Tagonto Project

Silvia Bindelli, Claudio Criscione, Mauro Luigi Drago

September 25, 2007

Chapter 1

Introduction

1.1 What is a TAG?

A tag is a simple term (although it could be composed by more than a single word) associated with an information of any kind. They were devised for purpose of classification and information retrieval, acting as keywords which can be used to identify resources. Each resource can have more than one tag associated: in their most common uses, tags are used to make clear both the content and the context of an information. There is yet another use for tags: personal tagging. Personal tagging is a far more ”natural” use case for tags: associating ”toberead”, ”tomorrow” and such transients (or less transients, like ”read”) tags to information in a way that only makes sense for the tagger.

Summing up, a TAG is a small (usually one word) piece of textual information, thus easily processable, associated with any other piece of information to specify its content or context, or defining some relative properties. TAG’s information is extremly unstructured and with little semantics, whereas one TAG can have almost any factual semantic content (as in the TAG ”Cute”), or a content depending on the context (as in the TAG ”Killer”, having completely different meanings in the contest of a ”Killer application” and of a Trial).

1.2 TAG systems in the internet

Tag systems are now a core part of the internet and one of the most (ab)used buzzwords of the Web 2.0. Some 2.0 sites were built around the idea of tags, del.icio.us () being the most famous: delicious is a ”collaborative bookmarking” website, where searches on the bookmarks database can be made using tags as their main driver. They both are also known as *folksonomies*, a concept which will be explained in next section.

Most of the usage of tags on the net is a ”composite” one, where tags are used inside a complex system: sites like YouTube and Flickr allow for searches on tags.

Often tools based on tag systems allow each user to visualize in a "cloud" all tags she used, highlighting with a bolder font the more frequent ones. This way the user can quickly reach all the information she tagged with a given keyword, and can immediately view contents more interesting to her. Moreover, this way of organizing tags suggests new words for her searches. An example of tag cloud is shown in figure 1.1.

Figure 1.1: Flickr Tag Cloud

Personal tags Personal tags are getting more and more popular on the internet, due to applications allowing the storage of personal tags locally and their association with any resource (like tagging file systems, mail tagging systems and so on). While this is a growing area, it must be noted that personal tags are usually not shared with others, and cannot be made sense of by anyone but the tagger.

1.3 Folksonomies

"Folksonomy" is defined as the practice of using tags in a collaborative way to annotate and categorize contents. The same concept is also known as "social tagging", but "folksonomy" better underlines the strong relationship to people: they imply metadata are created by contents' users and creators, and usually consist of a vocabolary familiar to them. This allows for information to be easier searchable and navigable.

The term "folksonomy" comes from the union of "folk" and "taxonomy", the latter being a subject indexing system.

One of the advantages brought by folksonomies is the empowerment of information retrieval capabilities of each user. Each user navigating in a tag system is able to view tag set of other users. This way she may find a user annotating contents in a way that makes sense to her too, and through this user's tag discover new related content. Moreover, the fact itself that a document is identified in a search by keywords assigned to them by users make it easier for it to be found, because the search will take place in a vocabulary well known to the searcher.

Folksonomies represent a very low cost way to add metadata to contents, making them more easily available to men and machines. But the metadata they allow for are *external*, in contrast with those usally provided by standards like the Dublin core. This implies that there is less control on their structure and correctness.

Indeed, folksonomies are often criticized because of their lack of terminlogical control, being freely developed by users, without the support of a shared and controlled vocabulary. Moreover, tags are not inserted in a logical structure which states objects and relation among them.

1.4 Tagonto support to Folksonomies

Tagonto's place in the context of folksonomies locates exactly here: Tagonto's purpose is to overcome problems which arise form the lack of an underlying structure exploiting all the help an ontology can give. Mapping tags onto ontology concepts allows for supplying a structure, improving their usefulness by offering related concepts (and with them related tags and then contents) when a search for a keyword takes place. This way both recall and precision get better, as explain in reserved section.

Tagonto then finds its collocation inside the idea of expanding folksonomies with ontologies.

1.5 State of the art

In this section we're offering a quick overview of what has already been done in the field of interaction between tags and ontologies. On the other side, we're not going to analyze the field of tags and of ontologies separately, because much has already been written about that.

What is clear is that not much work has been done till now in this area. Only a few efforts, coming out in the last months, show the interest that is emerging about it. People is now understanding how useful it would be to develop ontologies inside the context of Web 2.0, and some works are moving towards this direction.

1.5.1 Im Wissensnetz

Im Wissensnetz is an example of trying to find a collocation for ontologies in Web 2.0.

The problem it tries to solve is the developement of ontologies: at the moment there isn't a strong partecipation of users, who should be the more interested in them. But they are devoloped by models' experts, who don't know much about the knowledge domain they refer to. This seems quite far from what Tagonto is meant to do, as we will see later in this document.

SOBOLEO SOBOLEO (SOcial BOokmarking and Lightweight Engineering of Ontologies) is a tool, developed within the Im Wissensnetz project, which allows for tagging resources in the web using ontology concepts. It also lets its users to interact with the ontology, modifying the relations between concepts, their label and so on. It also has browsing features, which starting from concepts in the ontology lead the users to resources in the web tagged with the given concept. SOBOLEO also shows resources containing in their text a given searched term, just like any other search engine.

SOBOLEO offers functionalities far different from Tagonto's ones. In some way, we could say that it offers the "opposite" path than the one offered by Tagonto, which instead lets to map tags onto ontology concepts.

However, it seems an interesting point of view about how ontologies and tag-based syestems could usefully interact.

Chapter 2

TagontoLib

2.1 Introduction

TagontoLib is a java library, developed as a part of the Tagonto project, exposing facilities to map a tag on one or more concepts of a specified ontology. Mapping a tag onto a concept is not a trivial task. The natural language is by itself a complex object to analyze in an automatic fashion, especially since words can assume different meanings according to the context in which they are used. That's why we had to develop algorithms to generate the possible matches involving both syntactic and semantic checks. Furthermore, the need of doing some ontology reasoning and the strict performance requirements imposed by the necessity of working online (to serve real-time requests of users) lead us to develop a complex architecture we will explain in the following chapter.

2.2 Theory

Before describing in details the architecture of the library and how it works we have to define the theory that lies behind the task of mapping a tag onto a concept.

