---

# Tomorrow's music player : A spinning interface

Aurelia Rochat

EPFL
School of Computer and Communication Sciences

## Specialization Project

January 10th, 2009

**Professor**
Pearl Pu

**Supervisor**
Nicolas Jones
EPFL
CH-1015, Lausanne

# Contents


**1 Introduction** **3**

**2 Preliminaries** **3**

**3 The LAMA Framework** **4**
3.1 Server Side . . . . . 4
3.1.1 LAMA's structure . . . . . 4
3.1.2 Interaction with Last.fm . . . . . 6
3.1.3 Implementation . . . . . 9
3.2 Client Side . . . . . 10
3.3 Communication between the client side and the server side . . . . . 10

**4 Application** **12**
4.1 Objectives . . . . . 12
4.2 Server Side . . . . . 12
4.2.1 The Extension of LAMA . . . . . 12
4.2.2 The "Pure" Application Part . . . . . 15
4.3 Client Side . . . . . 16
4.3.1 Implementation . . . . . 16
4.4 Graphical User Interface . . . . . 17
4.4.1 How The User Can Interact With The Interface . . . . . 20
4.4.2 The Tools Used . . . . . 21

**5 Encountered Problems** **21**

**6 Future Work** **22**
6.1 Extension of LAMA . . . . . 22
6.2 Application and Interface Improvement . . . . . 23

**7 Conclusion** **23**

# 1 Introduction

Nowadays an important collection of music recommender systems is available on the Internet. Among them, Pandora (`www.pandora.com`) (accessible only by the US residents), Last.fm (`http://www.last.fm/`), MyStrands (`http://www.mystrands.com/`), and many others. These platforms build a profile for the users according to their music tastes, and then propose personalized recommendations. The user profile is computed on the base of explicit data collection, and on implicit data collection.
**Explicit data collection** can be done by asking directly the user to rate songs, to rank a set of songs, to make a choice between several items, etc. **Implicit data collection** consists in analyzing the user's actions on the site page (such as : 'did he click to view the full artist biography?'), observing the items viewed by the user in an online store, etc.

The new recommender systems that emerge tend to follow the trail designed by the first ones. They do not propose really innovative interfaces. Therefore the goal of this project was to design a prototype for an original interface, and to implement different policies for suggesting songs. The application had to use an existing recommendation system's features. It seemed appropriate to use Last.fm's features, since they provide free web services and protocols [3], facilitating the developer's task.

This report presents the work that has been accomplished for this project. First a framework was implemented, in collaboration with Lucas Maystre, to do the background tasks : the communication with Last.fm, the accesses to the database, the sessions management, etc. ; then an application using the framework was built; finally a graphical interface for this application was designed. The goal of this report is mainly to expose the different choices made when designing the framework and the application. For details on the framework code, please see the wiki at `www.whizz.ch/doc`. The chapter describing LAMA (section 3) was written by Lucas and myself.

# 2 Preliminaries

We had to choose among several possibilities to implement our application. The first idea was to use the **JLFM** library, available at `https://www.ohloh.net/p/jlfm`.
JLFM is a 'Java library to simply access the radio service of Last.fm and audioscrobbler'. Using JLFM would have spared us the "low level" part of the implementation, which is the interaction with the Last.fm servers, since it is implemented by the library. This would probably have allowed us to build more complex applications. The drawback of using JLFM is that the applications implemented could not work on top of another recommendation system than Last.fm.

Another choice was to modify the Last.fm official client, which code is available at `svn://svn.audioscrobbler.net/`.
The client is written in C++, and it is cross platform. The positive aspect is that the application would have been usable from the beginning of the project. The objectives of the project would have been slightly different. However, working on the base of a full application would have required to spend time to understand the core of the application code, and no other system than Last.fm could be used with this application.

The solution that seemed the most interesting and flexible was to build our own framework, using the Last.fm API, and our own web application on the basis of this framework. The main advantage of this solution is that it can be extended to use another recommendation system than Last.fm. The application part is separated from the framework part, meaning that different applications can be built easily on top of the framework.
Another positive aspect of building our own framework is that we had to use directly the Last.fm API, without using an intermediate library like JLFM. Therefore we have a now good knowledge of what is going on between our application and Last.fm.
We were also able to decide which programming language to use for our framework.

# 3 The LAMA Framework

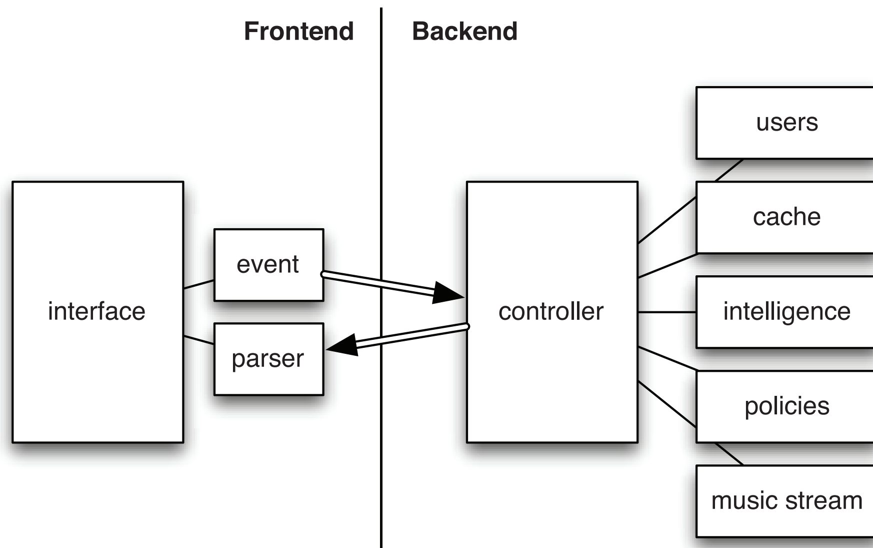


Figure 1: How LAMA works - simplified diagramm

The first part of the project consists of the design and implementation of the LAMA (Last.fm Musical Amplifier) framework. This framework allows to build applications involving a music recommendation system, using the tools provided freely by Last.fm.
The framework had to fulfill the following requirements: allow to listen to a radio channel; play the current song; display basic information about the current song; allow to continue listening to the current radio station automatically; propose alternative songs that the user must be able to choose and listen to; the alternative songs are proposed by a set of policies.

LAMA is designed in a generic manner. It should be easily extended, if, for example, one wishes to use another recommendation system than Last.fm. It is separated in several classes; the classes are grouped in folders, in a package-like fashion, as presented in the section 3.1.1.

The client part of LAMA is implemented as a *thin client*. That is, it performs as few processing as possible, leaving this task to the server side. Its roles are only to detect events in the interface, and to inform the server about these events, as well as to deal with the response received from the server, and update the interface content accordingly.

