Federated search scalability

Author: Beñat Txurruka Alberdi

Master in Innovation and Research in Informatics (MIRI)

Computer Networks and Distributed Systems (CNDS)

Facultad d'Informatica de Barcelona (FIB)

Universitat Politècnica de Barcelona (UPC)

April 29, 2015

Supervisor: Jaime Delgado
Departament d'Arquitectura de Computadors

Contents

List of Figures

List of Tables

Abstract

The search of images on the internet has become a natural process for the internet surfer. Most of the search engines use complex algorithms to look up for images but their metadata is mostly ignored, in part because many image hosting sites remove metadata when the image is uploaded. The JPSearch standard has been developed to handle interoperability in metadata based searches, but it seems that the market is not interested on supporting it. The starting point of this project is a broker based federated search for image retrieval which was supposed to be the backbone of the final software. However, due to out of date state of the previous project a new software has been developed. The objective of this approach is to support scalability on an image search system based on querying to the API of well known image hosting services and providing a tool to analyse images metadata for low level searches. A design is proposed to dodge the API rate limitations, the actual major problem regarding scalability, and a working software is developed to prove its viability. A set of tests are introduced to evaluate the performance of the approach and its results are interpreted. Finally a future work is suggested in order to improve the weaknesses.

Keywords

Federated search, web service, API, scalability, metadata, image search

Abbreviations

AJAX Asynchronous JavaScript And XML

API Application Programming Interface

DoS Denial-of-service

EXIF Exchangeable Image File format

HTTP Hypertext Transfer Protocol

IPTC International Press Telecommunications Council

JAX-RPC Java API for XML-based Remote Procedure Call

JAX-WS Java API for XML Web Services

JPEG Joint Photographic Experts Group

JPQF JPEG Query Format

JPSearch JPEG Search

SOAP Simple Object Access Protocol

W3C World Wide Web Consortium

WSDL Web Services Description Language

XML eXtensible Markup Language

XMP Extensible Metadata Platform

1 Problem definition

1.1 Introduction

Before 1975 when Steven Sasson as an engineer at Eastman Kodak invented and built the first electronic camera, all photos stored printed or in negatives, and most of the metadata was probably written on paper sheets along that paper piles and film rolls. The digital camera democratization arrived at the same time that the World Wide Web and since them the image quantity on the internet has increased exponentially. Today, thousand of webs are specialyzed on hosting images and hundred of search engines look into them to provide users the image they are looking for. Every image searcher tries to differentiate from the competence adding new searching parameters or techniques, but no one seems to bother about metadata. Metadata is "data about data". Data describing a full variety of information about, in this case, photos. Wouldn't be interesting to analyse it and obtain search results based on the photo information rather than complex algorithmic predictions?

1.2 Starting Point

This project was prepared to be the further work of a previous one, and to understand the accomplished work a brief explanation of what was the starting point is essential. Called "Buscador d'matges basat en un broker y reescriptura de queries" [1] or "Image searcher based on a broker and query rewriting" in English, its main objective was to create a multimedia browser that search on various image servers in a transparent way for the user, taking advantage of the metadata interperability approaches [2]. In general terms the final software was a image searcher application on the server and a web client for the user.

The image searcher application is responsible of making queries to different image and translating the user query to the characteristics of each image repository. This task was made by a broker [3], using the design of a federated search centralising all the tasks. On the other side, the web client was focused on the user, the web surfer who wants to search an image with so specific characteristics that they must be looked them up on the images metadata. It was developed a friendly design with a embeded map where fix the desired geographic location and with drop-down menus simplifying the

options available for the user.

That searching system was pioneering since all previous work about the subject was only theoretical. Even if the obtained result was acceptable, one major problem limited the usability of the system: The scalability. The image repositories being queried had to be accessed using their Application Programming Interface (API)s, and the limitations established by their owners along with the absence of focus on this direction in its design involved a poor scalability.

1.3 Scalability problem

Scalability is the ability of a computer application or product (hardware or software) to handle the increasing amount of work adequately so it can continue functioning appropriately or its ability to be expanded to deal with that increase. When third-party APIs are involved on the software, scalability is completely tied to their limitations and the ability to dodge this obstacles will determine how the system behaves. With the aim to solve the previous projects scalability difficulty this new project comes to life and the main focus will go on alleviating the limitations imposed by the API provider. To support this characteristic an alternative structure may be provided and extra service providers could be added.

1.4 General objective

The general objectives of the proposed approach throughout this project will be to provide an analysis of the scaling possibilities led by the limitations of querying APIs of well known image hosting services and contribute with a design of an architecture with the capability to manage in the best possible way that scalability obstacles. An analysis of the starting points project is needed to be made in order to evaluate its capacity to change its shape and adapt to the new design. The implementation of the design will be presented as a final software to search and analyse images throughout their metadata, proving its problem handling and architecture viability. Via a set of tests the performance of the product will be measured and evaluated.

1.5 Structure of the Document

The rest of the document is organized in eight sections. The second section exposes the state of the art and the third exposes the related work on the market. Once situated, in section four the previous project is analysed, and after in section five design and implementation proposed to achieve the objectives. In section six the developed software is explained and tested its performance in section seven. Finally in section eight conclusions and proposed future work are illustrated.

2 State of the art

As the word has gone computerizing, data became digital and metadata started being used to describe that data using metadata standards. In photography there are several standards to describe this metadata, being the most common Exchangeable Image File format (EXIF), International Press Telecommunications Council (IPTC) and Extensible Metadata Platform (XMP). The EXIF is filled by the camera when capturing the photograph and typically includes camera maker brand, model, timestamp, lens settings and more. The IPTC was developed in 1970's to record information related to press images and has evolved in time including now information such as photograph's byline, location, title, description and much more. However, nowadays its use is not limited to press photographs and is a widely used metadata standard for every photograph style. XMP is a standard developed by Adobe in 2001 and nowadays is an ISO standard. It is not a exclusive metadata for photographs and is used by Adobe for every format of its own like PDF or photographic editing software files. Regarding images, it replicates information from EXIF along with extra information such as editing tool identification, changes history and sources, and can also include some information from IPTC.

But filling internet with images wouldn't be so useful without image search engines. Since the first image searcher appeared, many competitors have grown up with its encouraging features. Simplifying, they work pretty much like Web searchers Web crawlers, indexing photo's file name and the text surrounding in the web page. Some also apply computer vision to implement Content-based image retrieval Still, search based on image metadata does not attract the attention of the market.

Google, Bing, Yahoo and Picsearch are common image web searchers and it exist an endless list of pages dedicated to this purpose. As time has gone by, they gone evolving and clearly looking in the mirror of each other, mimicking features. Following the mentioned content-based image retrieval, most of them implemented color-based search analysing the main distinguished colors in the image. When looking for a person many of them also uses face detection to improve rating of images with people inside it. Even you can choose if you want a photograph, a painting or a moving image. But till now, none of the implemented deeper information search, no one seems to index

images metadata to search within it.

API is a particular set of rules, protocols and specifications used when building software applications to communicate with each other. Similar to the way user interfaces provide an easy way to interact between humans and computers, APIs perform as an interface between applications to facilitate their interaction. Among many appliances, it is used to access remote site web services. Web providers publish web service endpoints so third-party applications can access to specific resources stored on their site. In this way, when retrieving images hosted in one page instead of crawling all the site and indexing every image like web searcher do, an API endpoint could be queried. This is a much more easier and resource saving way since all the searching job will be made in the provider server and the client only will need to ask and receive the answer. However, since all the hard work is done by the site proprietor, limitations are settled to avoid abuses and external Denial-of-service (DoS) attacks.

This API's limitations are the hardest obstacles when you want to scale your application. While everything you program is under your control, limitations are established from outside and cannot be modified.

3 Related work

There are many tools on the market related with the topic. A brief look at them can brighten the road to find a solution for the objectives and know the trends of current researches on the area.

3.1 Starting Point

The previous project standing as the starting point of this one is in fact the most related work. The analysis and decisions made at that point are taken into account setting the base of further analysis and the developed code is the background where this project will start working.

3.2 Federated Search

Federated search is an information retrieving technology based on a distributed and commonly heterogeneous search with a unified result in return. Federated search emerges to meet the need to seek multiple sources of content starting with a unique query from the user. It consist on a broker taking the initial query and transforming to each querying search engine syntax, broadcasting to that datasets or web services and returning the outcomes all together to the user. The results can be returned asynchronously as they get received, or synchronously representing all merged after every one has answered. Of course each one has its advantages, while asynchronous may be faster returning some results since it keeps responding as it gets them, synchronous can perform a post-processing algorithm to rank results or discard some when inappropriate.

3.3 Scalability on cloud

Scalability on cloud computing systems is a well studied subject. Centralized architectures where the architecture management is handled by a single component tend to lack on scalability and reliability [4]. Like in the starting software of this project, in case of failure of the central server all the system will be affected and the absence of a backup mechanism will knock down all the architecture [5].

3.4 JPSearch

The JPEG Search (JPSearch) Core Metadata Schema is design to provide metadata interoperability developed by the Joint Photographic Experts Group (JPEG) and is the main component in the ISO/IEC 24800 strategy. Along its five pars are specified minimum requirements to be fulfilled, including structure and rules. However, the creation of JPSearch does not expect the extinction of other well-established metadata schemas. The JPSearch Core Metadata contains a set of minimal core conditions that in order to support more parameters it can be extended. For this purpose it presents a framework divided in 6 different parts:

1. Global architecture: an overview of the global architecture of the JPSearch framework.

2. Schema and Ontology: describes the registration, identification, and management of schema and ontology.

3. Query Format: Part 3 of JPSearch is the most important for this project and contains the tools of the JPEG Query Format (JPQF). The objective of JPQF is to provide a unified tool to search among different image hosting sites with different services. The process is supported by a eXtensible Markup Language (XML)-based query language defining the format the query and the response must have in order to support the desired interoperability on a distributed image fetching system [6].

