asterisco, è necessario specificare l'origin corretta, ovvero l'indirizzo dell'entità[4] che sta facendo la richiesta. Per questo motivo, non conoscendo a priori l'origin di una richiesta,

[4] In questo caso del server HTTP sul client

il server estrae l'origin dalla richiesta ricevuta, per poi impostare l'`Access-Control-Allow-Origin` di conseguenza.

5.2 *La serializzazione dei Links*

Giunti a questo punto nasce un altro problema: il client invia i Links calcolati al server, ma in che formato? I Links dovranno essere serializzati e deserializzati durante il transito tra client e server, così come indicato in figura 5.3.

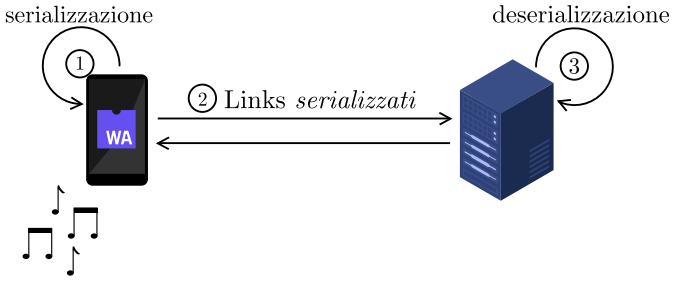

Figura 5.3: Schema serializzazione e deserializzazione Links

Si è deciso di evitare l'utilizzo di JSON, poiché il suo vantaggio dell'essere human-readable non compensa la sua scarsa efficienza nel trasportare dati[5].

Basti pensare che, mediamente, i Links estratti da una registrazione di 6/7 secondi sono circa 900. Ogni Link contiene due interi a 64 bit, ovvero 20 cifre decimali, che rappresentano rispettivamente l'hash e il numero della finestra di appartenenza. Se si rappresenta un Link in JSON come segue[6]:

```
[
    745129879060042, //hash
    307862961066999  //window
]
```

allora lo spazio occupato ammonta a:

$$len(\text{"[}\text{"}) + len(hash) + len(\text{","}) + len(window) + len(\text{"]}\text{"}) = 43B$$

Quindi 900 Links contenuti in un array JSON occuperanno:

$$len(\text{"[}\text{"}) + (len(Link) + len(\text{","})) \cdot 900 + len(\text{"]}\text{"}) = 39602B \cong 39KB$$

Al contrario, se si utilizza una serializzazione binaria, in cui si giustappongono i due interi Link in uno stream di byte (vedi figura 5.4), si occuperanno:

$$\texttt{sizeof(uint64\_t)} \cdot 2 \cdot 900 = 8B \cdot 1800 = 14400B \cong 14KB$$

Si ottiene quindi un risparmio pari al:

$$\left(1 - \frac{14400B}{39602B}\right) \cdot 100 \cong 64\%$$

[5] CJ Tauro, N Ganesan, SR Mishra, and Anupama Bhagwat. Object serialization: A study of techniques of implementing binary serialization in c++, java & .net. *Intl J of Computer Applications*, 45:25–29, 2012

[6] Ovvero come un array di due elementi

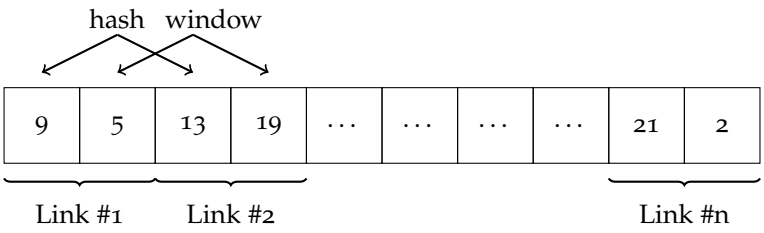

Figura 5.4: Rappresentazione serializzazione binaria Links

Esistono diverse librerie che gestiscono questo tipo di serializzazione[7], tuttavia si è scelto di scrivere un'implementazione custom. Questa scelta è stata dettata dal fatto di non voler aggiungere ulteriori dipendenze nel progetto, ma soprattutto dal fatto che la serializzazione/deserializzazione viene utilizzata una sola volta nell'intero sistema, ossia solo per inviare e riceve i Links.

