Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

# Identificación de piezas musicales en una base de datos

Trabajo Fin de Grado

**Grado en Ingeniería Informática**

**Autor**: José Manuel Mañogil López

**Tutor**: Pedro Alonso Jordá

2018/2019

# Resumen

Este proyecto se centra en el análisis y optimización de una aplicación C que realiza la búsqueda de una pieza de música clásica en una base de datos a partir de un fragmento dado de ésta. La idea principal es que la aplicación aproveche la máxima potencia de la máquina y obtenga un resultado en un corto periodo de tiempo. Además, se creará una estructura cliente-servidor mediante PHP y un bot de TELEGRAM, al cuál se le podrá enviar un fragmento de canción con objeto de averiguar la original.

**Palabras clave:** canción, búsqueda, optimización, Telegram bot.

# Abstract

This project focuses on the analysis and optimization of a C application that seeks a piece of classical music in a database given fragment. The main idea is the application to take full advantage of the power of the computer and to obtain a result in a short period of time. In addition, a client-server structure will be created using PHP and a TELEGRAM bot, to which it can be sent a song fragment with the aim to find out the original.

**Keywords:** song, optimization, search, Telegram bot.

# Tabla de contenidos

# Tabla de Figura

# Capítulo 1. Introducción

La identificación de piezas musicales es una facilidad que ha captado gran popularidad en los últimos años al permitir a cualquier persona identificar canciones que se están escuchando con bastante facilidad. Las aplicaciones orientadas a esto encuentran la canción original a partir de un solo fragmento de la canción que puede ser virtualmente pequeño. El proceso de búsqueda de la canción original a la que pertenece dicho fragmento debe ser rápido, pues la base de datos donde se busca suele estar compuesta por miles o millones de canciones.

La mayoría de aplicaciones que realizan este trabajo utilizan algoritmos que permiten la comparación del fragmento de la canción proporcionado con miles o millones de canciones en apenas unos segundos. Para obtener velocidad, este tipo de algoritmos suele simplificar la canción obteniendo solo los puntos más significativos del espectrograma del audio. El principal problema de este tipo de algoritmos es que solo aceptan fragmentos de canciones que sean prácticamente copias exactas de la canción original para poder devolver el resultado correcto, es decir, los fragmentos no pueden tener variaciones en el *tono* o en el *tempo*. Se da la circunstancia, además, de que la música clásica suele estar expuesta a variaciones en el *tempo* o en el tono, haciendo casi imposible para este tipo de aplicaciones el reconocer la pieza si el fragmento dado contiene alguna de estas variaciones.

Para realizar este tipo de comparación en la cual existen diferencias en la interpretación de la canción original y del fragmento dado, se puede utilizar un algoritmo conocido como *Dynamic Time Warping* (*DTW*), el cuál permite la comparación de dos canciones aunque en su interpretación la velocidad sea diferente. El mayor inconveniente de este algoritmo, sin embargo, es el elevado tiempo de cómputo que requiere la comparación de las canciones.

Este trabajo de fin de grado tiene el propósito de optimizar una aplicación que utiliza el algoritmo *DTW*.

## 1.1 Objetivos

El principal objetivo de este trabajo de fin de grado es el análisis de una aplicación en C que realiza la identificación de una pieza musical clásica en una base de datos con el fin de optimizarla y, por tanto, de reducir su tiempo de ejecución, manteniendo un porcentaje de aciertos alto. Como objetivo adicional, se creará una aplicación cliente-servidor para poder disponer de esta aplicación en cualquier lugar.

La aplicación cliente-servidor utilizará un *bot* de TELEGRAM, gestionado mediante PHP en la parte del servidor para recibir un archivo de audio. Una vez recibido este archivo de audio, éste se tratará con objeto de que tenga el formato adecuado que utiliza la aplicación en C. La aplicación en C recibe una lista de canciones procesadas y el fragmento de una de ellas que debe procesar. Debe obtener la canción original del fragmento utilizando el potencial de la máquina donde se ejecute para obtener el menor tiempo posible con un alto porcentaje de aciertos. Acabada la ejecución de la aplicación en C ésta debe devolver el título de la canción con mayor probabilidad de ser la misma, y en el menor tiempo de ejecución posible.

# Capítulo 2. Contexto tecnológico

Una de las aplicaciones más famosas mundialmente para el reconocimiento de canciones es SHAZAM, con más de 100 millones de descargas en distintos dispositivos, es la aplicación favorita para la identificación de piezas musicales.

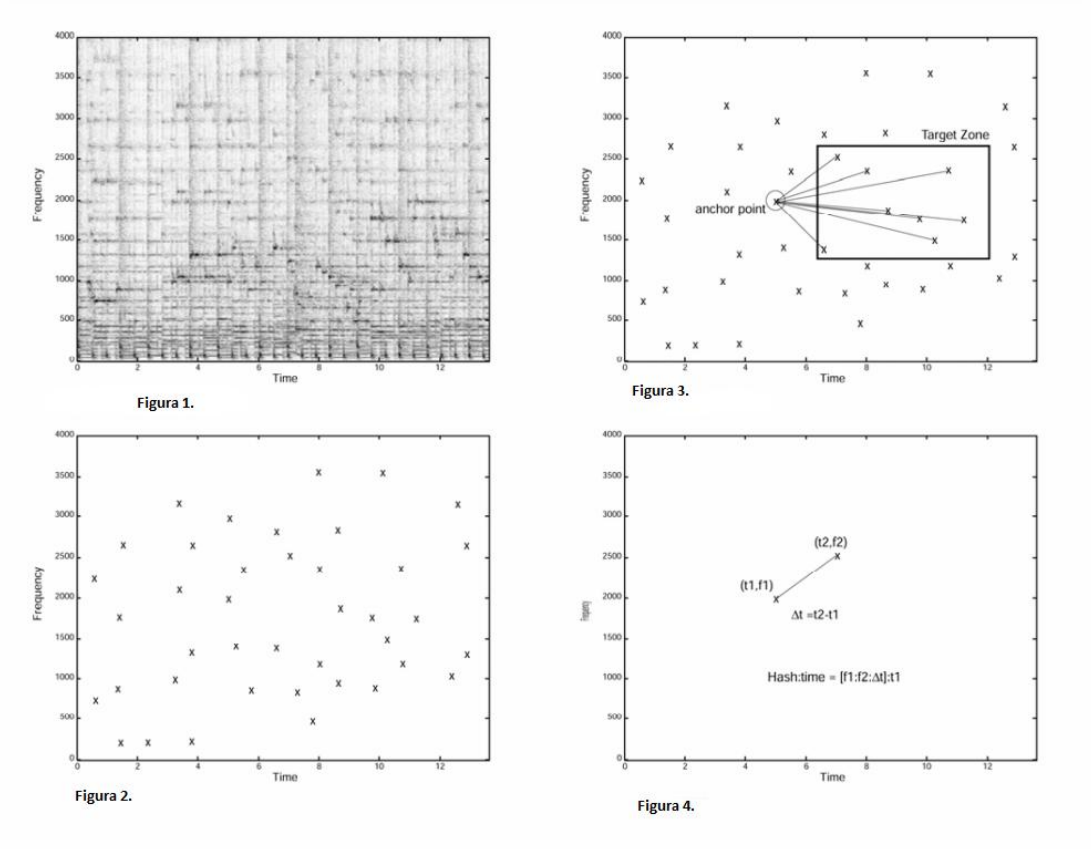


**Figura 1 Funcionamiento de SHAZAM**

SHAZAM [1] almacena las canciones en su base de datos en forma de huella digital de las cuales puede extraer *hash tokens* para la comparación con los fragmentos dados. El proceso para la obtención de estos *hash tokens* consiste en la reducción del espectrograma de frecuencia (Figura 1-1), en un mapa de puntos con las frecuencias más intensas (Figura 1-2). La generación de estos puntos debe ser semi-aleatoria ya que deben haber suficientes como para que de un fragmento de la canción se originen puntos de forma parecida a la original, pero no deben generarse demasiados puntos que eleven el tiempo de cómputo. A partir del mapa generado, a cada punto se le asocia una zona que constaría de alrededor de los 10 siguientes puntos, como se muestra en la Figura 1-3. A cada punto se asocia su zona generando *hash tokens* con los valores del primer punto, el par asociado y la diferencia de tiempo en la canción de estos, como apreciamos en la Figura 1-4.

Este tipo de almacenamiento permite la comparación entre pares en lugar de ir comparando cada punto. Con este se consigue una mejora de velocidad de ejecución de alrededor de 10.000 veces más rápido con el inconveniente de necesitar 10 veces más almacenamiento al generar 10 veces más de *hash tokens*.

