TraceTrack: a Recommender System Interface That Displays Your Listening Habits

Lucas Maystre

January 14, 2009

Abstract

In the internet era, where everybody can search through billions of pages on Google, receive informations on his mobile phone as soon as they come out, and stay connected nearly everywhere on earth, the question of how to handle the gigantic mass of informations is becoming more and more important. As getting the information becomes easier, what to do with it becomes a tougher challenge.

From there, one might ask himself two questions: how to filter that information? And how to visualize it in such a way that it helps getting the big picture and making decisions based on it? These questions give birth to two topics, data mining & interface design.

Recommender systems are at the very edge of these two topics. The goal of this project is to build an original music recommender and player, and give the user an original way to navigate through the millions of songs proposed by a web music provider, Last.fm.

During this project, a framework was built, which can be used to quickly build music recommender web applications. After that, a sample application, TraceTrack, was built on top of the framework to illustrate its possibilities and to propose an original recommender interface.

This semester project was done in collaboration with Aurélia Rochat, under the supervision of Dr. Pearl Pu and Nicolas Jonas, at the Human Computer Interaction Group at EPFL.

Contents

1 Introduction

In this section we will present a few points that motivated the project as it is, i.e. a framework that allows to easily develop web radio applications, and the concrete application built on top of the framework.

What is a recommender system? It is an information filtering technique that present information items that are likely to interest the user[1].

1.1 Music recommender systems: an overview

There are several music recommender systems already available on the web. Here we'll describe a few of them:

Pandora A famous american internet radio[2], built using the Music Genome Project[3], sadly only available in the U.S. due to licensing constraints. Very sophisticated and accurate algorithm to organize and classify music.

Last.fm Very successful british startup[4], bought in 2008 by CBS for several hundred millions of dollars. Emphasizes the social aspect. It is possible to compare our musical tastes againt our friends, and to find other people that have similar musical tastes. The underlying data mining and recommendation infrastructure is called Audioscrobbler[5].

Genius Launched in 2008 by Apple for its media player iTunes[6], it builds a playlist based on one song. The recommendations are computed from the listening habits of all the ITunes users.

Musicovery Interesting visualization paradigm, sees songs and their connections as a graph. Songs can be selected by mood[7].

There are also quite a few web services that deal with music[8]. Last.fm is interesting from this point of view, because it offers a very complete RESTful API for anything dealing with music (informations about songs, artist,

[1] source: Wikipedia article on recommender system
[2] `http://www.pandora.com/`
[3] `http://en.wikipedia.org/wiki/Music_Genome_Project`
[4] `http://last.fm/`
[5] `http://www.audioscrobbler.net/`
[6] `http://www.apple.com/itunes/`
[7] `http://www.musicovery.com/`
[8] `http://www.programmableweb.com/apis/directory/1?apicat=Music`

albums and even events). They also offer a radio API, such that it is possible to build an internet radio and play songs.

The question wasn't even really brought up: Last.fm was the logical choice, providing an underlying recommendation system on top of which we could work.

1.2 Motivations

Often, the music recommenders described above propose songs to users based on explicit user feedback. The current trend of recommender systems is to avoid this explicit feedback, and to take decisions based on implicit user actions. For example, the moment the user skipped as song, given his previous "skipping" habit, could have a particular meaning.

To describe it in one sentence, the goal is to extract the semantics from the flow of implicit informations provided by the user's actions. Possibilities are endless: maybe the mouse cursor position also reflects some semantic meaning?

From this starting point, the project was to facilitate the building of such recommender systems, by binding together the various components via a framework, and to build an application that would demonstrate the use of this framework.

It is the possibilities offered by the Last.fm APIs that also triggered this project, the fact that is was possible to handle a library of millions of songs and metadata associated with it. This was really an exciting perspective.

1.3 Different approaches

There were different approachs that could've been taken from there. How is it practically going to be developped? As a desktop application? On top of an existing software? We present here the alternatives that were considered.

- An existing application, called JLFM[9], was available, which already implemented the Last.fm radio API, allowing anybody to download songs from Last.fm. It was developed in Java, and wasn't documented. As it is distributed under a free software license, it could have been possible to extend JLFM and to fit it to our needs.

- Another possibility was to use the already existing Last.fm software[10], written in C++, whose source code was also open to everybody. It

[9] `http://code.google.com/p/jlfm/`
[10] `http://www.last.fm/download`

wasn't very well documented, and it would have been quite difficult to develop on top of such a complex software.