A questo scopo è stata definita la classe `ByteBuffer` nel namespace `fin::utils` della libreria fin. La classe rappresenta un contenitore di bytes[8] che permette di:

[7] Ad esempio: ProtoBuf di Google o Apache Thrift

[8] Nello specifico `uint8_t`

- aggiungere un dato arbitrario nel buffer stesso, attraverso il metodo:

```
template<typename T>
void add(const T &data)
```

- rimuovere un dato dal buffer, con:

```
template<typename T>
void remove(T &data)
```

La classe `ByteBuffer` viene utilizzata all'interno della classe `Links` nei metodi `toByteBuffer()` e `fromByteBuffer(ByteBuffer &byteBuffer)` :

- Il primo si occupa di serializzare i Links in un `ByteBuffer`
- Il secondo, una factory, deserializza i Links presenti nel `ByteBuffer`

Ne segue che il dialogo tra client Wasm e rest_server avverrà come di seguito descritto:

1. Il client Wasm calcola i Links
2. Il client Wasm serializza i Links in un `ByteBuffer` richiamando `toByteBuffer()`
3. Il client Wasm estrae lo stream di byte dal `ByteBuffer`
4. Il client Wasm invia lo stream di byte al rest_server
5. Il rest_server riceve lo stream di byte

6. Il rest_server converte lo stream di byte in un `ByteBuffer`
7. Il rest_server deserializza i Links nel `ByteBuffer` Links richiamando `fromByteBuffer(ByteBuffer &byteBuffer)`

6
L'eseguibile mock_client

L'eseguibile *mock_client* (figura 6.1) viene utilizzato esclusivamente nelle attività di test, il suo scopo è verificare il corretto funzionamento del sistema. È l'equivalente del Wasm client, ma:

- Dialoga direttamente col database senza passare per il rest_server
- Non utilizza il microfono per registrare un segmento audio da identificare
- È un eseguibile *standard* e non una RIA

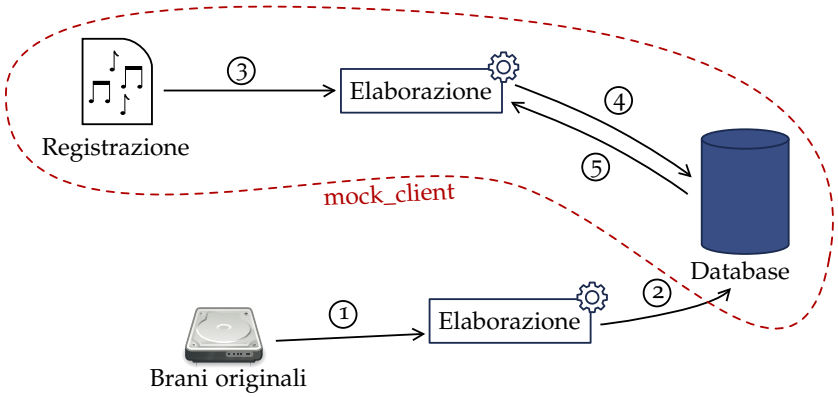

Figura 6.1: Schema architettura *mock_client*

Mock_client dipende da entrambe le librerie fin e fin_db. Prende in input il percorso di un file Wav da identificare e:

1. Carica i sample del file Wav utilizzando la classe `WavReader`
2. Calcola i Links a partire dai sample utilizzando la funzione `fin::computeLinks`
3. Serializza i Links in un `ByteBuffer`[1]
4. Deserializza i Links dal `ByteBuffer`

[1] L'unico scopo di questo passaggio è verificare il corretto funzionamento della serializzazione/deserializzazione

5. Dati i Links cerca nel database un match facendo ricorso a `fin::searchFromLinks`
6. Stampa a video il nome del brano identificato

7
L'eseguibile wasm_client

Il compito principale dell'eseguibile *wasm_client* è quello acquisire un breve segmento audio attraverso il microfono del client, estrarne i Links e inviarli al server_rest per avviare il processo di identificazione. Lo scope del client Wasm è quindi quello rappresentato in figura 7.1.