2.2.1 The Concept of Mapping

The main task of the Tagonto Lib, as we said before, is to map a tag on one or more concepts of a specified ontology, but until now we have not defined in detail what is a mapping. From a theorical point of view, a mapping can be defined as a relationship between a tag and a concept of an ontology. Obviously, since the natural language is ambiguous and the semantic of words depends on the context in which they are used, the multiplicity of that relationship is many to many, i.e. a tag can be matched onto many concepts and a concept can be matched onto many tags (fig. 2.1). In addition we must also specify what we mean using the terms tag and concept, in other words what are the sets onto which the mapping relationship applies. With the term *tag* we mean a vector of characters of any size, i.e. any word. The specification of *concept* instead is slightly more complicated, since by that term we mean any named concept in an

ontology (i.e. concepts that have been declared with a URI) and not anonymous ones (fig. 2.2).

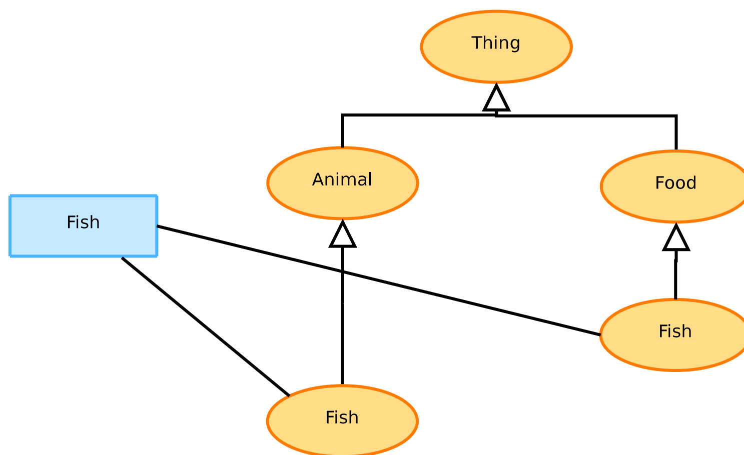

Figure 2.1: Multiple possible mappings

```
<owl:Class rdf:ID="Wine"> <!-- named concept-->
  <rdfs:subClassOf rdf:resource="&food;PotableLiquid" />
  <rdfs:subClassOf>
    <owl:Restriction> <!-- anonymous concept-->
      <owl:onProperty rdf:resource="#hasMaker" />
      <owl:cardinality rdf:datatype="nonNegInteger">
        1
      </owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="WineGrape"> <!-- named concept-->
  <rdfs:subClassOf rdf:resource="&food;Grape" />
</owl:Class>
```

Figure 2.2: Named Vs Anonymous concepts

The consequences of this choice, imposed by the mining technique we use to generate the matches, is that the set of concepts against which a tag can be mapped is smaller than the set of all the concepts declared in an ontology. However, we think that this limitation is not so critical, since this library was developed for a user-centric project and there are few users with the appropriate knowledge to understand a complex concept definition (since anonymous concepts have no name, the only way to give a description of the concept is to show its definition). Anyway, one of the weaknesses of this assumption is that names given to concepts are descriptive, i.e. the name of a concept summarizes the concept description, but this depends on who created the ontology and falls

outside this project.

With the definition of mapping we have given so far, as we have remarked when talking about multiplicity, a tag can be mapped onto one or more concepts and vice versa, but we have not defined what lets us distinguish between different mappings and, most important, which ones are better than the others. What lets us accoplish this task is the significance of a mapping (i.e. the weight assigned to the relationship, fig. 2.3). We will see in next chapter how those weights are calculated.

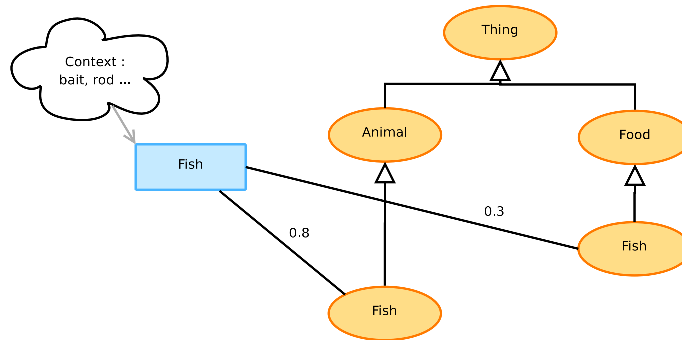

Figure 2.3: Multiple possible mappings with different weights

2.2.2 Mapping Generation

To generate mappings, both relationship tuples and their significance, TagontoLib uses two different types of heuristics:

- **Generative Heuristics**
  Their task is, given a tag and and an ontology, to generate the most significative mappings according to some metrics.
- **Choosing Heuristics**
  Their task is, given a set of mappings and an ontology, to modify the significance of the given mappings according to some metrics.

These two heuristics are then combined in a two step algorithm as shown in fig.2.4

Anyway, until now we have not specified how these heuristics really works, i.e. what are the metrics used. Mainly, we can divide metrics in two categories :

- **Syntactic Metrics**
  These metrics use only syntactic information to match a tag against a concept. This means that only the name of the concept and the tag are considered for the match, nor the context in which the tag was used neither semantic information residing in the ontology definition. Examples of these metrics are the usual text comparison metrics used in data mining, such as Levenshtein, Jaccard or Tanimoto metrics.

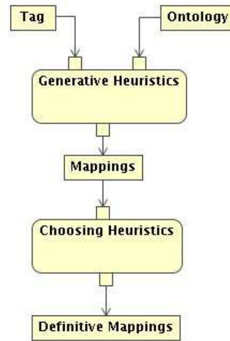

Figure 2.4: Main Algorithm

- **Semantic Metrics**
  These metrics use both syntactic and semantic information to match a tag againt a concept. The way we try to infer the semantic information depends on the particular method we adopt. For example, when using WordNet, we use an english vocabulary to search for synonims, hyponims and hypernims of a particular tag. Another example is the use of Google to search for the right context for a tag (we will se later how it works in details).

So far, we can categorize the methods we use to create mappings in a two space environment (table 2.2.2):

	Syntactic	Sematic
Generative	Exact Match Levenshtein Match Jaccard Match Google Noise Match	Wordnet Similarity
Choosing	Max Chooser Threshold Chooser	Links Chooser Friends Chooser GoogleChooser

Table 2.1: Heuristics categorization.