## 3.1 Server Side

### 3.1.1 LAMA's structure

In this section we will quickly describe the backend part of LAMA from the inside out. The reader is kindly requested to refer to the diagramm above, for a visual representation of what we explain here. We will cover the following subjects:

1. the document tree structure,
2. the session handling,
3. the event handling,
4. the radios handling,

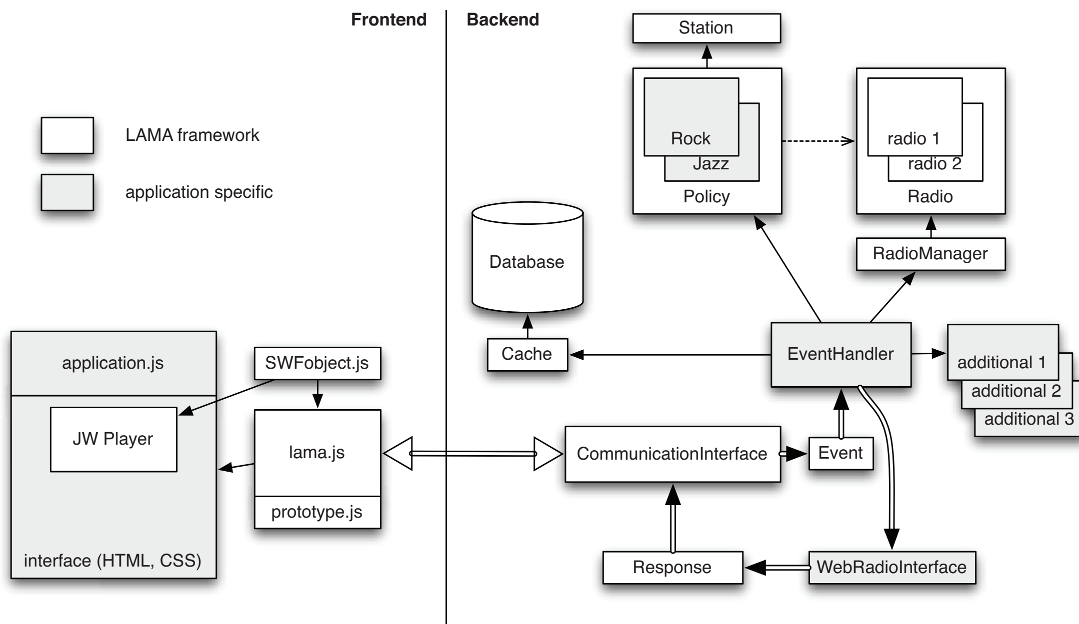


Figure 2: How LAMA works - diagramm

5. the policies,
6. and finally: the cache.

The backend of LAMA is entirely written in PHP. Therefore, we execute scripts every time a request is made to the server. We will discuss the pros / cons of this technological choice below. The files are structured into four directories, in a package-like fashion, allowing to distinguish big building blocks:

**default** contains all the files that couldn't fit elsewhere, e.g. classes that represent users, cache, database access; commonly used functions, etc.

**intelligence** contains all the scripts, classes, that play a decisional role. As these are often application dependent, there aren't many files. The classes are abstract.

**login** contains everything that's needed for a user to register and authenticate.

**stream** contains all the classes that deal with the music infrastructure. The goal of this package is to deliver music files and metadata associated to it.

The sessions are loosely handled in LAMA. The user has to register the first time he visits the application, and has to log in every time he wants to use the application. Two cookies are used, one is permanent and the other (the session cookie), identifies the session. It identifies the user uniquely. A user, when exiting the application, should click on the logout button that needs to be on every application, that launches some logout scripts and releases all the resources

allocated to the user. In case not every user behaves like this, it is necessary to set-up a cron job that will deallocate the resources for unused sessions after a given period of time.

Events are a the heart of the system. When an event is triggered on the front end, the file CommunicationInterface.php is requested. Events can be of different nature. Here is a list of typical events:

- initialization
- end of a track
- click on an element
- polling
- ...

When a call to CommunicationInterface.php is made from the frontend via AJAX, the originating event is transmitted with the request. An Event object is built from that. Another very central part of the process is the EventHandler class. The Event object is passed to an EventHandler object, that will take the appropriate actions relative to the originating event. E.g., if the originating event was the end of the track, we would like to send a new track to the client. The actions taken upon different events are of course application specific, a concrete subclass of EventHandler has to be defined in the application part. The ouput of the EventHandler object is passed to a WebRadioInterface object, which has knowledge of the representation of the interface (which div has which id, and so on), which in turn needs a Response object to format the resulting action into an XML file that can be parsed by the client. And voilà, "la boucle est bouclée" as french people say.

With Last.fm, a user can listen to a single radio (that makes sense). But the problem is that we can't show alternative songs to the user—at least not the actual metadata of the next song in the alternative channels. Therefore, LAMA uses a pool of fake users, whose radios are used by real users of the application. There's a limited number of fake users available, so there can only be so many users. These radios are stored in a specific table in the database, the table **RADIOS**, and whenever a user needs a radio, a flag is written in the table, and the radio is allocated to the user. The class LastFmRadio manages a single radio, providing functions for changing the station, getting a track, etc. The class Radiomanager handles several radios, allocates them or deallocates them following the user's need.

Now we come to the policies: the idea behind a Policy object is quite simple: get tracks that are related to a certain criterion. A Policy subclass handles Radio and Station objects to obtain songs that should correspond to a specific, intelligible topic, e.g. the songs that my closest (Last.fm) neighbour loved. They are of course application dependent, even if it would be interesting to build a big Policy library usable by any application.

The last component we want to talk about is the cache. It is needed, because the requests to the backend are independent. So we have to store the current state before sending the response and terminating the scripts. The class Cache offers a simple way to serialize objects and to store them in the database. Cache entries are local to the user (no global cache can be using Cache—but until now there was no need to have application-wide repercussions of the user requests. One interesting feature, though, would be to influence the recommendations of one user with the help of what happens to the other users). The cache works like a map, or an associative array, with (key, value) entries.

### 3.1.2 Interaction with Last.fm

The LAMA framework interacts with Last.fm servers in various contexts.

During the LAMA registration phase, the user is redirected on a specific Last.fm page, where it is recognized thanks to its Last.fm cookies. If the user owns no Last.fm cookies, it is redirected to the Last.fm login page. Once logged in, the user should be asked to allow LAMA to use its account, as described in [2]. However, the user arrives on its own Last.fm home page; therefore he has to log in LAMA once more to complete the authorization process. By clicking on a link on the authorization page, the user allows LAMA to access its Last.fm account and data.
Afterwards, the user is authenticated according to **last.fm authentication protocol** [2]. Last.fm authentication scheme provides the client with a session key that is used to sign calls to Last.fm API. The API includes functions that require the client to be authenticated. The following figure illustrates Last.fm authentication protocol:

(1) *Client* → *Server* : GET http://www.last.fm/api/auth/?api_key=xxxxxxxxxxx
(2) *Server* → *Client* : <callback_url>/?token=xxxxxxx
(3) *Client* → *Server* : GET http://ws.audioscrobbler.com/2.0/?method=
auth.getsession&api_key=xxxxxxxxxxx
&api_sig=xxxxxxxxxxx&token=xxxxxxxxxxx
(4) *Server* → *Client* : XML file containing user's session key

The first client request (1) redirects the client to the authorization page. The <callback_url> in the first server response (2) is the LAMA script that has to be called once the user has clicked on the authorization link. The second client request (3) is an API call to the auth.getsession function. The parameters are an api_key specific to LAMA, the token received in the previous response, and an *api_signature*.
The api_signature consists of a string containing the concatenation of all parameters for a given function ordered alphabetically; then a *secret* specific to the application (LAMA in our case) must be appended to this string. The MD5 hash of this string is then computed and results in a 32-character string. This string is the api_signature sent as a function parameter when calling the corresponding API function.
The XML file received in the second server response (4) contains a *session key* that the client will have to send with any further signed calls.