Existen otras aplicaciones parecidas, como SOUNDHOUND o MUSIXMATCH.

El principal problema de estas aplicaciones para la identificación de música clásica es que, para una misma pieza musical, diferentes actuaciones pueden variar en el *tempo* y en la *intensidad*. Una pieza clásica puede tener una indicación de *tempo andante*, que puede variar de entre 76 a 108 notas por minuto, dependiendo del director. Como se ha visto, SHAZAM utiliza el tiempo para determinar si un fragmento de una canción pertenece a ésta, haciendo que la identificación de piezas clásicas dé errores.

Una forma de solventar este problema consiste en tener la misma pieza musical con diferentes *tempos*, como se tienen varias piezas musicales iguales de la misma canción cuya diferencia es la zona de grabación, que puede ser un estudio u otro lugar en el que se ha grabado el concierto, entre otras. Otra opción sería utilizar *DTW*, un algoritmo que permite la comparación de archivos de audio sin que la comparación se vea afectada por la velocidad de la interpretación. El principal inconveniente de *DTW* es que su coste algorítmico, que es mayor que el algoritmo que emplea SHAZAM, por ejemplo.

La aplicación utilizada en este Trabajo Final de Grado se basa en este algoritmo, ya que está enfocada a música clásica es la mejor opción para tratar los cambios de velocidad en las piezas, siendo el principal objetivo de este TFG su optimización para que sea una aplicación competente.

# Capítulo 3. Tecnologías empleadas

En este capítulo se describen las tecnologías utilizadas para el desarrollo de este trabajo de fin de grado.

## 3.1 Telegram

La aplicación desarrollada sigue el esquema cliente-servidor. En el lado del cliente se utiliza la aplicación de Telegram, que es una aplicación de mensajería y VOIP basada en el protocolo *MTProto (Mobile Transport Protocol)* [7]. Esta aplicación permite, además, el envió de archivos multimedia. De Telegram se usará más concretamente su interfaz de programación Telegram Bot API, la cuál permite a cualquier persona de forma gratuita crear y registrar usuarios *bot*. Estos usuarios autónomos pueden ser configurados de diferentes formas tales que pueden reconocer comandos de texto, administrar canales, aceptar pagos bancarios y muchas otras opciones. En este caso el *bot* se centrará en gestionar archivos de audio.

## 3.2 PHP

Para la parte de servidor, se utiliza PHP, que es un lenguaje de programación general ambientado a la parte del servidor. PHP nos permite gestionar las respuestas del *bot* de Telegram, modificar el audio para que cumplan los requisitos adecuados y ejecutar el programa de identificación de piezas musicales.

## 3.3 Shizmidi

Para la identificación de piezas musicales se utiliza un código de libre software llamado ***Shizmidi***, que es una aplicación desarrollada en C y creada por la colaboración de las universidades españolas de Jaén, Oviedo y la Politécnica de Valencia. El objetivo de Shizmidi es poder identificar fragmentos de canciones que se le proporcionen a través de un archivo de audio o con el micrófono para poder identificar su pieza musical. A parte de esta función, también tiene la opción de utilizarse como acompañante al tocar una obra musical (*instrumento virtual*). En este proyecto solo se utilizará la parte de *identificación de piezas musicales, también conocida dentro de Shizmidi como ReMAS (Real-time Audio to Score Alignment*).

Como se puede apreciar en la Figura 2, la identificación de piezas musicales tiene dos fases bien diferenciadas. La primera fase o fase de pre-procesamiento es la aquella en la cuál se preparan las partituras originales que serán utilizadas en la siguiente fase. En la segunda fase o fase de alineación el fragmento de audio introducido es comparado con las partituras procesadas en la fase anterior.

### 3.3.1 Fase de pre-procesamiento

En esta fase es donde se prepara la base de datos que contiene las canciones que se utilizan en la fase de alineación con el fin de compararlas con el audio proporcionado. El objetivo de esta fase es convertir la partitura original, la cual es proporcionada en forma de partitura MIDI (Musical Instrument Digital Interface), en un vector de características en el tiempo de la forma $U = (u_1,..., u_n,..., u_N)$. Para llegar a este punto

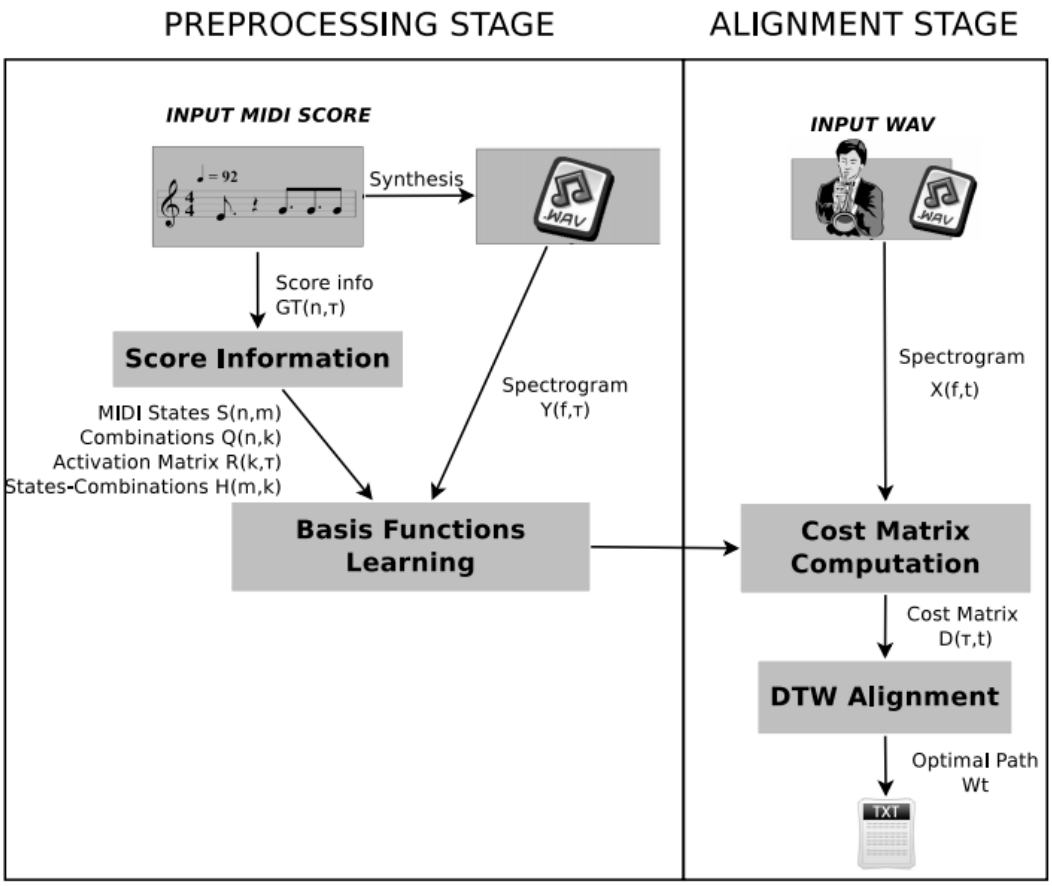


**Figura 2 Diagrama de ReMAS**

primero se analiza la partitura para detectar cuantas combinaciones únicas de notas ocurren durante toda la pieza musical. Cada combinación única de notas es llamada *unidad de partitura*. Una vez que la partitura ha sido estructurada en *unidades de partitura*, se aprende un patrón espectral o espectrograma único por cada *unidad de partitura*. Para realizar esto, la versión MIDI de la partitura tiene que ser convertida en un fichero de audio utilizando un sintetizador de software. El resultado es una señal de audio de baja calidad de la cual la activación de cada *unidad de partitura* es conocida. Esta señal es convertida a un espectrograma y descompuesta utilizando una matriz factorial no negativa (NMF). Como resultado se obtiene un espectrograma único por cada *unidad de partitura*.

Cada característica $U_n$ en la secuencia U es un espectrograma correspondiente a la activación de la combinación de notas en el instante n. Es decir, que para cada combinación de notas o *unidad de partitura* que aparece en la partitura sabemos el momento de activación en ésta y el espectrograma que se genera al tocar dichas notas con sus correspondientes instrumentos.

### 3.3.2 Fase de alineación

Esta es la fase donde se busca la canción original a partir de un fragmento dado. En la figura 3 se puede apreciar un esquema de los pasos que se siguen. La señal proporcionada debe tener unas características específicas tales como debe ser un audio monoaural, con una frecuencia de muestreo de 44,1kHz y una muestra de 16 bits. Una vez proporcionada dicha señal, lo primero que se realiza es la extracción de la información, para ello se debe obtener un espectrograma de bajo nivel. Primero, se aplica

una ventana de *Hanning* con un tamaño de 128 milisegundos y un salto de 10 milisegundos. Una vez obtenido la ventana se realiza la transformada rápida de Fourier (FFT) y el resultado se convierte de frecuencia linear a un archivo MIDI.