We will describe in details all these heuristics and how they are implemented in the architecture section.

2.3 Architecture

The architecture of this library was developed keeping in mind all the requirements expressed before and to make easy extending it to implement new features. At a high level of abstraction, Tagonto Lib has three main components (fig 2.5)

The Mapping Component constitutes the core of the library, while the other two components just expose facilities needed by the main component to accomplish its task in an efficient manner.

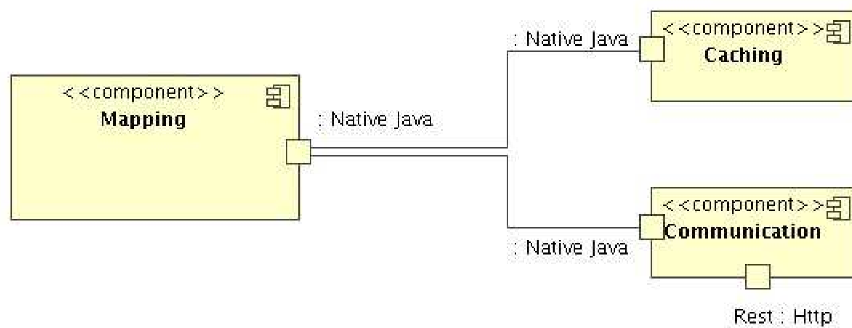

Figure 2.5: Main Components

2.4 The Mapping Component

The main task of the Mapping Component is, given a tag and an OWL ontology, to generate all the possible and significant mapping between the tag and one of the concepts of the ontology. The most important classes belonging to this component are shown in fig. 2.5.

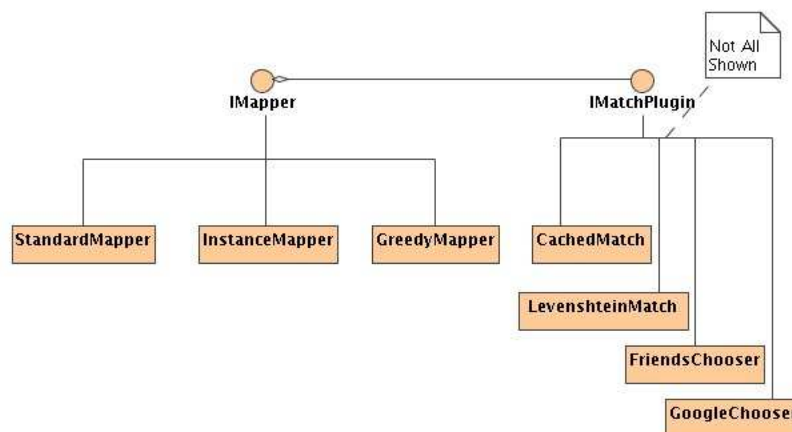

Figure 2.6: Mapping Component Main Classes

From a high level of abstraction we can define the main elements in this way :

- **IMatchPlugin**
  the implementation of a mapping heuristic
- **IMapper**
  the implementation of a mapping strategy, i.e. the specific algorithm we use to generate mappings. In other words, a mapping strategy is a collection of mapping heuristics and defines how we combine them, i.e. which is the order the plugins are called and how we merge their results.

Strategies	Standard Greedy Instance
Heuristics	Cached Match Exact Match Levenshtein Match Jaccard Match Google Noise Match Wordnet Similarity Link chooser Google chooser Threshold chooser Max chooser

Table 2.2: Implemented Mapping Strategies and Heuristics

As show in figure 2.6, we provide 3 main mapping strategies and 11 mapping heuristics.

In the following section we will describe all of them in detail.

2.4.1 String Match Plugins

Within this category we group all the plugins that generate a mapping using only syntactic information and that specifically use only string comparison metrics to generate the best mappings. Under this category we can list :

- **Exact Match**
  Does a string comparison ignoring case between the specified tag and the name of every concept declared in the ontology. If the two strings matches, a new mapping is generated with significance 1.

- **Jaccard Match**
  Does a string comparison using the Jaccard distance measure between the specified tag and the name of every concept declared in the ontology.

- **Levenshtein Match**
  Does a string comparison using the Levenshtein distance measure between the specified tag and the name of every concept declared in the ontology.

2.4.2 Cached Match

This plugin just queries the Tagonto cacher (we will describe it in the following sections) checking if a mapping for this tag has been already generated. The need for this functionality has been imposed by the strict perfomance required by an online-use.

2.4.3 Wordnet Similarity Plugin

This plugin uses a component taken directly from the XSom project. All it does is to invoke the imported component as many times as the number of concepts declared in the ontology. Then what the imported component does for every invocation is tokenizing the tag and the concept if possible, using wordnet to find synonims, hyperonims and hyponims and finally comparing the tag string and the found words with Jaccard and Levenshtein metrics. For further and more detailed information on the component imported from XSom see the XSom documentation.

2.4.4 Google Noise Match

This plugin is not really a mapping plugin, since it does not generate or modifies any mapping. We can define it as a facility for mapping, since it's task is trying to correct some misspellings with the use of Google. Basically, what it does is searching with google the tag and analyzing the response page, searching for a *Maybe Did You Mean* suggestion from the search engine. If a different word has been suggested, it calculates the noisyness of the original tag comparing it with the word suggested by google and returns both as a result. Then the invoking mapper, according to the strategy it realizes and the noisyness calculated, decides to repeat all the mapping process for the new tag returned by google or not.

2.4.5 Max and Threshold Choosers

The task of these plugins is modifying a collection of mappings. The max chooser finds the highest mapping significance in the collection and removes all the entries having a lower value. The threshold chooser instead removes from the collection all the mappings having a significance lower than the specified threshold.