LAMA must also interact with Last.fm servers in order to obtain playlists, that is, files containing the URLs of mp3 files. This is done by tuning on a radio station, and then by retrieving a playlist according to the station.
Last.fm proposes five different kinds of radio stations, each one being identified by a URL. A station can correspond to a *tag*, a *user's library*, a user's *recommendations*, a user's *neighbour's station*, or to an artist *similar* to a specified artist. This is summarized in the following table:

| **Station** | **URL** | **Value of <parameter>** |
|---|---|---|
| Tag | *lastfm://globaltags/<parameter>* | any music tag (rock, acoustic, classical...) |
| User's library | *lastfm://user/<parameter>/ personal* | a Last.fm user's username |
| User's recommendations | *lastfm://user/<parameter>/ recommended* | a Last.fm user's username |
| User's neighbour's station | *lastfm://user/<parameter>/ neighbours* | a Last.fm user's username |
| Artist's similar artist | *lastfm://artist/<parameter>/ similarartists* | an artist present in Last.fm database |

Table 1: Summary of Last.fm radio stations.

Here is a description of the **last.fm radio protocol** [1]. This protocol consists of three stages:

1. Handshake
2. Adjustment
3. Playlist retrieval

In all three stages, the requests are sent to *http://ws.audioscrobbler.com*, on the port *80*.

The **handshake**'s goal is to establish an authenticated session with a Last.fm server. The following figure illustrates the requests exchanged between a client and the Last.fm server at the handshake phase:

(5) *Client* → *Server* : GET http://ws.audioscrobbler.com/radio/handshake.php?username=<username>&passwordmd5=<password>
(6) *Server* → *Client* : a series of $\backslash n$ terminated lines

In the request (5), <username> is a Last.fm user's username, and <password> is the MD5 hash of the user's password.
The server response (6) is a set of key=value pairs, among which the *session* value is the session ID. The other values are not used further in our implementation.

The **adjustment** phase is used to determine a Last.fm radio station, and works as illustrated below:

(7) *Client* → *Server* : http://ws.audioscrobbler.com/radio/adjust.php?session=<session-token>&url=<lastfm-uri>
(8) *Server* → *Client* : a series of $\backslash n$ terminated lines

In the request (7) , <session-token> is the session ID returned in the handshake response. <lastfm-uri> is a valid Last.fm radio URI as explained previously.
As previously, the server response (8) consists of key=value pairs: the value for *response* has the value OK if the request succeeded, and FAILED otherwise. The value for *url* is the Last.fm radio URI. If the request failed, an error code is returned.

The **playlist retrieval** stage allows the client to obtain a playlist file, as shown below:

(9) *Client* → *Server* : http://ws.audioscrobbler.com/radio/xspf.php?sk=<session-token>&discovery=<discovery-mode>&desktop=<version>
(10) *Server* → *Client* : XML Shareable Playlist Format (XSPF) file

The <session-token> parameter in the request (9) is the session ID sent in the handshake response. The <discovery-mode> and <version> parameters must be set to 1. The use of these parameters is not explained in Last.fm documentation.
The playlist file returned (10) can contain zero or more tracks. For each track, the following information are given (among others): the title, the artist, the duration, and the URL where one can download the mp3 file.
The URLs provided in playlist files are usable only once. Once the file download has started, the link is not valid anymore. Furthermore, these URLs are not valid anymore if the user requests another playlist before downloading the files.

Last.fm web services API [3] provides us with a bunch of functions that allow access to information about users, artists, tags, tracks, etc. Each function returns an XML file. LAMA needs to call Last.fm web services mainly in the context of the *policies* that determine the tracks to

propose. For example, a policy which returns a track from a user's friend radio will need to first get a user's friend name, by calling the function *user.getfriends*, and then parse the resulting XML file. An example of such an XML file is shown below:

```
1  <lfm status="ok"\=>
2   <friends for="meryet">
3    <user>
4     <name>bengiuliano</name>
5     <realname/>
6     <image size="small">http://userserve-ak.last.fm/serve/34/2022414.jpg</
         image>
7     <image size="medium">http://userserve-ak.last.fm/serve/64/2022414.jpg<
         /image>
8     <image size="large">http://userserve-ak.last.fm/serve/126/2022414.jpg<
         /image>
9     <url> http://www.last.fm/user/bengiuliano</url>
10   </user>
11  </friends>
12 </lfm>
```

### 3.1.3 Implementation

As said above, LAMA is a web application framework. On the frontend, XHTML, CSS, Flash and JavaScript technologies are used, whereas the backend is developped with a scripting language, PHP. The communication between the frontend and the backend is done using non-persistent HTTP connections, as well as XML. The advantages are the following:

- Simple, robust, widely deployed technologies. Except for Flash, these are all open standards.
- No need of any software except for a browser and a webserver to host the application
- Because they're widely spread, there is a huge documentation for these technologies, and many possibilities to learn them.
- The use of HTTP as communication layer keeps things dead simple, facilitating the developement and the debugging.

But there are of course several drawbacks using these simple tools rather than very specific, sophisticated technologies. Here are a few of them:

- We have all the long known drawbacks for developping robust, cross-browser interfaces using the XHTML, CSS and JavaScript trio. AJAX is still at its beginning, and feels sometimes a bit fragile.
- HTTP is a one time use, stateless protocol, where the roles of the client and the server are clearly defined. It is impossible for the server (unless hacking the intent of the protocol) to choose the moment it wants to send data. We have to use AJAX polling to be updated periodically.
- Using interpreted scripts called by an already existing webserver is more simple, but doesn't leave us the flexibilty of what could have been a full-blown LAMA music server. We don't have the advantages of multi-threading, and it's a little more complex to store data between client requests (we have to use a database, and serialize objects).

One element is still missing in the list of technologies used: the database system. MySQL (`http://www.mysql.se/`) was used, as it is quite popular with PHP and very widely deployed as well. The database stores mainly informations about the users, the pool of radios and the cache.

## 3.2 Client Side

Let us briefly examine how the frontend is developed. It is made of the following components (see the diagram at the beginning of the chapter for a visual representation):

**index.php** the page where the user can register, or, if he is already registered, where he can log in and access the web application

**radio.php** the web application's interface and description, in XHTML and CSS. This is one of the two central pieces of the frontend architecture

**lama.js** the other central piece of the frontend. All the javascript code that ensures event detecting, client-server communication as well as response parsing is in this file.

**prototype.js** the javascript library used for AJAX communications and DOM manipulation.

**player.swf** the music player, written in flash by Jeroen Wijering (`http://www.longtailvideo.com`). Simple, free for non-commercial use, and... well, simple.

**swfobject.js** small JavaScript library used to embed flash objects into HTML documents, and manipulating them afterwards.

The javascript part of lama is in the file lama.js, which we will discuss now. It uses an underlying JavaScript library, namely the Prototype framework [5], which is very useful for AJAX communication and DOM manipulation. We describe quickly the features lama.js brings.

lama.js handles the integration of the flash player. It parses the radio.php, and recognises the place where the player should be embedded as well as the width it should have. It handles also the events sent by the flash player: ready, end of track, etc.

One of its very core functions is the event handling. It listens to differents type of event, e.g. for clicks on elements which have a parameter "lama-clickable" set to "yes", or for the end of the current track. Everytime an event is caught, it builds a request that is sent to the backend.

This brings us to the next feature of lama.js: it handles the communication between the backend and the interface, using AJAX requests. It is possible to display an AJAX status indicator by using a div with id "lama-status".

Once the server has responded to our request, the response must be parsed, and the appropriate actions must be taken. Based on the XML file we receive from the server, here are the different actions that lama.js takes when needed:

1. change the current track (in the music player)
2. update any content on the interface (i.e., the innerHTML of any DOM element)
3. trigger any javascript function (e.g. trigger a visual effect)
4. execute an arbitrary javascript code snippet (disabled by default for security reasons)

The next section gives a more detailed explanation of the client-server communication.