4. File Format: defines the file format that the metadata must have on images. It is also an important point for interoperability.

5. Data Interchange Format: defines a data interchange format between image repositories.

6. Reference Software: reference software to instantiate the functionality described in the standard.

Even if the standard is finished, it is still under development to include tools to help in its implementation. However, it has still not succeed on the market and it is not common to find a service implementing it.

4 Analysis and evaluation

4.1 Introduction

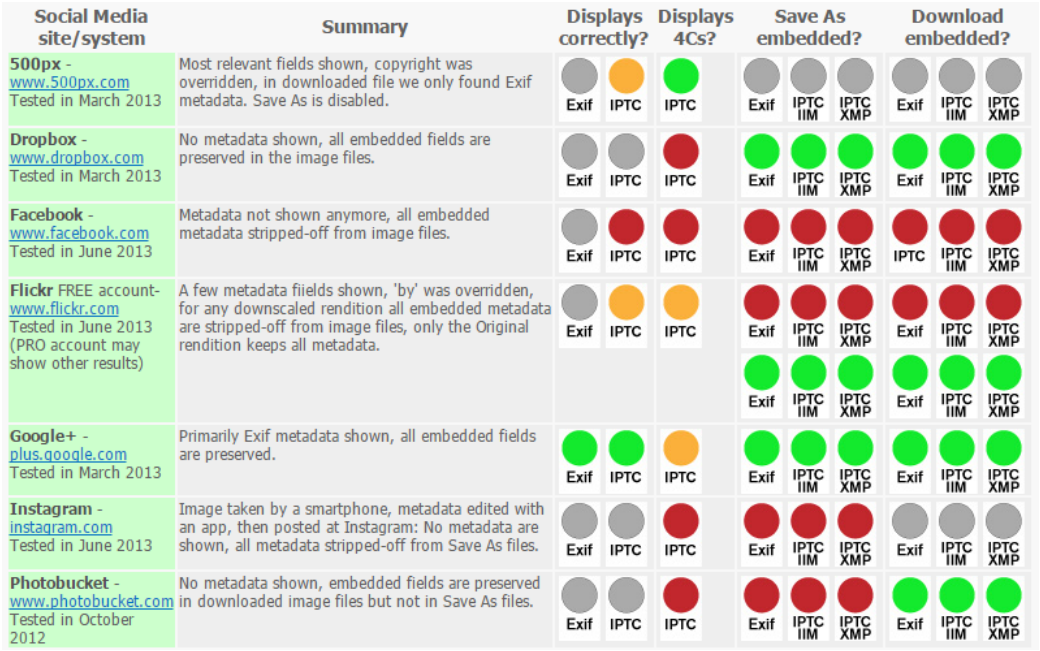
This section compiles the analysis of the image repositories on the market and the evaluation of the starting point project. First of all a survey is made about image repositories on the market and how they conserve the metadata so the supporting ones are chosen. Then the API provided by the selected repositories is reviewed to see the limitations imposed by each one, establishing the base of the decisions taken from now on. With the analysis done, before starting to change the previous work or propose improvements, the analysis of it is required. Being a project from 2010 the possibility of out of date technology was high and the assumptions taken on that time may be changed. In consideration of that, a whole evaluation of the software and the obsolescence of the used technology is done. Finally conclusions of this section are presented.

4.2 Image hosting pages and metadata conservation

The first step to understand the actual situation was to observe the state of the image repositories and their treat to metadata.

As can be seen on Figure 1, a small extraction of the results from "Social Media Photo Metadata test" [7] accomplished between 2012 and 2013 by contributors to the photo metadata survey of controlledvocabulary.com and members of the Photo Metadata Working Group of the IPTC, photographs metadata information usually is not accessible for the social media user, or at least not at all. In the picture, color gray will mean that the option is not available, green that all metadata is preserved and displayed, yellow that metadata is partially preserved or displayed and red that it is erased or not displayed.

According to the survey of Controlled Vocabulary, many social media just stripe metadata off the image when uploading to their servers. Others maintains it on the original image but is not replicated to other images derived from the original such as different resolution or thumbnail images. The survey explains how some services allege this metadata removal to user experience, decreasing downloading time when visualizing photos. However, despite the saved memory space is ridiculous in comparison to the global



Social Media site/system	Summary	Displays correctly?	Displays 4Cs?	Save As embedded?	Download embedded?
500px - www.500px.com Tested in March 2013	Most relevant fields shown, copyright was overridden, in downloaded file we only found Exif metadata. Save As is disabled.	Exif, IPTC	IPTC	Exif, IPTC IIM, IPTC XMP	Exif, IPTC IIM, IPTC XMP
Dropbox - www.dropbox.com Tested in March 2013	No metadata shown, all embedded fields are preserved in the image files.	Exif, IPTC	IPTC	Exif, IPTC IIM, IPTC XMP	Exif, IPTC IIM, IPTC XMP
Facebook - www.facebook.com Tested in June 2013	Metadata not shown anymore, all embedded metadata stripped-off from image files.	Exif, IPTC	IPTC	Exif, IPTC IIM, IPTC XMP	IPTC, IPTC IIM, IPTC XMP
Flickr FREE account- www.flickr.com Tested in June 2013 (PRO account may show other results)	A few metadata fiields shown, 'by' was overridden, for any downscaled rendition all embedded metadata are stripped-off from image files, only the Original rendition keeps all metadata.	Exif, IPTC	IPTC	Exif, IPTC IIM, IPTC XMP Exif, IPTC IIM, IPTC XMP	Exif, IPTC IIM, IPTC XMP Exif, IPTC IIM, IPTC XMP
Google+ - plus.google.com Tested in March 2013	Primarily Exif metadata shown, all embedded fields are preserved.	Exif, IPTC	IPTC	Exif, IPTC IIM, IPTC XMP	Exif, IPTC IIM, IPTC XMP
Instagram - instagram.com Tested in June 2013	Image taken by a smartphone, metadata edited with an app, then posted at Instagram: No metadata are shown, all metadata stripped-off from Save As files.	Exif, IPTC	IPTC	Exif, IPTC IIM, IPTC XMP	Exif, IPTC IIM, IPTC XMP
Photobucket - www.photobucket.com Tested in October 2012	No metadata shown, embedded fields are preserved in downloaded image files but not in Save As files.	Exif, IPTC	IPTC	Exif, IPTC IIM, IPTC XMP	Exif, IPTC IIM, IPTC XMP

Figure 1: An extraction from Social Media Photo Metadata test

image size, important data such as attribution information is removed in the process. Quoting David Weinberger, To a collector of curios, the dust is metadata, and like this curious man, social media cleans metadata as dust.

Since once the metadata is erased it cannot be acquired from anywhere, important image social media such as Facebook or Instagram are not longer appropriate for the purposes of this project.

4.3 API analysis

On the Table 1 are shown the query limitations imposed by the APIs, how many results it will return in total and how many will be displayed per page. Even if Facebook, Instagram and Dropbox are well known and heavily used services, since Facebook and Instagram erases all metadata on the image and Dropbox has no way to query its public images both are going to be removed from the analysis from now on. Instead, Panoramio and DeviantArt which are services used in the old application will be evaluated among the others

Since Picasa has been discontinued, we does not know it's API limitations.

	Query limit	Max results per query	Max images per query-page
Flickr	3600 queries/hour (86400 queries/day)	4000	30
Picasa/Google+	10.000 queries/day & 5 queries/second	Must be specified in the query	To be specified in the query
Panoramio	100,000 queries/day	100	To be specified in the query
500px	1,000,000 query/month	100	100 (default 20)
DeviantArt	Adaptive rate limiting	Adaptive	50
Photobucket	10000 queries/day	100	100

Table 1: API limitations

In any case, it is used as a back-end temporary solution for Google+ photo hosting, where we can find its limits are 10.000 queries per day and 5 queries per second.

4.4 Technology obsolescence

- JAX-RPC: Java API for XML-based Remote Procedure Call (JAX-RPC) allows a Java application to invoke a Java-based Web service using World Wide Web Consortium (W3C) standards such as Web Services Description Language (WSDL). It was the old Java standard for web service implementation. In 2006 when version 2.0 was released it was renamed to Java API for XML Web Services (JAX-WS), and in Java EE 6 it was deprecated. As further in this document is going to be explained, Web Services in the project needed to be updated due to various reasons and it would be anachronistic and senseless updating with a deprecated technology. Moreover, additional changes in the project or even the server could not be compatible with Java EE 5, the last working version with JAX-RPC, making this upgrade inevitable and the consequent JAX-WS update.

- Apache Tomcat Catalina: As Tomcat gets updating, so does Catalina, its Servlet Container. If the program is hosted in the latest Tomcat Server but the program is using an old Catalina package, the program could not run due to lack of compatibility so it should be updated. Nevertheless, servlets could be done with JAX-WS, making senseless to support two different libraries for the same purpose. However, since updating the project is not the main objective, a Catalina library was

added before involving in servlet migration into JAX-WS until all the errors not related with this issue got fixed.

- Spring Framework:

4.4.1 Out-of-date API libraries

Photobucket working, Twitpic partially and Picasa discontinued but API still working for Google+ back-end. Remaining API libraries became obsolete.

- Flickr: Due to its registered user amount and active community Flicker is the most valuable social network in the project. This also concern to Yahoo!, who maintains the site actively, implying frequent changes in their API. Because of this, the implemented version stopped working and its repair is vital. However, instead of calling their servers directly, a Java library was implemented in the project and due to its lack of update, a probable minor fix is instead a full change on Flickr involving parts on the project.