2.4.6 Friends Chooser

This plugin uses correlated tags and semantic informations derived from the ontology to disambiguate mappings. The first step this chooser takes is asking to TagontoNet services the tags correlated to the original tag and mapping every retrieved one with a Greedy strategy. Then for every original mapping, it modifies the significance using as metrics the *linkness* of the concept onto which the tag was matched in the original mapping with the concepts onto which correlated tags were matched (see fig.2.7 and fig.2.8). In other words, the more the concept onto which the original tag is matched is connected to concepts mapping correlated tags, the more the significance is raised and vice versa. The theorical basis that lies behind this heuristic is that if a mapping if correct (i.e. matches the tag onto the correct concept) then the concept must be connected to concepts mapping correlated tags (i.e. there must be ontology properties having as range the *original concept* and as domain the *correlated concept*, and vice versa).

safdasdf

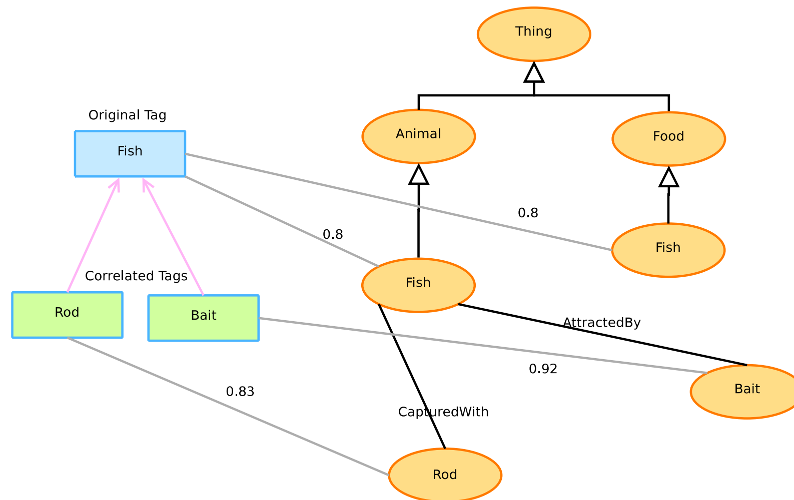

Figure 2.7: Mapping before correlated analysis

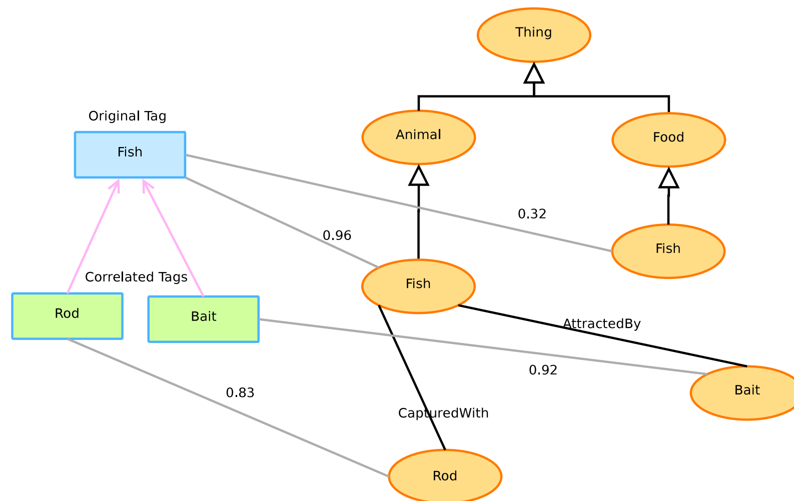

Figure 2.8: Mapping after correlated analysis

2.4.7 Google Chooser

This plugin uses Google search and semantic information derived from the ontology to disambiguate mappings. Essentially it acts the same as the Friends Chooser, the only difference is in the way it retrieves correlated words. Instead of invoking TagontoNet services and so retrieving correlated tags, this chooser uses the tag as a search query using Google and analyzes the first N results (i.e. the N html pages with the highest rank according to Google search metrics). How the analysis of these pages is done depends on the content of the page. If in the header of the html page the plugin finds the keywords meta information, it uses this list as the correlated words, otherwise a mining analysis is done on the page content (i.e. first removing all html tags, then using text mining and

indexing techniques to extract the most representative words).

2.4.8 Greedy Mapper

The Greedy mapper implements a greedy strategy for mapping, i.e. the algorithm terminates in only one pass. This means that the Greedy strategy only uses a subset of the possible plugins, in particular only syntactic plugins and the wordnet plugin. Heuristics using information embedded in the ontology (e.g. Friends Chooser and Google Chooser) are not embedded in this strategy since they need reasoning support, i.e. the completion time is too high for the strategy to be efficient. This mapper has been developed to efficiently map a tag onto a concept, and is used by other strategies when collateral mappings are needed to execute a particular heuristics (e.g. Google Chooser or Friends Chooser).

2.4.9 Instance Mapper

The instance mapper uses informations derived from instances of concepts to generate mappings. Since the standard behaviour of TagontoLib is to consider only concepts (their name and the semantic information that can be inferred from the ontology), we had to create this new mapper to take into account also instances. When this mapper is invoked, it uses syntactic heuristics (at the time of writing it uses only string comparison metrics since instances can be several and performace would be degradated) to check wether or not the tag can be matched onto one or more instances. If matches with high significance are found, for each match a new mapping is generated mapping the specified tag onto the direct concept (the most specific class the invidual is an instance). In other words, what this strategy does is not mapping a tag onto an individual, but mapping the tag onto the class whose the instance is an individual.

2.4.10 Standard Mapper

The Standard mapper realizes a complete strategy for mapping, all the heuristics we defined before are used. The first step of this mapper is invoking the greedy mapper to obtain a temporary mapping for the tag. Then it uses the instance mapper to generate new candidate mappings and finally the Google Chooser and the Friends Chooser to modify the significance of the greedy mapping as described before.

2.5 The Caching Component

Since one of the most important requirements for TagontoLib is being efficient and suitable to be used in an online fashion, the Caching component holds an extremely important role. The task of this component is not only caching all the mappings generated (not only the one generated for an explicit request but also *collateral* ones, e.g. mappings for friend tags or Google friends) but also caching information about loaded ontologies. Since semantic heuristics use information that can be inferred from the ontology specified for the mapping and ontology reasoning can be pretty slow, we had to precompute many of the results needed

by the heuristics at ontology loading time. Using this trick we can save much computation time during online-use, i.e. when TagontoLib receives a request to map a tag. At the time of writing, the caching component uses a Jdbc end point to store ontology informations and both an RDF enpoint and a Jdbc endpoint to store mapping information (fig.2.9). However, the jdbc endpoint holds a more important role if compared to the RDF endpoint, since the RDF enpoint is just a backup copy of the mappings and it's not used to read cached information for performance reasons.