## 3.3 Communication between the client side and the server side

The client side and the server side of LAMA communicate via AJAX (Asynchronous Javascript and XML) requests. The Prototype framework [5] has been used for this purpose. The communication takes place between the file **lama.js** on the client side, and the php script **CommunicationInterface.php** on the server side. The following piece of code shows how a request is sent using Prototype:

```
this.sendRequest = function() {
 new Ajax.Request(server, {

  //method can be the HTTP method GET or POST
  method: "post",(

  //the parameters that will be sent to the server
  parameters: this.param,
  onSuccess: function(transport) {

   //object specific to LAMA that will parse the response
   //the parameter ALLOW_SERVER_JS is also specific to LAMA
   new ResponseParser(transport.responseXML, ALLOW_SERVER_JS); },
  onFailure: function(){
   new ErrorLog("something went wrong with an AJAX request"); } });
}
```

The client side has to inform the server side about all the events that occur in the interface. To respect the 'thin client' fashion we decided to implement, very few kinds of such events have been defined: an *init* event, a *click* event, an *endtrack* event, and a *recall* event.
The *init* event is generated when the page is loaded in the browser. The *click* event is generated at any click on certain elements of the interface. Those special elements own an attribute called *lama-clickable*, meaning that a click on those elements must be handled by LAMA. An *endtrack* event is sent when the player has finished playing a track. The event called *recall* is generated when the previous server response's recall parameter is set to '1' (see below).
A special event called *poll* can also be generated, allowing the client side to send periodic request to the server. This feature is not used in this version of LAMA.

The server response is an XML message that is then parsed by the client side. Below is an example of such a message:

```
<?xml version="1.0" encoding="UTF-8"?>
<response xmlns="http://www.biocycle.ch/lama/">
 <track>
  <![CDATA[http://kingpin5.last.fm/user/97b314710d3b91bc10c345993473f580.
      mp3]]>
 </track>
 <content>
  <element id="artist-current"><![CDATA[Midnattsol]]></element>
  <element id="title-current"><![CDATA[Unpayable Silence]]></element>
 </content>
 <actions></actions>
 <recall>1</recall>
</response>
</lfm>
```

The use of the parameters of the XML response is the following:

- track: its value is the URL of an mp3 file that has to be played by the player immediately.
- content: can contain several <element> elements, each specifying the content that has to be replaced for a given item in the interface.
- actions: can contain several <action> elements, each specifying a Javascript function that has to be called, together with its parameters.
- recall: is set to '1' if the client side has to send an AJAX request with a *recall* event. This mechanism is used to limit the time interval during which no information is displayed in

the interface. It allows the server side to provide the client with new content immediately: the information about the next track in the "normal channel" are retrieved from the cache and sent to the client; only then the information about the alternatives tracks are fetched from Last.fm servers and stored in the cache. This is illustrated in Figure 1.

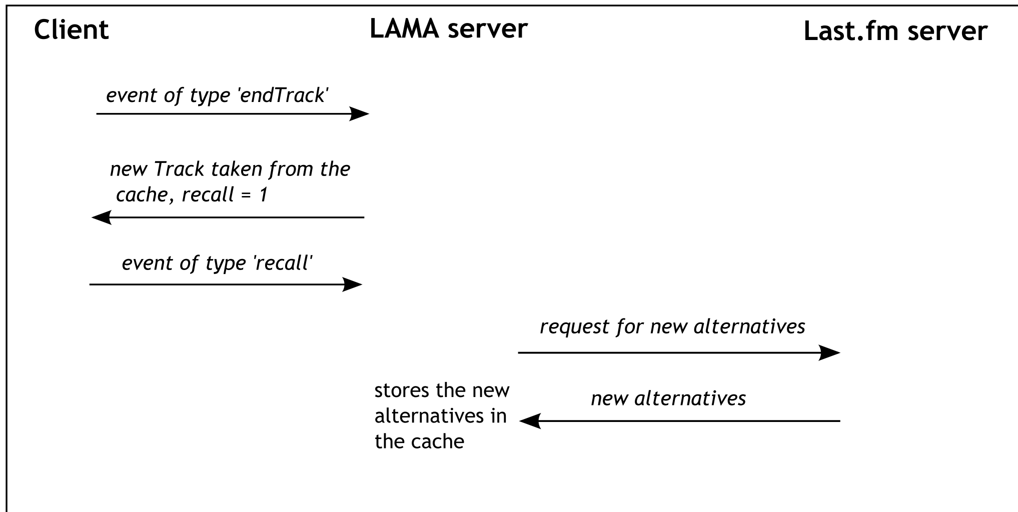


Figure 3: Use of the 'recall' parameter

# 4 Application

## 4.1 Objectives

The second part of the project consists of the application in itself, and also includes two distinct parts: the implementation of the application, and the design of the application's interface.

The goal of my application was to implement original songs suggestions (see Table 2). The application is also made of one server side, and one client side.

The interface had to be user-friendly, "intuitive", and propose an original display for its content. One objective of my interface was not to fix the number of alternatives, to keep some flexibility. I also wanted to define several categories of alternatives, and emphasize the existence of these two categories in my graphical interface.

## 4.2 Server Side

### 4.2.1 The Extension of LAMA

The application server side consists, among others, of several classes inherited from LAMA classes that are specific to the application: it has to handle the AJAX requests coming from LAMA's client side, and to send the corresponding response. Therefore it has to define the content of the XML response described in 3.4 depending on the interface. Thus the php classes that must exist in any application using LAMA are: **MyEventHandler.class.php**, **SimpleInterface.class.php**, **FullPolicy.class.php**. All other php files are purely specific to the application.

The class **MyEventHandler** is the "controller" of the application. That is, it receives the messages from the client side of LAMA, and, according to the type and content of the message, it computes the due response. The class structure is basically the same for each application

built on top of LAMA: it consists of a switch/case block parsing the type of the event received. Those types of events are defined in section 3.4.
Several attributes needed by the application have been added to this class: **$nb_alt**, which is the number of alternatives specified in the XML configuration file (see section 4.2.2); **$xml**, which is the XML content of the XML configuration file.
Specific variables are also used in this class: **$mapping**, which is interface dependent: as explained in the section 4.1, the user can choose the next song to play. When the user selects a song, this song is swapped with the default next song in the graphical interface. However, it is the content of the HTML elements that is swapped, not the elements themselves. Therefore one must keep track of the correspondence between the HTML element names, which are fixed, and what they contain. Since the tracks stored in the cache are ordered, this mapping mechanism allows find which track in the cache corresponds to which element in the page.

Another special variable is **$iface_config**, an array that stores information about the policies: as explained below, there are two kinds of policies. These two categories of policies must be differentiated in the interface; therefore, the response sent to the client side must determine how each alternative is represented according to the corresponding policy.

Several functions have been added to the class **MyEventHandler** in order to do some processing related to the interface: **determineInfo(...)**, which determines the information messages to be sent according to the chosen policies; **getOuterDiv(...)**, which is highly interface dependent, and knows the name of a specific container DIV in the page; **initMapping()**, which role is to set the variable **$mapping** to its initial value.