- Panoramio: Even owning the simplest API, Panoramio has changed its way of retrieving requests and it was not working when the project was received. However, due to the complexity and the size of the code, this change imply modifications on various classes on the project.

- DeviantArt: Designed to upload user-made artwork, DeviantArt is an important social network for hosting pictures and photographs. It goes evolving and so does its API, which actually only accept requests from Oauth-based user authenticated applications. In the old version of the project there was no logging option due to previous API implementation, but the new version apart of requiring to change querying methods also demand the development of a user logging section.

- Twitpic: When this project started Twitpic was another API to be updated. As Twitter is a hight used social network, many efforts were dedicated to its API update, but despite everything, suddenly Twitter annouced its discontinuity and shut down the service on 25 October 2014.

- Google Maps V1: Being one of Google's star service, the company keep maintaining it constantly. In this way, the first version of the

JavaScript API got obsolete and should be updated also in order to introduce latitude and longitude for location based searches.

4.5 Additional problem and conclusions of the analysis

Summing to the effort to fix every error, change all the obsolete technology update the out-of-date libraries, a new obstacle was found. It was detected that a code fragment of the software was not on the delivered previous project. This implied a reprogramming of a code meaning that the involving methods must be completely understood and everything should be written to fit in there.

At this point a decision was made. Since too many APIs were requiring to be completely changed and updated along with the out-of-date technology taking into account that a part of the project should be also rewritten with a complete ignorance of the followed algorithm, too much time would lost and the objectives of the project could not be achieved. Based on this conclusions it was decided that a small and lighter version of the application should be done since the beginning in order to afford some solutions to scalability issues.

5 Design and implementation

A proper implementation needs to be based on a precise design, and a precise design have to identify the main weaknesses and try to handle them in the more suitable form. Regarding to scalability, there were two main points where the previous project had its major problems and had to be kept in mind: Server workload and API limitation handling. While the servers resource scalability is easy to handle with a elastic resource provider such as Amazon EC2, Google Compute Engine or RedHat OpenShift, API limitation dodging should require a specific coding approach development. However, not everything concerning Server workload should be delegated to Cloud hosting services so a method to lighten its task would also be interesting.

5.1 Abstraction

Client-side metadata filter requests does not depend on the metadata analysis system, and Server-side web services are abstracted so analysis method could be changed in the future without implementing them again.

5.2 Client oriented load

In view of the fact that a new program should be developed from the beginning, was taken advantage of the opportunity to design it taking focus at implementing the objectives in the easiest and most efficient way. Actually, it was the maximum priority. In contrary of the old application version where the federated search was implemented on the server overloading it with every query, it was decided to give the power of searching to the client itself. Figure 2 and Figure 3 show the differences between previous and proposed Federated Search designs.

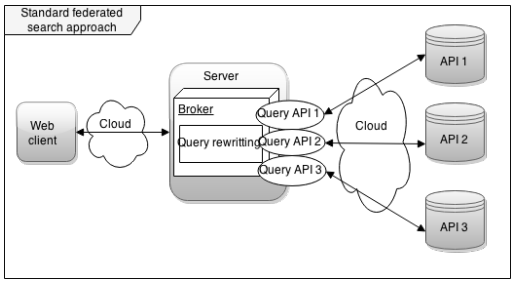


Figure 2: Common Federated Search design approach

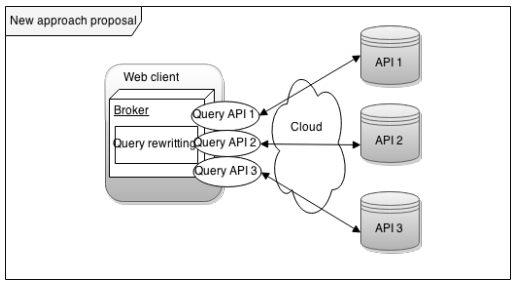


Figure 3: Proposed new Federated Search design approach

If instead of assigning the Server a Proxy task becoming an intermediary between Client and APIs, if the Client itself has the ability to directly query APIs the overload from the Server will decrease significantly. In other words, the client has its own broker implementing a small version of federated search, so the broker instead of federating the results from all the clients, it only federate results from that client instantiation. This way, the possible restrictions implied due to a unique machine querying massively may disappear or become at least a bit diluted. Figure 4 and Figure 5 show the differences between previous and new design connections and where the server load differences can be also deducted

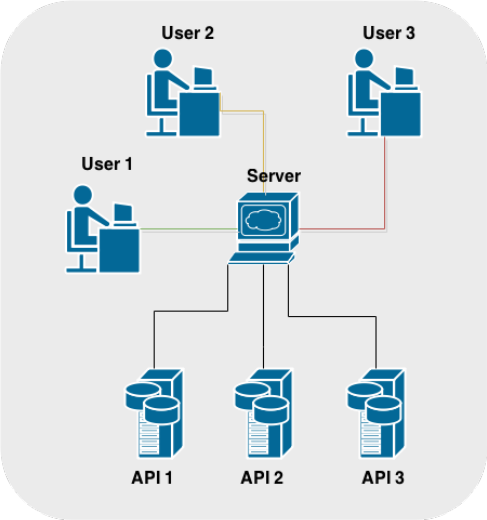


Figure 4: Old design connections

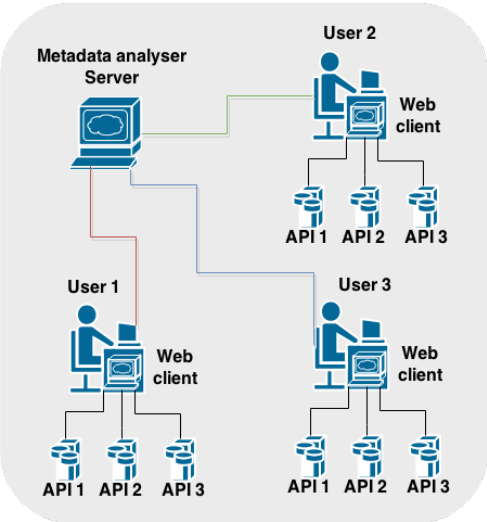


Figure 5: New design connections

To present a client application to the user implementing this design, a web page will be developed so the client can access it from any web browser across many platforms. It has to be noticed that the client only queries the repositories and it does not perform any metadata analysis. This part should be done by a more powerful application, so the figure of the server will not vanish.

5.2.1 jQuery requests

jQuery is a cross-platform JavaScript library designed to simplify the execution of scripts on the client-side instead of on the server-side. It is the perfect tool to develop Asynchronous JavaScript And XML (AJAX) applications which is what is needed to implement client-side query federation on an interactive web application. The jQuery cross-platform feature will lead to connect servers on other domain and making help to load data from the server without a browser page refresh. With AJAX when the submit button is pressed on the web application, JavaScript will accomplish all the requests and update the current screen.

5.3 Authentication

To prevent DoS attacks or developer abuses, many API providers require a developer key to make use of them. Table 2 depicts which of the analysed APIs requires an authentication to permit their access.

	Developer authentication required	Authentication type
Flickr	Yes	Key on query
Picasa	No	-
Google+	Yes	Key on Log In
Panoramio	No	-
500px	Yes	Key on query/Key on Log In
DeviantArt	Yes	Key on Log In
Photobucket	Yes	Key on Log In

Table 2: API developer authentication requirements

As mentioned before, an advantage when using client side querying is the possibility to dodge query limitations. The APIs without authentication may be eluded with queries coming from different addresses since it is not a common Key to be related. On the other side, even if an authorization requirement restricts the queries, the limitations from the ones requiring an authentication may also be possible to be relaxed. When looking at the API restrictions of this ones usually can be observed that the limitations can also be delegated to users. If instead of limiting the queries to the developers Key, the restrictions are transferred to each user, they will be hardly exceeded. Finally, when the developer Key is sent on the query itself it is not any way to avoid the restriction in this direction.

OAuth is an open protocol designed to provide a secure authentication on a standard way which is in its second version, not being compatible with the first one. The OAuth 2.0 authorization framework is a tool to obtain limited access to an HTTP service for third-party applications. OAuth 2.0 is commonly used as a way for web surfers to log into third party web sites using accounts from other major sites, without worrying about registering in every site.

5.3.1 OAuth 2.0 on JavaScript

Since all the client specific application side was being developed on JavaScript, a way to use OAuth 2.0 from it was needed.

Google has a proprietary library to establish OAuth 2.0 connections and arrange queries to their different APIs. Using it, the connection to Google+ achieved in a simple manner. However, Google+ has no specific way to retrieve public images, only user-own images could be requested with a single query. The proposed way to get public images is to get public posts and extract images from them. This alternative is not very useful since only a small part of publications have images and query limitations may run out quickly. So back again, the solution would be to use Picasa, in view of the fact that it does not require any authentication, although is logic since the service has disappeared and cannot be logged in in nowhere. In conclusion, Google's library was finally not used, but has been leaved commented in the code for further possible uses.

500px also has its own library for Oauth 2.0 implemented on JavaScript. It is little tricky because the code using it has to be in a separate JavaScript file because it is reloaded when callback is performed. To complete that callback, a provided Callback file without any modifications should be saved in the same root directory as the index file, that needs to be added to the white-list in the 500px application section in the Web page. Once all of this is configured, the integration of the library in the code is straight-forward following the steps on their documentation.

The leaving services does not own a proprietary JavaScript API, making inevitable to develop some code to implement OAuth 2.0 protocol in order to connect to their APIs. To do so, the algorithm of OAuth 2.0 protocol should be understood. It is important to remark that OAuth 2.0 has three use cases: For Web Server applications, Browser-Based applications and Movile applications. The objective was to implement in JavaScript so Browser-Based App was the chosen use case in this case.