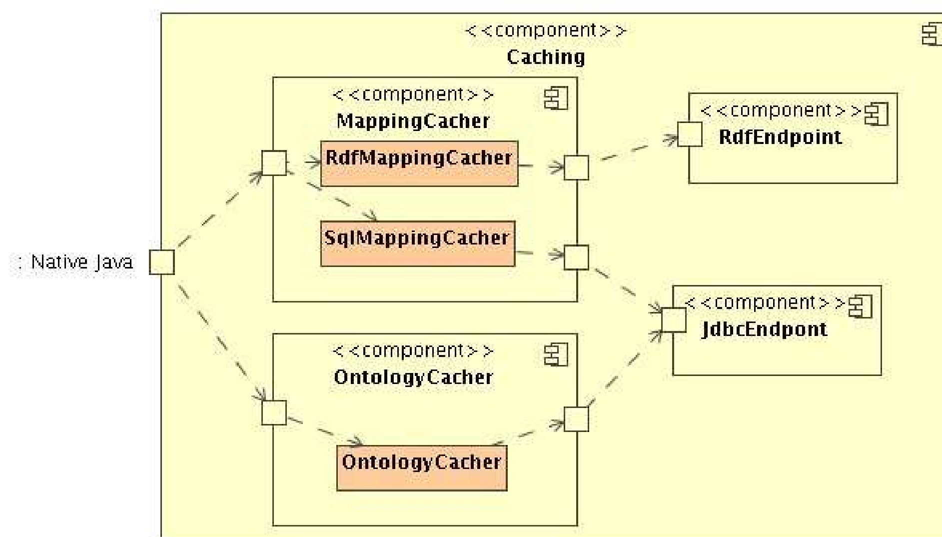

Figure 2.9: Caching Component architecture

Since the information stored about the ontology are a lot, the table 2.5 describes in detail what kind of information every database table keeps :

2.6 The Communication Component

The communication component was developed to enable the use of TagontoLib without accessing it with java. At the time of writing, the communication component realizes an Http proxy that enables the use of the main features of TagontoLib. The communication protocol implemented is based of the REST paradigm, for more details on how to call TagontoLib services and the format of the response see the internal documentation of TagontoLib avaible inside the distribution.

Table	Information Stored
ontologies	keeps the list of ontologies loaded into the system and information about the last time ontology information was refreshed
ontology_concepts	keeps the list of all named concepts defined in the loaded ontologies
ontology_declared_properties	keeps the list of all named propertied defined in loaded ontologies
ontology_properties	keeps statistics about properties, i.e. linkness of ontology concepts
reachable_instance	keeps information about individuals reachable for a specified named concept and property
subclass_of	keeps information about concept hierarchy for loaded ontologies
domain_range	keeps domain and range information for ontology properties
instances	keeps the list of all instances of every concept declared in loaded ontologies
mappings	keeps the list of all mapping generated
mapped_by	keeps information about which strategy was used to generate a mapping.

Table 2.3: Database Tables

Chapter 3

TagontoNET

3.1 TagontoNET - Tag retrieval engine

TagontoNET (TNET from now on) is a modular and plugin-based software for the interaction with tag systems on the internet. TNET offers two main services: a search-engine, able to retrieve resources associated with a given tag and a Friend-fetcher, able to retrieve tags often associated with the same resource on a given tag.

It's not possibile to give a general overview of these methods, since their internals are completely different for each plugin, but we may sum up the status of the present development. Currently, the plugins powering the *search engine* are using at least three different methods:

- API calls, where available.
- RSS feeds, where available.
- Scraping: web pages are fetched and parsed using regular expressions to obtain needed informations.

While most API calls will return tagged contents according to their relevance or popularity, it's hard to do the same things with scraping, and almost impossibile with RSS feeds.

Tag friends are obtained in a very similar fashion: either the APIs of the 2.0 website offer a proper method or webpages are parsed to retrieve tags. Either way, we were not able to obtain proper mathematical data to execute statistical analysis.

3.1.1 Tag representation system

Most systems use an internal representation of tags where every tag is a single tuple in the form of <TAG, RESOURCE, USER>. This form is suitable for ownership queries and allows for easy research in the database.

It's easy to see how the USER field can be an additional information source able to further enhance the semantic value of a tag: the same tag might have various meaning where used by different people, due to many reasons (like language differences, ambiguities, even irony).

While this is true on a single system, where the user is clearly identified and tracked, it's difficult to have the same idea implemented in an entirely different setting as TagontoNET.

During the design stage of TNET, we evaluated the possibility of taking into account both the TAG and the tagging user, assigning weights according to some heuristics yet to define. This idea was then abandoned due to two main problems:

- Some sites supporting tags do not store tagger's informations on their database, or the data are not accessible from outside. This would have caused a mismatch between sites supporting user's identification and those unsupporting it.
- It's not possible to track the user between different systems - while some heuristics can be inferred, as the nickname and the domains of the tagged resources, it's not likely they would achieve good results. This would greatly reduce the uselfuness of the information in the intended task.

TNET represents each tag as a simple string where the associated systems are properties of the given string, not identifiers. The internal PHP representation is a class serializable as <TAG, SYSTEMS>, thus allowing for easy navigation.

3.1.2 Architecture

TNET can be divided, from an high design standpoint, into three main parts.

1. A PHP library endpoint, able to offer its services through simple method invocations.
2. A RESTful web service endpoint.
3. A plugin system, where new plugins can easily be loaded at runtime.

The plugin system

TNET's plugin system is as simple as powerful: each plugin is composed by two required files and an arbitrary number of support files. The two required files are

- The *config.php* file, used to set each and everything config value needed by the plugin.
- The *manifest.php* file, containing the Plugin class.

Config The config file has to include at the very least an Unique Number, identifying the plugin, and a set of basic information about the plugin. The config file can then include any needed configuration like API key, username and such.