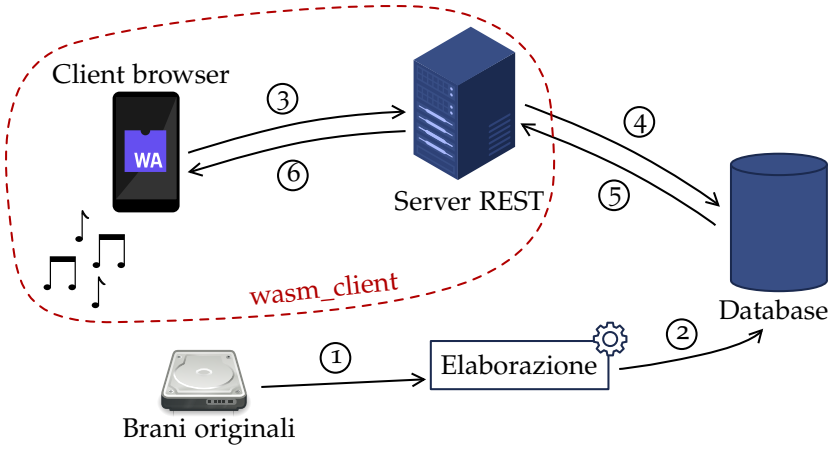

Figura 7.1: Schema architettura generale, dettaglio *wasm_client*

7.1 L'entry point

L'entry point del client Wasm è la funzione `main` che si occupa di:

1. Creare un `AudioContext`
2. Istanziare il *render thread* nel quale verrà eseguito il codice dell'`AudioWorklet`[1]

È stato infatti definito un `AudioWorklet`[2] per l'estrazione dei Links.

La creazione del render thread è, in realtà, un'operazione asincrona: una volta terminata viene richiamata la callback `audioWorklet-ProcessorCreated`.

[1] Paul Adis Sheppy, chrisdavidmills. Audioworklet, 2021. URL `https://developer.mozilla.org/en-US/docs/Web/API/AudioWorklet`

[2] Nient'altro che un `AudioNode` custom

7.2 *La callback audioWorkletProcessorCreated*

La callback `audioWorkletProcessorCreated` si occupa della creazione vera e propria dell'`AudioWorklet` per l'estrazione dei Links e della preparazione dell'interfaccia da proporre all'utente.

Il primo passo è la creazione dell'`AudioWorklet` configurato come segue:

- Un solo input
- Nessun output
- Una funzione per il processamento custom chiamata `processAudio`

A questo punto può essere creata la pagina HTML da mostrare all'utente.

Per prima cosa viene richiesto l'accesso al microfono dell'utente attraverso le API navigator JavaScript, richiedendo la disabilitazione dell'`echoCancellation` e del `noiseSuppression`, al fine di ottenere uno stream audio quanto più *raw* possibile. Inoltre è stato impostato il `channelCount` a 1 per avere uno stream mono[3].

[3] Esistono microfoni stereo

Viene quindi configurato il routing dei nodi nell'`AudioContext` già creato come indicato in figura 7.2.

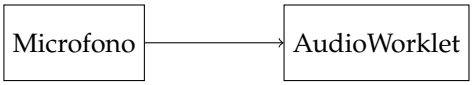

Figura 7.2: Routing `AudioNodes`

Infine viene aggiunto un bottone *Record* che innesca il processo di registrazione del segmento audio e la sua identificazione.

7.3 *La funzione processAudio*

Nello scope globale del client Wasm è stato dichiarato un `DummyReader`[4] che conterrà i vari campioni audio prodotti dal microfono.

[4] Vedere la sezione 2.1

La funzione `processAudio` è responsabile di leggere i campioni audio inviati all'`AudioWorklet` dal microfono e memorizzarli all'interno del `DummyReader`. Una volta che il `DummyReader` contiene abbastanza campioni il processo di estrazione dei Links può essere avviato.

Si noti però che una volta estratti i Links questi dovranno essere inviati al server, il che presuppone il poter utilizzare l'API JavaScript `fetch`. Tuttavia non è possibile usare questa API all'interno del rendering thread dell'`AudioWorklet`: bisogna ritornare nel main thread JavaScript. A questo scopo viene richiamata la funzione

```
void emscripten_audio_worklet_post_function_v(
        EMSCRIPTEN_WEBAUDIO_T id,
```

```
        void (*funcPtr)(void)
)
```

passando come primo parametro `EMSCRIPTEN_AUDIO_MAIN_THREAD` ad indicare di voler dialogare con il main thread e come secondo parametro `messageReceivedOnMainThread`, un puntatore a funzione di una callback che verrà eseguita nel main thread.