Three actors take part in the OAuth 2.0 protocol :

- The Third-Party Application: Is the application trying to access to the user's account. It is the client in the system and needs to connect to

the server to get the access grant permission.

- The API: It is the API server used to access the user's information. It is divided into Authentication server and Resource server.
- The User: The user owns the resource which the Client is trying to access and who will give it the permission.

Once the actors are clear, let see how the protocol works for Web-Browser applications: First of all, the application should redirect to the authorization server API end point to request a token. When doing that task, it will need to send the requested response type, the Client ID, the redirect URI and the scope. The scope will depend on the task wanted to do when access is gotten and in the API itself (each one defines it on a proprietary way), the redirect URI would be the direction where the response with the token must be send and the Client ID should be given by the API provider when registering the Third-Party application. Commonly the redirect URI must be added to a White-list on the API provider console too when registering the application. Continuing with the algorithm, a window will prompt to the user asking to connect their account to the requesting application, and if the user accepts it, the access would be granted sending the authorization server the access token to the specified redirect URI. Once the Third-Party applications has the access token it should be able to access to the resource server API end-points concerning the chosen scope, sending the request among that received access token. On Figure 6 the algorithm can be seen graphically.

Since at a first view the protocol is just sending and receiving Hyper-text Transfer Protocol (HTTP) messages between cross-origin platforms on JavaScript, jQuery appeared to be the best option to make AJAX calls. However, after trying in several ways with DeviantArt the connection was not achieved. Even if the request were made properly and the token were received, the application was no able to get and save the token, throwing always "Uncaught SyntaxError: Unexpected token" error. A possible explanation could be that setting the response type as "code" as is requested by DeviantArt documentation confuses the jQuery request, but the lack of documentation about that error made impossible to clear up the error and bring a solution.

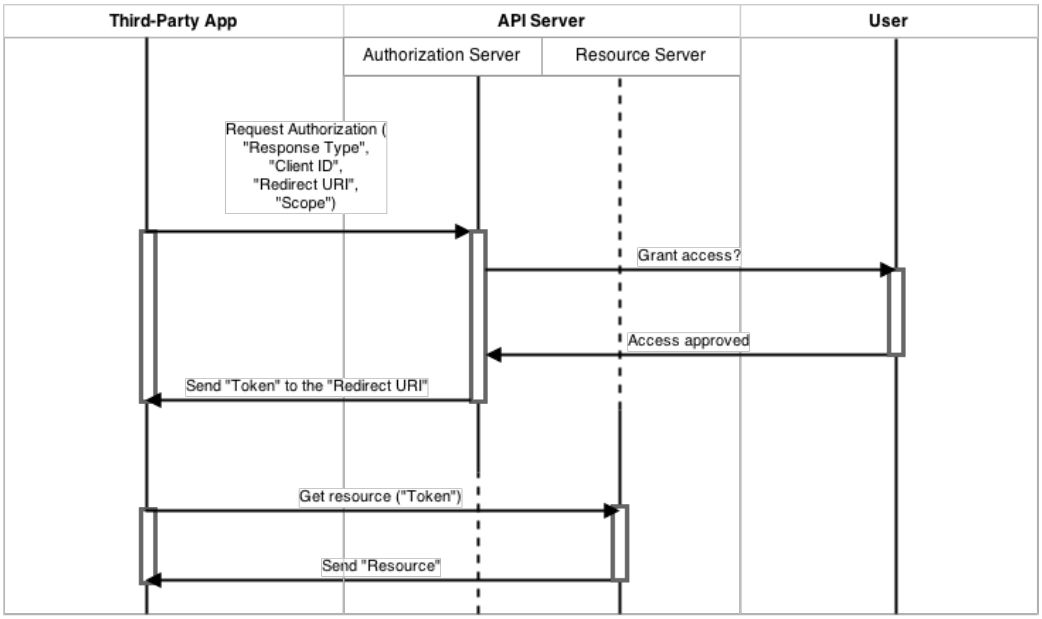


Figure 6: OAuth2 authentication flow on a Web browser application

The impossibility to achieve a successful connection in that way brought out to try Third-party generic JavaScript libraries implementing OAuth 2.0.

- JSO: Provided by "UNINETT AS", a non-profit company working for educational and research institutions in Norway, JSO provides a way to use OAuth 2.0 on client web applications. Yet, the annotations were not clear enough leading its misunderstanding to an incorrect implementation or it was not working with DeviantArt at all. Every time a JSO call was sent, the class was reloaded losing its instantiation and the callback did not work properly. After many attempts, the efforts to implement it were interrupted.

- Hello.js: A client-side Javascript SDK for authenticating with OAuth 2.0. It has been developed for different services depicted in their web page and each one has its own implementation. It abstracts the procedure so the developer only needs to specify to which API to connect, enter the required Key or ID and Hello.js will do everything. The problem comes if the wanted service provider is not supported, and that is what happened with DeviantArt.

- jsOAuth: As described in its documentation, it "aims to form the basis of custom clients such as Twitter and Yahoo". It had to be rejected after realising it was developed to not work on browsers, arguing "security reasons".
- js-client-oauth2: Another library to execute OAuth 2.0. However, it required node.js and even if it was tried briefly the integration of another runtime environment and its dependence was not the wanted solution so it was discarded.

In conclusion, not a single tried library was prepared nor tested to work with DeviantArt and all failed connecting to it. Finally, after testing failing all the attempts to connect to DeviantArt this procedure was pushed aside.

5.4 Metadata retrieval

Once the API connection feature is solved, is time to perform the required queries given the user indications and analyse its metadata. Until JPSearch succeeds in the market and service providers implements this framework, each one has its own way to operate. In view of this, its querying method should be analysed in order to identify common features and extract some pattern to develop this section.

The most common manner to ask for a photo set is demanding the images related to a word being it a tag, a part of the title, or a part of its description. However, Panoramio only accept queries regarding a geographical position. To afford this requirement a Google Maps API was used to embed a map on the web page so a bounding box with its latitude and longitudes could be obtained. Flickr also support querying about photographs geographically tagged inside a box delimited by latitude and longitudes, 500px only allows searching in a certain radius around a central point and Picasa has no way to query concerning location. In fact, Picasa only allows searching by words. The particularity of Picasa temporally performing as a Google+ infrastructure provokes to not have its own methods, and in the other hand the discontinuity of Picasa brings to the deactivation of most of the methods of it. The only information to filter Picasa images can only be gathered from its metadata. However, the metadata returned embedded on the image is the files basic information and four EXIF characteristics: Image size, software (allways Google since it is from Picasa), EXIF version and Google+ upload

code. With such a small information not much can be done. At this point, Picasa becomes until further notice extraordinarily limited.

As mentioned before, Panoramio only accepts queries regarding geographic location and the users requests would not be possible to filter in this way. Nevertheless, the photographs returned by Panoramio include all the metadata uploaded by its owner, and analysing and filtering through them would give the desired response. 500px is in a similar situation. Its API only accepts request by word and location, but the photographs also come with the full metadata. In the other hand Flickr apart of the mentioned locations based search, also support querying words, diaphragm aperture and focal length. However, the returned photograph does not include any metadata, and if any of this is wanted an extra query is demanded. This restriction combined with the limit related with the developer key reduces absolutely the possibilities to filter whatever is wanted maintaining the scalability. If the search is centered on words, location, aperture and focal length, there can be 3600 queries per hour, but if a deeper search is wanted then the searches are reduced to the half, 1800, doe to double querying for each search. On Figure 7 is displayed the reduction speed of remaining searches on both approaches.

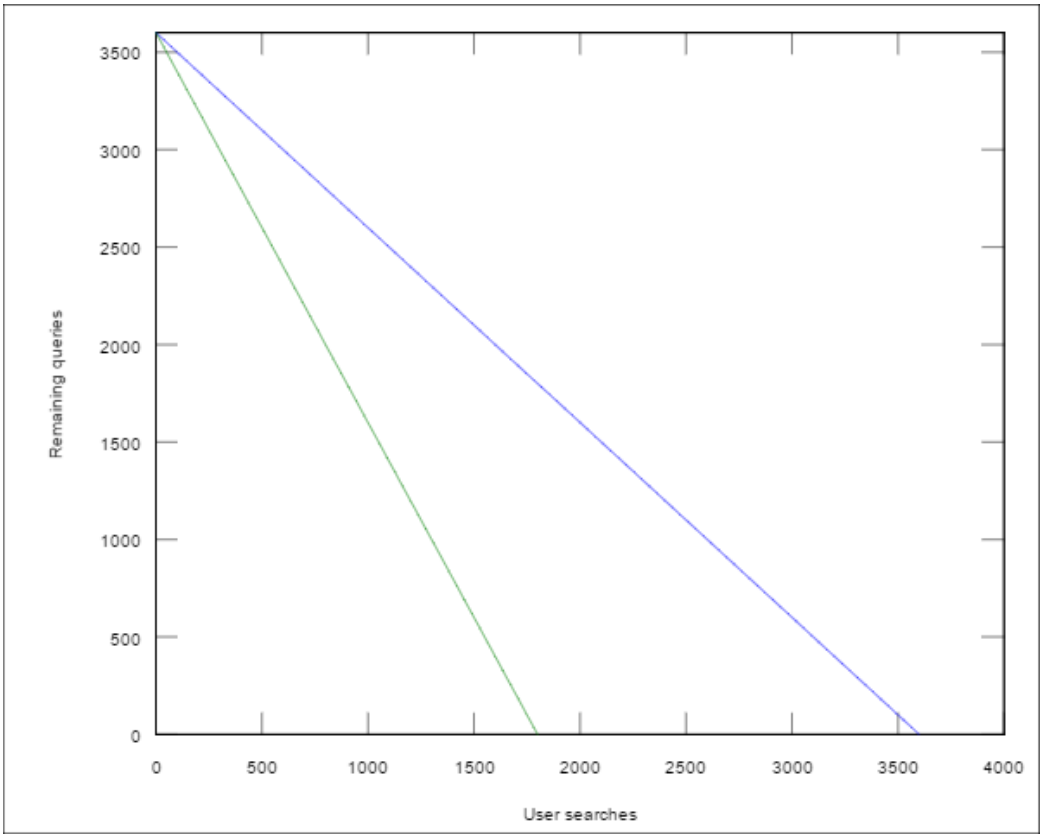


Figure 7: Speed of reduction of user searches capacity in relation with wanted amount of metadata filters

5.5 Metadata analysis and server implementation

Given the situation, in order to avoid unnecessary queries the priority was to focus on searches that Flickr could afford without extra queries, that is, along with words and location, searching by aperture and focal length and filtering on Panoramio and 500px through the metadata. To read and analyse metadata JavaScript was not enough and Server-sides application implementation had to be made, where Java was the chosen Server-side language to be used.