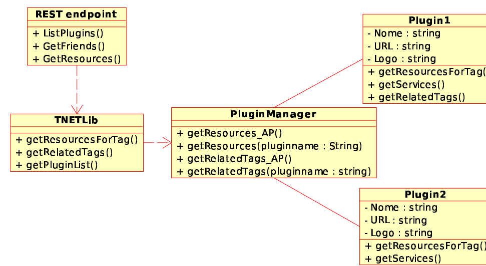

Figure 3.1: High-level architecture diagram

Manifest The manifest file contains the Plugin class, which in turn has to extend the class TagontoNetPlugin. The only needed instructions external to the Class definitions are the inclusions of the relative config.php file and of any needed libraries, plus the invocation of the static method of PluginManager plugin_subscribe, using as a parameter the plugin class' name.

Class' constructor has to setup the class properly, and it's the only required method along with getServices(), which will return an Human readable representation of the services offered by the plugin.

It's important to stress how plugin's code could theorically include any language and business logic, from using external webservices to running system commands. Thus, it's crucial for the developer to take into account the maximum execution time and memory limits of the script: since all the plugins are going to be activated on the same run, resources will have to be shared. Due to the inability of PHP to handle threads, time is an important factor when executing a plugin: this is why methods for querying just one plugin have been developed. Anyway, during the execution of a getRelated or GetFriends query, most of the times all the plugins will be activated (as in the case of TagontoLib) since the invoker has not a priori knowledge of the available plugins: extra care has to be taken in the coding of getFriends methods.

To offer a service, a plugin has to implement one of the methods described in the TagontoNetPlugin, with the correct returning type. Note that, since PHP is not a strongly-typed language, type consistency has to be checked by the developer. Once the method has been implemented, the PluginManager class will be able to detect it and use the plugin while answering a query for the given service. Once a query has been performed, plugins' results will be merged into a single result set and returned in a structured form (if TNET has been used as a PHP library) or as an XML document for the REST interface. See the WebService interface for more details.

RESTful endpoint

The REST interface is not actually a full REST implmentation, since it's not following the resource based paradigm: we are using the REST to describe a simple web service usable without complex SOAP methods and without a WSDL description.

Retrieving tagged content Tagged contents can be retrieved using the GetResources method. The result set will be organized in an array of Resources,

where every plugin is a Resource, containing basic resource-related information and an array of Results. Each Result is a tagged content, complete with its own url, title, description and shown content. Url and shown content may or may not be different, accordingly to the application logic of the plugin generating the result: the URL is the link to the resource, while the Shown Content may be any human readable representation of the resource. The Flickr plugin, for instance, will return the full page link as the URL and a thumbnail version of the image as the shown content. The Type of the Result serves this precise purpose, and can be used by the web service's consumers to identify textual (0) or image-based (1) content in the showncontent field. Management of the type field is completely handled by plugins.

Invocation To retrieve a tagged content, the GetResources endpoint must be invoked using the GET method with the following parameters:

- **tag** : the TAG for which resources have to be retrieved

pl : the plugin to be used. If pl is not submitted, all plugins will be used

Response The response of the server is always an xml document with the following syntax :

```
<tagonto requestSatisfied="true">
  <resource>
    <name>$PluginName</name>
    <url>$http://plugin.url</url>
    <logo>$http://pluginlogourl.any</logo>
    <results>
      <result>
        <url>$http://url.with.link.to.the.results</url>
        <showncontent>$http://data.to.show.to.the.user</showncontent>
        <title>$Title of the result</title>
        <desc>$Some description</desc>
        <type>[1-2]</type>
       </result>
    </results>
   </resource>
</tagonto>
```

Both the result and the resource elements can be repeated as many times as needed in the output.

In case of an error, the XML will be the standard error reporting XML described in this document.

Friend Tags Friend tags will be retrieved using all available plugins by default.

Invocation To retrieve a tagged content, the GetFriends endpoint must be invoked using the GET method with the following parameters:

- **tag** : the TAG for which resources have to be retrieved

- **pl** : an optional plugin to use instead of performing a global search

Response The response of the server is always an xml document with the following syntax :

```
<tagonto requestSatisfied="true">
  <tag label="$friendtag">
    <source>$PluginName</source>
  </tag>
</tagonto>
```

Obviously the *tag* tag will be repeted for each friend discovered. The *source* tag can be used to provide weights for each plugin or to power some heuristics. Tags are ordered on relevance - wherever possibile - and source system.

Error syntax Should TNET encounter any error during execution, it will report with an XML with this form:

```
<tagonto requestSatisfied="false">
  <error type="errortype">Error message</error>
</tagonto>
```

Type can have values fatal, warning or normal or even be missing. The error message will be in human readable form.

Chapter 4

Interface

4.1 Interface Overview

The choice of implementing a web interface is due to two main reasons. One of course is our desire to share our tool, making it easily accessible to everyone. The other one is a matter of coherence with respect to the aim of the tool itself: it seemed to us a natural choice the web for a tool which deals with topics such as ontologies and tags, the latter in particular being one of the more popular features of Web 2.0.

The interface has been designed as simple (and as readable) as possible. It is mainly divided into two parts horizontally: the upper part is related to the web search engine tag-based (TagontoNET), the lower part is dedicated to the ontology related results(Tagonto). Despite this division, there is only one search field, on the right of the top menu bar. The idea underlying Tagonto is to exploit the support of ontologies to emprove the searches of people in the web, giving a structure to something which, for its nature, doesn't have one, like tag-based systems.

The keyword typed in the search field will be both searched among tags in the web and in the ontology. Let's see now in details the two interface areas.

4.1.1 Web results area

The results of the search in the web are shown in the top area. They are organized in tabs, one for each web site. As explained in other sections, Tagonto supports the search in as many tag-based systems as the user prefers, being developed modularly. Each new system can be easily added to TagontoNET as a Plugin. The interface reflects this modularity through the use of tabs: every new plugin will be shown in a new tab, with no need to change something in the interface.

Results for each site are loaded just after a tab has been selected: we chose not to load them altogether because it's a computationally expensive task and it would have taken too much time. Moving from tab to tab, *part* of the resources tagged with the searched keyword in that site will be shown. I said "part" because we decided to reduce the number of results shown: they would have been too many, making the page less readable.

