Web Service Search on Large Scale

Nathalie Steinmetz[1,2], Holger Lausen[2], and Manuel Brunner[2]

[1] Semantic Technology Institute (STI) Innsbruck, University of Innsbruck, Technikerstrasse 21, A-6020 Innsbruck, Austria
nathalie.steinmetz@sti2.at
[2] Seekda GmbH, Grabenweg 68, A-6020 Innsbruck, Austria
firstname.lastname@seekda.com

Abstract. The Web is nowadays moving from a Web of data to a Web of services. In this paper we present our approach for Web Service discovery on Web scale, targeted to support flexible and on-demand Web Service usage on the Web. The approach starts with crawling the Web for Web Services, gathering on the one hand WSDL service descriptions and related documents, and, on the other hand, Web APIs. We describe our methodology for building unique service objects from multiple Web resources. Then we provide an overview of how we extract basic service information from all the data and use it to semantically annotate the resulting services.

1 Introduction

The Web is currently changing from a Web of pages to a Web of services, that is instead of mainly assembling static documents the Web is more and more collecting and offering access to Web Services. With Web Services technologies all possible functionalities can be exposed and used in multiple ways. They can be flexibly integrated both in traditional software systems and in Web pages like for example Web 2.0 style portals. This way they provide a new ground for interoperability of business logics. Most Web Services are published using either WSDL (Web Service Description Language) or following a RESTful (Representational State Transfer) approach. For users to be able to use a service they need first to be aware of the existence its particular features.

In the beginnings of the Web Service era, UDDI[2] was proposed as solution to publish and search services, but the standard has not prevailed in the domain of publicly available Web Services. Today Web Services are often registered on specific portals (e.g. XMethods[1] or ProgrammableWeb[2]) or are simply put on the Web together with some Web pages describing the features of the service. This leads to two main ways how services are searched today: over the specific portal's search functionalities or using standard search engines and keyword search. [1] and [5] discuss the efficiency of these approaches and outline some related

[1] http://www.xmethods.net
[2] http://www.programmableweb.com/

L. Baresi, C.-H. Chi, and J. Suzuki (Eds.): ICSOC-ServiceWave 2009, LNCS 5900, pp. 437–444, 2009.

problems like outdated or missing data. [6] provides a quantitative analysis of Web Service search using these methods.

Our approach allows discovery of publicly available Web Services, both WSDL and RESTful ones, by (1) performing a focused Web crawl, (2) identifying relevant documents and (3) aggregating available information to lightweight annotations of the services. Using this approach we collected the largest pool of (WSDL) Web Services known of (June 2009: more than 28.000 services from around 8.000 providers).

2 Crawling the Web for Services

The big success of search engines today is only possible due to efficient crawling solutions. A crawler exploits the fact that Web pages are interlinked with hyperreferences: by following the links found in a set of initial pages (seed) a crawler discovers more URLs. These (yet) unvisited URLs build the frontier of a crawl, distributed on multiple queues (e.g., one queue per host or per IP). The frontier is dynamic and grows according to the scope of a crawl. A scope defines which of the newly found URLs will be disregarded and which will be queued. Scoping and priority assignments to queues and URLs are the most important aspects in building a focused crawler like our Web Service crawler. To focus our crawler on that part of the Web that is relevant to Web Services is important, as due to (a) the size of the Web, (b) restricted resources and (c) time constraints, it is unrealistic to provide a complete coverage of the whole Web.

Our focused crawler is based on the Internet Archive open-source crawler Heritrix[3]. It has been designed in a modular way that allows extensions for all relevant aspects such as scoping of URLs, queue assignment strategies, URL precedence, etc. [8]. We focus our crawl on WSDL files, on related documents, as well as on Web pages that informally describe Web APIs (a.k.a. RESTful services).

2.1 WSDL Crawling Strategies

There are several aspects we need to take into account when crawling for WSDL Web Services and related information:

Seeds. The seed URLs that we use to start a crawl are relevant for the success. We collect them in a semi-automatic process that involves, e.g., screening of well known sites, like the specialized portals mentioned in Section 1 and a selection of URLs from previous crawls.

WSDL Identification. We concentrate our search on service descriptions and related documents, which are mostly stored in textual files. That said, we do by default reject a lot of content in our crawls, like images, audio or video

[3] `http://crawler.archive.org/`

files. We specifically want to look at pages like HTML, XML, PDF, other text documents, i.e. all types of files that could either contain a service description or a related information. During the crawl process we check whether a fetched XML resource is a valid WSDL description and whether it refers to publicly accessible endpoints.