Another difference between two applications is the content that is sent in the response, as well as the actions that must be taken, that is, the Javascript functions that must be called on the client side when the response is received. Below is a response specific to my application:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <response xmlns="http://www.biocycle.ch/lama/">
3   <track>
4    <![CDATA[http://kingpin5.last.fm/user/afea713e807b70367b741f8c0151899b.
       mp3]]>
5   </track>
6   <content>
7    <element id="artist-current"><![CDATA[Nemesea]]></element>
8    <element id="title-current"><![CDATA[Believe]]></element>
9    <element id="about_bio"><![CDATA[About Nemesea:]]></element>
10   <element id="user-age"><![CDATA[23]]></element>
11   <element id="user-country"><![CDATA[CH]]></element>
12   <element id="user-name"><![CDATA[meryet]]></element>
13  </content>
14  <actions>
15   <action name="set_cover">
16    <param name="src">
17     <![CDATA[http://userserve-ak.last.fm/serve/174s/10267549.jpg]]>
18    </param>
19   </action>
20   <action name="show_bio">
21    <param name="bio">
22     <![CDATA[Nemesea is a symphonic rock band founded by vocalist Manda
         Ophuis and guitarist Hendrik Jan de Jong]]>
23    </param>
24   </action>
25   <action name="setMapping">
26    <param name="mapping">
27     <![CDATA[lamaMapping['normal_next'] = 'normal_next';lamaMapping['alt1
```

```
        '] = 'alt1'; lamaMapping['alt2'] = 'alt2';]]>
28      </param>
29      </action>
30     </actions>
31    <recall>1</recall>
32  </response>
```

This XML response contains several actions that are defined in the application: "show_bio", "setMapping", and "set_cover".

The content of this response is defined in the class **SimpleInterface.class.php**. This class knows how the graphical interface is designed, meaning that it knows which are the elements of the interface and which are the available Javascript functions. Therefore it can specify the functions to call, and the content of the graphical elements: the row 7 of the above XML response means that the value of the graphical element named 'artist-current' will be set to 'Nemesea'. The rows 15 to 19 mean that the function 'set_cover' must be called with a parameter 'src', which value is 'http://userserve-ak.last.fm/serve/174s/10267549.jpg'.

The application must contain a set of **policies** that are used to define the "normal song" and the alternatives proposed. The role of a policy is to determine a Last.fm radio station; then a playlist is retrieved for the station and the mp3 file stored at the link indicated in the playlist can be downloaded. Two kinds of policies have been implemented: those that are related to the user, and those that are related to the current track, as summarized in the following table:

| **Policy Name** | **Description** | **Related to...** |
|---|---|---|
| UPersonalRecommendations | The user Last.fm recommendations | ...the user |
| UPersonalLibrary | The user Last.fm library | ...the user |
| UFriendPersonal | One of the user's friend's Last.fm library | ...the user |
| UTTSAP (User top track similar artist policy | An artist similar to one of the user's top track's artist | ...the user |
| URAS (User recent artist similar) | An artist similar to one the user recently listened to | ...the user |
| CSimilarArtist | An artist similar to the current one | ...the current track |
| CATTP (current artist top tag policy) | The top tag of the current artist | ...the current track |

Table 2: Summary of policies

The name of the policies related to the user start with a 'U'; those related to the current track must start with a 'C'. The various policies are managed by the **FullPolicy** class.

The php script called once the user has logged in is **radio.php**. This script is made of two parts: a PHP part, and an HTML part.
The PHP part generates the CSS files described in the section 4.2.2.
The HTML part is the page that is sent to the browser. Although it is specific to the application, there are elements that must be common to all applications using LAMA: three Javascript scripts must be included (prototype.js, lama.js, swfobject.js); there must be a DIV element with the identifier 'player', owning the attribute 'lama-width', which defines the width of the player in the page; the elements that are controlled by **lama.js** (see section 3.3) must have the attribute *lama-clickable* set to 'yes'.
The rest of the page depends on the application.

### 4.2.2 The ”Pure” Application Part

The application is designed so as to provide some flexibility: the number of alternatives is variable, and for each alternative, a policy can be chosen. The number of alternatives lies between 1 and 7. 7 seems to be a reasonable maximum, in order not to overload the graphical interface, and also not to confuse the user with too many choices. The number of alternatives as well as the policies are defined in an XML configuration file, as shown below:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <config xmlns="http://www.biocycle.ch/lama/">
4   <nb_alt>2</nb_alt>
5   <policies>
6    <policy>UPersonalRecommendations</policy>
7    <policy>CSimilarArtist</policy>
8    <policy>UFriendPersonal</policy>
9   </policies>
10 </config>
```

The variable number of alternatives implies that the positions of the alternatives in the interface are also variable. The positions of the alternatives in the interface are computed according to the parameter <nb_alt> of the config.xml file. This is done in the php file **songs_layout.php**, which computes the positions of elements to display in a circle fashion, and writes a corresponding CSS (Cascading Style Sheet) file. This file is called **songs_x.css**, where x is the number of alternatives.
The XML configuration file can be filled thanks to an HTML form that is included in the php script **config.php**. Therefore only specific values can be taken by the XML elements.
The generation of the file **songs_x.css** is illustrated in Figure 4. The label on the arrows indicates the output of the processing that occurs before; this is also the input of the processing that occur afterwards.

The idea behind this flexibility was to provide an ”intelligent” interface. Unfortunately, due to a lack of time (and exams preparation), this remained a theoretical idea.
The principle I would have liked to implement is that the interface adapts its content according to the user’s past actions. The more songs the users listened to, the more alternatives are proposed in the interface; and this, proportionally to the categories of tracks the user choosed: after having listened to a certain number of red tracks, an additional red alternative would be proposed. Therefore the interface would adapt the proposals to the user’s behaviour. This would require several additional features in the application, including a persistent cache table in the database.
The fact that the interface allows a flexible number of alternatives is the first brick to build an adaptive interface.

The application part also contains an error handler. Errors on the server side are mainly due to the fact that the application relies on Last.fm servers to answer API calls and to perform the radio protocol (see section 3.2.2). It might happen that the servers do not reply within a threshold delay of 30 seconds, or that the connection with a server cannot be established. In these cases, the error message generated automatically is sent in the AJAX response sent to the client side, and the response cannot be parsed correctly. Sometimes it is desirable that the error message is not sent: when the application server side is fetching the alternatives tracks to propose, a lot of connection to Last.fm servers need to be established: for each alternative, a tune on a radio station is performed; then several API calls are done to grasp information about the artists. Among all these connections and requests, one can fail, while all the others succeed. In this case it is more appropriate to ignore the connection failure, and to treat the successful ones normally. Thus the error message must not be generated, and the response is sent to the client side in the correct format.

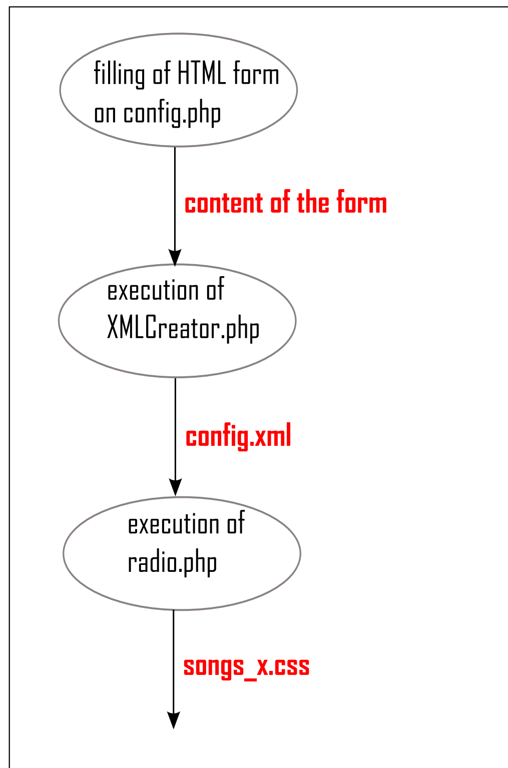


Figure 4: Generation of the file **songs_x.css** from the values inserted in an HTML form

To deal with this fact, the script **ErrorHandler.php** has been added to the application. This script contains only the fuction **error_handler($type, $msg, $file, $line, $context)**; this function is called by the PHP function **set_error_handler('error_handler')**, which automatically forwards the error message generated to the function specified as parameter.
For the moment, it ignores every error messages, except for the one indicating a time out at a the server ('Maximum execution time of 30 seconds exceeded').