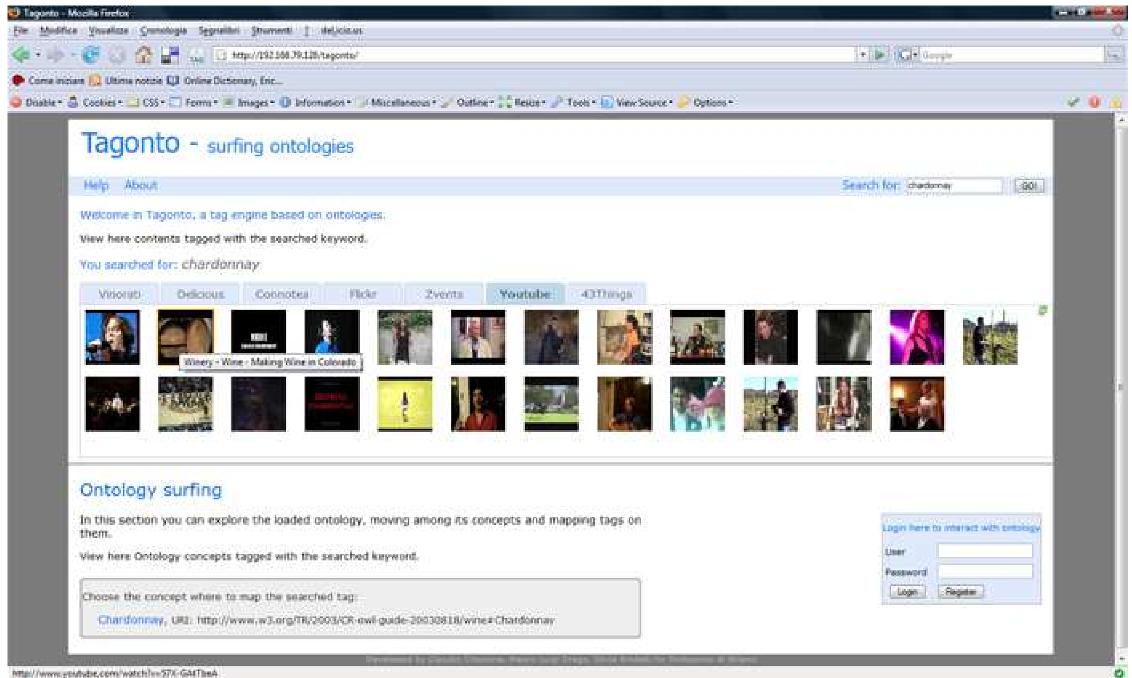

Figure 4.1: Screenshot of the interface

The results are shown in different ways, depending on their nature. For textual information, a link to the source is shown, together with a short description when available. For pieces of information from Flickr and Youtube, instead, a small preview of the picture (or video) is shown. The preview is itself a link to the resource, and its title is shown with a tooltip text. The kind of content is indicated by the Type of the Result (see section about Plugins in TagontoNET).

4.1.2 Ontology area

Let's observe now the lower part of the page. It offers functionalities linked to ontology navigation (or ontology "surfing", as we called that). Tagonto allows for the loading of different ontologies from the configuration side. The ontology used in this implementation is the wines' one, available at http://www.w3.org/TR/2003/CR-owl-guide-20030818/wine . When a keyword is searched, a grey box will appear in this area. This box lets the user choose among the ontology concepts retrieved by the reasoner the concept on which to map the searched tag. In other words, the concept in the ontology that better maps the searched keyword, from the user's point of view. This box is called "disambiguator".

Once a concept has been chosen, it will be shown together with some information about it. In detail: the concept's URI, the list of tags already mapped on that concept, the list of instances of that concept in the ontology. All this information will be included within a single box, with an orange border.

Together with the box representing the concept which is the object of the mapping, a group of other similar boxes (but with grey border) will be shown, each showing a concept linked to the given one by a certain relationship in

the ontology. These concepts are grouped in 3 macro-boxes: Sub-Concepts, Super-Concepts and Related Concepts.

The name of each concept is a navigable link which allows to navigate through the ontology: clicking over the name of a related concept, this will be shown in an orange box together with its related concepts and so on.

The information shown about the related concepts are similar to those shown for the current concept, but they also present the name of the relation which connects them to it.

The list of tags related to each concept is navigable too. Clicking over one of these tags a search for it will start, both in the web and in the ontology.

This feature particularly enables the support of the ontology to the search. The other tags mapped on the choosed concept and those mapped on the related ones can suggest to the user how to emprove its search, or a new direction where to develop it. Clicking over this tag a new search begins immediatly, with no other user action.

The istances' names are navigable too: they are treated as tags, and clicking over them a new search will start, both in the web and the ontology.

4.1.3 Personal area

Logging into the personal area through the blue box on the right, a registered user will have one more functionality available: she will be allowed to map a searched tag on an arbitrary ontology concept, if none of those offered in the disambiguator box satisfies her. In practice, a link on the bottom of the disambiguator box will open a new box containing a list of all the concepts in the ontology. A click over one of them will map the tag on it. From then on, the tag will be counted among the others mapped on that concept, and it will be assigned a weight (just what happen with a traditional mapping, see section about ontology mapping).

The login area is a really basic one. To register, the user has to enter a user name and a password and then click over the register button. From then on, she will be able to login typing them and clicking over the Login button. To logout, she will have to click over the logout link.

4.1.4 Configuration side

The configuration interface has been devoleped to make Tagonto's deployment easier.

Main page contains two link: one to set the ontology to be used, the other one to configure other properties of Tagonto, such as its databases.

To load an ontology, its URI is required, together with the location of the end point of Tagonto where to connect it.

The configuration of the other Tagonto properties is guided through a list of fields, and automatically generates a configuration file.

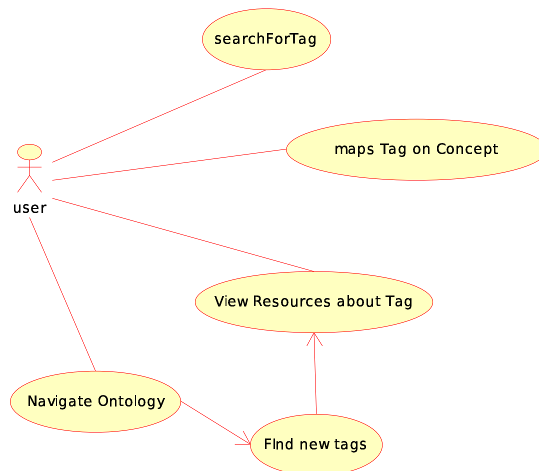

Figure 4.2: Interface Use Case

4.2 Implementation

4.2.1 Used technologies

Tagonto being a web application, its interface has been mainly developed in html, with the support of CSS stylesheets for graphical features.