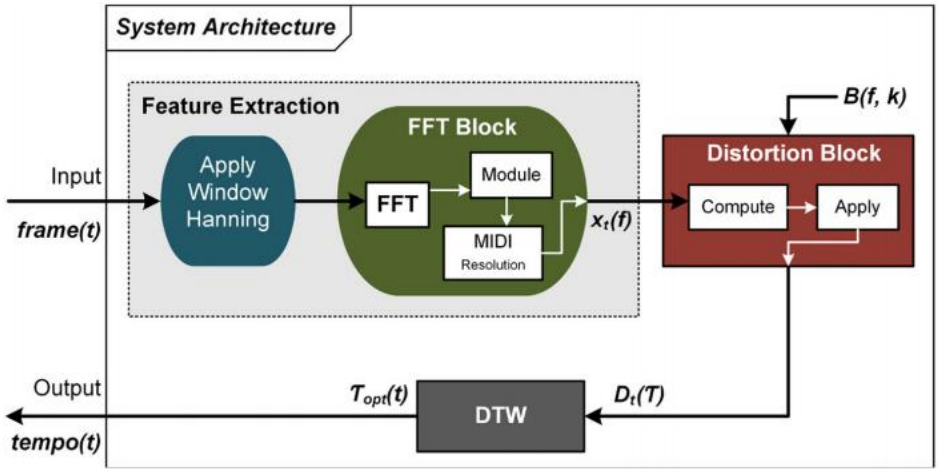


**Figura 3 Diagrama de alineación**

En el bloque de distorsión se calcula la posibilidad de que cada *unidad de partitura* sea activada en cada tramo. Para esto se analiza la probabilidad entre el espectrograma de la fase de pre-procesamiento y el obtenido en la etapa anterior de la señal de entrada. La matriz obtenida en esta etapa es utilizada en la última fase, la fase de alineación. En este último paso se utiliza el algoritmo *Dinamyc Time Warping* (DTW) para calcular el coste de comparación de dos audios.

DTW [5] es una técnica para la lineación de dos series de tiempo o secuencias. Las series se representan como dos vectores de características U = $u_1$,..., $u_i$,..., $u_I$ y V = $v_1$,..., $v_j$ ,.... $v_J$ donde i y j son los puntos que indican el tiempo en la serie. I y J representan la longitud de la serie de tiempo de U y v respectivamente. Como técnica de programación dinámica divide el problema en diversos sub-problemas, los cuales contribuyen al cálculo de la distancia acumulativa. Para más información de esta etapa acudir a [3].

La primera etapa del algoritmo de DTW es rellenar una matriz local de distancias de la siguiente forma:

$$D(i,j) = F(u_i,v_j).$$

Donde la matriz D tiene I x J elementos que representa el coste entre dos puntos en la serie de tiempo. La función de costes F puede ser cualquier tipo de función de coste que devuelve 0 si la comparación a resultado perfecta o cualquier valor positivo en caso contrario.

En la segunda etapa una matriz de deformación es rellenada de la siguiente forma:

$$C(i,j) = min\begin{cases} C(i,j-\ cj) + D(i,j), \\ C(i-c_i,j) + D(i,j), \\ C\left(i-c_i,j-\ c_j\right) + \sigma D(i,j) \end{cases}$$

Donde $c_i$ y $c_j$ son los tamaños de paso con rango de 1 hasta $\alpha_I$ y 1 hasta $\alpha_J$ respectivamente. $\alpha_I$ y $\alpha_J$ son los máximos pasos de cada dimensión. El parámetro σ controla la

parcialidad de los pasos diagonales. C(aj) es el coste mínimo del camino desde (1,1) hasta (i,j) , y C(1,1) = D(1,1).

Finalmente, en la última fase, el camino de coste mínimo w = $w_1$,..., $w_k$,..., $w_K$ es obtenido trazando el camino recursivo desde C (I,J). Cada $w_k$ es un par ordenado de ($i_k$,$j_k$) tal que (i, j) pertenece a w significa que $u_i$ y $_{vj}$ están alineados. Sin embargo, el camino debe satisfacer 3 condiciones.

1. Condición de frontera: $w_1$ = (1,1) y $w_k$ = (N, M).
2. Función monótona. $N_k$+1 >= $n_k$+1 ≥ $n_k$ para todo k ∈ [1, N − 1], y $m_k$+1 ≥ $m_k$ para todo k ∈ [1, M − 1]
3. Continua. $n_k$+1 ≤ $n_k$ + 1 para todo k ∈ [1, N − 1], y $m_k$+1 ≤ $m_k$ + 1 para todo k ∈ [1, M − 1]

Una vez realizado estos pasos nos devuelve el resultado que será el coste de comparación entre el fragmento dado y la canción con la que se ha comparado.

### 3.3.3 Resultados

Para la comprobación de la eficacia de esta aplicación se han extraído fragmentos de canciones con una duración de 5 y 20 segundos de una base de datos compuesta por 330 piezas musicales clásicas. Al realizar la búsqueda de estos fragmentos en la aplicación se ha obtenido un porcentaje de aciertos de 86% para los fragmentos de 5 segundos, llegando a alcanzar los 96,7 % de tasa de éxito para los fragmentos de 20 segundos. Como se puede apreciar es un sistema bastante eficaz que aumenta en cuanto mayor es la duración del fragmento proporcionado al disponer de más información a costa de un mayor tiempo de cómputo, el cual aumenta de forma proporcional a la duración del fragmento. En este caso los fragmentos de 20 segundos tardaban alrededor de 4 veces más de media que los fragmentos de 5 segundos.

Las pruebas han sido realizadas en un ordenador compuesto por un procesador AMD de arquitectura *Piledriver*, más concretamente el FX-8320 de 3,8 GHz acompañado de unas memorias RAM de 16 GB con una frecuencia de 1833Mhz y una latencia CAS de 11 ciclos. Al realizar las pruebas se obtiene un tiempo de ejecución de alrededor de 3 minutos con 45 segundos de media. El tiempo de ejecución es demasiado elevado para una aplicación que busca ser usada por cualquier persona para identificar canción que oigan en cualquier momento.

# Capítulo 4. Optimización del código

En este capítulo se van a presentar tres modelos de optimización de la aplicación descrita en el capítulo anterior con el fin de reducir su tiempo de ejecución.

## 4.1 Primera versión

Algoritmo 1. Primera versión

**Recibe**: Fragmento de una canción (*Can*) y la lista de canciones de la base de datos

**Devuelve**: La canción con coste mínimo

1: Transforma el fragmento

2: Crea la matriz de coste *C*

3: Crea la matriz de canciones *N*

4: Paralelización del bucle

5: **For** *i* = 1 hasta **sizeof**(*N*) **do**

6: ReMAS(*Can*, *N*[*i*])

7: Modifica la matriz *C*

8: **Fin For**

9: **For** *i* = 1 hasta **sizeof**(*N*) **do**

10: (pos) = Busca la canción con menor coste y guarda su posición

11: **Fin for**

12: **return** N[pos]

13: **Fin**

Una de las formas de paralelización más básica que se pueden realizar consiste en que cada hilo de la CPU del ordenador donde se ejecuta la aplicación realice la comparación entre el fragmento dado y una canción de la base de datos. Como se puede observar en el algoritmo 1, se ha representado un esquema del modelo que se busca describir. Primero se transforma el fragmento dado para que sea reconocido por el programa principal. Una vez realizado este paso, se crean las matrices que almacenan el coste de la comparación que tiene lugar con cada canción de la base de datos. Al mismo tiempo también se crea otra matriz que contenga el nombre de las canciones, la cual tendrá la misma posición que la matriz de costes. Una vez terminada la creación de matrices se realiza la comparación del fragmento con las canciones, que se realiza en un bucle que ha sido paralelizado en esta versión para ganar velocidad. La paralelización en cuestión consiste en que cada hilo realice una iteración y actualice la matriz de costes con el resultado obtenido al ejecutar ReMAS. Una vez finalizadas todas las comparaciones, se busca la canción que ha presentado menor coste y se devuelve el nombre de ésta.