7.4 *La callback messageReceivedOnMainThread*

Ritornati nel main thread JavaScript e con abbastanza campioni, si può avviare il processo di identificazione. Si procede come segue:

1. Vengono calcolati i Links dal `DummyReader` utilizzando la funzione `fin::computeLinks`
2. I Links vengono serializzati in un `ByteBuffer` attraverso `toByteBuffer`
3. Viene estratto lo stream di byte dal `ByteBuffer`
4. Viene configurata la `fetch` per eseguire una POST verso il server REST che ha come body lo stream di byte
5. Viene effettuata la `fetch`
6. Se la `fetch` ha successo e la registrazione viene identificata allora viene mostrato il nome del brano, insieme ad alcune informazioni accessorie

7.5 *La durata del segmento audio*

Sorge spontanea una domanda: qual è la lunghezza minima del segmento audio da registrare per avere un buon recognition rate?

Similmente a quanto già fatto nel paragrafo 2.2.1, si è predisposto un altro *ambiente di test*, composto dai già citati 400 brani e per ognuno di questi brani:

1. è stato estratto un segmento con una durata compresa tra 0.5 e 6 secondi, con un passo di 0.5 secondi
2. per ogni segmento è stato avviato il processo di identificazione[5]

[5] Facendo ricorso a `mock_test`

Infine, è stato contato quante volte l'identificazione ha avuto successo, ottenendo il grafico in figura 7.3.

In altre parole, con 3 secondi si ottiene un recognition rate più che accettabile, vicino al 98%.

Tuttavia, è opportuno considerare, ancora una volta, che le condizioni di test potrebbero non corrispondere esattamente a quelle di un

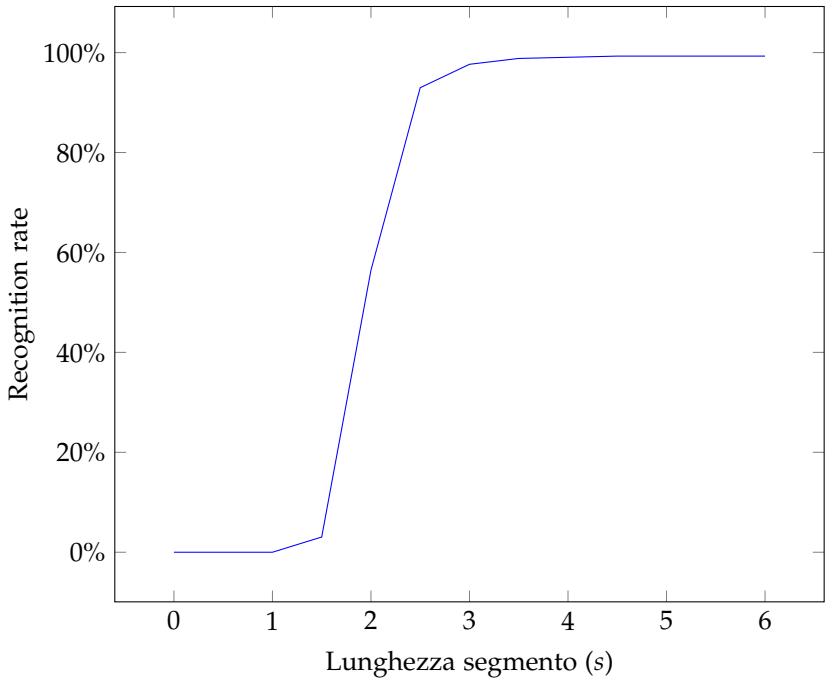

Figura 7.3: Recognition rate in funzione della lunghezza del segmento audio registrato

ambiente reale. Pertanto, per tenere conto dell'impossibilità di simulare perfettamente l'ambiente reale, sarebbe ragionevole scegliere una durata superiore ai 3 secondi.

Nel client Wasm viene utilizzata una durata pari a 5 secondi.

7.6 Ulteriori problemi col CORS

Wasm utilizza l'oggetto `SharedArrayBuffer` per consentire una comunicazione efficiente tra thread e migliorare le prestazioni. `SharedArrayBuffer` è un tipo di buffer condiviso, utilizzato, ad esempio, per la comunicazione tra main thread e rendering thread, permettendo contemporaneamente a entrambi di poter accedere in modo sincronizzato ai dati in esso contenuti.

Tuttavia, `SharedArrayBuffer` può rappresentare anche una minaccia per la sicurezza se utilizzato in modo improprio[6], poiché può essere utilizzato per attacchi di tipo side-channel tipo Spectre, che consentono a un sito web malintenzionato di accedere ai dati sensibili di altri siti web aperti contemporaneamente nella stessa finestra del browser. Per questo motivo, l'utilizzo di `SharedArrayBuffer` è subordinato all'implementazione di due policy di sicurezza sul server che ospita la RIA, anch'esse definite nel protocollo CORS: Cross-Origin-Embedder-Policy (COEP) e Cross-Origin-Opener-Policy (COOP).