XSL stylesheets have been used too, to translate into html the results of the web search, which are supplied in XML by TagontoNET.

The server-side part of the interface has been developed in PHP5, a reflective programming language. PHP5 allows object-oriented programming, which has been widely used within the implementation.

Another technology used in the implementation of the interface is AJAX. The choice of AJAX has been made to make the application quicker. As previously said, both the ontology reasoner and the web searcher are quite slow, dealing with very time-consuming tasks. To avoid the reload of the page at every query of the user, AJAX has been introduced, allowing the reload only of the piece of page affected by the action.

The use of ajax in practice consists of the use of some javascript libraries (mainly prototype.js and gwidgets.js, the latter used to generate the tabs view in the web results area). Using this approach, every content of the application is shown within the same page (only the menu bar and the configuration part will lead to different pages).

4.2.2 Interface structure

Except for the documentation pages and the configuration side, which are developed in a different way and are each included in an indipendent folder (*doc* and *config*), all the output of Tagonto is shown in the index page.

Its structure is given by divs, which represent the "web" and the "ontology" part, as previously defined.

The java functions called by Tagonto on user input are all collected in tagonto.js file, in the *js* folder. *js* folder contains all the javascript files, includ-

ing the libraries used to obtain ajax effects: prototype.js, gwidgets.js, base.js and effects.js. tagonto.js' functions are called through "onclick" html attribute, while param are passed them through php insctructions when available from the application, or directly through proper javascript methods when typed by user.

The shown output is put together in files located in the ajax folder. Each of them is invoked by the functions in tagonto.js, which pass them parameters using the GET http method. Then php files in *ajax* folder use methods and function made available by the other parts of Tagonto to show results.

They interact mainly with TagontoNETREST for the web part, to get results coming from the plugins, and with ReasoningManager and other php classes included in the *classes* folder for what concerns the ontology part.

In the *classes* folder, "manager" classes manage different aspects of the interface, while those classes with a name which refers to ontology are used to represent different elements of the ontology, where information about them are store when they are received by the engine.

ReasoningManager offers two main services, getConceptsByTag and mapTagOnConcept, respectively offering retrieving and mapping functionalities.

LoginManager offers services related to the login panel, used to register a new user and to manage session of logged users.

SearchManager offers services related to the "web part" of tagonto.

xslTranslator associates the stylesheet ResultsRendererPhp incuded in folder XSL.

Other php classes, as said, represent element of the system, mainly ontology elements.

Lib folder contains web plugins, together with their manager (see section about plugins for further information about this part).

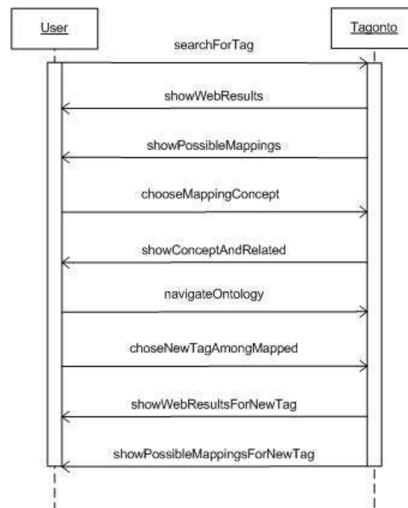

Figure 4.3: Interface sequence diagram

Chapter 5

Final considerations

5.1 Recall and precision

Measuring recall and precision of a folksonomy or tag-based system is not an easy task: there are no training sets, no standard testing methodologies, no widely accepted expected results.

We thus chose an heuristical approach to give at least a qualitative measure of the recall and precision performances of our application. As a comparison, we have taken a generic tag-powered web system with a tag driven search engine. If we give for granted that such a system can easily be included in TagontoNET with an ad-hoc plugin, we see that Tagonto's precision, if we provide just this very plugin, cannot be worse than the given system's, since all the results we would normally have from the system are included in the result: this way we now have a lower bound to the precision of our software.

While it's simple enough to set a minimum precision, we can only provide qualitative assertions about the expected increase in precision. A good example to understand how Tagonto can improve precision is using the tag Wine for a research. As a Tag wine has at least one homonym, a software product where Wine stands for Wine Is Not an Emulator: searching for wine in some tag based systems will result in concents related to both meanings of the word. On the other hand, if we suppose Tagonto to be powered by an Ontology related to just one of the meanings of wine - or otherwise able to resolve the homonymy conflict as described before in this document - the user will be provided with more terms, tags or concepts, related to the tag Wine. This way, he will be able to narrow down his search, thus increasing the precision of his query after only a couple of iterations given no prior knowledge of the search domain.

While tag clouds, a very popular meaning of tag association, are able to provide a service somewhat similar, they lack the semantic information an ontology contains and thus are unable to increase precision in the same way.

Recall rate can be analysed in a similar way. In the worst case, it will be the same as the given tag system, but we expect significant improvements due to the use of the ontology backend and the aggregation of more plugins. While the user can navigate various plugin, thus enlarging the result domain, we expect the

biggest improvement to came from the concept-tag mappings stored in memory and presented to the user: as in a semantic tag-cloud, a user can easily find resources tagged with tags similar to its own, thus enhancing recall.

5.2 Performances

In previous chapters we have discussed performance issues for the TagontoLib and the TagontoNet components. The reader might then wonder whether these problems do sum up, rendering the whole system so slow it cannot be actually used.

While these concerns do apply in a production environment, this is not the case in a testing-level infrastructure. Tagonto is able to answer most requests, according to the complexity of the reasoning involded, in no more than 15 seconds running on common personal hardware. Timeouts on remote resources and the use of asyncronous methods do improve user experience and let the product be usable. If mapping has been already cached, times can be cut down to about 5 seconds, while ontology navigation can be around 1 or 2 seconds.

This said, Tagonto should undergo a strong optimization and code cleaning to reach production-level: ontology based reasoning is still extremely expensive in term of computational resources, and the use of remote resources implies network imposed lags. From an architecture standpoint, however, Tagonto can scale very well to multiple server, its component being decoupled enough to reside on different servers.