To read metadata in this metadata analyzer server application a library called metadata-extractor is used, a library able to read EXIF, IPTC, XMP, JIFF and much more metadata segments. To analyse what the user re-

quested, three parameters are passed: The images, to be filtered, what field is wanted to be filtered and the value that the user entered. The application search on all the metadatas from the image until the required field is localized and if the image satisfies the condition, the image will be returned to the client.

Once the filtering part was implemented, only a way to connect with the client was remaining.

5.5.1 Web Services

A Web service is a method of interoperable machine-to-machine communication running on a variety of platforms and/or frameworks over a network. The W3C Web Services Architecture Working Group defined a Web Services Architecture in the way that it has to own an interface described in WSDL, a machine-processable format, being accesed using Simple Object Access Protocol (SOAP) messages and typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards. There exist many different frameworks on the market to implement Web Services: JAX-WS, Apache Axis2, Apache CXF, Oracle Metro... each one with its pros and cons. JAX-WS is the Java JDK built in Web Service implementation, with elemental components and basic SOAP messaging features. For high level security, resource management, policy advertisement and more, another more complex alternatives should be used. Apache CXF and Apache Axis2 are the most common alternatives. CXF uses AX-WS compliant based API while Axis2, which is the second version of Axis, is based on proprietary models. CXF also has a bigger community who updates much more frequently the source code fixing errors and security issues via fix packs, in contrary to Axis2 where a new release of the framework has to be waited to get that updates. In any case, both are highly efficient and each one has segments where performs better than the other. Despite CXF and Axis2 being more complete, JAX-WS was chosen due to its default integration with Java EE and simplicity. To implement basic Web Services it was enough and security and other characteristics implementation were postponed for future work.

For future better maintenance, Web Services were abstracted and so the metadata reader could be updated or changed if needed. To do so, a new Java class interface was created where metadata reader be invoked. Once this class

was working a WSDL was deployed so its methods could be accessed from the client. To import the wsdl on client Jax-WS uses wsimport command which is a little bit tedious, but since CXF uses JAX-WS as a base, it was much easier to get methods from the WSDL using the eclipse "Create Web Service Client" tool powered by CXF library. As soon as the methods were imported on the client only its connection with the JavaScript code weas left. However, there is not a direct way to make this calls, so the need to write an intermediary jsp class arose.

6 The software

The software developed for the project is divided mainly in two different parts: The client and the metadata analyzer server. Both are connected with web services to each other, but apart of that, they are completely independent. The following lines describes how is programmed each one and which technologies and functions are used.

6.1 The client web page

The client side application is the one the user interacts with. There is the Web page the user will search on the internet and will use to search images. The visual part and the connections with the APIs. Figure 8 shows the appearance of the web page after a search performed by a user.

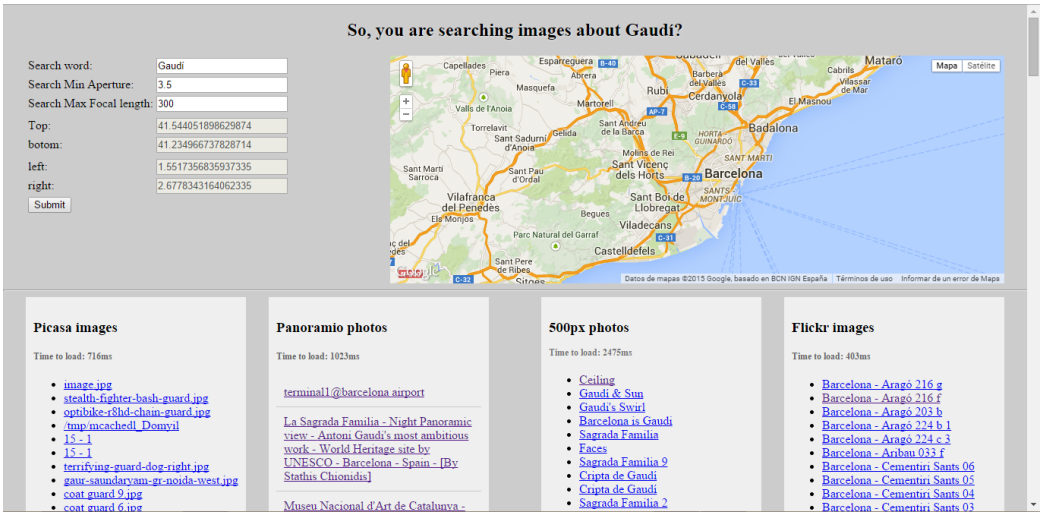


Figure 8: Client web page screeshot

6.1.1 index.jsp

The index is the main page. It is what the user sees when entering the site and what will be interacting with. It is mainly composed by three input boxes where the parameters to be searched are introduced and a map to zoom in or out to delimiter the geographic area where the images are wanted to be searched. These searching parameters correspond to:

- Search Word: Here are introduced the word or words to be searched.
- Search Min Aperture: Here can be introduced a numerical value to filter the images according to their camera lens aperture. The returned photographs will be only the ones with the aperture above the set value. If nothing is written, the filter will not be taken on account and the results will not be filtered by the camera lens aperture.
- Search Max Focal length: Here can be introduced a numerical value to filter the images according to their camera objectives focal length. The returned photographs will be only the ones with the focal length above the set value. If nothing is written, the filter will not be taken on account and the results will not be filtered by the camera objectives focal length.

The map on the page is a Google Maps Version 3 implementation. When zoom in and out is performed, the boxes geographical edges would change and they will be written dynamically on another four input boxes. They have the manual editing possibility disabled by default in order to only be allowed to change using the map.

Once the wanted parameters are introduced it is a submit button to send the form and perform the search. The results are displayed on four columns, one per API, representing a list of hyperlinks addressing to the returned image.

The reason to be a jsp and not just a HTML is because in order to support Google Maps API getting the maps parameter, JavaScript code should be included in the same file, and to do so a jsp has been written combining HTML and JavaScript.

6.1.2 script.js

It is the main script running on the index background. First, the variables are declared and related with the variables in the index. Second, the values of the search form are obtained and saved on the variables. And third, the queries to the APIs are executed. Each query works on the following way:

- Flickr: The variable "*flickrURL*" is set to the resource servers endpoint with the required developer key and is concatenated with the search word and the map bounding box parameters:

```
var flickrURL = 'https://api.flickr.com/services/
    rest/?method=flickr.photos.search&text='+
    searchWord+'&api_key=XXXXXX&extras=
    original_format&bbox='+left.value+","+bottom.
    value+","+right.value+","+top.value;
```

Then, if aperture and focal length are not empty, they are concatenated too to the "*flickrURL*";

```
if (aperture != "" || aperture != null){
        flickrURL = flickrURL+"&xah="+aperture;
}
if (focal != "" || focal != null){
        flickrURL = flickrURL+"&xfl="+focal;
}
```

With "*flickrURL*", the address where the request have to be directed, completely filled, is time to make the jQuery call. When doing this call, a function is executed where the images data is retrieved and attached to the list which is going to be displayed back on the index. This list will be filled with hyperlinks with the image titles and pointing to the URL. However, in Flickr when a photograph has no title, a blank field is returned in the response, so it has to be treated in order to write some text were be clicked in. In this case NoTitle was the chosen text. The listing shows the code to extract images information and attach to the list.

```
$.get(flickrURL, function(xml){
  photos = xml.getElementsByTagName("photo"); //get
      all photos from the result
  for (var i = 0; i < photos.length; i++){
    var flickrphoto = photos[i];
    if(flickrphoto.getAttribute('title')!=""){
       $flickrElem.append('<li␣class="flickrPhotos
          ">'+'<a␣href="https://farm'+flickrphoto.
          getAttribute('farm')+'.staticflickr.com/'
```

```
            +flickrphoto.getAttribute('server')+'/'+
            flickrphoto.getAttribute('id')+'_'+
            flickrphoto.getAttribute('secret')+'.jpg
            ">'+ flickrphoto.getAttribute('title') +
            '</a></li>');
        }
        else
        {
            $flickrElem.append('<li class="flickrPhotos
            ">'+'<a href="https://farm'+flickrphoto.
            getAttribute('farm')+'.staticflickr.com/'
            +flickrphoto.getAttribute('server')+'/'+
            flickrphoto.getAttribute('id')+'_'+
            flickrphoto.getAttribute('secret')+'.jpg
            ">NoTitle</a></li>');
        }
    };
}).error(function(){
    $flickrElem.text('Flickr Photos Error');
});
```

- Picasa: With Picasa, the "*GooglePlusURL*" variable is filled with the address. Since almost all the API is deactivated and it does not require any developer key, only the querying word is attached to the endpoint URL.

  ```
  var GooglePlusURL ="https://picasaweb.google.com/
      data/feed/api/all?q="+searchWord+"&alt=json";
  ```