[6] Eiji Kitamura. Making your website "cross-origin isolated" using coop and coep, 2020. URL `https://web.dev/coop-coep/`

In particolare, COEP con il valore *require-corp* specifica che l'origin dev'essere la stessa tra l'ambiente in cui è stato creato lo `SharedArrayBuffer` e l'ambiente in cui viene utilizzato.

8
L'eseguibile lyrics

L'eseguibile *lyrics* ha un comportamento simile a wasm_client[1], ma, oltre al processo di identificazione del segmento audio, viene presentato sul client il testo del brano riconosciuto, sincronizzato in tempo reale.

Lo scope di *lyrics* è quello in figura 8.1.

[1] Vedi capitolo 7

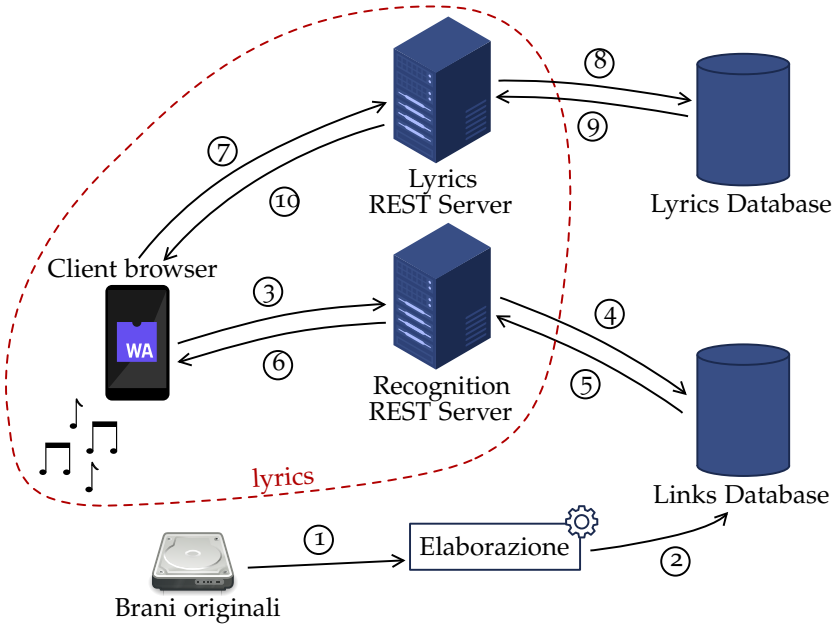

Figura 8.1: Schema architettura generale, dettaglio *lyrics*

Il processo dal punto ① al punto ⑥ è pressoché identico a quanto già presentato per wasm_client.

Le uniche differenze introdotte sono dovute al fatto che è necessario sincronizzare il brano che l'utente sta ascoltando col relativo testo: verranno analizzate di seguito.

In figura 8.2 un esempio di funzionamento.

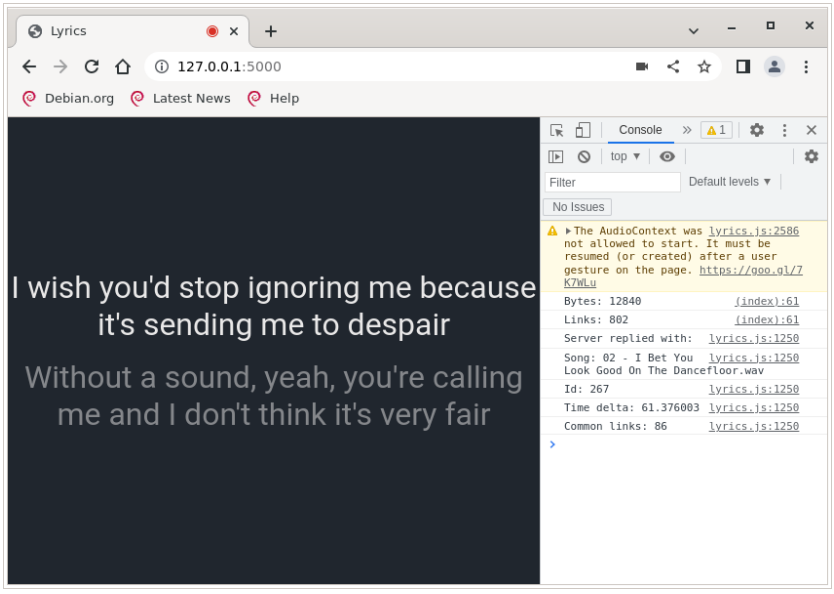

Figura 8.2: Screenshot funzionamento *lyrics*

8.1 *Il server REST lyrics*

Il *server REST lyrics* si occupa di fornire il testo di un determinato brano. Espone un unico endpoint: `/lyrics/{song_id}`, dove `song_id` è l'id del brano del quale si vuole ottenere il testo.

