POLITECNICO DI MILANO
Facoltà di Ingegneria dell'Informazione
Master of Science in Computer Engineering

MUSIC RECOMMENDATION SYSTEM BASED ON AUDIO SEGMENTATION AND FEATURE EVOLUTION

Polo Regionale di Como

Supervisor: Prof. Augusto Sarti
Co-Supervisor: Dr. Massimiliano Zanoni

Master Graduation thesis by:
Daniel Alberto Rodríguez Salgado, ID 752169

Academic Year 2010-2011

POLITECNICO DI MILANO
Facoltà di Ingegneria dell'Informazione
Corso di Laurea Specialistica in Ingegneria Informatica

MUSIC RECOMMENDATION SYSTEM BASED ON AUDIO SEGMENTATION AND FEATURE EVOLUTION

Polo Regionale di Como

Relatore: Prof. Augusto Sarti
Correlatore: Dr. Massimiliano Zanoni

Tesi di Laurea di:
Daniel Alberto Rodríguez Salgado, matricola 752169

Anno Accademico 2010-2011

Aut viam inveniam aut faciam
Hannibal

Abstract

The advent of new technologies, like Internet, digital audio formats, and portable media players, make it easier to produce and distribute music, exponentially increasing the offer, but making it harder to find songs that suits users' tastes. Therefore an important topic in research today is the development of music browsing, searching and organizing techniques and tools.

Music recommendation systems are one of the solutions to this problem. They focus on generating playlists according to the similarities between a chosen track and a predefined music collection. Similarities are generally based on the physical, perceptive, and acoustical properties of the audio signal (content-based approach), or on manually defined tags (context-based approach). Content information is obtained using Multimedia Information Retrieval techniques that extract descriptors out of a song, like rhythm, harmony, or loudness. Songs can then be compared using algorithms specialized on finding similarities between the extracted features, and matching items are proposed to the user as a playlist.

The purpose of this thesis is to extend an existing content-based music recommendation system, producing an application that generates playlists according to the acoustic features of the audio being played. This application works in both a local and a web environment, using a client-server infrastructure where the recommendation engine is not tied to the player. The core components of the application are exchangeable plugins. Similarity is measured according to two different approaches: local and global. In the local approach a song is segmented into a sequence of "cells" that represent highly homogeneous parts and similarity is evaluated over its descriptors. In the global approach, the similarity is performed over the whole song using the evolution on time of the extracted features. In both approaches the users are able to interact with the system by defining the desired feature values in order to improve the output.

Sommario

Con l'avvento di Internet, dei formati audio digitali e dei riproduttori musicali portatili, la produzione e distribuzione di contenuti musicali ha subito un'accelerazione notevole. L'abbondanza dell'offerta ha avuto come conseguenza un necessario ripensamento del nostro modo di reperire la musica che ci interessa. Così negli ultimi anni lo sviluppo di tecniche e di strumenti per la ricerca e l'organizzare di contenuti musicali sta acquistando una notevole importanza tecnica.

I sistemi di "music recommendation" sono una delle soluzioni a questo problema. Essi si occupano della generazione di "playlist" secondo criteri di similarità tra un brano e una collezione di contenuti musicali. Questa similarità si basa sulle proprietà fisiche, percettive e acustiche del segnale audio (approccio "content-based") e su i tag definiti manualmente (approccio "context-based"). Le informazioni sul contenuto si ottengono tramite tecniche di "Multimedia Information Retrieval" che estraggono i descrittori da un brano, come il tempo, l'armonia, oppure la rumorosità. I brani vengono comparati attraverso funzioni di similarità tra le feature estratte. La playlist proposta all'utente viene popolata sulla base di tali funzioni.

Lo scopo di questa tesi è l'estensione di un sistema di music recommendation content-based esistente. L'applicazione è in grado di generare playlist secondo le proprietà acustiche dell'audio riprodotto. Essa è in grado di funzionare sia in modalità locale sia in modalità web, secondo un'infrastruttura di tipo "client/server" indipendente dalla piattaforma. L'applicazione è stata dotata di funzionalità implementate attraverso plugin intercambiabili. La similarità viene calcolata in due modi: locale e globale. Nel modo locale il brano è segmentato in "celle" omogenee dalle quali vengono estratte le feature da comparare. Nel modo globale la similarità dipende dell'evoluzione nel tempo delle feature del brano completo. In entrambi i casi, l'utente può interagire impostando i valori dei feature per migliorare l'output.

Contents

Abstract **I**

Sommario **III**

1 Introduction **1**

2 State of the art **7**

2.1 Music Recommendation/Discovery Systems 8
2.1.1 Pandora 9
2.1.2 Mufin 10
2.1.3 Musicovery 11
2.1.4 The Echo Nest 12
2.1.5 SoundBite 13
2.2 Music Identification Systems 14
2.2.1 Musipedia 14
2.2.2 Midomi & SoundHound 14
2.2.3 Tunebot 15

3 Theoretical Background **17**

3.1 Feature Extraction 17
3.1.1 Harmony 20
3.1.2 Tempo 21
3.1.3 Brightness 21
3.1.4 RMS 22
3.1.5 Mood 22
3.2 Feature Based Recommendation 23
3.2.1 Similarity Functions 23
3.2.2 Cross-correlation 28
3.3 Ranking System 28
3.3.1 Gaussian Mixture Model (GMM) 30

4 System Design 33

4.1 The existing application . . . 33
4.2 Requested Features . . . 36
4.3 Architecture . . . 37
4.3.1 Streaming . . . 37
4.3.2 Server/Client Architecture . . . 38
4.4 Software . . . 39
4.4.1 Java . . . 39
4.4.2 NetBeans . . . 40
4.4.3 Apache Derby . . . 41
4.4.4 FFmpeg . . . 42
4.4.5 Java Simple Plugin Framework . . . 43
4.4.6 Jetty . . . 43
4.4.7 JOrbis . . . 44
4.4.8 Database Schema . . . 44
4.4.9 Track . . . 45
4.4.10 Anchor Type . . . 46
4.4.11 Anchor . . . 46
4.4.12 Feature Type . . . 46
4.4.13 Attribute Type . . . 46
4.4.14 Attribute . . . 46
4.4.15 Normalized Attribute . . . 47
4.4.16 Cross-correlation . . . 47
4.4.17 GMM . . . 47
4.4.18 GMM Point . . . 47
4.4.19 Cluster . . . 48

5 Implementation 49

5.1 User interaction . . . 49
5.2 System startup . . . 50
5.2.1 Server startup . . . 50
5.2.2 Client startup . . . 52
5.3 Main operation . . . 52
5.3.1 Searching for a song . . . 53
5.3.2 Choosing an audio segment . . . 55
5.3.3 Temporal files . . . 55
5.3.4 Recommendation generation . . . 55
5.3.5 Streaming . . . 56
5.3.6 Reproduction . . . 57
5.3.7 Autoselection . . . 59

5.4 NetBeans 59

6 Conclusions and future developments **63**

6.1 Conclusions 63

6.2 Future Developments 64

Bibliografia **66**

List of Figures

1.1 Amazon's rendition of a cloud based service 2
1.2 Workflow of a traditional recommendation system 4
1.3 Workflow of the system to be extended 5

2.1 Pandora's Graphical Interface 9
2.2 Mufin's self-denominated "interactive 3D music universe" . . 10
2.3 Musicovery's Flash interface in Italian 11
2.4 SoundBite's integration with iTunes 13
2.5 Musipedia's main menu 14
2.6 Midomi's "Click and Sing" interface 15
2.7 Tunebot's interface 15

3.1 Structure of a generic classification system 18
3.2 Workflow of a novelty based segmentation system 19
3.3 Unwrapped Chromagram 21
3.4 Key clarity 21
3.5 Tempo: onset detection 22
3.6 Brightness 22
3.7 Tempo similarity graph 24
3.8 Harmony similarity measure 25
3.9 The qualitative graph of harmony similarity 26
3.10 Harmony similarity measure 26
3.11 Graph of $compare_{harmony}$ 27
3.12 Mood bi dimensional plane 27
3.13 User's BPM selection 29
3.14 Normalized cross-correlation of two feature sets 29

4.1 A feature's xml representation 34
4.2 PolySound's Graphical Interface 35
4.3 Three Tier Architecture 39
4.4 Java Logo 39

4.5 NetBeans Logo 40
4.6 Apache Derby Logo 41
4.7 FFmpeg Logo 42
4.8 JSPF Logo 43
4.9 Jetty Logo 43
4.10 Database Schema 45

5.1 Steps of the server's startup 51
5.2 Steps of the client's startup 53
5.3 The system's workflow 54
5.4 The client's streaming workflow 58
5.5 The streamed segment split in three pieces 58
5.6 An early implementation of the GUI 60

List of Tables

3.1 C Major (left) and C Minor (right) similarity measures 24
3.2 The qualitative measure of harmony similarity 25
3.3 Mood similarity table 27

Chapter 1

Introduction

Music recordings have always been the target of many different technological evolutions. First there were analog storage mechanisms like vinyl records and cassette tapes. Then computers changed much of our lifestyle and music started to be stored on digital format using devices like CDs and MiniDiscs (MDs). Compression algorithms and codification schemes made it possible to store audio tracks in small portable files. The internet brought the possibility to share these files quickly and easily, and portable media players allowed users to take their music anywhere. Companies saw the opportunity to create a market where users can buy single songs instead of full albums and download them on different devices. A new concept of radio on the net gave birth to streaming technologies that allow users to start listening to music as soon as the file download begins, with an offer that includes different genres and audio qualities. Finally, the technological world is now shifting to the "Cloud": the separation between what's stored locally and what's stored remotely is disappearing thanks to the speed of internet and the processing power of new mobile devices. Now files can be stored in many different places, but retrieved at any time and any place. There isn't a unified definition of what the Cloud is[18][5], but a simplified description tells us that the Cloud is just a user friendly collection of systems that provide storage and uninterrupted services in a network (music in our case). What users gain by using the Cloud is the possibility if accessing all their files in different locations, without the hassle of having to synchronize everything manually device by device. As an example, figure 1.1 shows the way Amazon implemented the Cloud concept tied to it's commercial business.

The advent of these new technologies makes the production and dis-

Figure 1.1: Amazon's rendition of a cloud based service

tribution of music and audio contents much easier. As a consequence the amount of information made available to the average user is increasing over time, and as the offer grows, it becomes harder to find music that suits one particular taste. Helping the user find the music that he wants or may enjoy is now more important than before. There are different types of solution that can be implemented to improve the user's listening experience. The first one is called a "music identification system" and it's objective is to identify a song using a small sample as input, regardless of its quality or origin. Music identification systems can take small samples of audio and analyze them to identify the song to which they belong. As these systems become more advanced, they will be able to recognize suboptimal samples, even when noise, hiss, or artifacts (clicks and pops) are present. Depending on the techniques used, it is even possible to identify songs having simple melodies, sung lyrics, or even musical scores as the input.

A broader approach is used by "music recommendation systems". These systems may include a search function and a media player. They don't try to produce perfect matches like identification systems do, but instead focus on generating song playlists according to the similarities between a chosen track and a predefined music collection, trying to keep a balance between quantity and quality (similarity). "Music discovery systems" are an extension of recommendation systems, as their main purpose is to give the user the possibility of finding new music according to the song provided as an input, and a personal music collection or history of reproductions. We can obtain a discovery system by filtering the output of a recommendation system, removing any songs and artists that the user already knows. The result will still be similar to the initial input, but will hopefully be new to the user.

Music recommendation systems can be classified into two types: context-based and content-based. Context-based systems describe music using a set of global metadata, generally stored as labels or tags. These tags are in some cases defined manually and they can give some information about the song, like interpreters, genre, album, date, etc. There are also numeric tags that can be generated automatically according to information obtained from the user's interaction with the system (e.g. number of sales, number of reproductions, and ranking). There are two important methods worth mentioning when it comes to gathering such data: "collaborative filtering" and "social filtering". On the first one, every action of a user is recorded including reproduction statistics, history of selections, and the liked/disliked items. The chronology of reproductions serves as a method to understand the individual preferences of a user, but can also be matched with the records of other users to find patterns that can be exploited to create statistically "ideal" playlists. Social filtering tends to be very similar, but the information comes from users that belong to a more integrated social network, like Facebook, Twitter, or Google+, where the actions of friends or contacts may have a greater impact on what's recommended or not. In this case the playlist will reflect the predilections of the user's friends and tries to increase the possibility of discovering new songs thanks to the data gathered out of the publications and posts of those contacts. For example, labels could be similar to "recommended by x" or "n number of friends have listened to this song".

Content-based systems depend instead on the physical, perceptive, and acoustical properties of the audio signal, which are an objective source of information. This information is obtained using Multimedia Information Retrieval (MIR) techniques that extract descriptors out of a song. These descriptors are called "features" and the process is called "feature extraction". They can describe properties like rhythm, harmony, or loudness. Songs can then be compared using algorithms specialized on finding similarities between the extracted features. Similar items are then used to generate a reproduction playlist that's presented to the user. However, features are rarely used directly to generate a playlist. Instead, they are processed and clustered into similar groups that are then labeled to generate metadata similar to the one used by context-based recommendation systems, as can be seen on figure 1.2. This grouping limits the versatility of the recommendations and favours performance over detail. There is however the advantage of having a tag generated by a logical process instead of possibly faulty user

input.

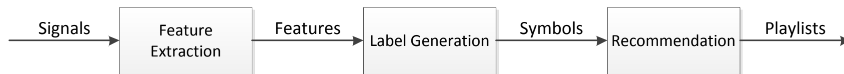


Figure 1.2: Workflow of a traditional recommendation system

One important difference between these two systems, on a user experience level, is that the content-based one can start generating playlists as soon as the music is analyzed without requiring any prior user interaction, since all the needed data is already contained in the song. On the other hand, context-based systems always require a "start-up" process where labels are created or provided, which is not an immediate process and may affect the availability and quality of the recommendation results. For example, a rating system is not useful when the number of votes is too low, and is useless when there are none. Another disadvantage of context-based systems lays in the use of text labels, since they may contain typographic errors, abbreviations, or idiomatic expressions.
However, content-based systems also have their own disadvantages. Recommendations based only on physical properties are not directly adapted to the user's tastes and predilections. Two songs with similar content descriptors may generate completely different reactions on the same person. To be able to learn from the user's experience and feedback, the system requires to implement a context-like ranking system.

The purpose of this thesis is to extend an existing application that implements a content-based music recommendation system that avoids the clustering/labeling process, and favours user interaction in its place. The objective is to let the user modify the value of these features dynamically using different input methods. The workflow can be seen on figure 1.3 and it shows how features are used directly into the recommendation process to keep a higher level of precision.

The application must be extended following a set of prerequisites that include new features mainly related to modularization, web streaming, and feature comparison. The modifications aim to give the user more control over the song's features and their evolution over time, and to improve the

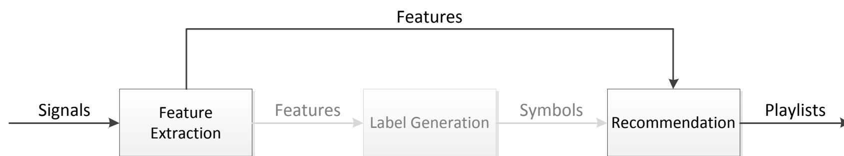


Figure 1.3: Workflow of the system to be extended

flexibility of the application on the programming and functional level. The resulting software shall be more robust and easy to modify and extend.