Related Information Identification. Related information may consist of provider documentation of the service functionality, provider Web pages, Wikis, Blogs, FAQs, user ratings and many more. The documents may be pointing to the service, the service provider its service definition or may also not directly be linked to the service. As a first step we consider the inlinks and outlinks of the WSDL documents, i.e., those resources that include links pointing to the service interface description and vice versa. We can gather this information from the crawl link graph that is being written during a crawl iteration; the crawler follows the outlinks in a given page and writes the from-link and its outlinks into a link graph. The task of collecting related information is split onto the crawl run-time and the post analysis of the data because those documents that point to the WSDL descriptions, i.e. the inlinks, cannot be identified during the running crawl and are collected in a post-processing step by iterating through the crawl link graph (see also Section 4).

But it is not yet sufficient to collect related information only by relating outlinks and inlinks to the service descriptions: this way a lot of information may stay hidden to us (e.g., a price page published by the provider but not linking directly to the service description). This leads to another way of detecting information related to services: looking at term vector similarities. We assume that by looking at the term vectors of pages we are able to assess the similarity between documents and services and can thus conclude that they are related. We though calculate at crawl run-time the term vectors of all fetched pages and store them. The analysis, i.e., the term vector similarity comparison, is done afterwards in the postprocessing step. Clearly we cannot apply this approach blindly on all fetched documents, as this would require far too much computing power and time. We restrict our approach to checking the similarity of the term vectors of services to the term vectors of documents fetched from their respective provider domains, which we screen more intensively than other domains.

Queue and URL Scheduling. As we mentioned already before, queue and URL scheduling are very important means to focus a crawl. The crawler creates new queues per top-level domain, i.e. per host. Influencing the URL and queue scheduling means (in the specific case of Heritrix) allocating costs or precedences to URLs and/or queues before they are being scheduled by the frontier (low cost or high precedence meaning the URL or queue is being scheduled more upfront). We have developed an approach for URL cost assignment that is targeted on the prioritizing of (assumable) Web Service related documents. We set the cost of each new URL by default to 20. Then we check the URL for negative features that we penalize by increasing the costs (e.g. a lot of subdomains, more than one query string, more than one path segment). Afterwards we start privileging

positive aspects of the analyzed URLs by reducing the costs (e.g. URLs that contain "?wsdl", "ws", "service", "api").

As last step we take into account a score that we calculate for the provenance page, i.e. for the 'from-link' URL whose outlinks we are currently assigning costs to. A high score means that it is rather probable that this page is somehow talking about Web Services; we assume that a page that is talking about services might with a high probability link to other pages that talk about services. We calculate the score based on the number and position of the occurrence of Web Service related terms in the page's content, taking as well into account HTML mark-up (e.g. words appearing in the title text or being marked bold). Finally we reduce the costs of the outgoing links by the score of the provenance page.

The aforementioned strategies to set, increase or reduce the costs of URIs cannot be applied to queues. Here we follow another approach: we set the precedence of the queues to the lowest cost that the URLs within those queues provide. This makes that URLs with low costs, i.e. interesting URLs, automatically enhance the precedence of the queue they are being scheduled in. This way the most interesting URLs should always be processed first. [11] provides a more detailed overview of the Service Crawler's queue and URL scheduling approach.

2.2 Web API Crawling Strategies

Detecting Web APIs on the Web is unlike harder than detecting WSDL files or even related documents. Web APIs are HTML documents, same as other Web pages, differentiated only by the fact that they expose a functionality that can be invoked by (in most cases) adding a specific query string to the URL that then calls a specific method in the background (e.g. `https://api.linode.com/api/?api_key=cakeisgood&action=domainGet&DomainId=45F33`). RESTful services, as introduced in [4], are usually a lot easier to create than WSDL services, use basic HTTP request methods (like GET, POST, PUT, DELETE) and are quite understandable for humans. We mostly use the term *Web API*, instead of REST service, as an API may represent a REST service, but it can also represent a service that is not strictly RESTful (following the definition in [4]). We have developed two different initial approaches to tackle the challenge of crawling Web APIs, which both are still in a rather experimental phase and not yet as well matured and evaluated as the WSDL crawling approach. We will outline the evaluation approach for the Web API crawling in Section 5.

Automatic Classification Approach. Our first approach follows a traditional data mining approach: text classification. Automated classification (also called categorization) of texts has become quite important as in recent years huge amounts of digital documents are becoming available[9]. There are two major types of text classification: supervised and unsupervised[7] learning approaches. In short, supervised learning works with a positive example set, i.e., a set of already classified documents, which is taken as input and used to produce a class label prediction (the so-called classification). Unsupervised learning functions are used when there is no training set available for the machine learning tool.