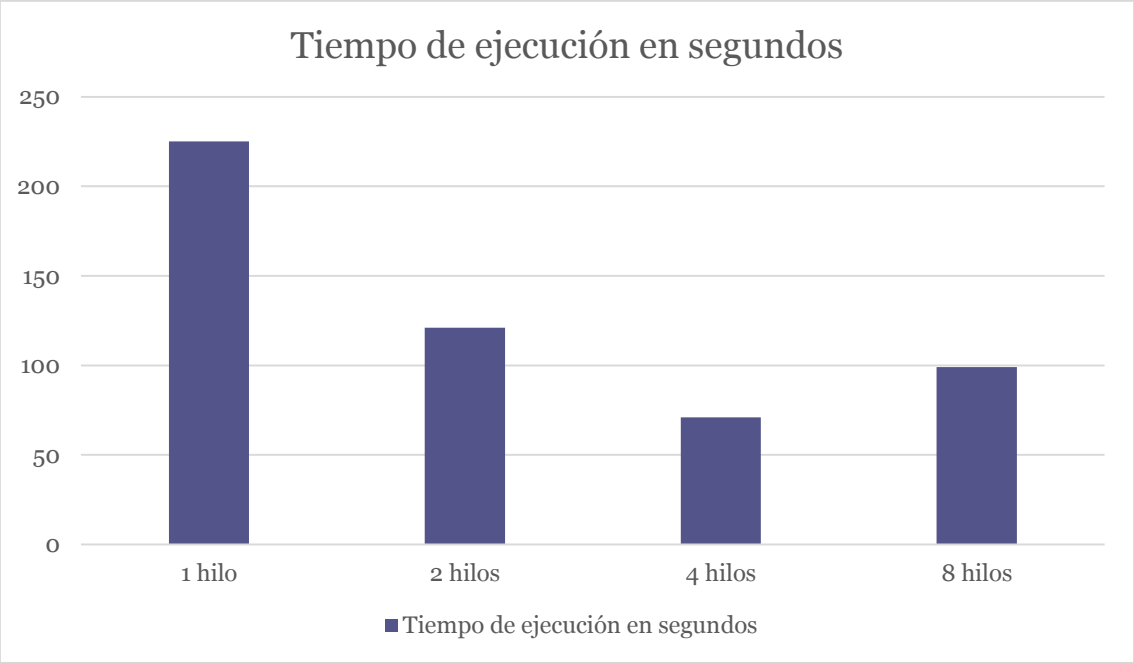


**Figura 4 Gráfica de tiempos de la versión 1**

En cuanto a la tasa de éxito de esta versión, ésta se mantiene, pues no se ha modificado la forma de comparar las canciones y se sigue realizando la comparación con toda la base de datos.

Respecto al tiempo de ejecución, como podemos apreciar en la Figura 4, se ha reducido considerablemente respecto a la versión secuencial, cuyo tiempo de ejecución rondaba casi 4 minutos. Ahora tarda, en el mejor de los casos, 71 segundos cuando utilizan 4 hilos. En la Figura 5 podemos observar la aceleración y eficiencia que presenta la aplicación dependiendo del número de hilos utilizados.

| | 2 hilos | 4 hilos | 8 hilos |
|---|---|---|---|
| Aceleración(S) | 1.85 | 3.16 | 2.27 |
| Eficiencia(E) | 92% | 79% | 28% |

**Figura 5 Resultados de la primera versión**

Como se puede apreciar, la aplicación tiene una reducción de eficiencia cuando supera los 4 hilos en ejecución. Al realizar las pruebas se observa un elevado consume de memoria RAM que llega a casi 1,5 Gb. en el caso de la ejecución de 8 hilos. Esto es debido a que las canciones de la base de datos deben cargarse en memoria provocando que, al utilizar muchos hilos, los accesos a la memoria puedan entorpecer la ejecución del programa. Aún utilizando la versión de 4 hilos que ofrece el mejor tiempo, 71 segundos sigue sin ser un tiempo adecuado para implementar una aplicación cliente-servidor, lo que nos lleva a tratar de realizar una mejora de esta versión.

## 4.2 Segunda versión

Viendo los resultados obtenidos en la primera versión, queda claro que la paralelización del bloque principal de la aplicación no presenta unos resultados muy buenos, aunque se consiga cierta eficiencia con pocos hilos, no se aprecia un buen escalado que permita reducir suficientemente el tiempo de ejecución al aumentar el número de hilos. Para mejorar las prestaciones, en esta versión se ha analizado el código de la aplicación en busca de las funciones con más coste computacional y se ha estudiado la optimización de estas.

Algoritmo 2. Segunda versión

**Recibe**: Fragmento de una canción (*Can*) y la lista de canciones de la base de datos
**Devuelve**: La canción con coste mínimo
1: Transforma el fragmento
2: Crea la variable coste *C*
3: Crea la matriz de canciones *N*
4: **For** *i* = 1 hasta **sizeof**(*N*) **do**
5: ReMAS(*Can*, *N*[*i*])
6: Paralelización bloque Distorsión
7: Paralelización bloque DTW
8: **if** costeActual > C **then**
9: **continue**
10: **Fin if**
11: *C* = costeActual
12: X = N[i]
13: **Fin For**
14: **return** X
15: **Fin**

Tras analizar el código se concluye que los bloques de *distorsión* y *DTW* son los más costosos con diferencia respecto al resto de funciones de la aplicación. Como podemos ver en el algoritmo 2, tenemos una versión de la implementación. En ella se ha cambiado la matriz de coste por una variable ya que las canciones se comparan de una en una al no paralelizar el bloque principal. Los bloques de distorsión y DTW presentan bucles de gran tamaño, los cuales se paralelizan teniendo en cuenta que las variables utilizadas no sean comprometidas por los hilos en cada iteración. Además, se ha añadido una condición de salida del bucle principal de ReMAS que funciona de la siguiente manera. Por cada *frame* del fragmento de audio que introducimos se lleva a cabo una ejecución del bloque de distorsión y el de DTW. En este último también se actualiza la matriz de costes para cada *frame* de la canción con la que se está comparando, una matriz que va en aumento según más se compara el fragmento introducido. Si en algún momento en esta matriz no se almacena un valor menor al coste de la canción menor que se haya encontrado, se considera que la canción que se está comparando solo puede tener un peor coste que el ya encontrado deteniéndose su comparación para pasar a la siguiente canción. Si

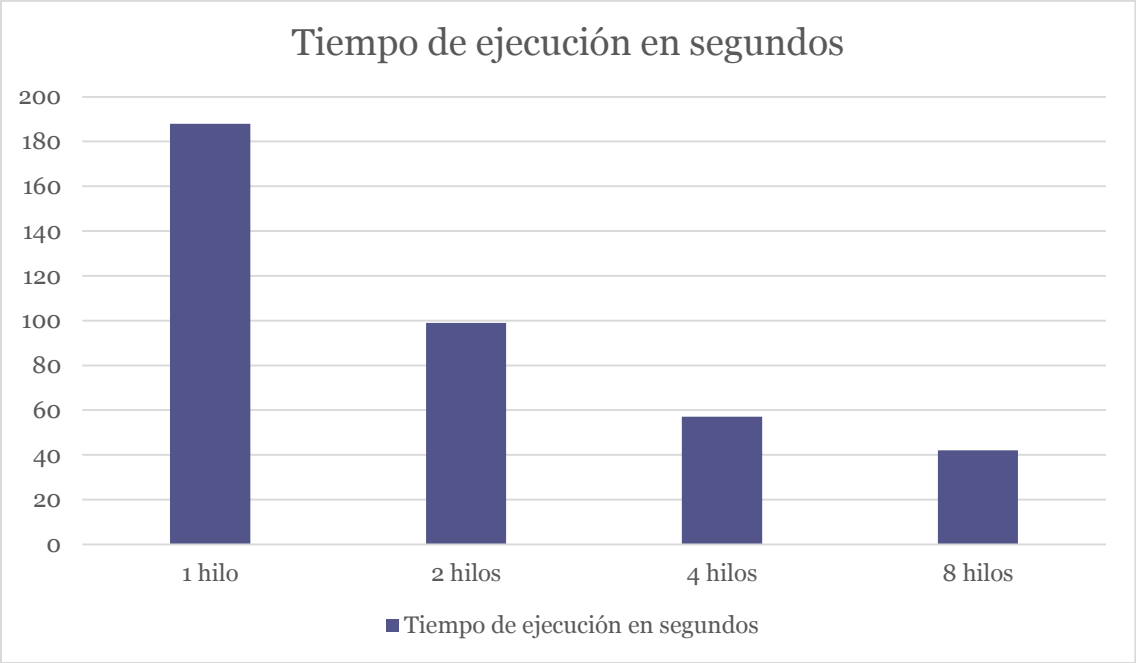


**Figura 6 Gráfica de tiempo de la segunda versión**

no sucede esta condición, se considera que la canción tiene menor coste que las ya comparadas y se actualiza el coste mínimo y el nombre de la canción. Una vez finalizadas todas las comparaciones se devuelve el nombre de la canción con menor coste encontrado.