La risposta ritornata è in formato JSON e segue uno schema come quello indicato di seguito:

```
{
    "song_id": song_id,
    "lyrics": [
        {
            "offset": 1.2,
            "text": "Line 1"
        }, {
            "offset": 2.2,
            "text": "Line 2"
        },
        ...
    ]
}
```

Dove `offset` indica l'offset temporale in secondi, rispetto all'inizio del brano originale, in cui viene cantato il testo `text`.

8.2 *La funzione processAudio*

La funzione `processAudio` è molto simile a quella già discussa per wasm_client nel paragrafo 7.3, le uniche differenze significative sono riportate di seguito:

- È stata introdotta una variabile globale `std::chrono::system_clock::time_point firstSampleTime`
- La variabile `firstSampleTime` viene valorizzata quando vengono inseriti i primi campioni nel `DummyReader`

In altre parole, `firstSampleTime` è un riferimento temporale a quando è stato acquisito il primo campione audio della registrazione da identificare.

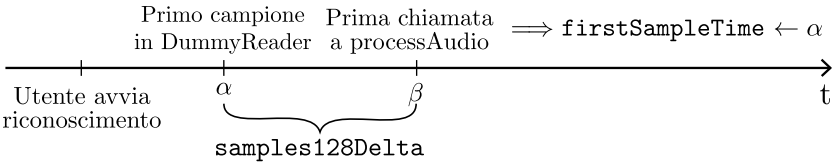

Figura 8.3: Timeline con dettaglio `firstSampleTime`

Con riferimento alla figura 8.3, `firstSampleTime` viene valorizzato come segue:

```
std::chrono::milliseconds samples128Delta(
    PROCESS_SAMPLES * 1000 / SAMPLE_RATE
);
firstSampleTime =
    std::chrono::system_clock::now() - samples128Delta;
```

È necessario, infatti, sottrarre `samples128Delta` in quanto all'assegnazione di `firstSampleTime` l'`AudioWorklet` ha già ricevuto 128 campioni.

8.3 *La funzione getElapsedTimeSinceFirstSample*

La funzione `getElapsedTimeSinceFirstSample` si occupa di calcolare quanti secondi sono passati da quando è stata valorizzata la variabile globale `firstSampleTime`.

È stato fatto in modo che questa funzione possa essere richiamata anche da JavaScript, sfruttando il meccanismo del *binding*:

```
EMSCRIPTEN_BINDINGS(my_module){
    emscripten::function(
        "getElapsedTimeSinceFirstSample",
        &getElapsedTimeSinceFirstSample
```

```
    );
}
```

In questo modo da JavaScript la funzione potrà essere richiamata con un semplice:

```
const elapsedTime =
    Module.getElapsedTimeSinceFirstSample();
```

8.3.1 L'utilizzo del clock corretto

Nel contesto di calcolo della differenza temporale tra due istanti, normalmente, è essenziale fare affidamento su clock garantiti monotoni crescenti, come ad esempio `steady_clock` o `high_resolution_clock`. Tuttavia, occorre considerare che la variabile `firstSampleTime` viene inizializzata nel rendering thread dell'AudioWorklet, mentre la funzione `getElapsedTimeSinceFirstSample` viene invocata nel main thread di JavaScript: in altre parole non è detto che il clock che valorizza `firstSampleTime` e quello utilizzato per calcolare la differenza temporale utilizzino lo stesso riferimento nel tempo. La mancanza di sincronizzazione può portare, quindi, a risultati privi di significato nel richiamo di `getElapsedTimeSinceFirstSample`. Pertanto, risulta necessario utilizzare il `system_clock`, che seppur non monotono crescente, garantisce l'utilizzo dello stesso riferimento temporale per ogni sua istanza, ovvero i secondi passati dalla *UNIX epoch*.