  With "*GooglePlusURL*" filled, the jQuery call is executed, performing as well the corresponding function to fill the Picasa list of the index with the hyperlinks to the photos like with Flickr. However, The returned photographs by Picasa always have a title so no processing is done for that.

  ```
  $.ajax({
    url: GooglePlusURL,
    dataType: "jsonp",
    success : function(response) {
  ```

```
    $googleHeader.text("Picasa images");
    picasaEntries =  response.feed.entry;
    for (var i = 0; i < picasaEntries.length; i++)
       {
      var pic = picasaEntries[i];
        $googleElem.append('<li class="gpPhotos">'
          +
            '<a href="' + pic.media$group.
               media$content[0].url + '">' + pic.
               title.$t + '</a></li>');
    };
  }
});
```

- Panoramio: Finally, its time for Panoramio. Similar to Picasa, to fill "*PanoramioURL*" there is no developer key to be attached, but instead of the search word, the map bounding edges should be concatenated to the endpoint URL.

  ```
  var PanoramioURL = "http://www.panoramio.com/map/
     get_panoramas.php?set=full&from=0&to=20&minx="+
     left.value+"&miny="+bottom.value+"&maxx="+right.
     value+"&maxy="+top.value+"&size=original";
  ```

  When doing the query to get photos, this time the procedure is more complex. To analyse lens aperture and objectives focal length like in Flickr, the power of the metadata analyzer server is needed. To connect to the web services and send data to it, the photo links and the values to be filtered are sent to "*metareader.jsp*" (see section 6.1.5 on page 40), which will do the remaining job until this point. However, it is not a direct way to send information from JavaScript to a JSP file, so jQuery is used again, setting the URL to "*metareader.jsp*". In the listing X can be seen the procedure.

  ```
  $.ajax({
    url: PanoramioURL2,
    dataType: "jsonp",
    success : function(response) {
      $panoramiotHeaderElem.text("Panoramio photos");
  ```

```
panoramas =  response.photos;
var urls  = new Array();
for (var i = 0; i < panoramas.length; i++) {
  var pano = panoramas[i];
  $panoramiotElem.append('<li class="panoramas
     ">' +
  '<a href="' + pano.photo_file_url + '">' +
     pano.photo_title + '</a></li>');
  if (aperture != "" || aperture != null ||
     focal != "" || focal != null){
    urls.push(pano.photo_file_url);
  }
}
    if ((aperture != "") || (focal != "")){
  console.log("Send panoramio photos to analyse
     !");
  $.ajax({
      url: "metareader.jsp",//servlet URL that
         gets first option as parameter and
         returns JSON of to-be-populated
         options
      type: "GET",//request type, can be GET
      cache: false,//do not cache returned data
      data: { param:JSON.stringify(urls), meta:
          aperture, filter: "Aperture"},//data
         to be sent to the server
      success : function(filtered){
        console.log(JSON.parse(document.
           getElementById('demo').innerHTML));
        console.log( "Panoramio filtered: "+
           filtered);
      }
  })
}
    var endPanoramio = new Date().getTime();
$panoramiotHeaderElem.append("<h5 id='panoramio
   -time2'>Time to load: "+(endPanoramio -
   start)+"ms</h5>");
```

```
    }
}
```

6.1.3 script500px.js

It is the script to use 500px.js [8] library included also in the project. The code inside this file is not in script.js because as mentioned on section 5.3.1 on page 24, after making the call the secript is reloaded, and if it is not separated other calls may be lost in the process.

To begin with 500px, first the variables are declared and related with the corresponding variables in the index and then the 500px library is loaded in order to make use of it. To load the library, the jQuery "*getScript*" function is called with the library name and its location directory. Inside the "*getScript*" call all functions regarding 500px are executed. The sequence of actions will follow this way:

1. Initialize 500px giving the developer key: "*_500px.init*" function is called and developer key setted.

2. Get Authorization from the user to log in on 500px if wanted: First with "*_500px.getAuthorizationStatus*" function the status of the user will be checked. If it is actually logged in, it will jump to step 3. Otherwise, the window to log in on 500px using OAuth will prompt to the user. Logging in will not show to the user any extra information, but it will make to count the limitation to it, instead to the given developer key.

3. Calculate parameters to search by location: In contrast to Panoramio and Flickr, 500px does not search the geographical position using a delimiting box, it searches around a central point with a given radius in kilometres and that requires extra calculations. The central point is easy to obtain, the central latitude is the midpoint of the latitudes and the central longitude is the midpoint of the longitudes.

```
var mapCentreLat = (parseFloat(top.value) +
    parseFloat(bottom.value))/2;
var mapCentreLng = (parseFloat(left.value) +
    parseFloat(right.value))/2;
```

The radius instead is not so straightforward: Since the map scale is changing, the radius will change every time the user zooms in and out in the map. To calculate the radius, the distance between two opposite points of the delimiting map-box has to be computed, and take its half as the radius. To compute the distance between the two points (bottom-left and top-right) the Haversine formula has been used. The haversine formula is an equation important in navigation which departing from the latitude and longitude of two points it calculates the great-circle distances between them. Listing shows the JavaScript code for the radius calculation.

```
function deg2rad(deg) {
  return deg * (Math.PI/180)
}

//Function to calculate the distance between two
   points using the Haversine formula
function getDistanceFromLatLonInKm(lat1,lon1,lat2,
   lon2) {
  console.log("Values:␣"+lat1+",␣"+lon1+",␣"+lat2+"
     ,␣"+lon2);
  var R = 6371; // Radius of the earth in km
  var dLat = deg2rad(lat2-lat1);  // deg2rad
     function declared above
  var dLon = deg2rad(lon2-lon1);
  var a =
    Math.sin(dLat/2) * Math.sin(dLat/2) +
    Math.cos(deg2rad(lat1)) * Math.cos(deg2rad(lat2
       )) *
    Math.sin(dLon/2) * Math.sin(dLon/2);
  var c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-
     a));
  var d = R * c; // Distance in km
  return d;
}

var distance = getDistanceFromLatLonInKm(parseFloat
   (b.value),parseFloat(l.value),parseFloat(t.value
```

```
    ),parseFloat(r.value));
var radius = distance/2;
```

With the centre and radius calculated, all the parameters for querying by location are determined and the variable "*geoParam*" is filled with them to send the data in the required format.

```
var geoParam = mapCentreLat.toString()+","+
    mapCentreLng.toString()+","+radius+"km";
```

4. Make the query and add the photos to the list: This query is made using "_500*px.api*" function. After receiving the response the photos, if aperture or focal length are required to be filtered they are sent to the metadata analyzer server, and if not whey will be attached to the list on the index as the final result.

6.1.4 ServideClient.java

It is the class calling the web services on the metadata analyzer server. It receives the data to be sent to the metadata analyzer server, establishes the connection to it, creates an instantiation of the web service and sends the data.

6.1.5 metareader.jsp

It is the class doing the connection between the "*ServideClient.java*" and "*script.js*" or "*script*500*px.js*". As mentioned while explaining "*script.js*" in section 6.1.2 on page 33, a jQuery call is made to send data to this class where they are saved on variables after checking that they have data. Once all data is retrieved, ans instantiation of "*ServideClient.java*" is called to perform the delivery to the metadata analyzer server.

6.1.6 callback.html

The callback HTML file that is obligatory for 500px library. It must be located in the same directory as "*script*500*px.js*" and should not be modified.

6.2 The metadata analyzer server

In the metadata analyzer server it is running the application which reads and filters the data. It is a stand-alone application which can work with the developed client or another different one sending data in the proper way to its published web services. It is programmed in Java and uses the "$metadata - extractor$" library [9] in its 2.7.2 version.

6.2.1 Reader.java

This is the heart of the application and it only contains one function, "$readAndFilter$" and is the one reading the images and making the filtering. It receives three parameters:

- url: A String containing the URL of the image to be read.
- metaValues: An Array of Strings, containing the value of the parameters to be analysed. In this approach, they will be the aperture and the focal length values.
- filterNames: An Array of Strings, containing the name of the parameters to be analysed. In this approach, they will be "$Aperture$" and "$Length$".

Only the parameters to be analysed will enter in the function, meaning that if the user has chosen to filter images by aperture, the "$metaValues$" and "$filterNames$" will be 1 size long, having only values corresponding to aperture. Inside the function using metadata descriptors will be declared and initialised.

```
ExifSubIFDDirectory exifSubIFdirectory = metadata.
    getDirectory(ExifSubIFDDirectory.class);
ExifSubIFDDescriptor exifSubIFdescriptor;
XmpDirectory xmpDirectory = metadata.getDirectory(
    XmpDirectory.class);
XmpDescriptor xmpDescriptor;
```

Next, metadata directory names are saved on "$directories$" String array and then starts the filtering. The filtering compares the value passed by the user and the value of that filter read from the image, and if satisfies the condition, a "$true$" value is added to the "$results$" list.

```
exifSubIFdescriptor  = new ExifSubIFDDescriptor(
    exifSubIFdirectory);
auxiliarString = exifSubIFdescriptor.
    getApertureValueDescription();
if (auxiliarString==null) auxiliarString =
    exifSubIFdescriptor.getMaxApertureValueDescription()
    ;
if (auxiliarString!= null){
   auxiliarString = auxiliarString.substring(1);
   auxiliarString = auxiliarString.replaceAll(",", ".");
   auxiliarValue = Float.parseFloat(auxiliarString);
   if (auxiliarValue <= Integer.valueOf(metaValues[0])){
     results.add(true);
   }else{
     results.add(false);
   }
}
```

Listing 1: A fragment of the method where a condition is checked

Once all conditions are checked, a loop iterates into the "*results*" list to see if any condition returned "*false*". If only one condition returns "*false*" the "*readAndFilter* will return a "*false*" value meaning that the image does not satisfy all the conditions established by the user, and if not, it will return a "*true*" value meaning that it can be returned to the user because it satisfies all the constraints.