The objective of this project is to produce a content-based music recommendation application that generates playlists according to the properties of the audio being played, like tempo, mood, harmony, brightness, and loudness. This application must work in both a local and a web environment, with the possibility of being separated into a client-server infrastructure where the recommendation engine is not tied to the player. The core components of the application must be exchangeable, to give the possibility of growth and further improvement. It must be able to compare the properties of the audio segments of a song, or of the complete song if chosen, using the extracted audio features or their evolutions on time, and it must let the user interact with these properties to alter the output and improve the playlist generated. All this must be done while trying to maintain the look and feel of the original application that's being extended.

The thesis is organized as follows:

- In chapter 2 we'll take a look at the state of the art of music recommendation systems. We are going to see how most current applications work and how they have different approaches for the creation of playlists.
- In chapter 3, we define some of the most important theoretical concepts that are used throughout this document.
- In chapter 4 the system design details are presented. This includes the architecture of the system and software used.
- Chapter 5 explains the details behind the implementation of the new software.

- In Chapter 6 we evaluate the modifications done to the original application.
- Finally, chapter 7 presents conclusions and guidelines for even further future improvements and evolutions.

Chapter 2

State of the art

As introduced previously, most recommendation systems are aiming to consolidate themselves as platforms that can store, classify, filter, and stream music, independent of where the user is and how he is connected. Today, the most popular services are: Apple's iTunes[1], Amazon Cloud Player[2], and Google Music[3].

Apple has recently launched a couple of new services called iCloud[4] and iMatch[5], that allow paying clients to upload not only music, but also documents and pictures to Apple's servers, so that they can be permanently access through any device that is part of Apple's product offer, like Macs, iPads and iPhones, without any extra effort on the user's side. iMatch is a music identification system that analyzes the user's music collection finding matches with the iTunes database. If a match is found, the song is not uploaded to iCloud, but instead the user gains access to a high-quality 256 Kbps AAC DRM-free file (which means that the file can then be shared between devices). This set of functionalities becomes a music recommendation system when mixed with the iTunes service and its playlist generator, iTunes Genius[6].

Amazon's cloud service has a simpler approach: the user can upload his music collection and listen to it on the web or on Amazon's tablet, the

[1]Apple Inc., http://www.apple.com/itunes/

[2]Amazon Inc., https://www.amazon.com/gp/dmusic/mp3/player

[3]Google, http://music.google.com

[4]Apple Inc., http://www.apple.com/icloud/

[5]"A Clear Explanation of iTunes Match", http://www.macrumors.com/2011/11/14/a-clear-explanation-of-itunes-match/

[6]"What is iTunes Genius?", http://ipod.about.com/od/itunes/g/itunes_genius.htm

Kindle. The recommendation system that it uses is the one seen on Amazon's shopping web site, which simply recommends songs based on the user's acquisition history and Amazon's most-sold tracks. Google Music instead relies on the Android Market for purchasing music directly from the web and from mobile devices, and uses a recommendation system only improved by the social filtering done thanks to its connection with the social network of Google+.

Google Music and Amazon Cloud Player are music recommendation systems that mainly use context-based techniques to describe and offer their music. Other systems mix both content-based and context-based approaches to increase the accuracy of the results. An example of this is Apple's iTunes Genius system, which advertises the complexity of it's algorithms without publicly revealing them. Other systems that use mostly context-based techniques to create their playlists are: Last.fm[7], using an engine called "Audioscrobbler", keeps a record of what users listen to, and works mostly as a music discovery system based on collaborative and social filtering; Spotify[8], which is focusing on music discovery through social filtering using Facebook's Open Graph protocol[9]; Jango[10] and Grooveshark[11] stream music using recommendations based on labeling and social filtering; and finally we have Stereomood[12], an Italian web site that streams music based on social filtering and labeling, classifying music according to "mood tags".

Since our application is related to content-based recommendation, we'll now introduce some of the applications or services, either academic or commercial, that belong to this field. This list is divided in two sections: music recommendation/discovery systems, and music identification systems.

2.1 Music Recommendation/Discovery Systems

The following applications and services include many different functions like search, recommendation, reproduction, and commercialization of music. Their main objective is to not only help the user find what he's looking for, but also to help him discover new music that may be of his liking. This

[7]Last.fm Ltd., http://www.last.fm
[8]Spotify Ltd., http://www.spotify.com
[9]Facebook, http://ogp.me
[10]Jango, http://www.jango.com
[11]Escape Media Group, http://http://grooveshark.com
[12]Facebook, http://http://www.stereomood.com/

is usually done automatically as part of the reproduction of an audio track, without requiring any particular user interaction.

2.1.1 Pandora [13]

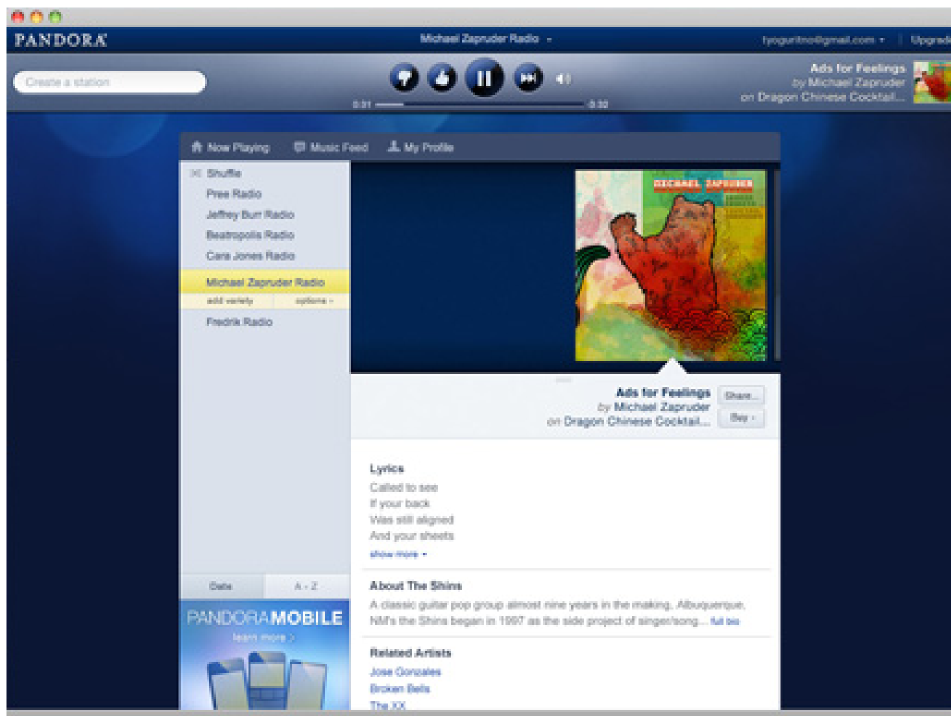


Figure 2.1: Pandora's Graphical Interface

Being based on the Music Genome Project, a document related to Music Recommendation Systems can't ignore the work done by the Music Genome Project and Pandora. It aims at describing songs by using many different attributes obtained as the result of an analysis of their musical qualities. Furthermore, these songs are also organized and filtered according to the user's input and collaborative filtering.
The types of features it handles are: Melody, harmony, tonality, rhythm, instrumentation, lyrics, vocals, influences, orchestration, etc. There are more than four hundred different types, which they call "genes", and they give Pandora the possibility of finding accurate similarities between different musical pieces. Also, in the case that a user doesn't like or approve a particular recommendation, there's the option of rating the piece or actually banning

[13]Pandora Media, http://www.pandora.com

it from playing again within a predefined amount of time (an example of collaborative filtering).
Pandora's user interface can be seen in figure 2.1.

2.1.2 Mufin [14]

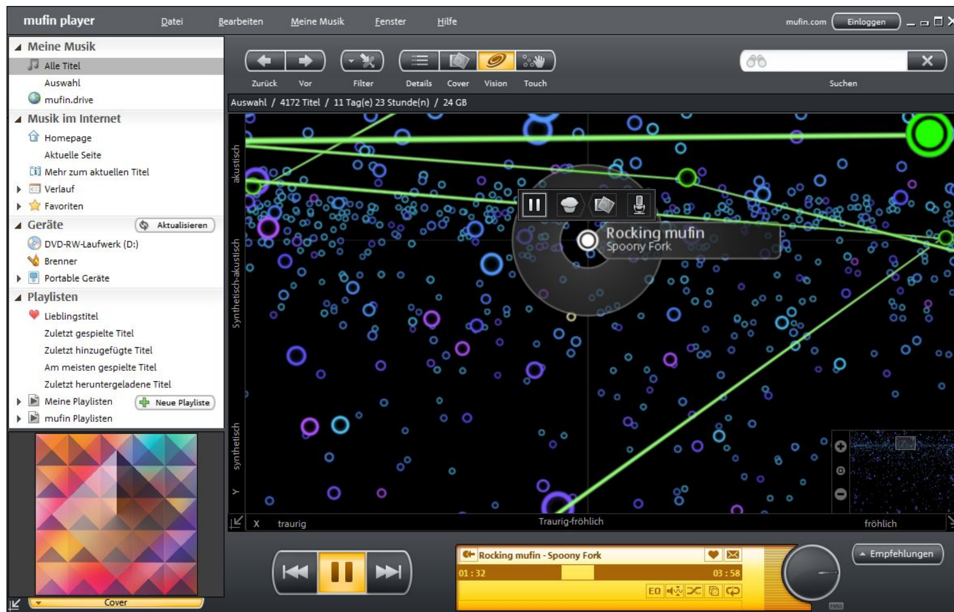


Figure 2.2: Mufin's self-denominated "interactive 3D music universe"

Mufin (Music Finder) is an offshoot from the Fraunhofer Institute. It began as part of an investigation on "audio fingerprinting", which resulted on a technology called "AudioID"[1]. It offers a service that combines mood based music recommendation and cloud storage functionalities to allow users to listen to the music that suits their current emotional state on any supported device among PCs, mobiles, and web browsers.

Mufin requires that the user uploads the songs before they are analyzed. In this way the system can determine their acoustic properties and classify them as happy, sad, or as a mix of different classifications. As we have seen on section 1, this is an example of a content-based approach that uses classification to label songs.

[14] Mufin GmbH, http://www.mufin.com

What really distinguishes Mufin from the other services is its implementation of the playlist concept. Tracks can be displayed in a 3D environment ("Audio Landscape")2.2, where the user can navigate around using the relationships between different items. It can also generate automatic playlists with similar songs to the one currently playing.

2.1.3 Musicovery [15]

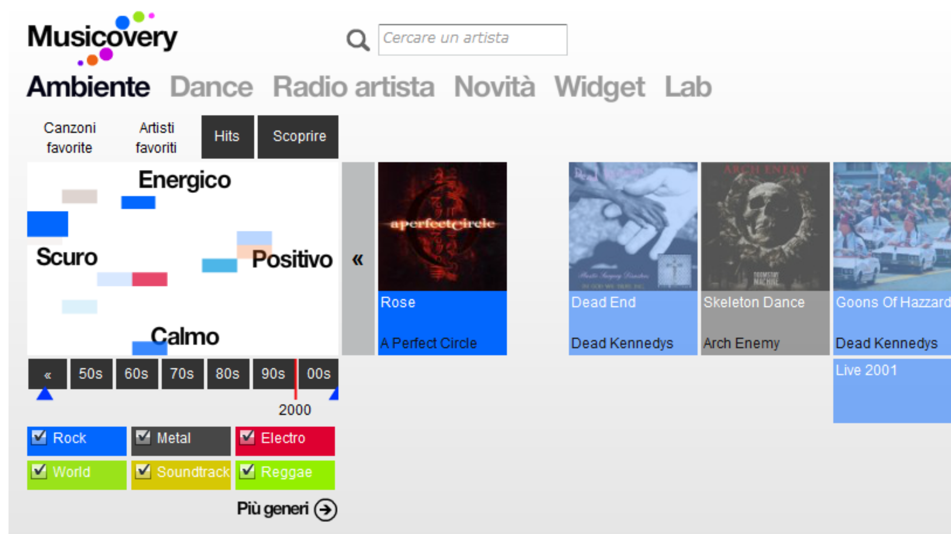


Figure 2.3: Musicovery's Flash interface in Italian

Musicovery is a service that allows the listener to choose songs according to their mood using a matrix ("mood pad") that exposes many different values between 2 axes: dark/positive and calm/energetic.
The system extracts 40 acoustic parameters of each song, each parameter taking between 3 to 20 values. An algorithm converts these values into a position on the mood pad (seen on the top left of figure 2.3).[16] The mood pad's concept is constantly evolving and improving and, as an interesting feature, changes its recommendations according to the geographic position of the user, to include local songs and genres and customize the interface, as can be easily seen on the included snapshot.
This service also gives the user the opportunity to rate the songs (represented as a heart for positive rates and a broken heart for the negative ones) to include collaborative filtering.

[15] Vincent Castaignet & Frederic Vavrille, http://www.musicovery.com
[16] http://musicovery.com/aboutus/aboutus.html

2.1.4 The Echo Nest [17]

The Echo Nest is a platform ("music intelligence platform") that offers an extensive set of tools related to multimedia information retrieval, co-founded by two MIT PhDs after 12 years of research and development done by the MIT in conjunction with the Universities of Columbia and Berkeley[7]. It crawls the web looking for musical content that is later analyzed to extract and store features like tempo, mode, beats, segments, pitch, key, timbre, and loudness. This information is exposed through a real-time API to registered application developers that wish to create music related software, including music recommendation systems, without having to implement the required feature extraction techniques.

To obtain this information, the platform uses advanced techniques that include data mining (analysis and filtering of data according to relevance), natural language processing (using artificial intelligence to improve the system's understanding of human defined labels), acoustic analysis (feature extraction), and machine learning (using artificial intelligence to improve the system's classification and pattern recognition).

One of the features offered by the platform is the possibility of automatically creating static or dynamic playlists that may use one or different types of recommendation according to the input given. For example, using collaborative filtering it allows skipping particular artists or "boosting" them so that they appear more often. Technically, the call is made using a simple HTTP GET and the answer can be either in JSON, JSONP, XML, or XSPF format, something that makes the platform compatible with most modern systems, if not with all.
More interesting for our current approach is the possibility of searching songs using the parameters depicted above. The platform even permits the identification of songs (either with an HTTP GET or POST) thanks to two custom "fingerprinting" systems: The Echo Nest Musical Fingerprint (ENMFP) and the Echoprint. The ENFMP is a closed source database of roughly 30,000,000 songs and works better on full file recognition with speeds around 20x the real time speed of the song. The Echoprint is an open source database with approximately 200,000 songs publicly available, and supports "over the air" recognition, with speeds up to 100x real time speed.
As an interesting fact, The Echo Nest has made a surprising deal with EMI

[17]The Echo Nest Corporation, http://the.echonest.com

Music to allow developers to create innovative commercial applications for artists under a revenue sharing model. As part of this partnership, users are given access to different "sandboxes" containing music from various artists like Gorillaz, Pet Shop Boys, and Eliza Doolittle[18].

2.1.5 SoundBite [19]

The Centre for Digital Music (C4DM[20]) of the Queen Mary University of London has created different applications that do feature extraction and music recommendation. Feature extraction is done with programs like Sonic Annotator[21], Sonic Visualizer[22], and the QM Vamp plugins[23]. Music recommendation is done with SoundBite, a plugin that integrates with iTunes and Songbird (a music organizer and player), adding a new icon on the interface that can create a playlist according to the currently selected song (figure 2.4). To be able to do this, once the plugin is installed, SoundBite does a content-based analysis of the user's music collection. The technique in

Figure 2.4: SoundBite's integration with iTunes

which SoundBite is based is called "Audio Features Ontology"[24][8], which is part of the research done by the C4DM on "Music Ontology"[17], a method for describing music on the semantic web[25].