## 4.3 Client Side

### 4.3.1 Implementation

The client side of the application has been implemented for Mozilla Firefox. It consists of the following files: two CSS files, one generated as explained in the section 4.2.2 (**songs_x.css**), and one permanent file (**default.css**); one Javascript file **application.js**. The images used for the interface are in the folder **GUI**, and other images are in the folder **images**.
Another Javascript file, **Effect.js**, is used to add some graphical effects in the interface. This file is part of the Scriptaculous framework [6].

The interesting part of the client side reside in the **application.js** file. This file defines the function and the events specific to the application. It contains an *ApplicationEventController* object which triggers actions depending on the event type, and on the event source. These events can be of various types: 'mouseover', 'mouseout', and 'click'. In case of a 'mouseover' or 'mouseout' event on an element that has the 'gui-sensitive' attribute set to 'yes', the event controller will trigger the corresponding action. The same applies in the case of a 'click' event on an element owning the 'gui-clickable' attribute.
The 'mouseover' and 'mouseout' events are used to display or hide some elements. The 'click' event is used to open or close the "help" page.

A few animation effects are part of the interface: the effect used to display the biography is the Scriptaculous *BlindDown* effect.
Another Scriptaculous effect was first used to display the DIV containing the information about the alternatives; this effect is *Appear*, coupled with the effect *Fade* to hide the DIV: when the user "mouseovers" on a given zone, the DIV should be displayed. When the mouse leaves the zone, the DIV should disappear. Unfortunately this function was bugguous: it was not able to handle correctly the limits of the sensitive zone. Therefore this function has been replaced by the function **fadePic()** [7]. This new function has been slightly modified: to implement the "fade" effect, the function diminishes the opacity of the object in steps; it ended with an opacity of 0.01, where an opacity of 0 is required. The same applied for the "appear" effect, which increments the opacity of the object up to 0.99, where an opacity of 1 is required. The modified function gives a better result. However, it is still not perfect: in case of fast movements over the sensitive zone, the image blinks for a while and disappears.

Two functions relative to the DOM tree have been implemented: **findFirstChild(element, type)** and **findFirstSibling(element, type)**. Although they are not used anymore in the application, I decided not to remove them from the script. They can be used to find the first child, or the first sibling, of a certain type, for a given element.

An important function is the function **swap(...)**. This function is called whenever the user selects an alternative song. It swaps the content of the elements received as parameters: this consists in inverting the title of the songs, the name of the artists, as well as changing the background image and the note image.

The **swap(...)** function uses a function called **style** [8] which determines the value of a given CSS property for an element.

## 4.4 Graphical User Interface

The last part of the project was to propose a graphical interface suitable for the content that was to be displayed. The interface must contain elements that allow to display the following information: the title and artist for the current track, the title and artist for the next track on the same channel, and the title and artist for each alternative track. It must also display some information about the current artist, such as the biography. It must contain a music player, and a mechanism to skip the current song.
The various components of the interface have a fixed size. It would have been better to have an interface that adapts to the screen dimensions; but due to the amount of images that are included in the interface as DIV backgrounds, it was much simpler to fix the size of the elements. Transforming this interface into a scalable one is part of possible improvements.

The basic idea to present the alternatives on the interface is to place them in a circle fashion; since the number of alternatives can vary, this allows to compute very easily the positions of the alternatives. Therefore the interface is divided in two parts : a left part, containing the player, and displaying information about the current track; the right part, containing the alternative choices, being the part with which the user can interact. Based on this scheme, several versions of the interface were designed.
Several trials have been designed for the first version of the interface. Two are shown in Figure 5 and Figure 6.

It turned out that this interface had several important problems, concerning the design and the semantics: design problems were that too many different colors and shapes were used, and the background was too dark; semantics problems were that there was no relation between the left and the right part of the interface; there was no indication between the content and the way this

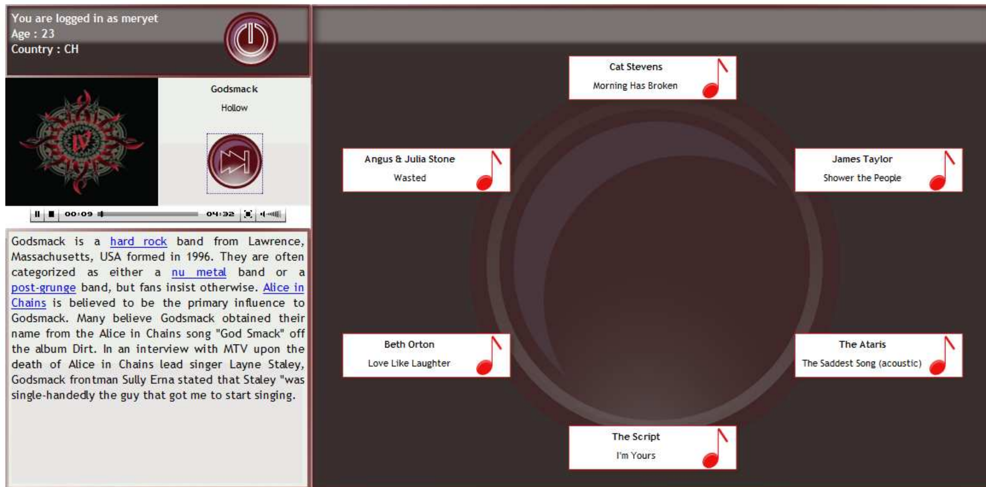


Figure 5: Version 1.1 of the graphical interface

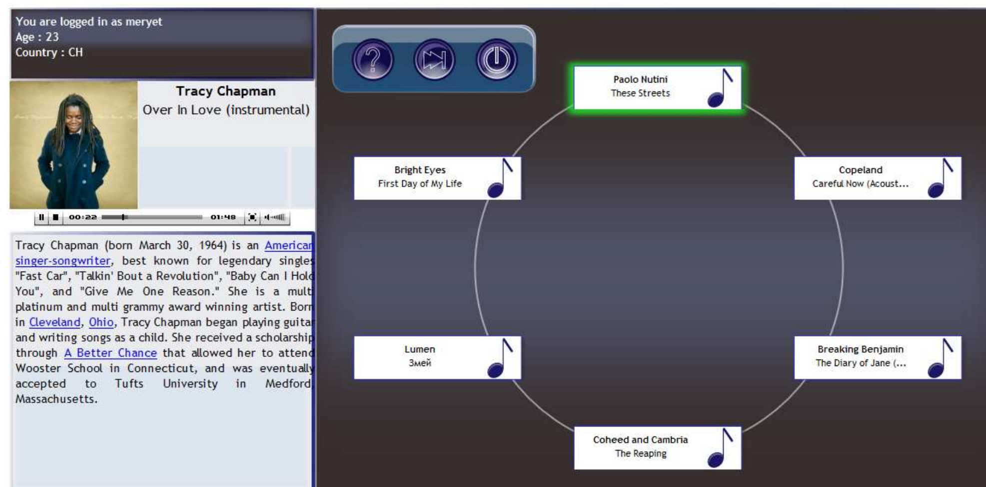


Figure 6: Version 1.2 of the graphical interface

content was presented. Therefore a second work phase on the interface was needed to improve it. The resulting interface is shown in Figure 7 and Figure 8.