6.2.2 MetaReadersWebServices.java

It is an interface class which abstracts functions. This abstraction is to allow changing their functioning in the future without changing the web services themselves. There are two web services "*readListMetadata*" to read and filter a list of images, and "*readMetadata*" if only one image is wanted to analyse.

6.2.3 MetaReadersWebServicesImplementation.java

Is the implementation of the abstract functions "*readListMetadata*" and "*readMetadata*" declared on "*MetaReadersWebServices.java*".

- "*readListMetadata*": First a "*Reader.java*" instantiation is created, and then iterating using a For loop it analyses calling "*readAndFilter*" function each image from the list received. If the image image satisfies the condition, "*readAndFilter*" returns "*true*" and it is added to a list for further return to the client.

- "*readMetadata*": It works in the same way as "*readListMetadata*" but only with one image, without the For loop to iterate between images.

"*readListMetadata*" runs a for loop to pass through each image direction in the array. Inside them a Reader object is called and

6.2.4 WSPublisher.java

It is the class publishing the WSDL with the declaration of the web services.

7 Performance analysis

To get conclusions from the proposed architecture, its performance has been measured. This procedure has been made in 5 different steps in order to cover the achievements of diverse characteristics.

7.1 Execution time

The first test is about the time that is needed in order to present the images from each service provider. One major aspect of the quality of service is the response time of a web page, and since this is a client oriented software it should be observed how it behaves. To accomplish the task some timers has been programmed. First of all when the user presses the search button a timer gets the time in milliseconds. Secondly, after receiving the response from the API and adding to the list for the user, a second timer gets the time. Then the final time and the starting time are subtracted and the difference obtained, and finally this time is returned so it can be wrote down. The Figure 9 shows the execution times, or time taken to return images to the user, over 15 difference searches.

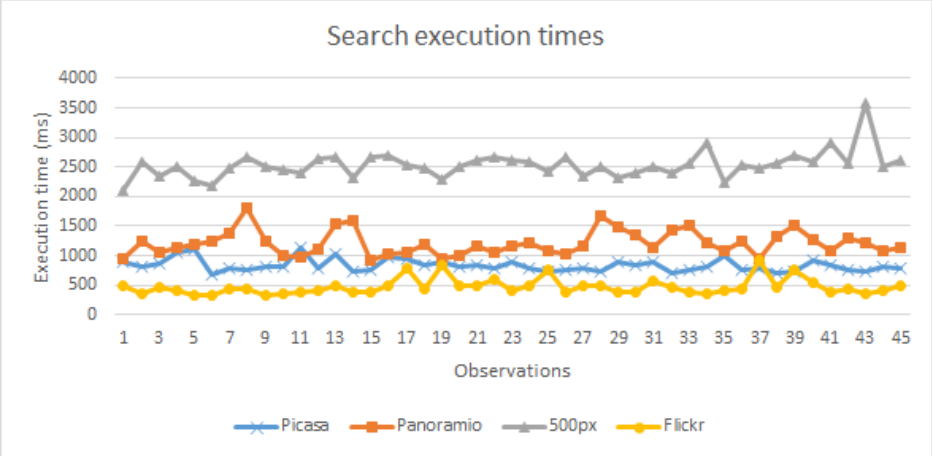


Figure 9: Execution time differences

In each observation the four APIs has been measured simultaneously so the network or computing performance affected all in the same way. Looking at the graphic it can be perceived that some are more regular than others.

Looking into the average execution time on Figure 10 it can be seen that Flickr is the speediest, and 500px the slowest with a really big difference.

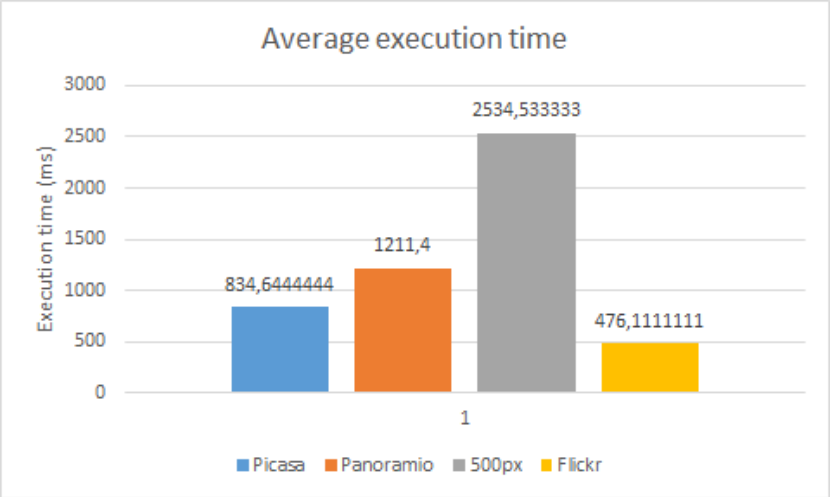


Figure 10: Average Execution times on milliseconds

The average of two seconds and a half that 500px gets to return the query exceeds widely the 1.0 second limit established by Jakob Nielsen as the limit for the user's flow of thought to stay uninterrupted, indicating a too slow performance from the users point of view. Much of this time is due to the Harvesian formula calculation, which when temporally disabling has been observed that speeds up 500px about 500 milliseconds. It is left for future version improve this algorithm for location based search.

7.2 Number of image retrieval cost

Another facet to examine is how the receiving amount of images could affect the response time. In the previous test 100 images has been requested. Since for this project the primary limitations is the amount of queries to be made, the most possible amount of images are wanted for each query in order to get as much as possible images to show to user at once, and avoid this way another unnecessary query. However, more images require more data to be transferred, to be read, to be analysed and to be listed. That is the motivation for this analysis. The data figured on Table 3 show the times collected for this test.

Observations on 100 results queries			
Picasa	Panoramio	500px	Flickr
887	954	2094	481
807	1235	2589	352
864	1054	2349	459
1058	1145	2516	417
1118	1186	2270	340
677	1253	2176	340
787	1379	2477	449
762	1820	2672	446
808	1239	2500	334
822	1007	2465	344
1129	986	2389	384
778	1105	2643	402
1022	1535	2661	484
742	1604	2325	379
753	927	2666	389

Observations on 50 results queries			
Picasa	Panoramio	500px	Flickr
815	733	1670	301
648	701	1904	315
602	657	1789	488
674	746	1653	313
647	605	1362	338
583	659	1660	368
581	660	1517	297
584	616	1921	326
677	711	1426	276
573	597	1432	380
561	734	1437	345
599	582	1544	301
580	943	1485	341
569	765	1605	357
613	801	1505	308

Table 3: Execution times on milliseconds, depending on result amounts

Figure 11 shows the execution time when requesting 100 images and Figure 12 when requesting 50 images. It can be seen that the behaviour is similar, but it seems that if more images are required more time it takes and that in fact the execution time does depend on the requested amount of images.

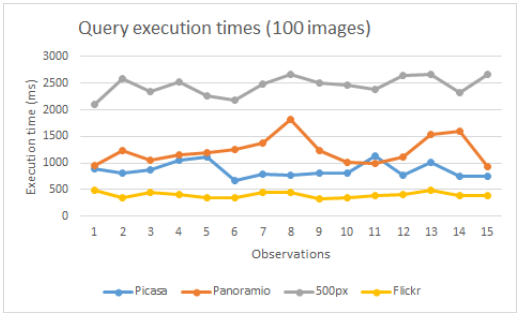


Figure 11: Execution times on milliseconds with 100 results

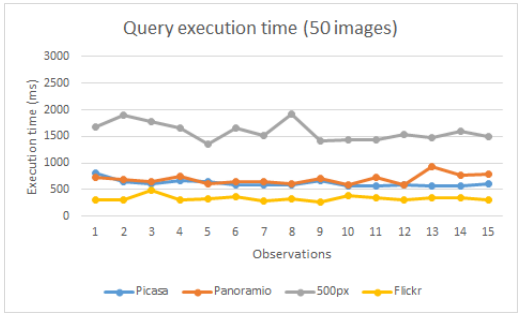


Figure 12: Execution times on milliseconds with 50 results

To visualize the difference the averages are calculated and represented on

Table 4. It definitely confirms that more results take more time.

	Picasa	Panoramio	500px	Flickr
100 images	867,6	1228,6	2452,8	400
50 images	620,4	700,6666667	1594	336,9333333

Table 4: API average execution times depending on result amounts

Figure 13 depicts more graphically the data from the table and differences between both searches can be clearly perceived.

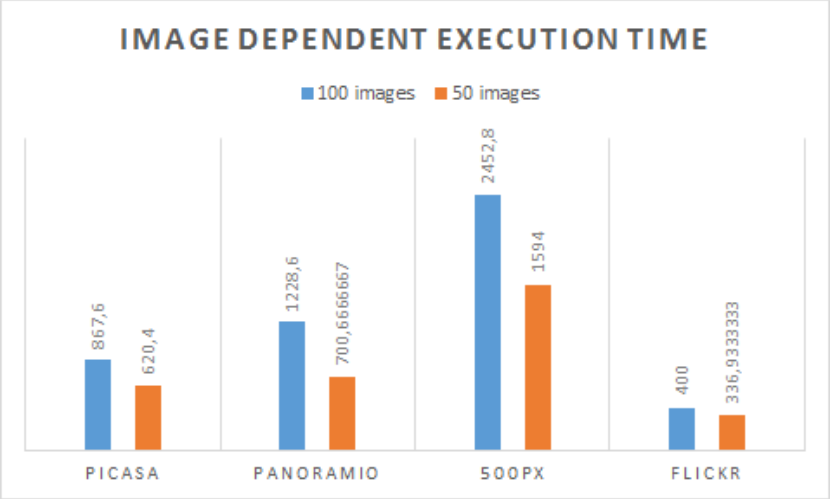


Figure 13: Comparison of API average execution times depending on result amounts

In conclusion, more images at once spends more time on being processed. In contrary, less images at once implies possibility of more queries per user, and hence, in global. However, while Flickr does not take much more time in comparison, with 100 images request Panoramio almost doubled its time and 500px increases its execution time in almost one second.