In our approach we use a supervised learning algorithm, concretely the Support Vector Machine (SVM) model[7]. We used Web API documents that we collected from ProgrammableWeb[4] as positive example set. The automatic classification is done within the crawler, by adding a classification processor into the regular crawl environment. As classifier we use the open-source data-mining tool RapidMiner[5].

Term Frequency Approach. Our second approach is based on term frequencies and tries to tackle, amongst others, the weaker aspect of the automatic SVM approach: the fact that it is only based on words and does not take into account HTML structures and mark-ups. We might as well want to take into account the URL of a Web document, which often contains words describing the topic of the page. Another relevant aspect covers the syntactical properties of the language used in Web API homepages. Most times they contain a higher amount of camel-cased words than random pages (e.g. `getDocument`) and often they contain fewer external links than usual. Often Web API homepages also contain internal links that target to the same domain, e.g., example calls for the described API.

We have created three indicators that group all the relevant parameters: *API*, *Documentation* and *Web*-related. The *API* indicator takes into account the appearance of keywords like "api", "developer", "lib", "code", "service", etc. in the URL and/or content of a page and looks for a high amount of camel-cased words. The *Documentation* indicator looks for keywords like "dev", "doc", "help", "wiki", etc. in the URL of a page and checks the page's content for the number of outlinks and camel-cased words. The *Web*-related indicator takes into account keywords like "rest", "web service", "api", etc. in the URL and/or content of a page and looks for a high amount of inner domain links in the page's content. [11] describes the parameters and indicators in more detail. Each of these indicators is regarded individually and is assigned a score that indicates to what level a specific document complies with this indicator: the three indicator scores need to be over a specific threshold in order to mark the specific page as Web API.

3 Building Unique Service Descriptions

After having harvested the Web for Web Service descriptions - both WSDLs and Web APIs - we remain with a large amount of service descriptions and related documents. But not all of these service descriptions correspond to exactly one unique service. That is, we do not have a one-to-one mapping from service descriptions to actual services. E.g., one WSDL can contain more than one single service, each bound to different endpoints. But even more usual is the case that multiple WSDLs are out there that resume to one single service. Often service

[4] `http://www.programmableweb.com/`
[5] `http://rapid-i.com/content/blogcategory/38/69/`

descriptions are hosted on more than one server, even sometimes from more than one provider.

We have developed an approach to deduplicate WSDLs, i.e. to build unique service objects that each represent single unique services. Our first step is to extract the provider from the service description endpoint. This is a non-trivial step, as it is not clear what is an authority and what is a registered domain. Since there is no algorithmic method for finding the highest level at which a domain may be registered for a particular top-level domain, we use the Public Suffix List[6] instead. An example would be the URL `http://www.library.uibk.ac.at/test.wsdl`, where the provider resolves to uibk.ac.at, the domain of the University of Innsbruck. Next we build a new unique (seekda) URL for the service. This URL contains first the provider's name and is then completed with the service name (e.g. `http://seekda.com/providers/cdyne.com/IP2Geo`).

If one service assembles a set of WSDLs under one umbrella, we, as last step, need to choose one service description that we present as the main one to the user. We do so by choosing the URL that has the shortest path and the less subdomains and - if available - belongs to the service provider domain. While this might not always be the right choice, we think of it as a good starting indication. This deduplication approach is so far restricted to the crawled WSDL service descriptions; for Web APIs we create a similar unique identifier for each, which contains the provider name and the hash value of the Web API URL instead of the service name (as we do not know the name of a service in that case).

After a crawl iteration where we have more than 200.000 WSDL service descriptions (see Section 2), we apply our algorithm for service deduplication on it and remain with more than 28.000 unique Web Services.

4 Automatically Enriching Service Descriptions

We do not stop the analysis of the data we gathered during the crawl at the deduplication of WSDL services. To each one of the unique service objects we try to append some more information. We store this service meta-data in RDF triples, using as structure ontologies that have partly been developed in the scope of the Service-Finder project: *Service-Finder Ontology*[7] and *seekda Crawl Ontology*[8].

The first meta-data that we store is the relation between services, their providers and their related documents, whereas these documents refer to both WSDLs and other documents. As already described in the sections 2 and 3, we collect this information by (a) going through the link graphs stored by the crawler and (b) when building the unique service objects.