[18]"EMI Partnership", http://developer.echonest.com/sandbox/emi/
[19]C4DM Queen Mary University of London, http://www.isophonics.net/content/soundbite
[20]C4DM, http://www.elec.qmul.ac.uk/digitalmusic
[21]C4DM Queen Mary University of London, http://www.isophonics.net/SonicAnnotator
[22]C4DM Queen Mary University of London, http://sonicvisualiser.org
[23]C4DM Queen Mary University of London, http://www.isophonics.net/QMVampPlugins
[24]Music Ontology tools, http://motools.sourceforge.net/doc/audio_features.html
[25]Music Ontology, http://musicontology.com

2.2 Music Identification Systems

Some applications are not designed to recommend music to the user, but to allow him to find a song he's looking for using friendly and intuitive methods. Searching for a song may yield a list of results consisting of different matches, in a similar way to how recommendation systems generate playlists. In fact, a recommendation system generally uses the same type of techniques to compare tracks and create recommendations.

Shown below are some popular web applications that use music identification techniques to help users find a particular song.

2.2.1 Musipedia [26]

Figure 2.5: Musipedia's main menu

Inspired by Wikimedia, this web site helps people find the music they are looking for by allowing them to use sound instead of text as a search parameter. Users can whistle on the microphone, use a virtual piano, or just tap the rhythm of the song on the keyboard by choosing the corresponding option on the web site's main menu, as seen on figure 2.5. The system uses the melody, defined as pitch and rhythm, the melodic contour (description of the notes as simply "going up" or "going down", also called "the Parsons Code for Melodic Contours"), or just the rhythm to try and find coincidences within its database. This search engine is called "Melodyhound" and it allows the user to retrieve music sheets, MIDI files, and information about songs and composers.
The site also includes a free SOAP Interface that enables consuming applications to search their database using Web Services.

2.2.2 Midomi & SoundHound [27]

Similarly to Musipedia, Midomi allows users to sing, hum, or whistle on their microphone, using the interface shown in figure 2.6,to look for a particular song. As a commercial solution, Midomi provides client application for the

[26]Rainer Typke, http://www.musipedia.org
[27]Rainer Typke, http://www.midomi.com

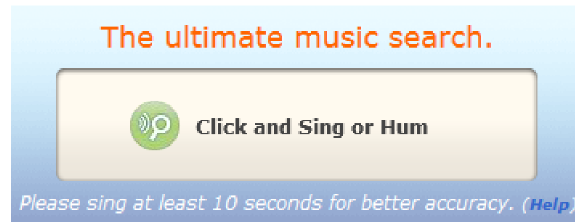


Figure 2.6: Midomi's "Click and Sing" interface

mobile market, called SoundHound.
The most obvious problem of Midomi is that it can only find a song if it already belongs to the service's database. This database is fed with user recordings and input, which is then compared with the searched melody.

2.2.3 Tunebot [28]

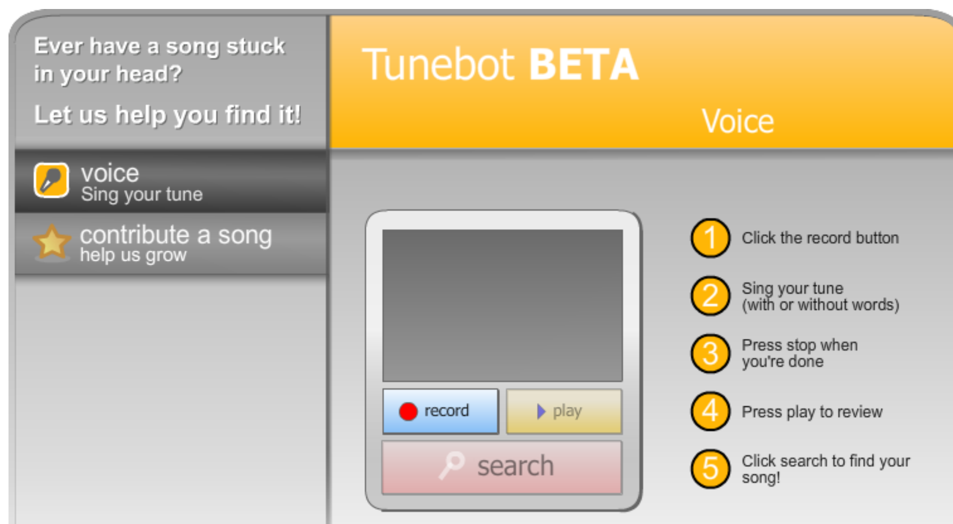


Figure 2.7: Tunebot's interface

Tunebot is a project done by the Interactive Audio Lab of the Northwestern University (Evanston, USA), funded in part by the National Science Foundation. Just like previous services, it allow users to search for a song using as input a recording of the user singing or humming. The system also allows music notation, even if the main interface doesn't display that option to the user (figure 2.7). The main difference with the previous solutions

[28]Northwestern University Interactive Audio Lab, http://tunebot.cs.northwestern.edu

is that this service doesn't store actual songs, but instead it keeps only the recordings uploaded by users, which are then analyzed and associated to the real song in Amazon.com. The only way to find a song is if someone else has already sung it or hummed it before on Tunebot.
The theoretical basis for this service includes research concepts like "query by humming"[15][11], "online training of music search engines"[14], and also "searchable melodies"[4].

Chapter 3

Theoretical Background

In this chapter we go deeper into the detail of what is a musical feature, how these features are extracted, and how content-based recommendation systems find similarities between tracks as part of the playlist generation.

3.1 Feature Extraction

To have a good recommendation system, first a good feature extraction has to be done. If the quality or the fidelity of the features is not good enough, it doesn't matter how good the recommendation system is, the results will not be acceptable. Our system will allow the user to interact directly with the feature values, so they need to be easy to understand and associate with the properties human can perceive when they listen to a song. Features like tempo (measured as the number of beats per minute or BPM) or loudness (measured as the RMS value of the signal) can be extracted after a low level analysis of the audio signal, and they offer a good frame of reference to evaluate song similarity since they are characteristics that people are already familiar with, and can be easily related with concepts like "slow music" or "loud music". On the other hand we have features that can be grouped to describe more subjective measures (high level features). For example, mood can be defined as a function of its timbre, intensity, and rhythm [16].

On section 1 we introduced the way in which recommendation systems work, and how they do a classification using a set of features (labeling). Figure 3.1 displays the structure of a typical classification system. The part of the process that is of special interest for us is the pre-processing: inputs must be monophonic, de-noised, down sampled audio signals. Also, since features are rarely uniform during the whole track, the signal has to be

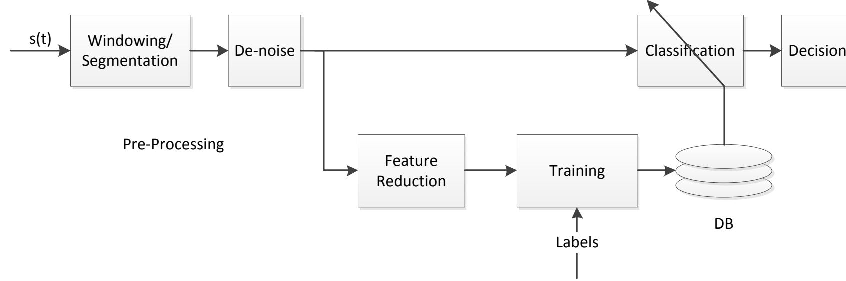


Figure 3.1: Structure of a generic classification system

split into smaller meaningful pieces. These pieces are generally defined by changes in tempo, harmony, spectrum, etc. and can be selected manually (human interaction) or automatically as part of the extraction logic.

The particular technique that concerns us is called "measure of audio novelty" [9]. Based on the concept of self-similarity, it is calculated using a workflow (figure 3.2) that includes the following set of steps:

- Audio is parameterized using standard spectral analysis.
- Frames are tapered with a Hamming window and an FFT is applied.
- The logarithm of the FFT is used as the Power Spectrum of the signal.
- High frequency components (> Fs/4) are discarded as not relevant for self-similarity.
- The resulting vectors are embedded into a matrix by means of a (dis)similarity measure D:

$$D_c(i,j) \equiv \frac{v_i \cdot v_j}{\|v_i\|\|v_j\|} \tag{3.1}$$

- The resulting matrix S contains the similarity metric for all the frame combinations.
- Since S tends to look like a checkerboard, if we perform the convolution along the diagonal with a checkerboard looking kernel (figures 3.3(a) and 3.3(b)), we will obtain as a result a one-dimensional function that corresponds to the measure of novelty. This Gaussian chequerboard

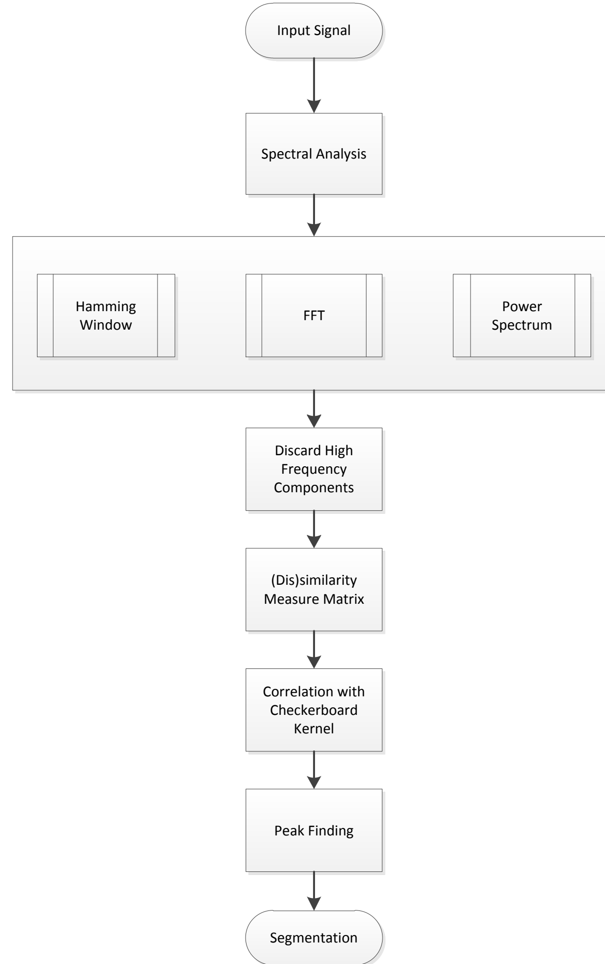
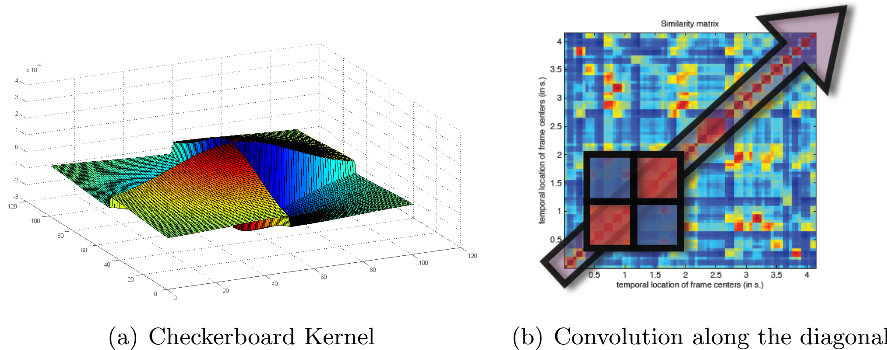


Figure 3.2: Workflow of a novelty based segmentation system

kernel is obtained from a point to point multiplication between the bi-dimensional Gaussian function and the following function:

$$f(x,y) = \begin{cases} +1 & \text{if } sign(x) = sign(y) \\ -1 & \text{otherwise} \end{cases}$$

- The track is then segmented based on the peaks found on the results.



(a) Checkerboard Kernel (b) Convolution along the diagonal

The resulting pieces are then processed to extract the features, but avoiding the labeling process that most systems do after the extraction.
Next we will describe some of the features that are used in this project:

3.1.1 Harmony

A simple definition of harmony is provided by the Encyclopedia Britannica:

> Harmony, in music, the sound of two or more notes heard simultaneously. In practice, this broad definition can also include some instances of notes sounded one after the other. If the consecutively sounded notes call to mind the notes of a familiar chord (a group of notes sounded together), the ear creates its own simultaneity in the same way that the eye perceives movement in a motion picture. In such cases the ear perceives the harmony that would result if the notes had sounded together. In a narrower sense, harmony refers to the extensively developed system of chords and the rules that allow or forbid relations between chords that characterizes Western music.

The key words here are "notes" and "chords". The simplification we use in this project is that harmony can be represented as a key (C, D, E, D#, Bb, etc.) and a mode (major, minor). It's obtained through a pitch analysis, using a "chromagram" obtained as a redistribution of the spectrum energy along the different pitches ("chromas") of the FFT of the signal, after limiting the frequency range to cover only a few octaves and the magnitude to only the 20 highest dB (figure 3.3).

Determining the harmony requires the computation of the key clarity (the probability associated with each possible key candidate) through a cross-correlation of the wrapped (octave information discarded) and normalised chromagram, with similar profiles representing all the possible tonal-

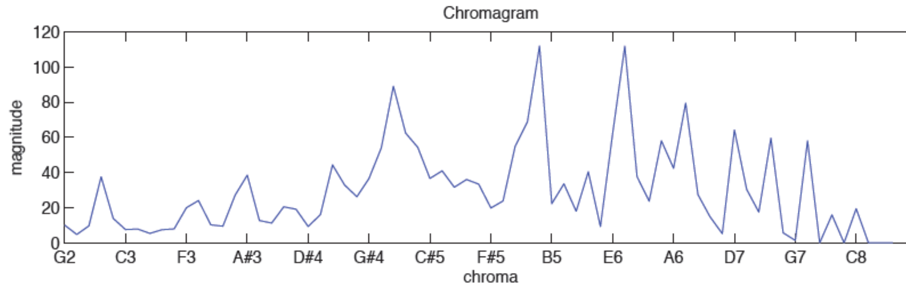


Figure 3.3: Unwrapped Chromagram

ity candidates[12][10]. The resulting graph indicates the cross-correlation score for each different tonality candidate (Figure 3.4), and the maximum value corresponds to the selected harmony.

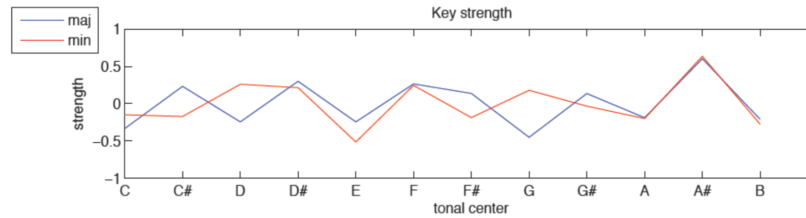


Figure 3.4: Key clarity

3.1.2 Tempo

As referenced before, it's expressed in BPM, and can be found through the computation of an onset detection curve, applied to multiple functions (signal envelope or spectrum), showing the successive bursts of energy corresponding to the successive pulses (Figure 3.5). Peak picking is then applied to the autocorrelation function of the onset detection curve.