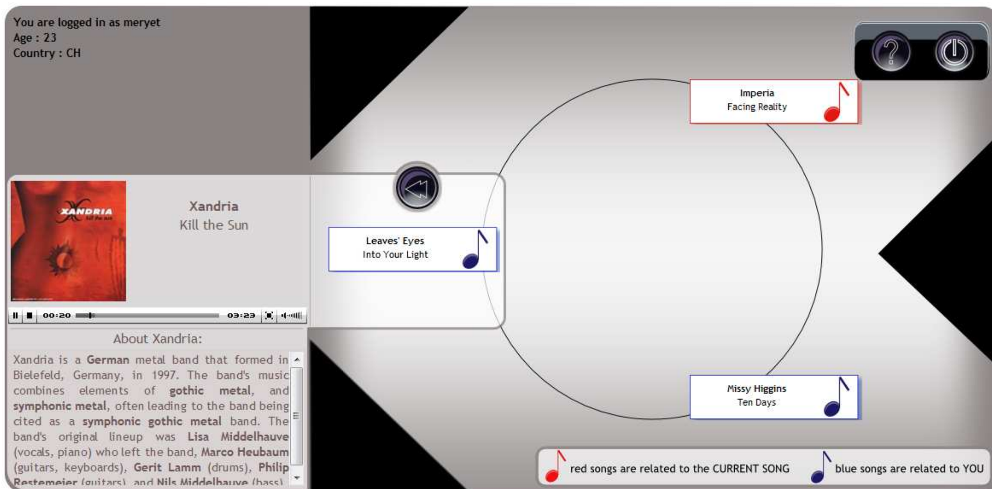


Figure 7: Version 2.0 of the graphical interface

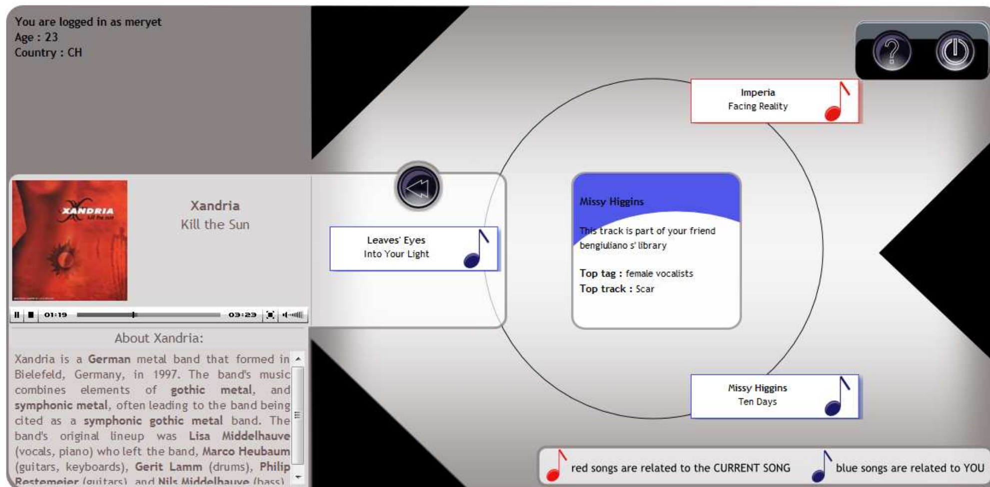


Figure 8: Version 2.0 of the graphical interface

The major changes between the first and the second version are indicated in Figure 9, and explained below:

1. The background color is much lighter. Since two colors were needed to distinguish the two kinds of alternatives, a neutral color was appropriate for the background.
2. The same shape is used whenever some content needed to be enclosed.
3. The player part includes the next track to be played, showing the relation between the left and the right part of the interface. It also contains the 'skip' button, making more clear that clicking on the button will push the track in the player.
4. The relation between the two parts is emphasized by the black triangles : they are disposed in a funnel shape, indicating that only the song closer to the player can pass through, and therefore, can be played.

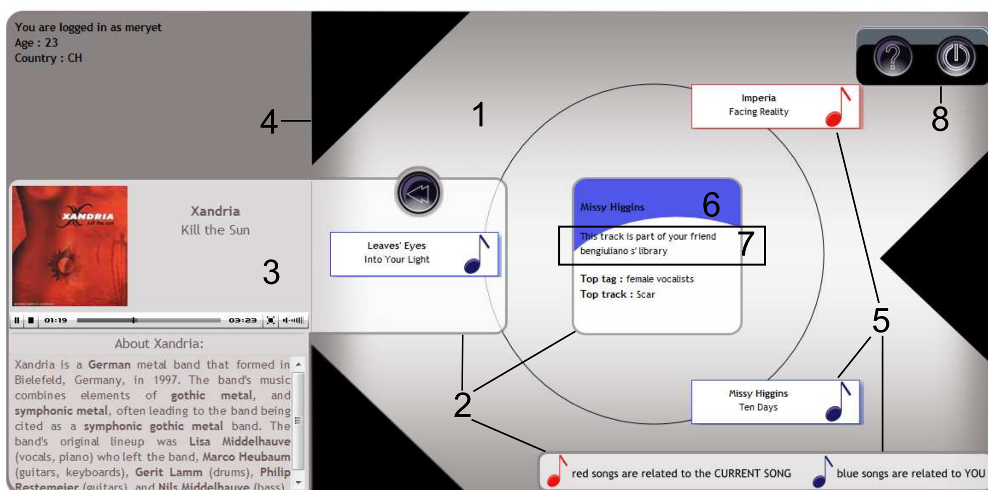


Figure 9: Version 2.0 of the graphical interface

5. Two colors are used to differentiate the alternatives, and the meaning of these colors is explained in the interface.
6. The color of the element that presents additional information relative matches the corresponding category of alternatives.
7. A specific indication states why this track is proposed, according to the policy used.
8. The icon bar is positioned on the top right corner of the interface.
9. The whole right part of the interface is arrow-shaped, symbolizing that the information goes from the right part of the interface to the left part.

The biography of the current artist had to be placed on the left part of the interface, since it has no relation with the alternatives. The same occur for the information concerning the user.

Two animations are used in the interface. They are quite discrete, to keep the interface sober, while adding some movement. One animation is used to display the biography of the current artist. The other animation is used to display the DIV element containing the information about the alternatives.

### 4.4.1 How The User Can Interact With The Interface

The user can interact with the interface on various contexts: first, it can simply interact with the flash player. Secondly, it can interact with the icons present in the 'icon bar', to log out, and to open the help page.
The most interesting way to interact with the interface is to play with the alternative propositions. A click on an alternative will place it in the 'next track' position, near the player. This is intuitive for the user, since the mouse cursor becomes hand-shaped when it is moved over the alternative, indicating that the element is clickable.
The other way to interact with the alternatives is to place the mouse over a 'note' image. Again, the mouse cursor becomes a 'hand', and simultaneously, the information about the corresponding alternative is displayed in the center of the right part.

Although the interface is easy to use, a quick help is available when the user clicks on the 'help' button. This help page gives information about how the user can interact with the interface.

### 4.4.2 The Tools Used

Not having followed any courses on interfaces, and having no previous knowledge on that domain and on color theory made this part of the work quite difficult, but very interesting.
Every single graphical elements of the interface has been made using **Inkscape**. A bunch of good tutorials for Inkscape are available on the Internet [9], making it easy to learn. No existing elements have been taken on the Internet. This way, it was easier to design the required elements and to integrate them than to integrate already existing elements.

# 5 Encountered Problems

The implementation phase began quite late in the semester; during the first 5 weeks there were no possibilities to meet and discuss the project in details. Once the architecture of LAMA has been defined, only 6 weeks out of 14 remained, which is very few to implement a whole application and design an interface.