- The third solution, the one that was chosen in the end, was to develop, from scratch, a new system that would work as a web application. This solution was seducing because web applications are easy to build (or at least to prototype) and easily maintainable. The Last.fm APIs seemed not too hard to implement, at least not harder than it would have been to understand the applications described above. The technical implementation, such as the technologies used, were still to define.

2 The LAMA framework

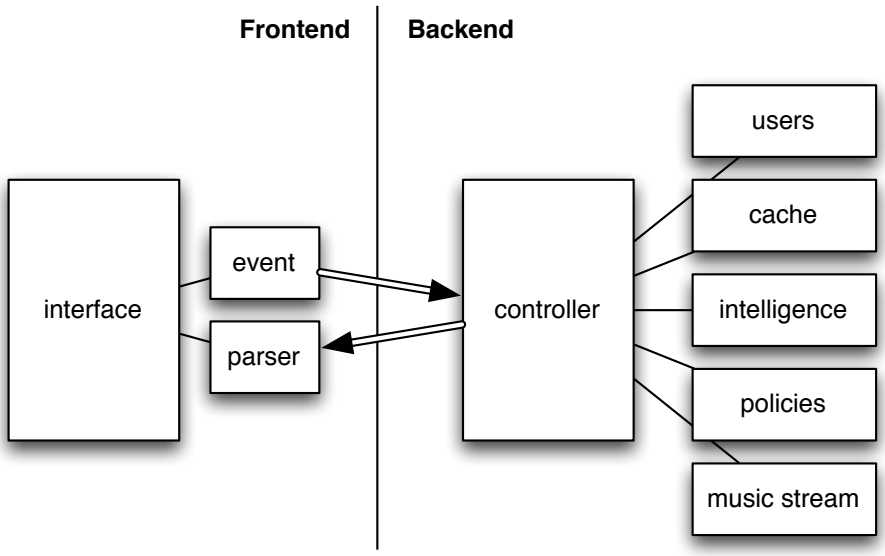

Figure 1: How LAMA works - simplified diagramm

The first part of the project consisted 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 2.1.1.

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.

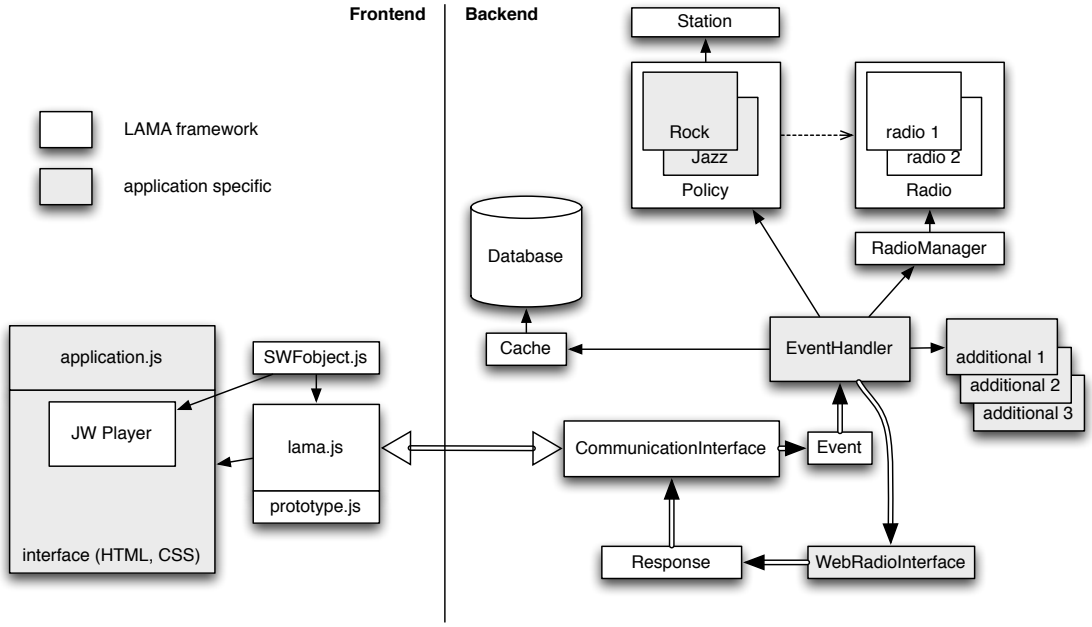

Figure 2: How LAMA works - diagramm

2.1 Server-side

2.1.1 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 the fig. 2 for a visual representation of what we explain here. For technical details about the

implementation, please refer to the online wiki[11]. We will cover the following subjects:

1. the document tree structure,
2. the session handling,
3. the event handling,
4. the radios handling,
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 frontend is in a separate folder, and might even be in a physically separated place. They are as much separated as it possibly could be. The only information the frontend has to know about the backend is the location of one script (CommunicationInterface.php).