3.1.3 Brightness

The brightness of a track is roughly based on the idea of sound having a similar behaviour to visual brightness, and can be expressed as the power

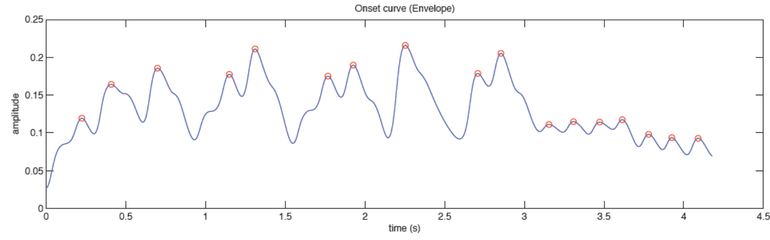


Figure 3.5: Tempo: onset detection

of the low-frequency bands over the power of the high-frequency ones. This ratio usually ranges between 0.2 and 0.8. Considering a threshold close to 1500 Hz, as displayed on figure 3.6, brightness can be calculated by the following formula:

$$brightness = \frac{\int_{threshold}^{+\infty} X(\omega)}{\int_{0}^{threshold} X(\omega)} \tag{3.2}$$

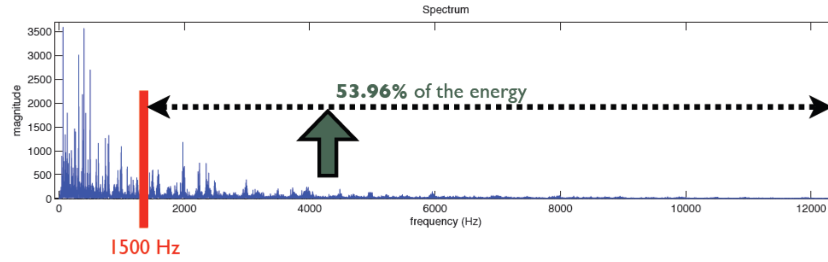


Figure 3.6: Brightness

3.1.4 RMS

The "root mean square" measure gives us the global energy of the signal (closely following its envelope) using the formula:

$$x_{rms} = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} = \sqrt{\frac{x_1^2 + x_2^2 + x_3^2 + ... + x_N^2}{N}} \tag{3.3}$$

3.1.5 Mood

Mood is a high-level feature describing the emotional content of a musical piece. The lack of a standard set of emotions implies that any implementation is free to use any useful psychological description. For our project we use

mood as the interaction between a measure of intensity (based on loudness) that tries to define a particular segment of audio as energetic or not, and an emotional component defined by the corresponding timbral analysis[3].

3.2 Feature Based Recommendation

Once we have extracted the features, the task of a recommendation system is to take those results and use them to generate a track playlist based on a given input. Since we are looking for similar items, this input will usually be an array of features corresponding to the segment of audio being currently played. Also, for this project the comparison process may take as input free values defined directly by the user, but always within the valid range of the feature. In this project we use two approaches to this process: using similarity functions over the features corresponding to each segment of the track(the cells obtained as part of the feature extraction process explained on section 3.1), or by considering the features of a song as an array of values changing over time. Since we were looking for a method that allowed us to compare the evolution of a function over time, without the substantial loss of precision that comes as a side effect of clustering methods, we decided to use a short delay cross-correlation algorithm based on a one by one comparison done with similarity functions.

3.2.1 Similarity Functions

Some of the features we use are just numeric values that can be compared directly using the following function:

$$compare_{feature}(...) = 1 - |feature_1 - feature_2| \tag{3.4}$$

This applies to brightness and rms. However, harmony, tempo, and mood require different approaches. Tempo, for instance, uses a distance function calculated as a normalized Gaussian function centred in one of the two BMP values and with an appropriate variance (Figure 3.7). It returns high values when the two tempos are near and reasonably decreasing values when they move away.

Harmony and mood instead rely on similarity tables and formulas that differ according to the software implementation chosen. In particular, harmony's similarity functions may vary according to the type of music that's being analyzed. For example, Western music has different harmonic rules than Eastern music. For this project we use the Western rules. In Table 3.1 and Figure 3.8 we present the values of the compare function between (C, maj)

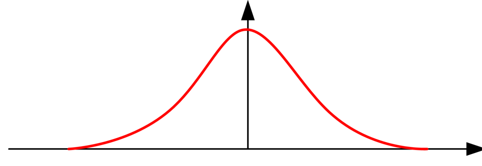


Figure 3.7: Tempo similarity graph

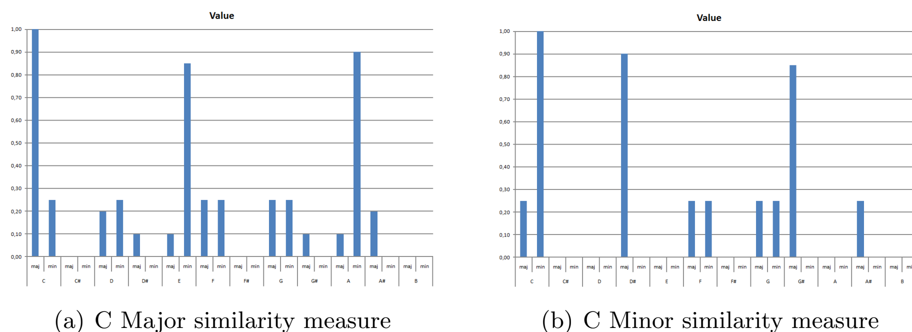
and all other harmonies, and between (C, min) and all other harmonies. The other values could be found musically transposing the notes.

Key	Mode	Value
C	maj	1.00
	min	0.25
C#	maj	0.00
	min	0.00
D	maj	0.20
	min	0.25
D#	maj	0.10
	min	0.00
E	maj	0.10
	min	0.85
F	maj	0.25
	min	0.25
F#	maj	0.00
	min	0.00
G	maj	0.25
	min	0.25
G#	maj	0.10
	min	0.00
A	maj	0.10
	min	0.90
A#	maj	0.20
	min	0.00
B	maj	0.00
	min	0.00

Key	Mode	Value
C	maj	0.25
	min	0.00
C#	maj	0.00
	min	0.00
D	maj	0.00
	min	0.00
D#	maj	0.90
	min	0.00
E	maj	0.00
	min	0.00
F	maj	0.25
	min	0.25
F#	maj	0.00
	min	0.00
G	maj	0.25
	min	0.25
G#	maj	0.85
	min	0.00
A	maj	0.00
	min	0.00
A#	maj	0.25
	min	0.00
B	maj	0.00
	min	0.00

Table 3.1: C Major (left) and C Minor (right) similarity measures

Beside this, the similarity measure also considers the *keyClarity*, i.e. the confidence of the detected harmony. The key clarity not only expresses the



(a) C Major similarity measure (b) C Minor similarity measure

Figure 3.8: Harmony similarity measure

probability of the correctness of the given harmony, but gives hints about the amount of inharmonic noise present in the piece. If this value is very high, a strong harmonic component is perceived by the listener. On the contrary, when this value is low, the segment does not present a well-defined harmony.

When two segments have both a high key clarity, the overall harmony similarity function should consider the values defined in Table 3.1. However, when two segments have both a low key clarity, the actual value of the key is not important since the value has a low confidence.

In particular given two audio samples, with key clarity $keyClarity_1$ and $keyClarity_2$ respectively with the key similarity computed as shown before (see Table 3.1), the harmony similarity measure should show the qualitative behaviour described in Table 3.2.

	High		Low		$keyClarity_1$
	High	Low	High	Low	$keyClarity_2$
High	$keySimilarity$	Medium	Medium	High	
Low	$keySimilarity$	Medium	Medium	High	
$keySimilarity$					

Table 3.2: The qualitative measure of harmony similarity

.

The qualitative graph of the similarity function is shown in Figure 3.9; when both key clarities are zero, the overall similarity is high, when both are one, the value is the real key similarity.

The overall similarity measure is obtained by the combination of two functions:

$$f_1(...) = (1 - keyClarity_1) \cdot (1 - keyClarity_2) \tag{3.5}$$

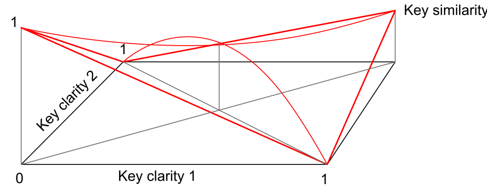
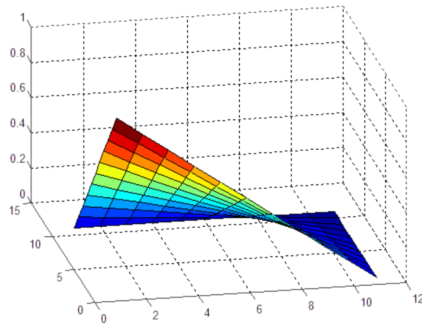


Figure 3.9: The qualitative graph of harmony similarity

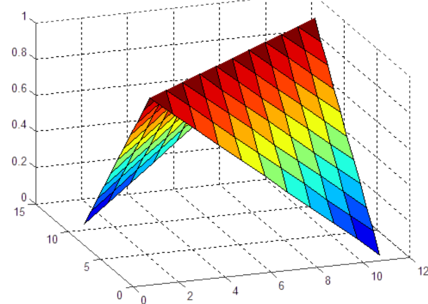
that is maximum when both $keyClarity_1$ and $keyClarity_2$ are zero, and the following:

$$f_2(...) = keySimilarity \cdot (1 - |keyClarity_1 - keyClarity_2|) \tag{3.6}$$

whose value decreases when $keyClarity_1$ and $keyClarity_2$ are distant. Figure 3.10(a) shows the graph of f_1, whereas Figure 3.10(b) shows the graph of f_2.



(a) Graph of f_1



(b) Graph of f_2

Figure 3.10: Harmony similarity measure

The harmony similarity is obtained combining the contributes of the two functions:

$$compare_{harmony}(...) = f_1(...) + (1 - f_1(...)) \cdot f_2(...) \tag{3.7}$$

In Figure 3.11 an example of similarity function is plotted (key similarity is set to 0.5). We can see that the qualitative behaviour resembles the one in Figure 3.9.

Mood in our project is represented by four psychological classes: exuberance, anxious, contentment, and depression. The distance measure among the classes is summarised in Table 3.3. Using a bi-dimensional plane like the one shown in figure 3.12 to calculate the mood[13], the similarity is 1.00

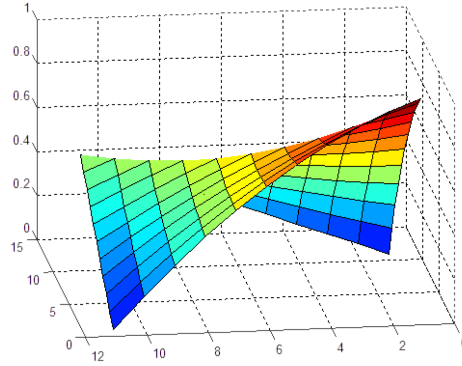


Figure 3.11: Graph of $compare_{harmony}$

when the two moods are the same, 0.33 when they are in the same column, 0.40 when they are in the same row and 0.10 otherwise.

	Exuberance	Anxious	Contentment	Depression
Exuberance	1.00	0.40	0.33	0.10
Anxious	0.40	1.0	0.10	0.33
Contentment	0.33	0.10	1.0	0.40
Depression	0.10	0.33	0.40	1.0

Table 3.3: Mood similarity table

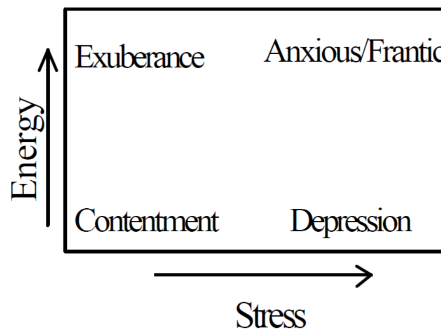


Figure 3.12: Mood bi dimensional plane

3.2.2 Cross-correlation

The cross-correlation between two signals gives us a good idea of how similar they are. This is attained by applying a constantly increasing delay to one of them and calculating the dot product between the correspondingly aligned samples. The circular cross-correlation of two signals x and y can be defined by:

$$\frac{1}{N}(x \star y)(l) \triangleq \frac{1}{N}\sum_{n=0}^{N-1}\overline{x(n)}y(n+l),\ l = 0, 1, 2, \ldots, N-1. \tag{3.8}$$

The lag l is an integer value, and the resulting values go from -1 to 1. A value close to 1 means that both series are similar at that particular delay. In the same way, a -1 indicates that the series are exactly opposite at that delay. It's also important to note that the cross-correlation is not commutative:

$$(x \star y) \neq (y \star x) \tag{3.9}$$

This means that even if we use some sort of cache on the system to further improve performance, our calculations have to be done in both directions anyway.

The use of cross-correlation in this project allows the user to specify the evolution of a particular feature or feature set, which will then be compared with the whole feature database using the sum of the cross-correlations and a weight system. For example, the user may want to listen to a song that is slow on the beginning, fast on the middle, and slow again on the end, just like figure 3.13 shows. After normalizing the data, we can calculate the cross-correlation with another feature set (in red). The inputs and the result (in green) can be seen on figure 3.14
We can see that after the first five samples, the cross-correlation value starts to decrease since the shifted shapes of the signals are no longer similar. As a conclusion of this we can consider using a lower delay since similar series will give us the bigger cross-correlation within the first few samples. Furthermore, the results will be finally filtered using a weighting mechanism and a predefined threshold to avoid "false positives".

3.3 Ranking System

An important part of the original application that this project is expanding, is the ranking system. This is a mechanism that learns the taste of the user

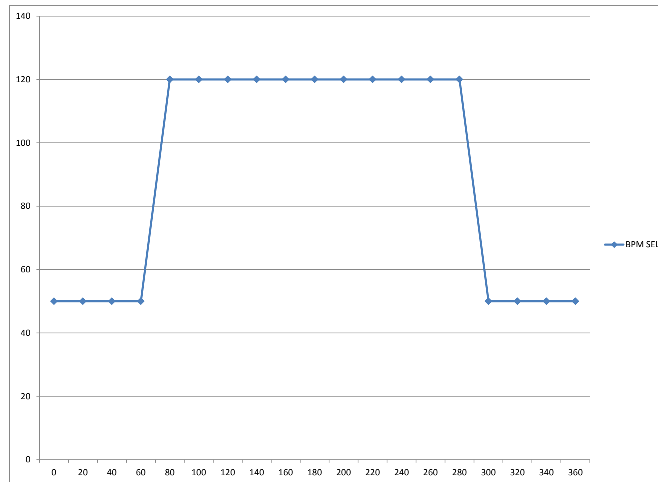


Figure 3.13: User's BPM selection

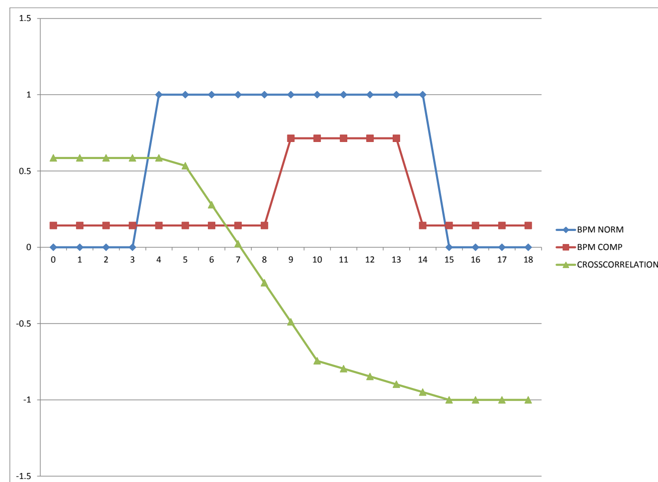


Figure 3.14: Normalized cross-correlation of two feature sets

and ranks the music according to the history of the user's preferences. The more the system is "trained", the more it adapts to the inferred preferences of the user, improving the quality of the recommended music. Since this

process runs on the background, the user implicitly interacts with the learning algorithm by selecting items from the proposal list.