7.3 Authentication time overhead

Due to all the process of checking if the user is logged in and token validation, the speed could be affected to. Since 500px is the only API supporting

OAuth 2.0 from the implemented ones, the measures are only considering its performance and conclusions will be deducted from its observation. Future implementation of more OAuth 2.0 supporting services may shed more light on the question. Figure 14 shows the speed difference of the application with authorization algorithm and without it when requesting 100 images. In the other hand Figure 15 shows the performance when requesting 50 images. It is needed to point that the authorized queries does not count the time spent by the user introducing its credentials, and only analyses if once the user is authenticated the validation algorithm slows down a perceptible time.

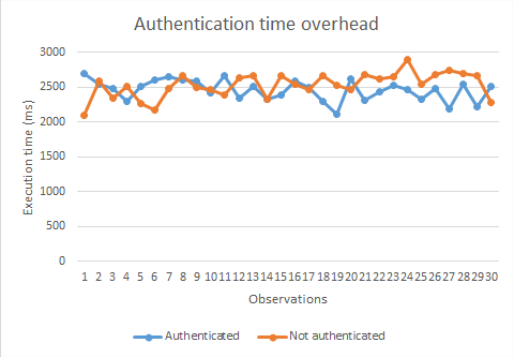


Figure 14: Authentication time overhead with 100 results

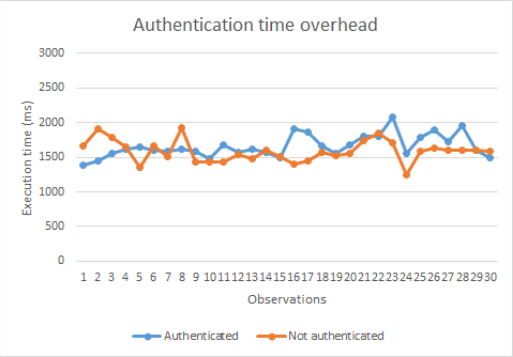


Figure 15: Authentication time overhead with 50 results

It cannot be noticed any significant pattern despite unstable execution times. Figure 16 shows the difference of average times to make easier to see the difference. At this time, the difference is not significant and in fact is contradictory in comparison, meaning that it has no effect and that the difference is only due to unstable execution times, so the hypothesis from the beginning is rejected.

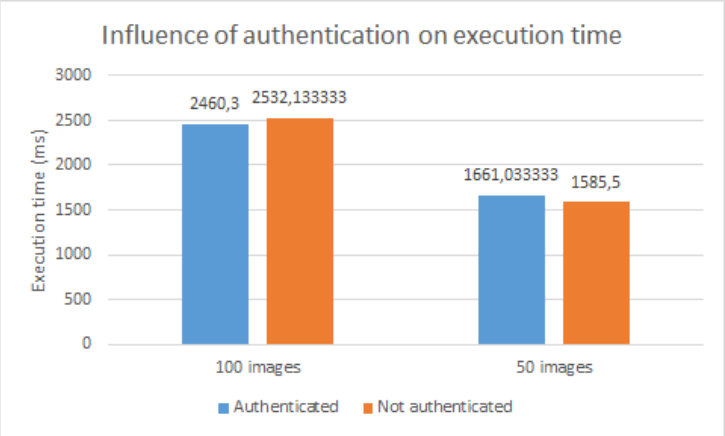


Figure 16: Influence of authentication on execution time

7.4 Extra time for metadata analysis

In all of this tests the metadata analyzer server has been separated in order to only measure client side implications. Searching by aperture and focal length means sending data to server to analyse, and this clearly slows down the system. This amount of time needs to be measured as well to judge the performance of this part of the design. To do so, timers has been included in the metadata analyzer server application as it has been done early on the web page. The measured time represents the cost taken in analysing 100 images metadata.

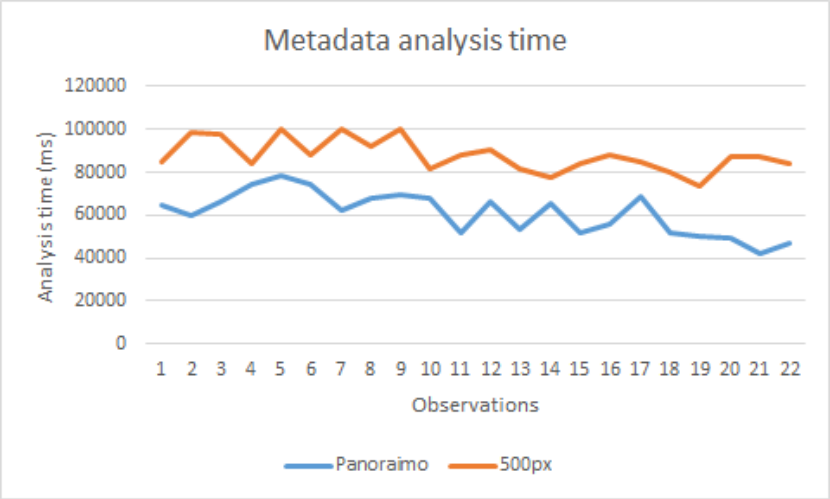


Figure 17: Time cost for metadata filtering

The results on Figure 17 show that as suspected the metadata analysis on the metadata analyzer server slows down the overall speed. For each image lot of time is spent, which added to Panoramio and 500px, the slowest APIs of the client, increases too much the global time to be waited until the search is completed.

8 Conclusions and future work

- A required analysis have been done about the state and the limitations of different image repository APIs. This analysis is the base for all the decisions taken throughout the project and the implemented design.

- A necessary software has been developed to access image repositories and provide user the requested images. The client and server side are both completely functional. Server-side application can also work for other client services accessing its web services, and client-side web page can do basic image search without depending on the server.

- A different design has been proposed based on client side federation. The limitations of the APIs of image repositories pushed to propose an alternative architecture of the common centralized one on federated systems. Federating on the client application led to alleviate that limitations. However, due to API restrictions there are limitations to improve scalability.

- As seen on the performance tests, decisions made to handle scalability issues affected speed for returning images and, hence, in some way usability. If scalability is benefited then usability is sacrificed, and otherwise.

- The metadata analysis on the metadata analyzer server slows down the overall speed. Until JPSearch succeeds on the market, analysing the metadata manually will be a problem for a slow execution time.

- Client broker approach helps incrementing scalability, although it does not resolve the problem completely. Server broker centralized everything and despite it has a common approach for federated searches and is a good option, due to API limitations it was not viable.

- The difference of JPSearch and the software developed along this project is that with a JPQF supporting client, a new JPQF supporting image provider can be added without much complications, and in the other hand, with the system implemented on this project the complexity of increasing the image repositories reside on the metadata given by the provider, the required extra analysis, the programming of a possible different querying format etcetera. With a good design using JPQF the

addition of new repositories could be automated, but without a standard functioning in the provider servers, like the actual environment, the addition of new repositories implies deep changes on the code.

8.1 Future Work

For further development some characteristics could be improved.

- The implementation of CXF could be studied for increasing web service security and improvement on of JAX-WS technology.
- It is interesting to analyse the possibility of the reliability to develop a database where store images with its databases for a limited time.
- Implement repositories using OAuth2 and the authentication method for all the actual supporting services.
- The appearance of the web page could be improved showing results on a more visual way instead of a list.

9 Bibliography

References

[1] Pere Toran and Jaime Delgado. Buscador d'imatges basat en un broker i reescriptura de queries. Master's thesis, Universitat Politecnica de Catalunya (UPC BarcelonaTech), Barcelona, Spain, 2010.

[2] Ruben Tous, Jordi Nin, Jaime Delgado, and Pere Toran. Approaches and standards for metadata interoperability in distributed image search and retrieval. In *Database and Expert Systems Applications*, pages 234–248. Springer, 2011.

[3] Pere Toran and Jaime Delgado. Image search based on a broker approach. In *11th International Workshop of the Multimedia Metadata Community*, 2010.

[4] Mohamed Adel Serhani, Abdelghani Benharref, Elarbi Badidi, and Salah Bouktif. Scalable federated broker management for selection of web services. *The Computer Journal*, 55(12):1420–1439, 2012.

[5] Dong Nguyen, Thomas Demeester, Dolf Trieschnigg, and Djoerd Hiemstra. Federated search in the wild: the combined power of over a hundred search engines. In *21st ACM Conference on Information and Knowledge Management, Proceedings*, pages 1–5, 2012.

[6] Mario Doller, Ruben Tous, Frederik Temmermans, Kyoungro Yoon, J-H Park, Youngseop Kim, Florian Stegmaier, and Jaime Delgado. Jpeg's jpsearch standard: Harmonizing image management and search. *MultiMedia, IEEE*, 20(4):38–48, 2013.

[7] Contributors to the photo metadata survey of controlledvocabulary.com and members of the Photo Metadata Working Group of the IPTC. Social media sites: photo metadata test results. `http://www.embeddedmetadata.org/social-media-test-results.php`, 2013.

[8] Tye Shavik. 500px javascript sdk. `https://github.com/500px/500px-js-sdk`, November 2012.

[9] Drew Noakes. metadata-extractor. `https://github.com/drewnoakes/metadata-extractor`, 2015.