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. These cookies identify the user

[11] `http://whizz.ch/doc/doku.php`

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 frontend, 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 songs 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.

2.1.2 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. 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**[12]. 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:

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*.

[12] `http://www.lastfm.fr/api/webauth`

(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 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 table 1.

Here is a description of the **last.fm radio protocol**[13]. 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:

[13] `http://www.lastfm.fr/user/dahnielson/journal/2007/08/07/8nrm\_last.fm\_radio\_protocol`.

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.

(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* : GET 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* : GET 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[14] 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>
```

[14] http://www.lastfm.fr/api

```
10    </user>
11   </friends>
12  </lfm>
```

2.1.3 Implementation

As said before, 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.

2.2 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[15], which is very useful for AJAX communications 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.

[15] `http://www.prototypejs.org/`

One of its very core functions is the event handling. It listens to different types of events, 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.

2.3 Communication between the client and the server

The client-side and the server-side of LAMA communicate via AJAX (Asynchronous Javascript and XML) requests. The Prototype framework[16] 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:

```
1 this.sendRequest = function() {
2  new Ajax.Request(server, {
3
4   //method can be the HTTP method GET or POST
5   method: "post",(
```

[16] http://www.prototypejs.org/

```
6
7   //the parameters that will be sent to the server
8   parameters: this.param,
9   onSuccess: function(transport) {
10
11    //object specific to LAMA that will parse the response
12    //the parameter ALLOW_SERVER_JS is also specific to
        LAMA
13    new ResponseParser(transport.responseXML,
        ALLOW_SERVER_JS); },
14   onFailure: function(){
15    new ErrorLog("something went wrong with an AJAX
        request"); } });
16 }
```

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:

```
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/97
      b314710d3b91bc10c345993473f580.mp3]]>
5  </track>
6  <content>
7   <element id="artist-current"><![CDATA[Midnattsol]]></
      element>
```