This ranking system was implemented by training a Gaussian Mixture Model:

3.3.1 Gaussian Mixture Model (GMM)

GMMs are used in the field of speech processing, mostly for speech recognition, speaker identification and voice conversion, because of their capability to model arbitrary probability densities and represent spectral features. The GMM approach assumes that the density of an observed process can be modelled as a weighted sum of component densities $b_m(x)$:

$$p(x|\lambda) = \sum_{m-1}^{M} c_m b_m(x) \tag{3.10}$$

where x is a d-dimensional random vector, M is the number of mixture components and $b_m(x)$ is a Gaussian density, parameterised by a mean vector μ_m and the covariance matrix Σ_m. The coefficient c_m is a weight that is used to model the fact that the different densities have different heights in the probability density function. The parameters of the sound model are denoted as $\lambda = \{c_m; \mu_m; \Sigma_m\}$, m = 1, ..., M. The training of the Gaussian Mixture Models consists in finding the set of parameters λ that maximises the likelihood of a set of n data vectors.

The Expectation Maximisation (EM) algorithm is one of the alternatives available to perform such estimation[2]. It works by iteratively updating the vector λ and the estimation of the probability density function $p(m|x_i, \lambda)$ for each element in the training set. In the case of diagonal covariance matrices the update equations become:

$$\mu_m^{new} = \frac{\sum_{i=1}^{n} p(m|x_i, \lambda) \cdot x_i}{p(m|x_i, \lambda)} \tag{3.11}$$

$$\Sigma_m^{new} = \frac{\sum_{i=1}^{n} p(m|x_i, \lambda)(x_i - \mu_m)^T (x_i - \mu_m)}{p(m|x_i, \lambda)} \tag{3.12}$$

$$c_m^{new} = \frac{1}{n} \sum_{i=1}^{n} p(m|x_i, \lambda) \tag{3.13}$$

the value $p(m|x_i, \lambda)$ is updated at each iteration by the following equation:

$$p(m|x_i, \lambda) = \frac{c_m b_m(x_i)}{\sum_{j=1}^{M} c_j b_j(x_i)} \tag{3.14}$$

Let us now consider the decision process: if we have a sequence of $L \geq 1$ observations $X = x_1, x_2, ..., x_L$ and we want to emit a verdict, we have to choose the model among $\lambda_1, \lambda_2, ..., \lambda_K$ that maximises the a posteriori probability for the observation sequence:

$$\hat{k} = \arg \max_{1 \geq k \geq K} P(\lambda_k|X) = \arg \max_{1 \geq k \geq K} \frac{P(X|\lambda_k)P(\lambda_k)}{p(X)} \tag{3.15}$$

The computation can be greatly improved: in fact $p(X)$ is the same for $k = 1, ..., K$. Furthermore, assuming that $P(\lambda_k)$ are equal for each class of sounds, and using logarithms and the independence between observations, the sound recognition system computes:

$$\hat{K} = \arg \max_{1 \geq k \geq K} \sum_{l=1}^{L} \log(p(x_i|\lambda_k)) \tag{3.16}$$

Chapter 4

System Design

The purpose of this project was to improve an existing music recommendation application made in Java, adding some new features to extend its functionality and prepare it for a larger implementation and use.

In this chapter we present the whole system design, the architecture chosen, the software used, and the premises that lead to most of the decisions taken in the process.

4.1 The existing application

The existing software, called "PolySound", is a content-based recommendation system created in Java[6]. When it reproduces a song, it also generates a playlist according to the feature similarity between the currently playing audio segment and the segments of the other available songs found in the user's music collection. The user also has the option to define manually the value of the features that are used to generate the recommendation by using graphical components and different types of input. The features it includes are: tempo, brightness, rms, mood, and harmony.

This existing Java application was implemented as a NetBeans project, organized in approximately 16 packages divided on libraries and functionality modules. It didn't provide Feature Extraction functionalities, as this part was done offline with a series of scripts using Matlab. The application had some hardcoded paths indicating the folders corresponding to the audio tracks to be played and their feature information.

The features were stored in a series of XML files, one per feature, named with the schema "Song name_feature type" (e.g. "All I got.wav_tempo.xml"). There was also a file per track indicating the segmentation of the features, named with the same schema, but with the text "anchors" as the suffix (e.g. "All I got.wav_anchors.xml"). Feature files also had segmentation data (position in frames), similar to what anchors contain. An anchor's position was defined as a number of frames, which is dependant on the format of the source file. This format was in this case the same one used to extract the features: 16 bit monophonic PCM at 11025Hz audio.

```xml
<feature id="tempo">
        <data>
                <dataitem position="1">
                <attributes>
                        <attribute name="bpm">93</attribute>
                </attributes>
                </dataitem>
                <dataitem position="220501">
                <attributes>
                        <attribute name="bpm">94</attribute>
                </attributes>
                </dataitem>
                <dataitem position="330751">
                <attributes>
                        <attribute name="bpm">116</attribute>
                </attributes>
                </dataitem>
        </data>
</feature>
```

Figure 4.1: A feature's xml representation

Running the application meant loading all the features into memory and looping over them every time a recommendation needed to be done, using a similarity function to rate them. The result was then displayed to the user as a vertical list of tracks were the recommended piece was highlighted.

On the panel of the right, as can be seen on figure 4.2, there are 6 different tabs. Some of the most relevant ones are:

- Values: where the user can "override" a feature value to modify the output of the system.

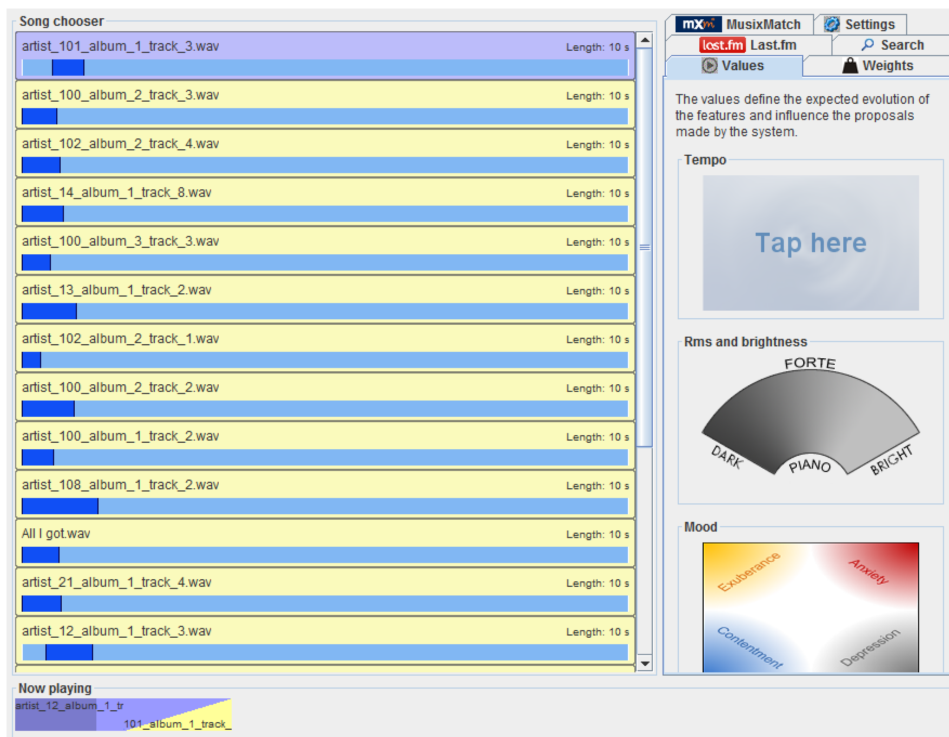


Figure 4.2: PolySound's Graphical Interface

- Weights: the user can choose how much one feature affects the recommendation output by changing its weight from 0 to 100 (%).
- Settings: found here are options that affect the rendering of the song or the way the recommendation system works. The user can choose to have one or many recommendations per playlist item, to automatically add the current song to a "Taboo List" (TabuList) so that the song doesn't get repeated again, to use beat-matching during the cross-fading of two segments (speeding up or down the audio pieces to obtain a speed match between them, without affecting the pitch), and to select the type of reproduction of the song, which can be normal (the user can change songs when he wishes), skipped (the program automatically selects the next recommended item), or continuous (the song gets played from beginning to end without offering a playlist).
- Search: Find a track by name.

The application had no Play, Pause, or Stop functionality. It chose a song randomly on startup and kept playing the audio until closed (when a track

was finished, it automatically chose the next one).

4.2 Requested Features

Improving the current application implied modifying, and in some cases rebuilding, the existing code. For this purpose we defined some requirements at the beginning of the project, which we shall now explain in detail. The first one is modularization: the new application must be neatly organized to allow for easy modification of the code. It shall also be organized in clearly defined packages and keep the functionalities distributed according to roles and scopes.
The implementation must include "over the web" features, since most current applications are starting to point to the cloud for storage and transmission of content. Of course, the infrastructure needed for a solution that big would simply be too much for our project, so we are limiting the scope to web streaming and a server-side common pool of songs.
Another important requirement is the use of a pluggable feature code implementation. This means that if a new feature is to be added or eventually removed from the system, it should be done as part of a process as simple as possible. We are not aiming at "hot swapping" of components, but the idea is to be able to add .jar files to a particular path allowing the software to load the code on startup and use it as needed. This plugin system should be available for the client (graphical interface), as well as for the server.
Since the application must have web features, a request was made to include the possibility of working online and offline, or maybe in a mixed mode. In addition to this the client used must be exchangeable, meaning that anyone can develop a client application for any device (PC, mobile, or web), using any operative system and any programming language, and then connect to our server. For this reason we have to choose carefully the core functionalities of the system, since not all clients may be able to support advanced functions.
Finally, as indicated on section 3.2, the system must use two recommendation approaches: the recommendation based on the features corresponding to single segments (cells) done by the original application, and a recommendation based on the feature evolution of a track over time to find similar songs, instead of cells. Both options shall be made available to the user, so he can choose which one to use.

4.3 Architecture

The main concern when creating the new architecture was about the extension to the web-based paradigm. As a consequence we decided to migrate to a "Server/Client" architecture, instead of the original "local only" implementation. What this means is that the recommendation logic could eventually be located on a remote server instead of on the client's machine. However, since the recommendation system needs to be able to recommend songs from its own pool, streaming becomes the only feasible option.

4.3.1 Streaming

On-demand streaming works as follows: the client asks the server for a file, the server responds with a header that indicates the type of the file (usually the MIME type) and the length of the download in bytes (if available). The rest of the response corresponds to the binary data of the file that gets buffered by the client. Depending on the format and length of the download, the client may begin the reproduction of the audio stream after enough information has been buffered and decoded. If the user decides to change the track, the process begins again.
It's important to notice that this process can't be done in real time: the initial buffering time depends on the quality of the connection, so there's always latency between the moment the user chooses an item and the moment the file starts playing.
Since the existing application is based on track segmentation by predefined anchors, the streaming logic has to overcome many different obstacles to give an good user experience. For example, the server must be able to send only the requested piece of audio to the client, instead of the whole file, since it's possible that the next request belongs to a different file altogether. In fact, the client will be able to change pieces constantly (there's actually an option for this on the settings menu), and has to do cross-fading between pieces when changing tracks, slowly decreasing the volume of the first song, while at the same time increasing the volume of the starting one. This means that the buffer cannot be fed directly to the sound card, since the signal may have to be pre-processed before reproduction. This leads to a "last minute buffering" strategy that may be troublesome for large pieces.
Another thing to consider is that the original files may be in an inappropriate format for streaming and therefore may need to be converted. The format must be chosen keeping into account file size and sound quality.

All these considerations where analyzed and taken into account when creating the architecture, but the last one is considerably more complicated since we can't ignore the possibility of having limited bandwidth clients (e.g. mobile phone clients) in situations where it's unaffordable to download buffer segments that may not be played at all.

4.3.2 Server/Client Architecture

The requisite of making a web-based solution lead us to choose a server/client architecture, but another requirement was that the solution should be able to work in stand-alone mode too. This means that the user should be able to use the application offline, but having the whole set of functionalities offered by the server. Therefore the approach taken for this project was to create a lightweight server for Data Storage (features and audio tracks), Music Recommendation, and Audio Streaming; and a client for Audio selection, reproduction, and preprocessing. Both can be executed on the same machine when needed, without any extra installation of software.
The server is implemented with an embeddable Web Server and Database (to replace the current application's xml storage), which means that it's portable and doesn't require any special configuration. The music and album art pictures are not stored in the database. They are loaded directly from Hard Disk. Finally, the server exposes a Web Service that allows the client to gather information about the tracks and segments, and to upload configuration data.

We implemented a basic three tier architecture, where the Data Access Layer (DAL) and the Business Logic Layer (BLL) are located on the Server, and the Presentation Layer (PL) that includes the GUI is handled by the Client. In figure 4.3 we can see how the DAL includes the persistence layer that connects to the database. The BLL can also be seen and it includes the core and the audio transcoding units. Audio is loaded from the file system of the computer. It also loads the feature processing plugins. Finally the GUI is in a different layer and it loads the graphical components for feature interaction and the decoding code. The dotted lines in the figure indicate pluggable components that may be modified without affecting the whole solution.

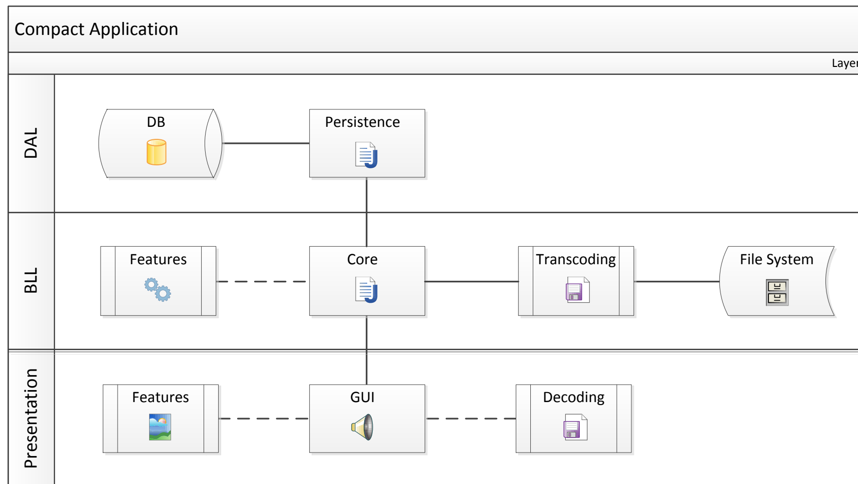


Figure 4.3: Three Tier Architecture

4.4 Software

The software used in the project includes the following applications or libraries:

4.4.1 Java [1]

Figure 4.4: Java Logo

Java is a programming language originally created by Sun Microsystems (now owned by Oracle). It uses a syntax similar to C and C++. Execution of Java code is done within a "sand box" environment: the compiled code is executed by a software called Java Virtual Machine (JVM) which runs, controls and monitors the code, acting as an interface that gives access to the low level Operative System's resources according to a predefined security structure. This is partly done to avoid instability problems (e.g. blue screens of death in Windows) caused by applications that access private memory or physical devices using incorrect drivers. This also allows the code to be executed on different devices with different hardware and operative systems, since only the JVM needs to be modified and ported.

[1] Oracle Corporation, http://www.java.net

Java's features can be extended with libraries. Some are already included in the official distribution as part of the Java Runtime Environment (JRE), like the ones required to play audio and read/write files. Others must be downloaded and included in the application's Classpath (a parameter that indicates the physical path were each application should look for classes on startup). There are also more advanced components that are included within the Java Development Kit (JDK), a set of components and tools like compilers, consoles, debuggers, etc.