Other meta-data that we collect refers to basic information that we can extract around the fetched Web Services. One information bit that we gather around the service endpoint is the geographic location of the service, that is the country

[6] `http://publicsuffix.org./`
[7] `http://www.service-finder.eu/ontologies/ServiceOntology`
[8] `http://seekda.com/ontologies/CrawlOntology`

where the service is located (i.e. hosted). Another basic information that we extract for all services is their liveliness, i.e. their availability and response times. seekda is monitoring and storing these data on a daily basis, and provides a corresponding availability graph with the service details.

Concerning the Web APIs we store initial meta-data concerning the three indicators that we have described in Section 2.2, i.e. the indicator scores, the number of camel-cased words, the number of external or inner links, etc. This information can be used in later stages to refine the focused crawl approach for this service type.

All the meta-data extracted about the two kinds of services as described above can be concluded by the crawler or a direct postprocessing analysis of the fetched data (without the need of complex information extraction, e.g.). This data can be used in several ways: to improve semantic service discovery, to provide service ranking (based e.g. on the availability of services) or to provide the users of a service discovery engine with more information on a service than only its technical description (see e.g. Service-Finder Portal, SOA4All studio).

5 Evaluation

We have evaluated our service crawling approach according to several indicators, which are different for the WSDL and Web API approaches.

The WSDL and related information crawling approach was evaluated by, on the one hand, pure performance measure indicators (e.g., documents crawled per second, kB crawled per second) and, on the other hand, indicators that refer to the quality of the resulting data, i.e. how much relevant information could be found (e.g., number of WSDL documents, number of extracted service identifiers). [3] provides detailed evaluation results using these indicators, comparing three different crawl iterations.

For evaluating the Web API crawling approach we have created three data sets: one set with random Web pages, one set with Web API Homepages from ProgrammableWeb.com and one set with Web pages from the domain of "programming languages", taken from the dmoz.org directory. We have then run our classifier over these three sets and evaluated the results. Detailed results of the evaluation can be found in [10].

6 Conclusion and Future Work

In the scope of this paper we have described our approach for discovering Web Services and related information on large scale, taking into account both WSDL services and Web APIs. We have showed how we focus a Web Crawler to retrieve as many as possible services and service-related information. The fact that there is no one-to-one mapping between WSDL service descriptions and actual services has led us to introduce new unique service objects and identifiers that assemble all duplicate services under one umbrella. We have shown how we relate the crawled data, i.e. the related information and the WSDLs, to the

services and how we store this meta-data. Also we have provided an overview of what other meta-data can currently be extracted from the raw crawl data. Finally we have provided an overview of our evaluation approach for the three crawling approaches: one WSDL and related information approach and two Web API crawling approaches.

Some of the major issues that can be tackled in the future to improve the Web Service crawling and analysis include the deduplication of related documents, the detection of new means to find service related information on the Web, the refinement of the Web API crawling and the unification of the current two Web API classification approaches.

Acknowledgements

The work is funded by the European Commission under the projects Service-Finder and SOA4All.

References

1. Bachlechner, D., Siorpaes, K., Lausen, H., Fensel, D.: Web service discovery - a reality check. In: 3rd European Semantic Web Conference (2006)
2. Bellwood, T., Clément, L., Ehnebuske, D., Hately, A., Hondo, M., Husband, Y.L., Januszewski, K., Lee, S., McKee, B., Munter, J., von Riegen, C.: Uddi version 3.0 (July 2002)
3. Brockmans, S., Celino, I., Cerizza, D., Valle, E.D., Erdmann, M., Funk, A., Lausen, H., Schoch, W., Steinmetz, N., Turati, A.: D7.3 - assessment of tests for alpha release and revised testing scenarios and evaluation criteria for beta release (2009)
4. Fielding, R.T.: Architectural Styles and the Design of Network-based Software Architectures. PhD thesis, University of California, Irvine (2000)
5. Lausen, H., Haselwanter, T.: Finding web services. In: 1st European Semantic Technology Conference (2007)
6. Lausen, H., Steinmetz, N.: Survey of current means to discover web services. Technical report, STI Innsbruck (August 2008)
7. Moens, M.-F.: Information Extraction: Algorithms and Prospects in a Retrieval Context. Springer, Heidelberg (2006)
8. Mohr, G., Stack, M.: An introduction to heritrix. In: 4th International Web Archiving Workshop (2004)
9. Sebastiani, F.: Machine learning in automated text categorisation. Technical report, Consiglio Nazionale delle Ricerche (1999)
10. Steinmetz, N., Lausen, H., Brunner, M., Martinez, I., Simov, A.: D5.1.3 - second crawling prototype (2009)
11. Steinmetz, N., Lausen, H., Kammerlander, M.: Crawling research report - version 1 (2008)