```
8    <element id="title-current"><![CDATA[Unpayable Silence
        ]]></element>
9   </content>
10  <actions></actions>
11  <recall>1</recall>
12 </response>
13 </lfm>
```

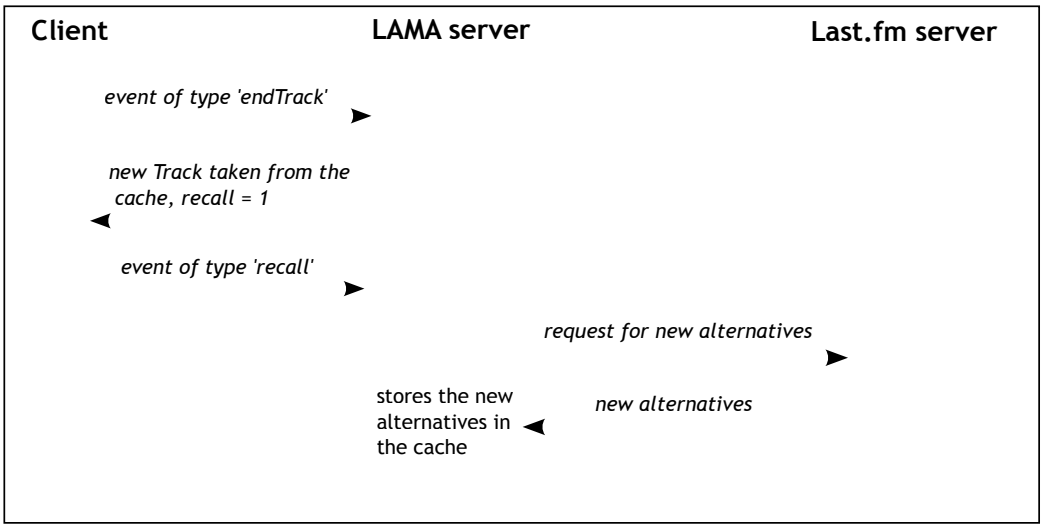

Figure 3: Use of the 'recall' parameter

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 3.

3 My application: TraceTrack

This third section talks about the application that was built on top of LAMA, to demonstrate its possibilities and to create a real music recommender interface. It is called TraceTrack, because it keeps a visual trace of all the songs that the user listens to. First we'll see what the key concepts behind TraceTrack are, then we will see how the interface was made, how the interaction works. In the end we'll have some insights on how it works on the backend part.

3.1 The idea

Here are a few key ideas that drove the building process. The application's interface should:

1. be simple to use,
2. be very modular,
3. be visually attracting,
4. have a main "channel", as well as several alternatives,
5. have a central music player, with the alternatives floating around,
6. keep a visual trace of what the user listened to.

The application has been named after the last point, because it keeps somewhere on the screen all the songs that were played by the music player.

At first, the application was proposing totally random recommendations. It was more some kind of a showcase of what could be accomplished with the framework than a real, useful, recommender system. After that, a second version, visually identical, has been developed, with a meaning behind the recommendations.

Here are some thoughts on how the recommendation process works in the final version of TrackTrace:

1. it should stay as simple as possible,

2. it should take heavy profit of Last.fm recommendation features,
3. without the user's action, it should continue to play the same channel,
4. it should propose 2 alternatives to the current song that is played,
5. one of the alternatives should be directly connected to the song (same tag or similar artist),
6. the other alternative should be connected to the user (his library, his recommendation, or his neighbour's recommendations).

It is very simple, because there are alway two alternatives, and the choice of the recommended songs are given by Last.fm. It would have been very interesting to build a much more powerful and original recommendation engine, but this was beyond the scope of this project.

3.2 Interface & user interaction

Fig. 4 presents the visual appearance of TraceTrack's interface. The chosen colors are:

black for the current music player and the current channel. The font color is thus different, too: it is white instead of black. This is a nice distinction between the current channel and the alternatives.

blue for the alternative associated with the user. Why blue? It is a light blue, quite neutral. This alternative is only influenced by the user, not at all by the song, therefore it is more general, and permits to totally change the style of music.

orange for the alternative associated with the song. It is a little more outstanding than the blue recommendation, because it seemed logical to emphasize the recommandation closely related to what is being played than a more general, user-defined recommendation that has nothing to do with the current song. On the other hand, it isn't possible to totally change the kind of music you're listening to with an orange alternative.

There are four places where an alternative can be: "top left" (not seen on fig. 4), "top right"—the orange alternative, "middle"—there's always the current channel, and "bottom"—the blue alternative. The current channel stays always in the "middle" place, but the alternatives can be anywhere in the three remaining places. The place is chosen at random, so there's no real

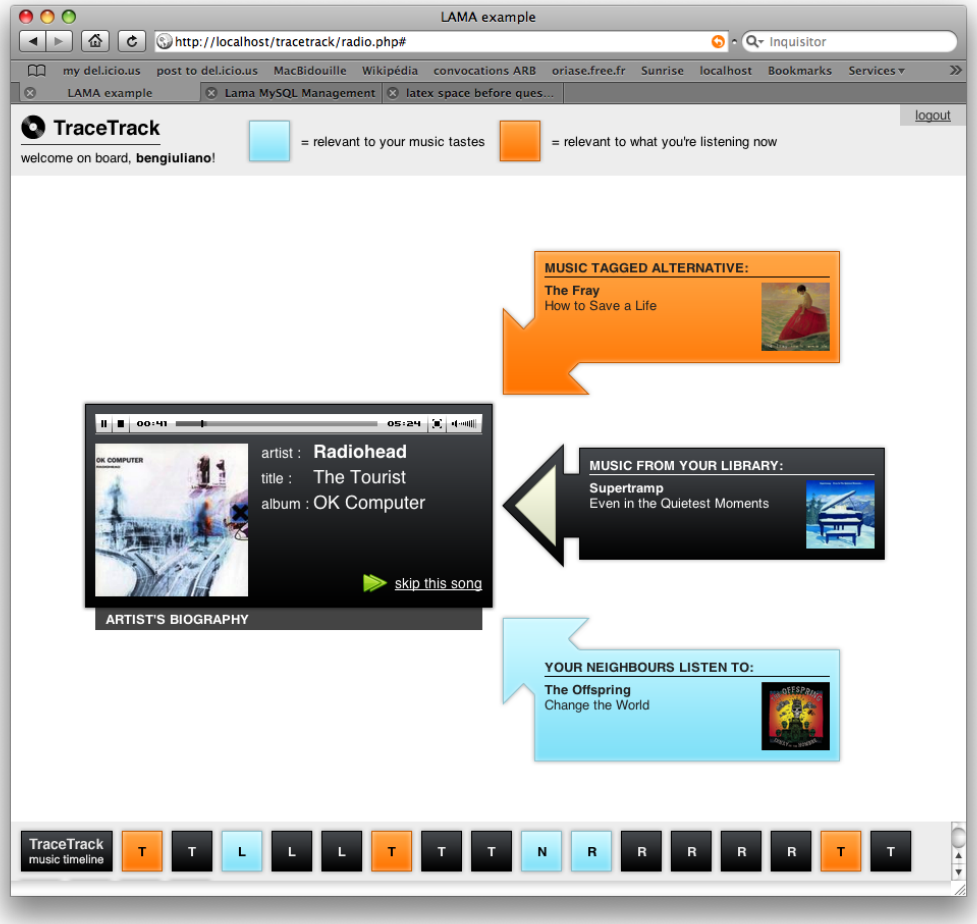

Figure 4: The TraceTrack interface.

meaning associated with the place of an alternative, but we don't want the user to know in advance where which alternative is going to be, because the way the alternative is chosen is changing every time, even if there's two main categories, *user* and *song*.