En cuanto a la tasa de éxito, ésta se mantiene igual que en las otras versiones, pues se sigue comparando con toda la base de datos, con la única diferencia de que, ahora, las canciones que superen el coste mínimo se descartan sin llegar a analizarlas completamente. En cuanto a las mejoras en el tiempo de ejecución, como podemos apreciar en la Figura 6, se obtienen unos tiempos mejores que en la versión anterior. Con solo la condición de salida obtenemos una mejora del 15%, llegando a necesitar solo 40 segundos para 8 hilos que, en comparación a los 3 minutos y 40 segundos iniciales, supone una gran mejora. Sin embargo, este valor sigue resultando insuficiente para implementarlo en un modelo cliente-servidor.

| | 2 hilos | 4 hilos | 8 hilos |
|---|---|---|---|
| Aceleración(S) | 1.89 | 3.29 | 4.47 |
| Eficiencia(E) | 94% | 82% | 55% |

**Figura 7 Resultados de la segunda versión**

En la Figura 7 tenemos la aceleración y la eficacia que se obtiene al paralelizar esta versión. Como puede observarse la aceleración va aumentando con el uso de más hilos, lo que indica que esta parte de la aplicación es muy adecuada a ser paralelizada. En cuanto a la eficiencia, ésta va disminuyendo a un nivel aceptable, al contrario que en la versión anterior dado que esta no presenta problemas con los accesos a memoria.

## 4.3 Tercera versión

Algoritmo 3. Tercera versión

**Recibe**: Fragmento de una canción (*Can*) y la lista de canciones de la base de datos
**Devuelve**: La canción con coste mínimo
1: Transforma el fragmento
2: Creación la variable matriz *C*
3: Creación de la matriz X con valores para acortar el fragmento
4: Creación la matriz de canciones *N*
5: **For** *i* = 1 hasta **sizeof**(*X*) **do**
6: **For** j hasta **sizeof**(N) **do**
7: ReMAS(*Can*, *N*[*i*])
8: Paralelización bloque Distorsión
9: Paralelización bloque DTW
10: Actualiza la matriz *C*
11: **Fin for**
12: (N,media) = media de coste y actualiza la matriz N con los de menor de coste
13: **Fin for**
14: **For** *i* = 1 hasta **sizeof**(*N*) **do**
15: ReMAS(*Can*, *N*[*i*])
16: Paralelización bloque Distorsión
17: Paralelización bloque DTW
18: **if** costeActual > C **then**
19: **continue**
20: **fin if**
21: *C* = costeActual
22: X = N[i]
23: **fin For**
24: **return** X
15: **Fin**

Como se observa en la versión 2, el código aun precisa de una mayor optimización pues sigue presentando elevados tiempos de ejecución. Para aumentar las prestaciones, no solo se necesita paralelizar las funciones más costosas dentro del programa, sino que

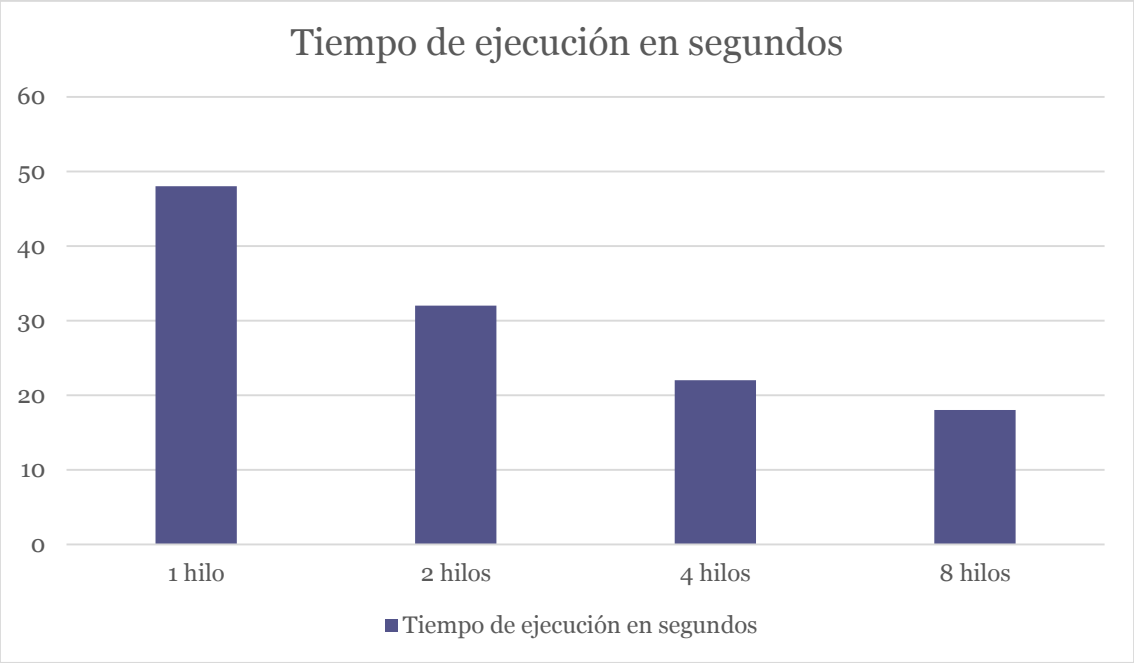


**Figura 8 Gráfica de tiempo de la versión 3**

además, es necesario modificar o ampliar parte del código con el fin de que no realice tantas comparaciones.

Como podemos ver en el algoritmo 3, la modificación planteada consiste en ir analizando pequeños trozos del fragmento de audio introducido con el fin de ir descartando las canciones que presente a priori un resultado negativo. Esto se realiza mediante una ampliación del código, el cual funciona de la siguiente manera. Primero se vuelve a crear una matriz de costes que guardará los costes de las canciones para analizar cuáles deberán seguir siendo comparadas. Una vez creada esta matriz, se ejecutará una versión de ReMAS con un pequeño trozo del fragmento de audio para reducir el tiempo de ejecución. Esta pequeña comparación permite obtener unos costes orientativos de las comparaciones. Una vez acabadas estas comparaciones, se obtiene la media de costes de todas las canciones que se han comparado y se eliminan las canciones que hayan superado la media de coste. Este proceso se repite varias veces quedando al final una pequeña porción de la base de datos que deberá realizar la comparación final. En la base de datos de ejemplo, de las 330 canciones que deberían compararse al completo, quedan al final una media de 10 canciones para comparar. Una vez finalizada la reducción de la base de datos, simplemente quedaría aplicar a las canciones restantes la comparación completa, que en este caso se utiliza la misma modificación de la aplicación que la explicada en la versión 2.

Como se puede apreciar en la Figura 8, se ha reducido considerablemente el tiempo respecto a la versión anterior, donde con solo 1 hilo obtenemos el mismo tiempo que con 8 hilos en la anterior, llegando incluso a solo 18 segundos para el uso de los 8 hilos disponibles. Estos tiempos, más concretamente el de 4 y 8 hilos ya son tiempos aceptables para el modelo cliente-servidor que buscamos implementar. Respecto a la tasa de éxito

| | 2 hilos | 4 hilos | 8 hilos |
|---|---|---|---|
| Aceleración(S) | 1.5 | 2.18 | 2.66 |
| Eficiencia(E) | 75% | 54% | 33% |

**Figura 9 Resultados de la tercera versión**

de esta versión, al no comparar las canciones de forma completa e ir descartando estas, se da lugar una reducción en la probabilidad de acierto. Utilizando los mismos fragmentos de 20 segundos que se han utilizado en la base de datos de ejemplo obtenemos una tasa de aciertos del 89%, menor que la original de 96,8% aunque aceptable debido a la gran mejora de ejecución que se ha obtenido.

Como se puede observar en la Figura 9, esta versión presenta unos resultados paralelos peor que las otras dos versiones debido a la ampliación de código que se ha tenido que realizar para aumentar la eficiencia del programa, es decir, esta versión aunque presente una aceleración creciente con el uso de hilos indicando que no presenta problemas al paralelizar, la eficiencia de estos hilos es peor que las otras dos versiones indicando que no va a aprovechar tan bien la potencia de la maquina donde se ejecute.