Since Java is an Object-Oriented programming language, code is written by creating classes associated to a particular namespace or package. This namespace simplifies the access to the classes' methods and define a logical grouping of functionalities. When compiled this code is exported to a compressed file with either the .jar, .ear, or .war extension according to its purpose.

4.4.2 NetBeans [2]

Figure 4.5: NetBeans Logo

The whole implementation was done in Java using NetBeans version 7.0.1. NetBeans is not only a Java IDE, but also an application platform that enables developers to create applications for the web, desktop, and mobile market. The latest version supports the Java SE 7 specification with JDK 7 language features. The platform also provides enhanced integration with Oracle databases and with GlassFish[3], an open source Web Server. NetBeans also provides an easy way to do Swing GUI development, which translates into richer Java components for the graphical user interface of the application.
Since one of the requests made was to modularize the previous application, we chose to use a project of type "NetBeans Platform Application". A "Module Suite" might have been equally useful, but the platform application includes some code and libraries that makes it easier to create a fully-fledged solution.
The code was made using Java 1.6 Source Level. In total, five platform applications were created:

- Server Application, containing the DAL, BLL, and Feature plugin loading code.

[2]Oracle Corporation, http://netbeans.org
[3]Oracle Corporation, http://glassfish.java.net/

- Client Application, containing the playlist GUI and the Feature Component/Audio Decoding plugin code.
- Feature Plugins Suite, where all the plugins are grouped and compiled.
- Feature Components Plugin Suite, where all the component plugins are stored and compiled.
- Decoder Plugin Suite, where one decoder plugin is stored (may eventually contain more if needed).

The last three could have been implemented as independent Java Projects, but keeping all the similar ones under the same suite saves a lot of time when configuring common libraries and settings.

4.4.3 Apache Derby [4]

Figure 4.6: Apache Derby Logo

Since our application has to be executed even on a standalone client, we need an embeddable database, robust but portable. There were two clear options: JavaDB or Derby. Actually JavaDB is a ramification of the Derby project that was made available with the JDK. Sadly, this branch is not updated as much as the original Derby project, and hence we decided to go directly with Derby.

Apache Derby is an open source relational database implemented entirely in Java, with a very small footprint, which means that it's highly portable. It can also be configured as a server, but in this case we needed to use the embedded mode to avoid extra configurations.

To access the database we used a persistence unit (JPA 2.0 with EclipseLink[5]) on NetBeans that handles the connection pool, the authentication, and the queries, so we don't have to write any Derby specific code or stored procedure. In this way we completely isolate the database from the code, allowing for fast switching and migration to different databases when needed. In fact, if the server is to be deployed on its own machine to do heavy operations, it would be recommended to switch the database to a professional level product, like Oracle[6], MS SQL[7], DB2[8], or even MySQL[9].

[4] Apache Software Foundation, http://db.apache.org/derby
[5] The Eclipse Foundation, http://www.eclipse.org/eclipselink/
[6] Oracle Corporation, http://www.oracle.com/us/products/database/index.html
[7] Microsoft Corporation, http://www.microsoft.com/sqlserver
[8] International Business Machines Corp., http://www.ibm.com/db2
[9] Oracle Corporation, http://www.mysql.com/

4.4.4 FFmpeg [10]

Figure 4.7: FFmpeg Logo

Since our application must be able to load any of the songs available on the user's music collection, it needs to have a transcoding engine able to recognize different formats. In a "server only" solution it would be possible to keep all the music stored in a single format to avoid some of the transcoding complexity, or maybe even acquire licensed transcoding libraries (which can be very expensive) for a particular subset of formats. But as one of our requirements implied, this solution must work even on a standalone client, and we can't ask a user to convert his whole music collection to the format we choose, since that would be unrealistic and troublesome at the least. We must also realize that most music collections are still stored in MP3 format, which is a licensed format. This means that to be able to do streaming of MP3 music, royalties have to be paid to Thomson[11], which is something we obviously can't afford. Therefore we decided to find a single transcoding unit that could handle most formats and gives us as an output whatever format we decided.

FFmpeg is free (depending on the codecs configured on compile time) cross-platform software, that uses the libavcodec library to decode and encode audio and video. It's kept updated thanks to a big community of developers, which means that it's constantly improved and fixed.

In "client mode" we use FFmpeg's command line to decode the original track, getting a WAV audio file (PCM 16 bit LE) that can be passed directly to Java's audio line. In "server mode" we use it to transcode from the original format to Ogg Vorbis[12], which is a lossy patent free format that gives us a smaller file that can be easily streamed to the client.

The server executes the FFmpeg command line using Java's ProcessBuilder to generate a temporal file that gets streamed when requested by the client. We also attempted to use JNI[13] and JNA[14], Java libraries that enable applications to load native libraries (like the ones generated with C++), in an attempt to access directly FFmpeg's libavcodec, something that would yield more efficiency on the transcoding process since we could do everything in memory instead of doing expensive I/O writes and reads. In reality, the stability of FFmpeg's libraries development makes it unreliable to code against (at least in Java), and was thus discarded as an option.

[10] Multiple developers, http://ffmpeg.org

[11] Thomson, http://mp3licensing.com/royalty/software.html

[12] Xiph.Org., http://www.vorbis.com/

[13] Oracle Corp., http://java.sun.com/javase/7/docs/technotes/guides/jni/index.html

[14] Twall, http://jna.java.net

4.4.5 Java Simple Plugin Framework [15]

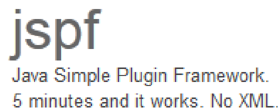


Figure 4.8: JSPF Logo

Another requisite insisted on making it easy to add new features to the solution. This implied creating disconnected code for the similarity functions and cleaning all the code from possible references to particular feature types. In fact, the new application references the features only through interfaces, passing objects based on the entities generated from the database. Jspf allowed us to define an interface, both in the server and in the client, which is consumed by the plugin suites. Those suites implement classes that inherit from the plugin class exposed by jspf and follow the rules defined in the interfaces. This way we can always rely on having a fixed set of methods available on each and every plugin, which is read and stored on memory on run time.

4.4.6 Jetty [16]

Figure 4.9: Jetty Logo

We evaluated many different Web Servers when designing the architecture. The server had to be embeddable and portable, according to our needs. There are many different offers available, some including support for EJBs and .war deploy files. We needed something that could handle web services and web streaming, and hopefully small enough to be included in an installation file.

NetBeans, depending on the installation pack, can automatically install a Web Server: GlassFish. GlassFish is a very capable server, with many advanced options and native integration with NetBeans. It can also be used in embedded mode. However, even in embedded mode, the jar files required for proper execution of the server occupy between 20 and 30 MBs of disk space, which is too big compared to the rest of the solution items. On the other hand, we also evaluated the Tiny Java Web Server (TJWS)[17]. With a really small footprint (100 - 200 KB) and support for servlets and .war files, it can deal with average server tasks. Unfortunately we didn't succeed on making it work with our current solution, meaning that we couldn't start it up directly from code, which is what we wanted.

At the end we opted for Jetty, a server that offered everything we needed, with an acceptable footprint (< 2 MB), with a simple API and with an

[15] R. Biedert, N. Delsaux, and T. Lottermann, http://code.google.com/p/jspf/

[16] The Eclipse Foundation and The Codehaus, http://jetty.codehaus.org/jetty/

[17] Dmitriy Rogatkin, http://tjws.sourceforge.net

optional XML based configuration file system. We use Jetty to publish a Web Service on the server machine, accessed on the clients with Java Architecture for XML Binding (JAXB), and two streaming servlets (without EJBs) that the client has to call alternating the order each time (for buffering reasons). The port used can be changed by modifying the included XML file.

4.4.7 JOrbis [18]

When working in "client mode" there is no need to compress the data that the client gets from the server since they are located on the same machine (or in the same LAN if needed). For this configuration, the server gives the client uncompressed PCM streams, since the bandwidth cost is irrelevant but the quality is expected to be high.
In "server mode" the client will receive audio encoded in Ogg Vorbis format (the output format can be easily modified in the settings file) since it gives us at least 10x compression (at 64 Kbps) without much of a noticeable loss in quality. The client uses the plugin interface to load a decoder when first run. For this case we decided to use the decoding library JOrbis to generate a proper PCM stream that the client software can consume normally.
JOrbis provides the tools to decode the audio, but it won't directly generate an output file. It's the responsibility of the coder to convert the output into something that Java's audio system will understand and reproduce.

4.4.8 Database Schema

The original application used an xml-based approach for data storage. As a solution grows and gains users, it starts storing more data, and there's a point in which loading xml files and then parsing them may become a bottle neck for the system's performance. When designing a software solution, it's important to foresee this possibility and to use a more robust system for data storage. The best option in this case is to use a database since it's optimized to handle large amounts of information, with efficient indexing and optimized querying capabilities that give us the best performance independent of the amount of data consumed by the system.
As can be seen in figure 4.10, the database is organized in eleven tables:

[18] JCraft Inc., http://www.jcraft.com/jorbis/

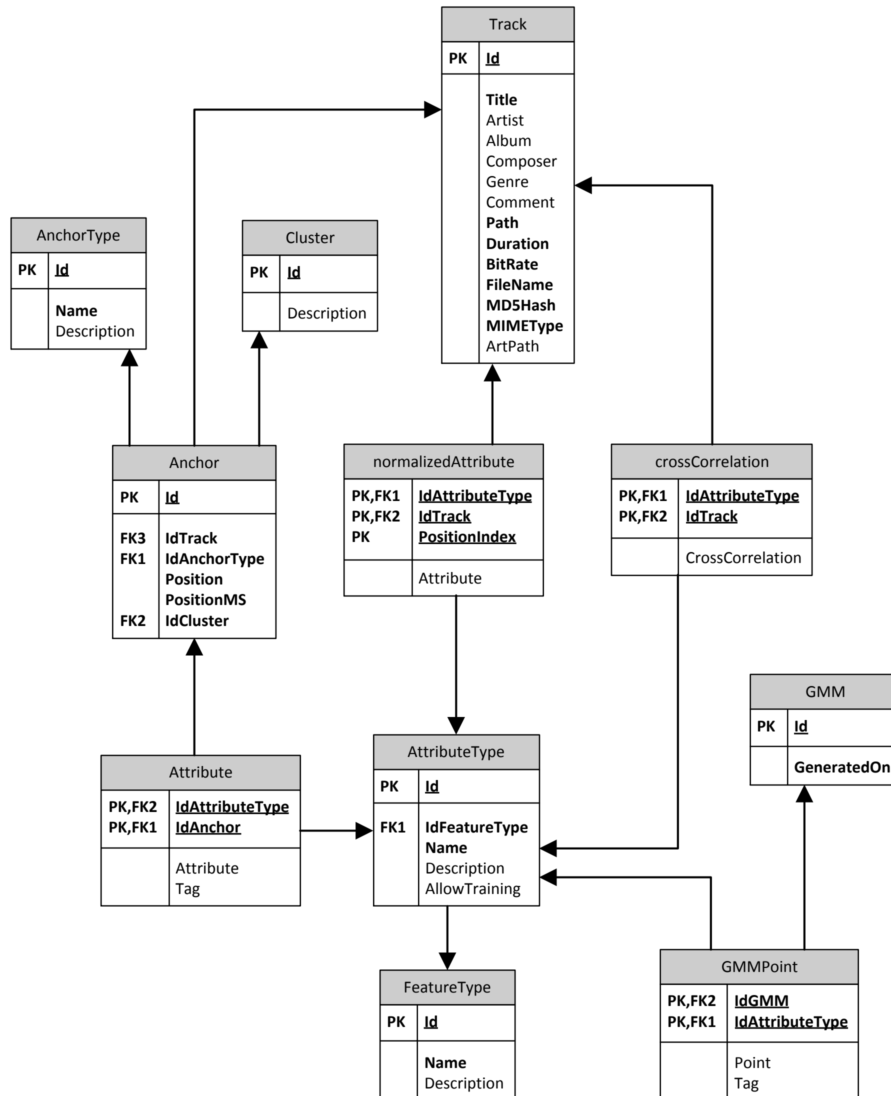


Figure 4.10: Database Schema

4.4.9 Track

The Track table contains all the typical information pertaining to a song: artist, album, genre, composer, etc. It also contains the data we need to be able to use the track on the system: the path of the file, the file's name, the duration of the song in milliseconds, and the location of the Album Art picture (if available).

4.4.10 Anchor Type

The system has been designed considering two types of segmentation for a song: variable anchors and fixed anchors. Variable anchors are defined according to the analysis of the intra-homogeneity of the track. The more homogeneous the song, the smaller the number of anchors (see section 3.1). Fixed anchors, as the name indicates, define equally long segments all along the track. They are used to define the feature sets used to calculate the similarity of songs using the cross-correlation (see section 3.2.2). Variable anchors are meant to be evaluated separately, hence rendering the system able to give recommendations on a segment level. Fixed anchors are used as vectors or series of points, equally distanced from each other, that are eventually used to compare the whole track instead of the segments with all the other tracks on the database using cross-correlation (see section 3.2.2). This table contains information for both anchor types with a small description of each one.

4.4.11 Anchor

Anchors indicate the position of a new segment in the track. The table contains: the id of the track, the id of the anchor type, the position in samples, and the position in milliseconds.

4.4.12 Feature Type

In this table the system stores all the features that shall be made available to the client. It's simply defined by a name and a short description.

4.4.13 Attribute Type

A feature may be composed of one or more, different measures. For example, tempo is measured as BPMs, brightness is simply brightness, but mood is represented with a key, a mode, and a key clarity. We defined all these values as attribute types, a subset of the feature types. The table contains all the same fields of the Feature Types table, plus the Feature Type's id.

4.4.14 Attribute

What we call an attribute is simply the feature's value. It's defined by an attribute type id and an anchor's id. The value was defined as a decimal (length 24, precision 20), but since the system in the future may have to support string values, a "tag" field was also included.

4.4.15 Normalized Attribute

Since we need to compare whole feature sets using the cross-correlation, we need vectors of the same length. This means that we need to find the longest vector size, and resize all the other vectors to have the same number of points through interpolation. This is an expensive process, thus we decided to create this table that gets loaded the first time the vector comparison algorithm is executed. Next time the process is launched, if there haven't been done any modifications to the saved features, these pre-calculated points can be loaded directly from the database, improving performance. If a new feature set is added to the database, and it has more anchors than the previous maxima, the algorithm reconstructs the whole normalized set to match the new vector length.

4.4.16 Cross-correlation

Once we have the normalized attributes, we still have to do the cross-correlation between every single feature set in the database (for the "fixed" anchor type). Since we don't want to be doing this every single time a recommendation needs to be generated, we also store the results on the database. As we said before cross-correlation is not commutative, which means that we have to calculate and store the result for $x \star y$ and $y \star x$. It uses a small portion of hard disk space, but it can save a lot of processing time, especially when the user is not searching for particular features or feature evolutions.

4.4.17 GMM

The original application defined the basis for a system to classify user choices and improve the sorting of the playlist recommendations. This table is created for legacy reasons, as it replaces the current xml file used to store the GMM point info.

4.4.18 GMM Point

The previous table is basically an index table, while this one contains the actual training points. The only difference with the previous xml schema is the inclusion of a tag field, emulating the modification done to the attributes to include string data.

4.4.19 Cluster

In the case we need to improve the performance of the system, a non-labeling clustering algorithm would have to be implemented to avoid looping over the whole database with each recommendation. For this reason a cluster table was created and some extra fields were added to the Anchor and Attribute Type tables: Anchors have a Cluster Id, and Attribute Types have an "allow training" field, which tells the system if the type contains data that should be used to classify the values.