The alternatives and the current channel are presented in arrows, that point towards the music player. It reinforces visually the fact that these blocks are connected with the music player.

There is a album cover picture for all the songs presented at a given time to the user. This piece of visual information is pretty important, because that allows the user to recognize immediately an album he already knows from elsewhere, or that he listened to in a previous session in TraceTrack.

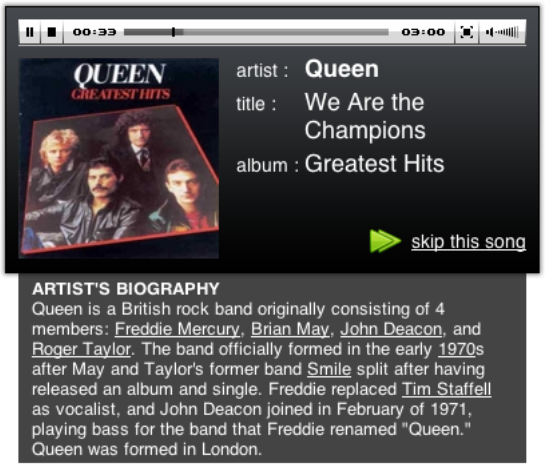

Figure 5: Displaying the artist's biography.

It is also possible to display the current artist's biography, by clicking on the bottom of the player (fig 5). Initially, the intent of this feature was to capture the action of clicking on the button "Artist's biography" and to interpret that as a implicit sign that the user likes very much the current song, and recommend the next songs partially based on that fact. But finally it was a bit too complicated to continue in this direction.

The TraceTrack special feature is the visual trace of all the song the user listened to. Everytime a new track is sent to the music player, it automatically adds a little square in the line on the bottom. The colors of the traces match the color of the alternatives. The letter in the squares is the policy that was used to get the song. It can be:

- T: from a tag,
- S: from a similar artist,
- L: from the user's library,
- R: from the user's personnal recommendations,
- N: from the neighbour's library.

The user can get the meaning of any letter by placing his mouse cursor above the letter.

The user can interact the following ways with the interface:

show biography as explained above, the user can display the biography of the current artist by clicking on the dedicated button. The biography then slides down.

alternative whenever the user clicks on an alternative, it becomes selected, and at the end of the song, the selected alternative is played. The visual indication that an alternative is selected is a white arrow (can be seen on the current channel in fig 4).

music player controls These events are handled by the flash music player, they aren't spread further away

skip This has the same effect as the end of a song. It immediately replaces the song that is played with the track in the selected alternative or in the current channel (if no alternative was selected). The current channel is changed if an alternative was selected, and the next song in the current channel as well as new alternatives are created. The visual appearance of the wole process is the following:

1. The song is immediately changed in the music player, as well as the metadata associated (title, artist, album, cover).
2. The arrow from the alternatives fade out from the interface, as well as the current channel's next song—but not the black arrow.
3. The new current channel's next song fades in, as well as the new alternatives.

logout There's a small logout button on the top right. When the user clicks on "logout", the application is shut down and he's taken to a page displaying the LAMA logo.

3.3 Behind the scenes

TraceTrack relies of course on LAMA, but also on several additional files. On the frontend, there's an additional javascript file, `application.js`, containing several functions that handle the interface. These functions are built with the help of the script.aculo.us[17] visual effects library. It expands the already used by LAMA library prototype.

These functions handle the display of the traces, the display of the alternatives when they fade in and out, as well as several other functions that are not used anymore in the current version of TraceTrack, such as changing the opacity of the alternatives.

On the server-side, a few additional files have been created (we only describe the most useful files):

[17] `http://script.aculo.us/`

MyEventHandler.class.php this is the TraceTrack specific implementation of the EventHandler class, describing how to react to events on the client-side.

SimpleInterface.class.php this is the TraceTrack specific implementation of the WebRadioInterface class, describing how to translate abstract actions provided by the EventHandler into concrete modifications on the interface.