One challenge in this project was to coordinate my work with Lucas'; we also had to discuss LAMA's architecture in detail, to find an agreement on how it had to be implemented. It was a good experience to collaborate on a semester project with someone; I think we always learn more when we can share our opinions and discuss our problems with someone.

Concerning the interaction with Last.fm, there are problems for which we cannot propose any solution: LAMA is dependent on Last.fm servers, as explained in section 4.2.2
The radio protocol is slow, and this delay is visible in the interface. LAMA's good functioning is dependent on the state of Last.fm servers, and their ability to send a fast response. If one server does not respond, we can only propose a solution to allow the application to continue working (see section 4.2.2.).
Another issue with Last.fm is that it often returns an empty playlist file when we ask for the user's personal library, of for its personal recommendations. We think that this occurs more often if the user's library size is below a certain threshold, but we found no official explanation for that.
It might occur that Last.fm protocols result in an unexpected outcome. The 'SimilarArtistPolicy' returns the radio station of an artist similar to the current artist. If there is no current artist, then we try to tune on the radio station of an artist similar to one which name is empty (see 5th row of Table 1). Unexpectedly, Last.fm radio protocol succeeds, and a playlist file is received. Therefore we can listen to the radio of an artist similar to " (empty string).
These several facts allow to say that even though Last.fm tools are very well documented, they are not as reliable as they should.

A few problems encountered while using Scriptaculous (see section 4.3.1) allow me to say that, although this library provides a simple framework to bring cool effects in an interface, I would consider that it is still too fragile to be used in a more important context.

The conception of the interface revealed that not having taken the Human Computer Interaction course would be a problem. Indeed I focused too much on the appearance of the interface, meaning that I wanted it to be graphically pleasant, while the important point was its functionality, and the relation between the content, and how it is displayed. I realized a bit too late that advises on how to design a 'good' interface would have been really useful before we started the design phase. This would have allowed us to spare one week or two.

An strange implementation issue was discovered in the application page (radio.php), in which I placed a link on a div element :

```
1 <a href="#">
2  <div id="div-a" class="track" lama_clickable="yes">
3   ...
```

```
4    </div>
5  </a>
```

This worked fine until I uploaded my application on the Netoxygen server to test it non locally: instead of placing the link on the whole div element, it sometimes appeared to be on the content of the div. This did not happen every time, and only on one such element in my page. I tried my application from several client computers, and this seemed to happen only on mine. After some research on the Internet, it came out that placing a link on a div element was not standard compliant; this would explain why Firefox may misinterpret it. The bug was easily corrected in my page, since the link was present only to transform the mouse pointer into and hand-shaped one. I only had to replace the above piece of code by :

```
1  <div id="div-a" class="track" lama_clickable="yes" style="cursor:pointer
      ">
2    ...
3  </div>
```

However I regret that web browsers might have such a 'random' behaviour.

# 6 Future Work

As always, there were more ideas than time to make this ideas concrete. This section briefly describe what could be done in the future to improve the framework and my application.

## 6.1 Extension of LAMA

Several features could be integrated into the LAMA framework. Those features may include scrobbling. The scrobbling protocol is fully explained in [4].

An system using **explicit data collection** (see section 1) to rate songs could also be added to LAMA. This would be done thanks to Last.fm API, which contains a *track.love* and a *track.ban* function. These two functions require authentication (see section 3.2.2); since every API call sent by LAMA are automatically authenticated, this should be an easy improvement of the framework.

One could also implement a system using **implicit data collection** (see section 1) for filtering the artists, based on the use of the 'skip' button: whenever the user skips a song, a weight is given to the song according to the proportion of the track the user has listened to. The shorter is the part of the song, the weaker is the weight. A song listened completely would receive a weight of 1. When the same artist is proposed several times, the weight given to its tracks is compared, and if the average weight does not reach a minimal threshold, the artist is excluded from the user's recommendations.

Last.fm API provides a method to add tracks, albums, or artists to the user's library : *library.addTrack*, *library.addAlbum*, and *library.addArtist*. LAMA could provide the possibility to add such elements to the user's library.
Personalizing the home page according to the user's identifying cookie would also be a possible extension of the framework.

The songs proposed to the user could be recorded in a log file; this would allow to filter the user's recommendations, eliminating the songs that have been proposed recently.

Another important thing that has to be improve in LAMA is the login and registration phase. Indeed, no verification is done concerning the validity of the password, which the user does not even need to enter twice. Several improvements need to be done to make the application secure.

## 6.2 Application and Interface Improvement

The most important improvement that might be brought to the interface is the one explained in section 4.2.2: turning the application in an adaptive, intelligent one.

The policy which chooses a user's friend and tunes on its station (see Table 2, row 3) could be improved in such a way that it chooses a friend that has a high profile similarity with the user. Last.fm API function *tasteometer.compare* allows to compute a similarity score between two users, and returns a list of artists shared between these two users. Proposing the radio of a friend that has a high score may be more accurate.

A ranking system for the tracks could be added to the application, giving each track a similarity factor: the more similar to the current track, the higher the similarity factor. The challenge would be to integrate this in the interface. Placing the tracks on a circle intrinsically defines a distance between each alternatives and the current track. This distance could be determined by the similarity factor: the track that has the lowest similarity factor could be placed at the highest distance from the current track. However since the selected track is moved near the player, the meaning of the distance is lost when the user selects an alternative track: if the user chooses the least similar song, that is, the furthest one, then this song is swapped with the default one; thus although it is the least similar, it takes the closest place to the current track. Therefore one should find a way to represent this similarity factor such that its meaning is kept whenever the user chooses another song.

# 7 Conclusion

The results of this project are the LAMA framework, and an application using this framework.

Designing and implementing the framework was the "software-engineering" part of the project. During this phase we learned how to use the development tools provided by Last.fm.
Last.fm is a very well documented framework. It was quite easy to work with Last.fm web services and protocols, although they are not as reliable as one can whish.

The implementation of the application required a lot of thinking on what was to be displayed in the interface, how the user would be able to interact with the content, and how to implement it on top of LAMA.

The design of the interface, which was the "creative" part of the work, was very challenging. My lack of previous experience in that domain made this task very difficult (much more than I first thought), and thus took more time than what I planned to spend on it.

The whole project covers various domain: software architecture, software implementation, graphical design; this makes it a very interesting project. It is unfortunate that we did not manage to start the implementation phase earlier: this would have allowed us to implement one or two additional features among those mentioned in section 6.1 and 6.2.

# References


[1] last.fm radio protocol,
http://www.lastfm.fr/user/dahnielson/journal/2007/08/07/8nrm_last.fm_radio_protocol.

[2] last.fm authentication protocol,
http://www.lastfm.fr/api/webauth.

[3] last.fm API,
http://www.lastfm.fr/api.

[4] audioscrobbler protocol,
http://www.audioscrobbler.net/development/protocol/.

[5] prototype,
http://www.prototypejs.org/.

[6] scriptaculous,
http://script.aculo.us/.

[7] function appear/fade,
http://www.developpez.net/forums/d278036/webmasters-developpement-web/communaute-developpement-web/contribuez/src-petit-script-fading-dimage/.

[8] function style,
http://blogs.telerik.com/tervelpeykov/posts/08-09-16/JavaScript_Get_Any_CSS_Property_Value_of_an_Object_using_style.aspx.

[9] Inkscape tutorial
http://inkscapetutorials.wordpress.com/.

[10] Tutorial to create icons with Inkscape
http://www.starfishwebconsulting.co.uk/tasty_icons.