## 4.4 Conclusión

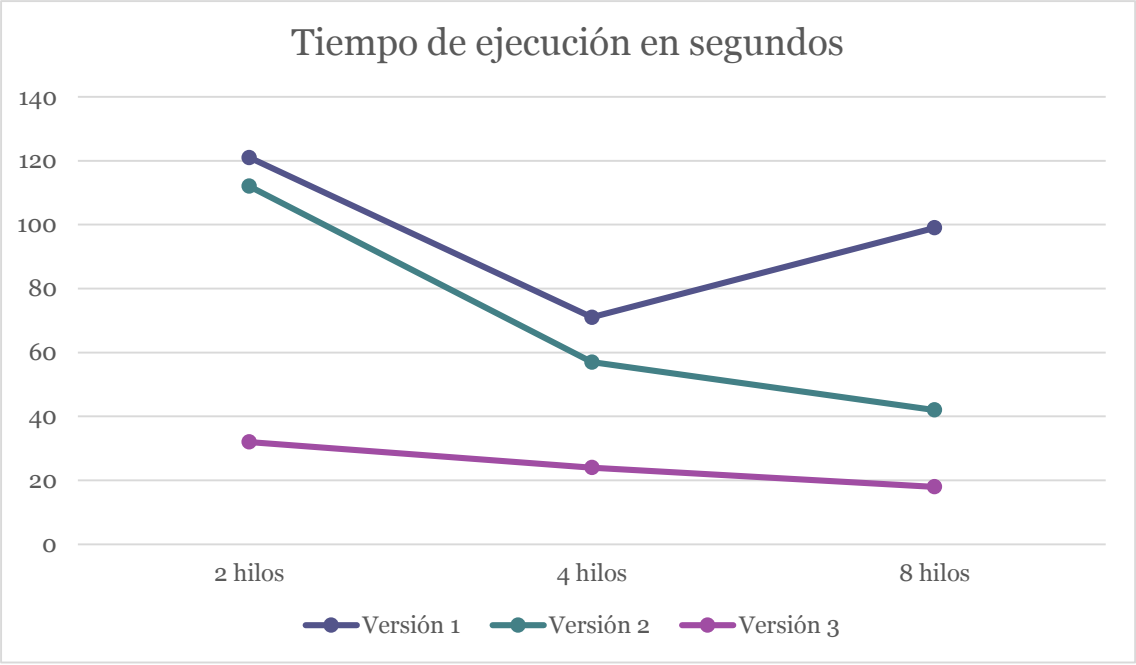


**Figura 10 Resultados de las tres versiones.**

En la Figura 10 podemos observar las comparaciones en tiempo de ejecución entre las tres versiones de optimización planteadas. En la versión 1 destaca el incremento de tiempo que sufre al utilizar más de 4 hilos debido a los problemas que presenta en los accesos a memoria. Respecto a la versión 2, se ha cambiado el enfoque de paralelización evitando los problemas que planteaba la primera versión, ofreciendo la mayor eficiencia de las tres versiones con diferencia, aunque sigue teniendo un tiempo de ejecución demasiado elevado. Finalmente, con la versión tres se decide, no solo optimizar paralelizando, sino además modificar el código de la aplicación en busca de reducir el número de comparaciones que se realizan. Consiguiendo este objetivo, se reducen de forma notoria los tiempos de ejecución, llegando a necesitar solo 18 segundos cuando se emplean 8 hilos. Aunque es una gran mejora, la modificación del código también trae algunos inconvenientes, tales como la reducción de la eficiencia respecto a las otras dos versiones,

pues la modificación implica que la aplicación debe realizar más procesos de forma secuencial.

Es esta última versión, la versión 3, la que va a ser utilizada en la implementación cliente-servidor al presentar los mejores tiempos de ejecución manteniendo una elevada tasa de aciertos respecto a la ejecución original.

# Capítulo 5. Implementación

Una vez optimizado el código de la aplicación Shizmidi para que cumpla con una respuesta aceptable en cuanto a tiempo, se realiza la implementación de la aplicación. Esta implementación consistirá en una interfaz que reciba un audio y responda con el nombre de la canción que presente el menor coste en la base de datos.

La arquitectura consiste en:

- Front-end o capa de presentación: Es la interfaz con la que el usuario va a interactuar para poder mandar audios y recibir las respuestas.
- Back-end o capa de acceso a datos: Es el motor principal de la aplicación donde se va a procesar la información recibida, ejecutar Shizmidi y gestionar las respuestas.

## 5.1 Front-end

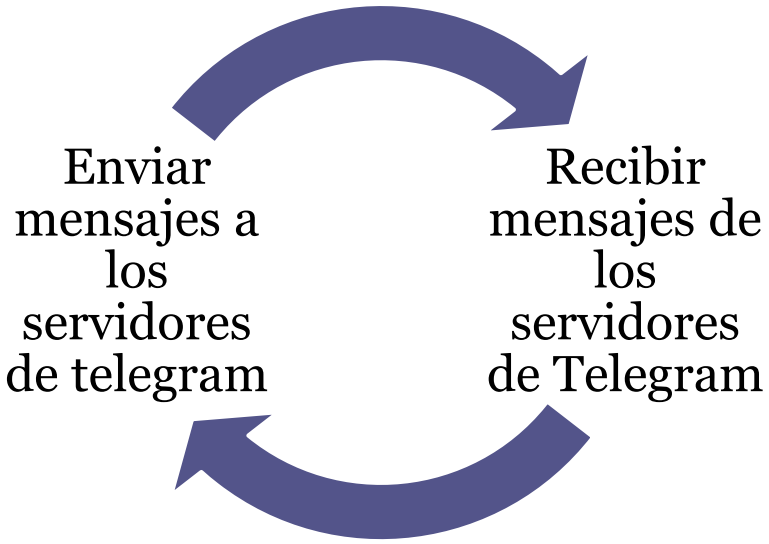


**Figura 11 Esquema del funcionamiento de un bot de Telegram**

En esta parte se va a emplear un bot de Telegram de forma de interfaz. El bot, como se puede ver en la Figura 11, tiene una funcionalidad muy básica. Puede recibir los mensajes que el servidor de Telegram le envía, entre estos mensajes se encuentran las respuestas a los usuarios. La otra funcionalidad es enviar a los servidores de Telegram los mensajes que ha recibido de los usuarios. Estos mensajes son procesados en los servidores y almacenados durante un periodo de un día de tiempo a la espera de que sean solicitados. En caso de que no sean solicitados y el tiempo expire, Telegram borrará estos mensajes de sus servidores. Como se puede apreciar, la verdadera funcionalidad de un bot está diseñada en la parte del servidor, donde los mensajes son utilizados como se desee.

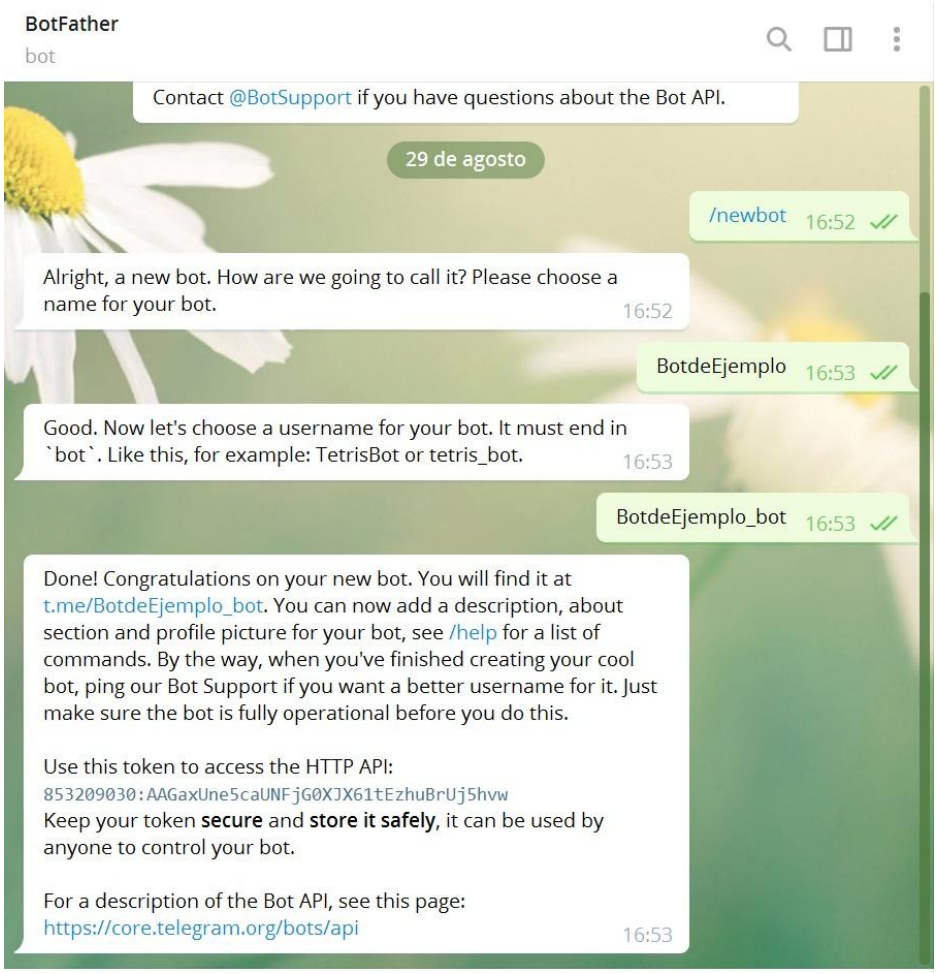


**Figura 12 Creación de un bot**

La creación de un bot de Telegram es relativamente fácil, ya que en la propia aplicación contamos con todo lo necesario para hacerlo, tal como se puede apreciar en la Figura 12, donde se ha creado un bot de ejemplo. Los pasos a seguir son:

1. Se establece una conversación con el bot BotFather, que es un bot de Telegram cuya función es ayudar a los usuarios a crear y gestionar sus bots.
2. Se envía el comando */newbot* para indicar que queremos crear un bot.
3. Se introduce el nombre del bot y el alias con el que las personas pueden encontrar al bot.

Con estos tres sencillos pasos ya tendríamos creado un bot, aunque no realiza nada todavía. Para dotar de funcionalidad al bot hace falta crear alguna aplicación que controle el funcionamiento del bot, gestionando los mensajes que recibe y las respuestas que envía. Para ello, Telegram nos proporciona el *token* del bot con el cuál podemos realizar peticiones a su API para gestionar los mensajes que recibe el bot, los que se quieren enviar o para modificar algunos parámetros del bot. Este *token* será utilizado por la capa de acceso a datos.

Para más información acerca de los bots consultar [8].

## 5.2 Back-end

En esta parte se implementa un programa desarrollado en PHP para controlar las acciones del bot de Telegram creado y la aplicación ReMAS de Shizmidi. Este programa funciona de la siguiente manera.

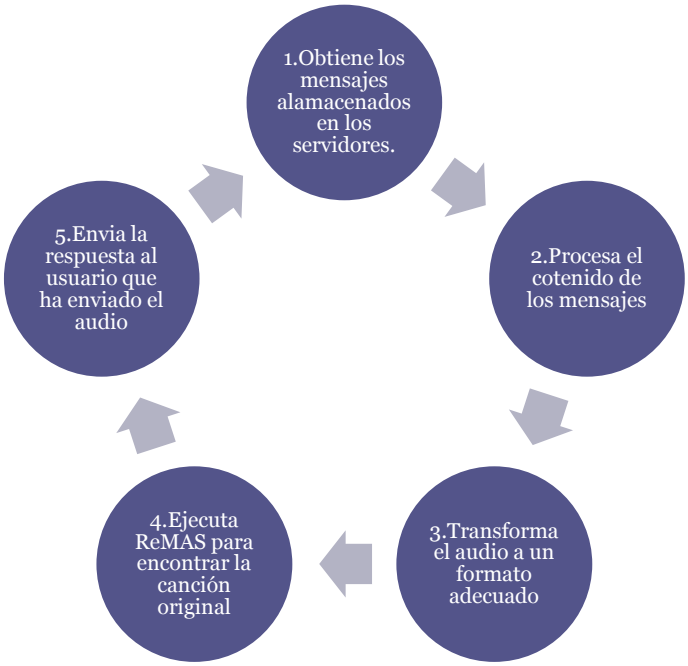


**Figura 13 Esquema del funcionamiento del servidor.**

Como se puede ver en la Figura 13, se tiene un esquema básico del funcionamiento de la aplicación. Lo primero que realiza la aplicación es obtener los mensajes almacenados en los servidores de Telegram, Figura 13-1. Para obtener los mensajes hace falta realizar un *request* a la API de Telegram. Esto se realiza mediante la función del código 1, donde se envía un mensaje con el *token* del bot, en este caso el creado en el ejemplo y la opción */getUpdates* para que la API nos envíe los mensajes almacenados. La respuesta es un string json que debe ser decodificado para que pueda ser fácilmente manipulable por el programa. Para más información acerca de los *request* de Telegram [9].

```
$a = file_get_contents('https://api.telegram.org/bot853209030:AAGaxUne5caUNFjG0XJX61tEzhuBrUj5hvw/getUpdates');
$b = json_decode($a);
```

**Código 1. Petición de mensajes.**


**Figura 14 Esquema de los mensajes de Telegram**

Una vez obtenidos los arrays decodificados de los mensajes almacenados podemos empezar a procesar la información que estos contienen, Figura 13-2. La estructura de los mensajes de Telegram tiene una forma como la que presenta la Figura 14. En este caso, la figura representa dos mensajes, el primero un audio *mp3* y el segundo un audio grabado con el micrófono. Como se observa cuando el archivo enviado es un audio se crea el subarray audio o *voice* cuando es grabado con micrófono, este tipo de subarray será los que se buscarán al procesar los mensajes en busca del audio que se quiere comparar. En caso de no presentarlos se le envía un mensaje al usuario indicando que no ha introducido un audio. Además, en la estructura también se puede obtener la id del usuario o la id del chat en caso de ser un grupo, que tienen id negativa, para poder enviarles mensajes con las respuestas.

```
function audioMP3($me,$audio,$tokenBot) {
    $audioObjaux = "https://api.telegram.org/". $tokenBot ."/getFile?file_id=". $me;
    $audioObj = file_get_contents($audioObjaux);
    $audioObj = json_decode($audioObj);
    $audioPath = $audioObj-> {'result'}-> {'file_path'};
    $audioFileUrl = "https://api.telegram.org/file/".$tokenBot."/". $audioPath;
    $audioFile = file_get_contents($audioFileUrl);
    file_put_contents("MusicBot/". $audio."", $audioFile);
    shell_exec("mpg123 -w  audio.wav -m audio.mp3");
    unlink("MusicBot/audio.mp3");
}
```

**Código 2. Función audioMP3.**

Una vez se sabe que los mensajes contienen audio hay que procesarlos para obtener el fichero de los servidores de Telegram y transformarlo para que cumpla con las características de ReMAS, Figura 13-3. En la función audio MP3, donde $*me* es la id del audio, $*audio* el nombre de este y $*tokenBot* el *token* del bot, es donde se obtiene el audio del

servidor y se transforma de la siguiente manera. Primero se realiza un *request* a la API con el fin de obtener la información del fichero con id en $me. Cuando se realiza este *request* la API devuelve la información del fichero y prepara este para que pueda ser descargado. En la variable $*audioPath* es donde se guarda la dirección del fichero en los servidores de Telegram, esta variable solo dura 1 hora. A través de otro *request* descargamos el fichero finalmente y lo guardamos en la máquina. A continuación se transforma el audio mp3 a WAV mediante el reproductor mpg123 a través de líneas de comando mediante un Shell. Finalizada la transformación, se borra el fichero mp3 original. En el caso de que sea un archivo de *voice* en vez de audio, se realiza el mismo procedimiento pero con una transformación más. Es decir, de ogg a MP3 y luego a WAV.

Obtenido el audio transformado al formato adecuado se llama a la aplicación ReMAS mediante una línea de comando donde los parámetros necesarios son el audio obtenido y la base de datos, Figura 13-4. Una vez finalice la comparación devolverá el nombre de la canción que ha presentado menor coste en la comparación. Este nombre tiene que ser enviado al usuario que ha enviado el fragmento de audio, Figura 13-5.

```
function sendMessage($chatID, $messaggio, $token) {
    $url = "https://api.telegram.org/" . $token . "/sendMessage?chat_id=" . $chatID;
    $url = $url . "&text=" . urlencode($messaggio);
    $ch = curl_init();
    $optArray = array( CURLOPT_URL => $url, CURLOPT_RETURNTRANSFER => true);
    curl_setopt_array($ch,$optArray);
    $result = curl_exec($ch);
    curl_close($ch);
}
```

**Código 3. Función para enviar mensajes.**

La función *sendMessage* realiza un request a la API de Telegram para enviar un mensaje con el nombre de la canción al chatid del mensaje que contiene el audio. Para realizar esta función se utiliza la biblioteca url cliente [10]. Cuando se envía el mensaje, Telegram devolverá el mismo mensaje en caso de que se haya podido realizar el envío correctamente.