Alternative.class.php this class is just a representation of an alternative. It contains information about where it is placed, what kind of policy was used to generate it, etc.

FullPolicy.class.php this class handles all the alternatives and the current channel, and connects them with the radios.

TrackProvider.class.php this class is used to generate the alternatives as well as to keep track of which channel has been selected.

policies five new policies have been created for TraceTrack. They are used to generate the alternatives, as it is explained below.

Now we'll describe how the recommender system works. The initial policy used for the first song is music tagged "acoustic". This is totally arbitraty, but anyway there has to be a beginning. At the beginning, and every time the track changes (after a click on skip, or the end of the song), two alternatives are generated as follows:

- The policy of the orange alternative, relevant to the song, is chosen randomly between two possibilities: `GenericTagPolicy` and `GenericArtistPolicy`. The keyword to instantiate these policies—respectively, the tag or the artist—is retrieved from the song that is currently being played. In the case of the tag, we need to make a request to Last.fm to get the top tags of the current track, and then we choose the first tag.

- The policy of the blue alternative, relevant to the user, is chosen randomly between three possibilities: `LibraryPolicy`, `RecommendationsPolicy` and `NeighbourPolicy`. The keyword to instantiate these policies is always the same: the user's username.

As one can see, it is in fact very simplistic. But it takes all it can out of the Last.fm system; it uses every type of basic Last.fm station. It would of course be very interesting to develop this part further, but it's enough to

give quite good recommendations (from a very subjective personal point of view).

This concludes our discussion on TraceTrack.

4 About the project

4.1 Difficulties

Several unsuspected difficultes arose during the developement of LAMA and TraceTrack. We quickly describe some of them:

- the Last.fm API infrastructure ended up being quite unreliable. To begin with, the infrastructure is not very solid: sometimes the server just doesn't respond, sometimes the requests return an empty response. The radio API has strange behaviors, too: a playlist request takes more than ten seconds sometimes. And not to talk about empty playlists.
- informations, metadata and such, requested from Last.fm, weren't always very acurate either. But this is not surprising, since Last.fm is a social, user based database. It's very difficult to get homogeneous results.
- the script.aculo.us library used for the TraceTrack interface is quite fragile as well. Sometimes, the visual effects simply don't play in the browser! Quite embarassing when you have to display an alternative. If there was more time left, this library should be replaced.
- organisational problems proved to be a great challenge during this project. This was a difficulty, but in the end it was very instructional.

4.2 The future of LAMA

A consequent part of this project was the developement of the LAMA framework. Developing software has no ending, it can always be futher improved. In this section, we will talk about some possible next steps in the development of LAMA.

The first improvement one would want to make to LAMA is making it more robust, more fault tolerant to API request errors. Handle exceptions is often a tedious part in software development, and with LAMA, it would be for sure a priority. Re-attempt to connect on failed requests, alone, would enhance the framework significantly. The frontend part of LAMA, especially `lama.js`, can also benefit from a little more robust error handling.

Another improvement would be to have an option to integrate the application even more tightly to the Last.fm audioscrobbling technology. Implementing the scrobbling would allow to play more with the Last.fm recommendation algorithm. This isn't very difficult to do, but it takes some more time.

On the complete opposite, it would be interesting to have another musical service provider integrated into LAMA, as it was designed to be more or less music provider independant.