Chapter 5

Implementation

Due to the complexity of the system and to the requirements defined at the beginning of the project (section 4.2), the software was designed as a server/client solution. The server role is responsible for executing most of the memory and processor consuming tasks (playlist generation, audio streaming, web service, etc.), therefore it was very important to keep performance under control while developing it. The client is responsible for audio reproduction, song searching, feature value settings, feature weighting, and special effects.

On this chapter we explain the programming technicalities behind the application, according to what was already explained on the design chapter, we'll explain the workflow of the software, and then we'll define how the server and the client work in detail.

5.1 User interaction

Before we explain how the system works, it's good to remember what a user can do with the software. First, the graphical interface was created trying to imitate the original program, with the playlist on the left and the options on the right (see figure 4.2). Using this interface the most basic things that the user can do are: search for songs to play, choose segments or songs to reproduce, and pause or stop the reproduction. The advanced functionalities include: choosing the type of recommendation to use (by piece, or for the full track), defining fixed feature values to modify the output of the recommendation system or to search for songs (or segments), changing the weights that tell the system how relevant a single feature should be for the

recommendation algorithm, and switching between reproduction styles and effects (e.g. continuous playing, beat-matching).

5.2 System startup

To begin using the system, the server needs to be executed before the client. The reason for this is that the client needs to fetch important configuration data from the server on startup.

5.2.1 Server startup

The first thing the server does when started, as can be seen on the top of figure 5.1, is to load all the preferences stored on the configuration file. This file is created automatically the first time the application is run, and its administration is done automatically using Java's Preferences class and NetBeans' NbPreferences class, called using a class called settings.java. The configuration file is created on a folder within the application's root while debugging, or within the user's personal preferences folder when run normally. Within this file the application can find different settings, like the path of the FFmpeg program, a flag indicating if the server should encode it's output, the encoding preferences, the maximum number of recommendations that can be done in one call, the path of the plugins, and some of the error and information messages that can be passed to the client.

The next thing the server does is to read all the feature types available on the database. After this step, the system loads the feature plugins, .jar files that contain the code necessary to find the similarity between features of a particular type. We discard the classes that correspond to feature types not available on the database and store the rest on a shared dictionary. Next, the server loads all the attributes available on the database and creates a series of dictionaries that will be used later to calculate the similarities between features.

Since we have to calculate the cross-correlation between features, it would be better to do it offline. However, for practical effects this process can be done on startup, after checking if the values stored on the database need to be updated or not. This means that if a new song is added to the collection, the process has to be run for the missing items. Also, if the number of anchors in one of the new songs surpasses the previous maximum, a

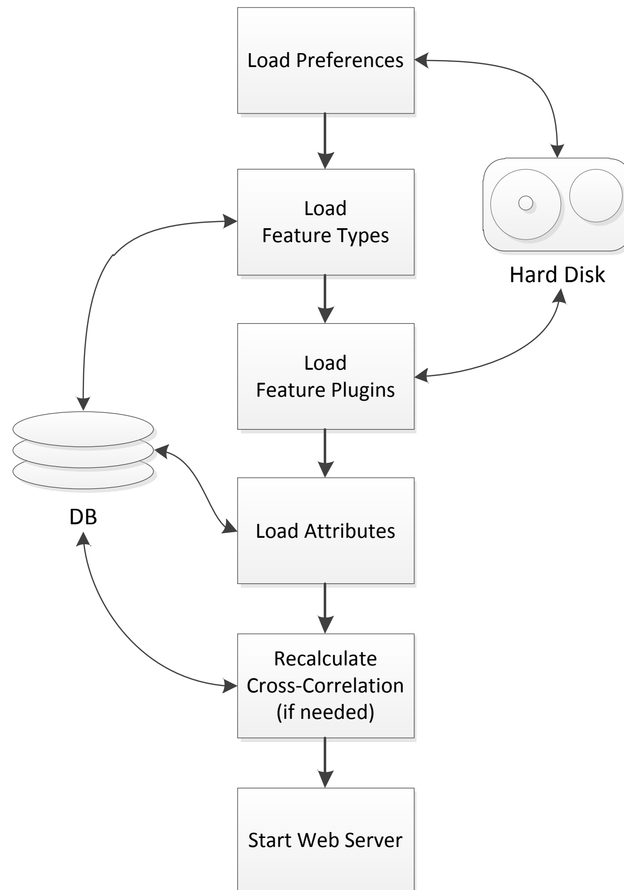


Figure 5.1: Steps of the server's startup

process of re-interpolation will have to be executed before calculating the cross-correlation. More details on this topic can be found on section 5.3.4.

Finally the web server is started. Jetty loads its own configuration file and publishes the web service and stream servlets. The web service is responsible of handling all the non streaming related interaction between the client and the server. This includes queries to the database, uploading and downloading of settings, requesting audio pieces (but not downloading them), and eventually shutting down the web server. The streaming servlets do the streaming when called by the client, passing as a result the piece that was

requested suing the web service.

It's important to add that we implemented the client using a singleton pattern, which means keeping only one version of an item in memory, instead of constantly recreating it as needed. This improves performance and extends the scope of the variables, making them easier to use.

5.2.2 Client startup

When the client is initialized, the first thing it does is to ask the server about the anchor types available on the database (using the web service). This is shown in figure 5.2. Then it asks for the available feature types. This is important since the client must render controls that depend on this information to give the corresponding options to the user. After the server responds, the client proceeds with the loading of the settings. The configuration file of the client is different from the configuration file of the server and are stored in different paths.

Once the application has finished loading and the rendering of the UI is complete, the plugin manager loads the feature components that allow the user to interact with the feature values, and the decoder that shall be used when the system is working on "server mode" (please refer to section 4.3.2). The feature components are then added to the options tab panel, and the decoder is stored on a shared variable.

5.3 Main operation

The client application needs to communicate with the server application to ask for a piece of audio and then to stream it. The main token we use to do this communication is the database entity "anchor". Anchors, as seen on section 4.4.11, indicate the position of the different segments of a song. It has a track and a type and is a small element that we can send and receive without spending too much bandwidth. Anchors can be received as a result of two processes: searching for a song or as a recommendation after choosing a segment of audio to play.

We now proceed to explain the workflow of this communication, as seen on figure 5.3, step by step:

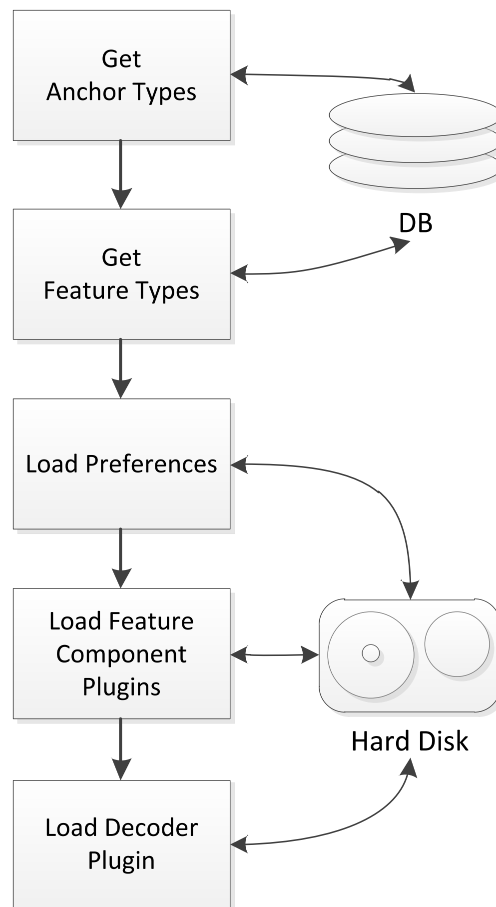


Figure 5.2: Steps of the client's startup

5.3.1 Searching for a song

To start using the application the user has to choose an initial song. The original program would load a random song on startup, but we decided to change this in our project because of possible bandwidth issues. In our case, the client can search by title, artist, or album using text. An additional option is to perform a search using feature values directly, accessed through the feature components that were loaded on startup.

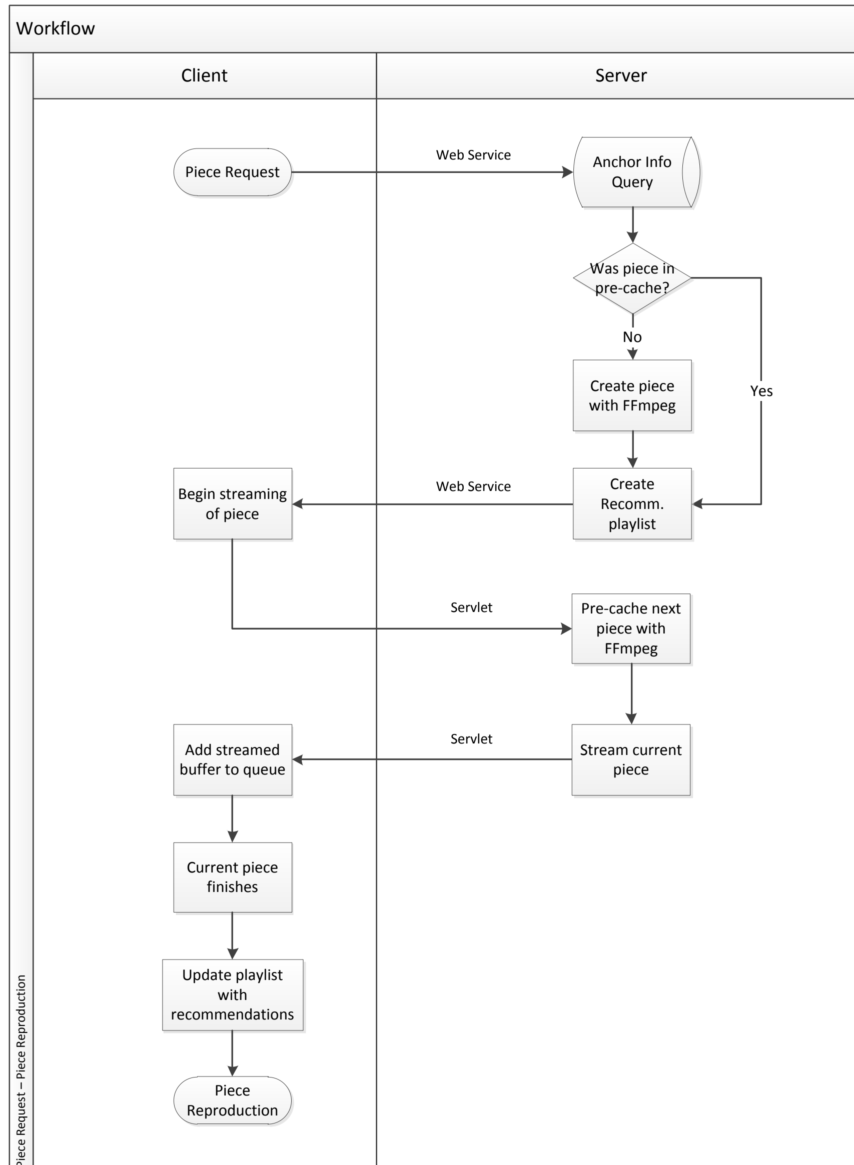


Figure 5.3: The system's workflow

The server will always respond with a list of anchors (elements from the database) that can then be chosen and played. It doesn't give us tracks directly, but anchors are always logically connected to tracks once they are retrieved from the database. This means that once the list of anchors arrives,

the software can generate a list of the tracks corresponding to the imported anchors. Then, with another call to the web service, we can get all the other anchors that correspond to each track. This method is also used to create the recommendation playlist, as will be explained in due time.

Once a result list is obtained, the user can make a choice between any of the different tracks available. This is a big modification from the way the original application worked. The PolySound application would only allow the user to play recommended segments. We decided to give more freedom to the user, allowing him to pick any segment, even if it wasn't recommended by the server.

5.3.2 Choosing an audio segment

When the client application renders the playlist, the user can choose any piece of audio he wishes by clicking on it with the mouse. At this point, the client calls the web service's method "playPiece", passing as parameters the id of the anchor, the current feature weights, and the feature values fixed by the user. When the server receives this information, it reads the anchor data from the database and launches two process in parallel: creation of the temporal file corresponding to the chosen piece and the generation of a recommendation list that fits it.

5.3.3 Temporal files

The server uses FFmpeg to create the temporal files that get streamed to the client. We keep two configurations stored in the settings file: one for uncompressed audio (client mode) and one for compressed audio (server mode). The format we use for uncompressed audio is 16 bit LE PCM at 44 Khz, this gives us good quality audio at the price of having big temporal files. The compressed format we use is Ogg Vorbis with a quality of 64 Kbps, which gives us a quality similar to that of an MP3 file at 96 or 128 Kbps, but with a smaller size.

5.3.4 Recommendation generation

Since our system doesn't use clustering techniques, the recommendation algorithm is implemented as a loop that compares anchors or tracks, feature

by feature. If the user has fixed a feature value using the GUI, that value replaces the one native to the anchor. If the process is done segment by segment ($anchorType = 1$), just like the original application did, the anchor's features are passed as parameters to the corresponding feature plugin's similarity function (see section 3.2.1), in a process that gives us a normalized decimal value as a response: 0 means no similarity, 1 means a perfect match.

When the comparison is done track by track ($anchorType = 2$), the system creates one vector for each feature, and loads it with the features that correspond to each anchor of the track. This vectors are then normalized in length using interpolation. The resulting normalized vectors are then cross-correlated with all the other vectors of the same feature type. This cross-correlation is done using the same similarity functions used to compare the single segments, but give us a match corresponding to the evolution of the features on the song. To improve performance, these results are stored on the database so the system doesn't have to recalculate them every time.

In both cases, once we have the similarity for each feature we need to evaluate its relevance according to the value of the weight that the user has chosen. If the user chose a weight of zero, the feature must not be taken into account. This is calculated by multiplying the similarity by the weight. The final average similarity is then calculated as the sum of the weighted similarities, divided by the number of features:

$$S_{avg} = \frac{\sum_{i=1}^{n} S_i * W_i}{n} \tag{5.1}$$

To avoid recommending songs with a very low similarity average, the system uses a threshold (loaded from the settings) to decide which items are worthy of being sent to the client as part of the playlist. The resulting list of anchors is first sorted according to the GMM training data stored on the database (see section 3.3.1), and then it is sent as a response to the client, so it can be rendered and displayed to the user.

5.3.5 Streaming

The generated playlist is received and stored by the client. Immediately after, the application calls one of two streaming URLs on the server to get the next audio segment to reproduce. The reason to keep two of them was

to avoid problems with simultaneous access, and to be able to swap segments easily during execution. When the server receives the call it loads the correct piece from hard disk, and feeds it to the client with a header that indicates if the content belongs to a .wav file or a .ogg file, and the size of the file. Once the whole file has been sent to the client, the server launches a new thread to create the temporal file corresponding to the next segment of the song. By doing this we save some time when the client asks for the next piece, since the web service won't have to create it and can focus on generating the playlist.

When the client starts receiving the audio file it follows the workflow shown in figure 5.4. If the file is in Ogg format, it gets decoded by the client to obtain a PCM stream. Once we have a PCM stream, either streamed or decoded, the application splits the segment in 3 smaller pieces just like in figure 5.5. The purpose of this is to store 3 different buffers that will allow the application to do special effects, like cross-fading or beat-matching. As soon as one of the smaller pieces is available it gets stored on a shared "BlockingQueue", a Java object that stores items on a FIFO (First In, First Out) queue that's thread safe, so we can store new pieces at the same time that the application is retrieving older items.

When the user chooses a different song, the system needs to do cross-fading between segments. This process is done between the initial piece of the next anchor, and the final piece of the current one, which has been stored in a fourth shared buffer. The result replaces both pieces in the queue and allows for a smooth transition.