Una vez finalizado el envió, se borran el resto de los archivos que queden y se vuelve a repetir el mismo proceso para cada mensaje que aún quede por procesar. Finalizado todos los mensajes hay que volver a realizar el *request* a la API para obtener los mensajes nuevos.

Telegram no borra los mensajes hasta dentro de 1 día, para borrar los mensajes que ya hemos procesado del servidor de Telegram hay que realizar el mismo proceso que en el código 1 pero con una modificación. Al final del *getUpdate* hay que añadir *getUpdate?offset=$id+1* donde $id+1 es la última id del mensaje recibido más uno. Con esto la API borrará los mensajes que nos envía. Este proceso debe repetirse cada poco tiempo para ir atendiendo los mensajes que vaya recibiendo.

## 5.3 Pruebas

A continuación se han realizado unas pruebas para comprobar el funcionamiento correcto de la implementación.

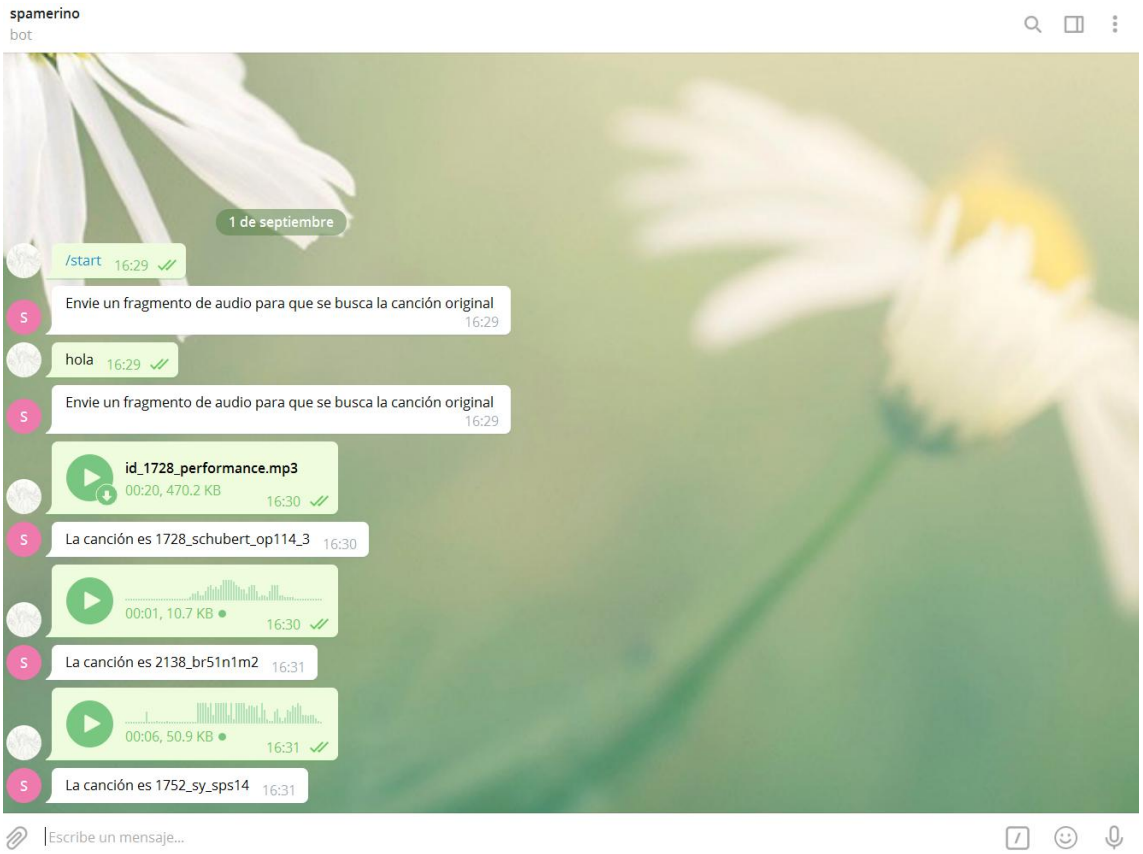


**Figura 15 Resultados de la implementación**

En la Figura 15 se muestra varias ejecuciones del bot. Al iniciar éste nos indica que se introduzca un fragmento de audio para iniciar el proceso. Si le enviamos un texto responderá que enviemos un audio. Si le enviamos un fragmento de audio nos responderá con la canción en la base de datos con menor coste. En este caso la ejecución del proceso tarda alrededor de 20 segundos en completarse, desde que se envía el mensaje hasta que el bot responde.

# Capítulo 6. Conclusión

En la realización de este trabajo de fin de grado se ha conseguido llevar a cabo los objetivos planteados. El objetivo primordial, que es el análisis y optimización de una aplicación que realiza la búsqueda de una canción en una base de datos, se ha realizado de forma satisfactoria a través del planteamiento de tres posibles optimizaciones de la aplicación y con el análisis de aceleración y eficiencia se ha elegido la más óptima, consiguiendo al final una optimización deseada que cumple con unos tiempos muy favorables para la implementación cliente-servidor. Esta parte ha sido la más complicada del proyecto pues se ha tenido que analizar el funcionamiento interno de una aplicación para averiguar sus partes más críticas en cuanto a la optimización.

Se ha realizado la implementación cliente mediante un bot de Telegram que funciona como interfaz entre el usuario y el servidor, una interfaz bastante simple pero que cumple con su función. En cuanto a la parte del servidor, la implementación en PHP realiza la función deseada de recibir y tratar la información. Como se ha comprobado en el Capítulo 6, la implementación cumple con los objetivos descritos en un tiempo bastante razonable. Esta última parte ha presentado algunos problemas pues es la primera vez que utilizo PHP, además, la comunicación entre PHP y C han sido un tanto tediosa de manejar.

En este proyecto se han podido aplicar conocimientos adquiridos durante la carrera como los aprendidos en las asignaturas de computación paralela o lenguajes y entornos de programación paralela, los cuales se han aplicado a la primera parte del proyecto.

Respecto a la segunda parte se han utilizado tecnologías que no se han dado a lo largo de la carrera como es PHP.

# Capítulo 7. Trabajos futuros.

Como se ha podido observar en la Capítulo 4, una de las optimizaciones planteadas presenta problemas de escalabilidad en la eficiencia y esto es debido a que la aplicación realiza un gran número de acceso a memoria, recordemos que las canciones se cargan en memoria antes de ser comprobadas y pueden llegar a ocupar bastantes megas. Una reducción de estos accesos podría aumentar las prestaciones que la aplicación presenta de forma notoria además de poder plantear  nuevas formas de optimización que podrían ser mejores de las ya planteadas.

En cuanto a la parte de la interfaz, aunque Telegram es una aplicación gratuita y de fácil acceso ya sea en el móvil o el PC, tener una interfaz diseñada exclusivamente para este propósito permite tener más control sobre su funcionamiento. Además de poder monetizar si se quisiera la aplicación, se podría ampliar para que realizara más funciones como, por ejemplo, la otra funcionalidad de ReMAS que permite el acompañamiento de piezas musicales.

# Bibliografía


[1] Avery Li-Chun Wang, An Industrial-Strength Audio Search Algorithm.

[2] Alonso P, Vera-Candeas P, Cortina R, Ranilla J (2017) An efficient musical accompaniment parallel system for mobile devices. J Supercomput 73(1):343–353

[3] Carabias-Orti J, Rodríguez-Serrano F, Vera-Candeas P, Ruiz-Reyes N, Cañadas-Quesada F (2015) An audio to score alignment framework using spectral factorization and dynamic time warping. En: Proceedings of ISMIR, pp 742–748

[4] Muñoz-Montoro A, Cabañas-Molero P, Bris-Peñalver F, Combarro E, Cortina R, Alonso P (2017) Discovering the composition of audio files by audio-to-midi alignment. En: Proceedings of the 17th International Conference on Computational and Mathematical Methods in Science and Engineering. pp 1522–1529

[5] Alonso P, Cortina R, Rodríguez-Serrano FJ, Vera-Candeas P, Alonso-González M, Ranilla J (2017) Parallel online time warping for real-time audio-to-score alignment in multi-core systems. J Supercomput 73(1):126–138

[6] P. Cabañas-Molero, Raquel Cortina-Parajón, E. F. Combarro, Pedro Alonso, F. J. Bris-Peñalver (2018) HReMAS: hybrid real-time musical alignment system.

[7] MTProto Moble Protocol : https://core.telegram.org/mtproto

[8] Información sobre los bots de Telegram: https://core.telegram.org/bots

[9] Funciones del bot de Telegram : https://core.telegram.org/bots/api

[10] Biblioteca url cliente de php: http://docs.php.net/manual/es/book.curl.php