8.4 La callback messageReceivedOnMainThread

Inizialmente la callback `messageReceivedOnMainThread` si comporta esattamente come quella già presentata per wasm_client nel paragrafo 7.4. Tuttavia, dopo il processo di riconoscimento, oltre all'id `song_id` del brano riconosciuto, viene memorizzato anche l'offset temporale `time_delta` tra la registrazione e il brano originale[2].

[2] Vedi paragrafo 3.2

A questo punto:

1. Viene reperito il testo del brano con id `song_id`, facendo una richiesta verso il server REST lyrics
2. Viene richiamata la funzione `getElapsedTimeSinceFirstSample`
3. Viene schedulata la visualizzazione del testo al tempo corretto: il $\texttt{text}_i$, ovvero il testo i-esimo, dovrà essere visualizzato all'istante (vedi figura 8.4):

$$\alpha = \texttt{offset}_i - \texttt{elapsedTime} - \texttt{time\_delta}$$

La schedulazione avviene facendo ricorso alla funzione `setTimeout` di JavaScript.

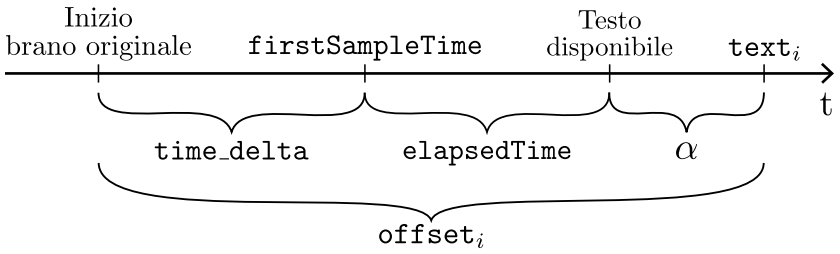

Figura 8.4: Timeline visualizzazione lyrics

9
Utilizzi futuri

9.1 Sincronizzazione di contenuti provenienti da sorgenti differenti

Uno scenario tipico nella post-produzione di materiale audiovisivo è la sincronizzazione di materiale audiovisivo proveniente da più fonti. Si pensi ad uno scenario come quello in figura 9.1:

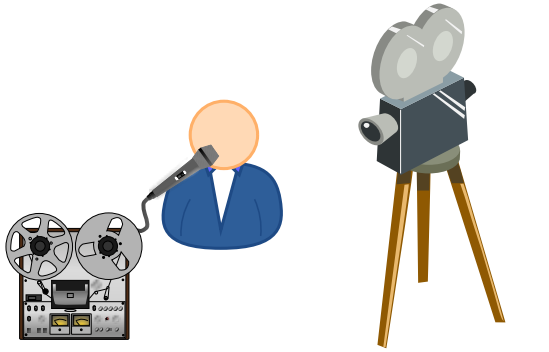

Figura 9.1: Scenario acquisizioni multiple

1. Una videocamera acquisisce un contenuto audio e video, l'audio viene registrato attraverso il microfono interno della videocamera
2. Un registratore audio acquisisce attraverso un microfono (per esempio un lavalier) l'audio di un operatore

In situazioni di questo tipo è quindi necessario sincronizzare l'audio «ambientale» registrato dalla videocamera con quello proveniente dal registratore esterno.

Tipicamente si fa ricorso ad un sistema basato su timecode. Il timecode fornisce un riferimento comune di tempo che consente di coordinare con precisione più fonti multimediali, facilitando il

montaggio e la post-produzione. Tuttavia, l'impiego del timecode richiede attrezzature specifiche, potenzialmente aumentando i costi e la complessità del processo di produzione. Inoltre, in caso di errori o discrepanze nella generazione o nella lettura del timecode, potrebbero verificarsi problemi di sincronizzazione che richiedono ulteriori sforzi di correzione.

Un alternativa, soprattutto in ambienti meno «professionali», potrebbe essere quella di utilizzare un sistema basato sull'algoritmo descritto precedentemente. Si faccia riferimento alla figura 9.2:

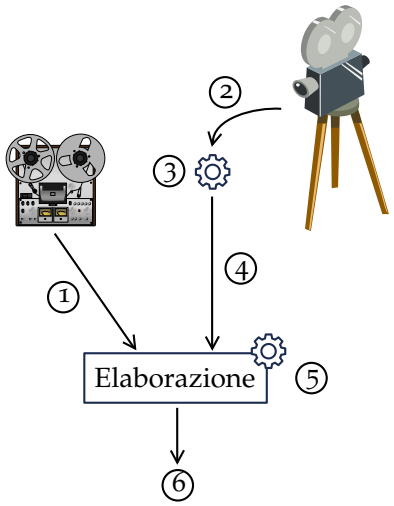

Figura 9.2: Sincronizzazione acquisizioni multiple

1. L'acquisizione audio è disponibile
2. L'acquisizione audio/video è disponibile
3. L'audio viene estratto dall'acquisizione[1]
4. L'acquisizione audio estratta è disponibile
5. Viene avviato l'algorimo per identificare l'*offset temporale* tra le due acquisizioni
6. L'offset temporale viene visualizzato all'utente

[1] La sincronizzazione avviene sul segnale audio e non sui fotogrammi

A questo punto, l'operatore di post-produzione può sincronizzare le due acquisizioni, evitando di utilizzare sistemi basati su timecode.

Si potrebbe pensare, inoltre, facendo riferimento alla figura 1.1, di adottare un'architettura semplificata del sistema:

- Il client, il server REST, il database e i segnali audio da analizzare risiederebbero tutti sulla stessa macchina

- Sebbene si continuerà ad utilizzare un database per le operazioni di memorizzazione dei Link[2], si potrebbe utilizzare un DBMS più lightweight come SQLite
- Il database dei Link andrebbe creato e distrutto ad ogni esecuzione dell'algoritmo

[2] Vedi paragrafo 2.4

10
Bibliografia

Sabah A Abdulkareem and Ali J Abboud. Evaluating python, c++, javascript and java programming languages based on software complexity calculator (halstead metrics). In *IOP Conference Series: Materials Science and Engineering*, volume 1076. IOP Publishing, 2021.

William L Briggs and Van Emden Henson. *The DFT: an owner's manual for the discrete Fourier transform*. SIAM, 1995.

Jianjun Chen, Jian Jiang, Hai-Xin Duan, Tao Wan, Shuo Chen, Vern Paxson, and Min Yang. We still don't have secure cross-domain requests: an empirical study of cors. In *USENIX Security Symposium*, pages 1079–1093, 2018.

Hongchan Choi. Audioworklet: the future of web audio. In *ICMC*, 2018.

Piero Fraternali, Gustavo Rossi, and Fernando Sánchez-Figueroa. Rich internet applications. *IEEE Internet Computing*, 14(3):9–12, 2010.

Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 185–200, 2017.

Pankaj Kamthan. Java applets in education. *Electronic Resource] Retrieved on May*, 17, 1999.

Eiji Kitamura. Making your website "cross-origin isolated" using coop and coep, 2020. URL `https://web.dev/coop-coep/`.

Dan Maharry. *TypeScript revealed*. Apress, 2013.

Thiago Nicolini, Andre Hora, and Eduardo Figueiredo. On the usage of new javascript features through transpilers: The babel case. *IEEE Software*, pages 1–3, 2023.

Xiaolong Pan, Weiming Wu, and Yonghao Gu. Study and optimization based on mysql storage engine. In *Advances in Multimedia, Software Engineering and Computing Vol. 2: Proceedings of the 2011 MSEC International Conference on Multimedia, Software Engineering and Computing, November 26–27, Wuhan, China*, pages 185–189. Springer, 2012.

Paul Pedersen. The mel scale. *Journal of Music Theory*, 9(2):295–308, 1965.

Prajoy Podder, Tanvir Zaman Khan, Mamdudul Haque Khan, and M Muktadir Rahman. Comparative performance analysis of hamming, hanning and blackman window. *International Journal of Computer Applications*, 96(18):1–7, 2014.

Guillermo Rauch. *Smashing node. js: Javascript everywhere*. John Wiley & Sons, 2012.

Adam D Scott. *JavaScript everywhere: building cross-platform applications with GraphQL, React, React Native, and Electron*. O'Reilly Media, 2020.

Charles Severance. Javascript: Designing a language in 10 days. *Computer*, 45(2):7–8, 2012.

1valdis Sheppy, chrisdavidmills. Audioworklet, 2021. URL `https://developer.mozilla.org/en-US/docs/Web/API/AudioWorklet`.

CJ Tauro, N Ganesan, SR Mishra, and Anupama Bhagwat. Object serialization: A study of techniques of implementing binary serialization in c++, java & .net. *Intl J of Computer Applications*, 45:25–29, 2012.

MW Trethewey. Window and overlap processing effects on power estimates from spectra. *Mechanical Systems and Signal Processing*, 14 (2):267–278, 2000.