5.3.6 Reproduction

The client contains a class for reproducing the audio found on the shared queue that runs on its own processor thread. This thread keeps reading the queue and sending the audio to Java's SourceDataLine so it can be played by the soundcard. This thread keeps running until the queue is empty or until it's killed by the main class. Once it has played the set of pieces corresponding to a full segment, it tells the main class that it's finished so the GUI can be updated and the recommendation playlist that was stored before can be displayed to the user.

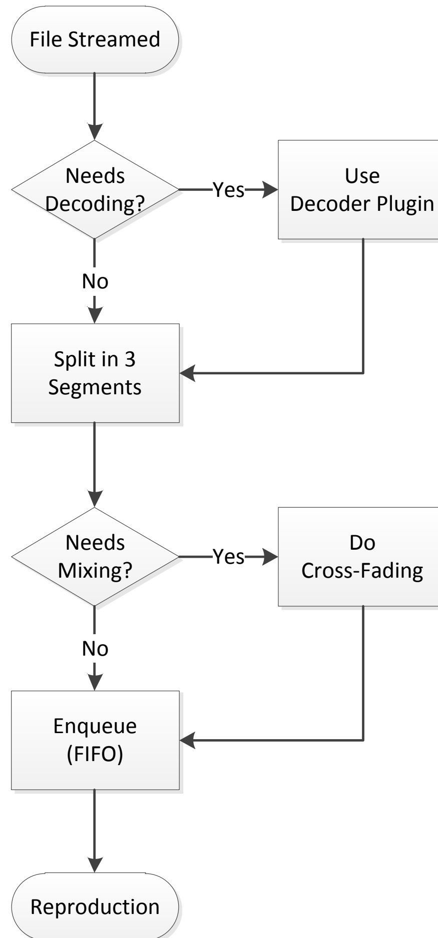


Figure 5.4: The client's streaming workflow

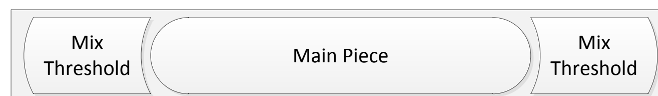
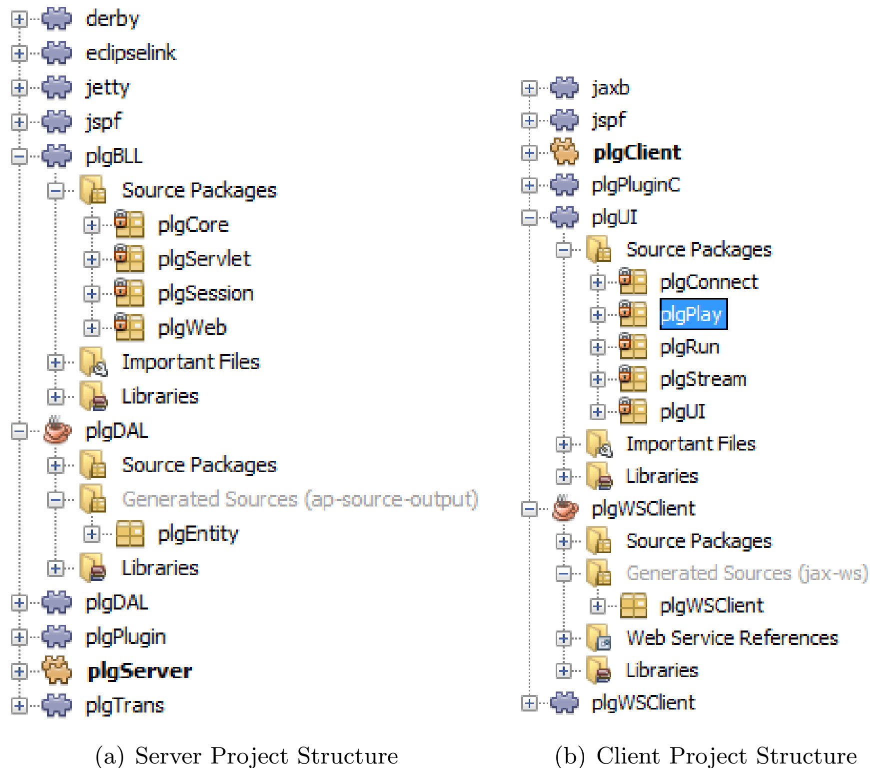


Figure 5.5: The streamed segment split in three pieces

5.3.7 Autoselection

To keep the system always running, even in the absence of the user's intervention, there's a timer that gets enabled as soon as the reproduction of a segment is started. As soon as the timer reaches a predetermined number of seconds (loaded from the settings), the system automatically blocks the playlist and chooses the next piece of the track by calling the web service's "playPiece" method. In this way we try to guarantee that the client has enough time to download the next piece of the song, avoiding a silent gap in the reproduction.

5.4 NetBeans



(a) Server Project Structure (b) Client Project Structure

The workflows and processes explained above were implemented entirely on NetBeans. The project distribution can be seen on figures 5.6(a) and 5.6(b). The server contains references to the derby (database), eclipselink (persistence), jetty (web server), and jspf (plugins) libraries described on section 4.4. The module plgTrans contains the code that loads the FFmpeg command line and does the file transcoding. Then we can see the Data

Access Layer (DAL) called plgDAL, a library responsible of connecting to the database using entity classes that represent its tables and relationships. plgPlugin contains the interfaces that the feature plugins must implement to be loaded by the application. Finally, there's the Business Logic Layer (BLL) responsible for all the main processing of the server. It is distributed in four packages: plgCore, where resides the code that creates the recommendation playlists; plgServlet, where the streaming pages (servlets) are kept; plgSession handles all the queries to the DAL and contains the facades that expose the CRUD (Create, Read, Update, Delete) functionality; plgWeb contains the installer class that handles the startup of the server, the settings handler, a small utilities class, and the web service code.

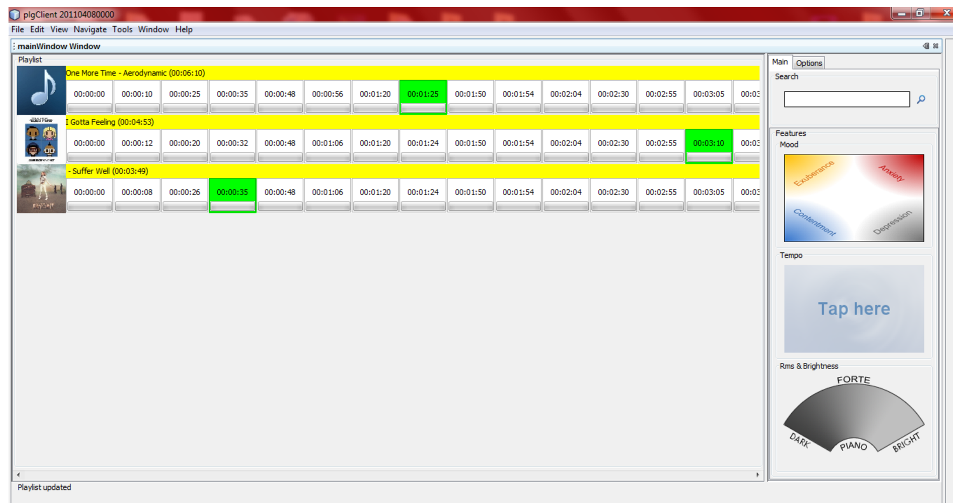


Figure 5.6: An early implementation of the GUI

The client project contains references to the jaxb (web service client) and jspf (plugins) libraries. There is a library called plgWSClient that contains the code necessary to access the server's web service. plgPluginC contains the interfaces for the plugin components and the decoder. plgUI is the main module and it contains several packages: plgConnect contains all the classes that access the web service through plgWSClient calls; plgPlay is responsible for playing the music and detecting the audio format of the streamed audio, telling the application if there's need of using the decoder plugin; plgRun, just like plgWeb in the server, is in charge of handling the startup of the client with an installer class that also stores most of the singleton objects that are used across the program, keeping settings synced, and offering a few

simple utilities; plgStream is the class that downloads the audio pieces and enqueues them for the classes on plgPlay; at last we have plgUI, containing all windows, panels, and controls that conform the graphical interface of the application.

The GUI is done using Java's swing components, plus some custom components, including modified versions of the feature components offered by the original PolySound software. A snapshot of an alpha version of the interface can be seen on figure 5.6, where it can be seen that it aims at keeping the same "look and feel" of the original one.

Chapter 6

Conclusions and future developments

6.1 Conclusions

This work proposes an extension on an existing music recommendation application adding code modularity, "over the web" functionalities, pluggable components, support for different clients, and a new recommendation approach based on the feature evolution of a track over time. The final result is a server/client architecture implemented as a pair of NetBeans Platform Applications, and three NetBeans Module Suites for plugins.
The two applications can be executed on the same machine if needed, but offer better performance when used on different machines. The communication between them is done using a web service and the audio pieces are downloaded by the client as a binary stream. Audio is transcoded using FFmpeg using temporal files that are stored on the server's hard disk. The client can decode the audio using a library contained in an exchangeable plugin. The recommendations can be based on the similarity of the features belonging to an audio segment ("variable anchor"), or the feature set of the whole song ("fixed anchor").

Audio decoding performed on the client side can be troublesome if the audio segments are too small (< 4s), because the time it takes to convert the next piece into PCM data may be longer than the time it takes to finish the current piece. This can generate silent gaps in the audio output. The system however shouldn't work with pieces this small, so this problem can be neglected. However, long pieces may also cause problems as the encoded

piece needs to be streamed completely before being split into the 3 buffers used for mixing (see section 5.3.5). One approach to solve this problem is explained in section **??**.

The web features of the server (web services and streaming) were not implemented using Enterprise Java Beans (EJB), a Java technology used for enterprise applications. The code, however, was made to be compatible with EJB and can be easily modified by adding the corresponding metadata. The use of plugins works like expected and makes it much simpler to modify the components without worrying about breaking the functionality of the application. Plugins must always comply with the provided interfaces before they can be used on the system, which means that they must follow the rules defined by the application. Plugins are also free to use their own libraries and resources, keeping them independent of the resources or libraries used by the main software.

6.2 Future Developments

There are many ways in which this solution could be improved in the future, that weren't implemented because they were either out of the project's scope, or they contradicted the specified requirements. Also, future developments may take advantage of the new client/server architecture, since it opens up new possibilities to programmers. Some of the most important improvements that could be developed in the future are listed bellow.

Using the libavcodec library directly, instead of the FFmpeg command line for audio transcoding, would greatly improve the overall performance of the system. As was explained on section 4.4.4, FFmpeg uses libavcodec to perform encoding and decoding of music. Using this library directly would be faster than using FFmpeg. The inconvenience is that the libavcodec is a native library and is meant to be used from low level applications using C++ language. It is only possible to reference this library in Java using interfaces like JNI and JNA, but this requires advanced C++ knowledge and familiarity with the FFmpeg code.

The time it takes the system to create a playlist will increase as the number of songs in the collection grow. Eventually this may cause timeout errors because the server will take too long to respond. Also the memory usage of the server could increase beyond Java's supported limits for the same reason. This are technical problems that may arise when the num-

ber of songs increases to a number not foreseen during the design of this project, and may be a result of the hardware (physical memory) and software (Java) used. Within acceptable values (hundreds of songs) this is not a risk, and the application behaves correctly. To be able to use a bigger music collection, the recommendation algorithm will have to be revised and tested.

One of the biggest bottlenecks of the application is the need to keep and fill many different buffers to be able to create special transition effects on the pieces. An improvement in performance could be achieved by passing the streamed data directly to the reproduction system (see section 5.3.6), instead of the enqueueing system, and doing a low level mixing process directly on the reproduction buffers (instead of our queue buffers). This would eliminate the problems with small and long pieces explained in section 6.1 since audio would be decoded on the fly. The low level mixing would probably have to be implemented using native code (C++).

The current solution was implemented for a single user. The system can be adapted to work on a multiple user environment if needed. It would require the creation of new tables on the database to store user information (name, age, location, password hash), to indicate the songs that belong to a user, and to keep personal preferences realtive to the server. The web service could accept the user id as a new parameter, and the stream servlets could also accept a user token as part of the URL to differentiate between users.

Giving the user the possibility to upload his own music to the system would turn the application into a Cloud Service (as described in section 1). The user would then be able to keep his songs on the server, and stream the audio using any supported client. This would require the availability of hard disk space on the server, and the definition of a standard format for the uploaded song files.

Another interesting extension of the program would be a feature extraction system. The current application uses features extracted offline using Matlab. Creating an application that extracts the features automatically would help the user to update his music collection, which is something that he currently can't do.

Bibliography

[1] E. Allamanche, J. Herre, O. Hellmuth, B. Fröba, T. Kastner, and M. Cremer. Content-based identification of audio material using mpeg-7 low level description. In *Proc. Int. Symposium on Music Information Retrieval (ISMIR)*, 2001.

[2] Samy Bengio. An introduction to statistical machine learning - em for gmms -, 2003. Dalle Molle Institute for Perceptual Artificial Intelligence (IDIAP).

[3] Deirdre Bolger. An exploration of timbre: its perception, analysis and representation, 2005.

[4] M. Cartwright, Z. Rafii, J. Han, and B. Pardo. Making searchable melodies: Human vs. machine. In *Proceedings of the 2011 AAAI Workshop on Human Computation, San Francisco, USA*, 2011.

[5] Clovis Chapman, Wolfgang Emmerich, Fermín Galán Márquez, Stuart Clayman, and Alex Galis. Software architecture definition for on-demand cloud provisioning. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, Chicago (USA)*, 2010.

[6] Luca Chiarandini, Massimiliano Zanoni, and Augusto Sarti. A system for dynamic playlist generation driven by multimodal control signals and descriptors. In *IEEE International Workshop on Multimedia Signal Processing. Hangzhou (China)*, 2011.

[7] Daniel P.W. Ellis, Brian Whitman, and Alastair Porter. Echoprint - an open music identification service. In *Proc. Int. Symposium on Music Information Retrieval (ISMIR)*, 2011.

[8] G. Fazekas and M. Sandler. Ontology based information management in music production. In *126th Convention of the AES, Munich, Germany*, 2009.

[9] J. Foote. Automatic audio segmentation using a measure of audio novelty. In *IEEE International Conference on Multimedia and Expo (ICME), New York City, NY (USA)*, 2000.

[10] E. Gómez. *Tonal Description of Music Audio Signals*. PhD thesis, Universitat Pompeu Fabra, 2006.

[11] A. Huq, M. Cartwright, and B. Pardo. Crowdsourcing a real-world online query by humming system. In *Proceedings of the SMC 2010 - 7th Sound and Music Computing Conference, Barcelona, Spain*, 2010.

[12] Krumhansl. Cognitive foundations of musical pitch. *Oxford UP*, 1990.

[13] Cyril Laurier and Perfecto Herrera. Mood cloud: A real-time music mood visualization tool, 2008.

[14] D. Little, D. Raffensperger, and B. Pardo. Online training of a music search engine. Technical report, Northwestern University, Evanston, IL, 2007.

[15] D. Little, D. Raffensperger, and B. Pardo. A query by humming system that learns from experience. In *Proceedings of the 8th International Conference on Music Information Retrieval, Vienna, Austria*, 2007.

[16] D. Liu, L. Lu, and H.J. Zhang. Automatic mood detection from acoustic music data. In *International Symposium on Music Information Retrieval, Baltimore, Maryland (USA)*, 2003.

[17] Y. Raimond, S. Abdallah, M. Sandler, and F. Giasson. The music ontology, 2007.

[18] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. In *ACM SIG-COMM computer communications review*, 2009.