University of Innsbruck

Semantic Technology Institute (STI) Innsbruck

A thesis submitted for the degree of
Master of Science

Web Service Location

handed in by
Nathalie Steinmetz

supervised by Univ.-Prof. Dr. Dieter Fensel
and co-supervised by Dipl.-Inf. Holger Lausen

Innsbruck, August 2, 2010
Contents

1 Introduction 4

2 Background 9
   2.1 Web Crawling 9
   2.2 Service Discovery 16
   2.3 Service Ranking 17

3 Web Service Location 18
   3.1 Web Service Crawling 18
   3.2 Analysis and Output of the Crawl Data 38

4 Web Service Ranking 51
   4.1 Feature Aggregation Based on Ontologies 51
   4.2 Multi-valued Ranking 54

5 Evaluation 60
   5.1 Service Crawling 60
   5.2 Service Ranking 67

6 Conclusion And Future Work 71

A Appendix 77
   A.1 System Requirements 77
   A.2 Installation and Launch 78
   A.3 Configuration 81
   A.4 Crawling Output 85
List of Tables

3.1 Web Service Domain Term Vector ........................................ 30
3.2 Example Term Vector ...................................................... 31
3.3 Raw Crawl Data - Crawl A .................................................. 41

5.1 Service Crawler Throughput Performance Indicators ............... 61
5.2 Service Crawler Focus Performance Indicators Crawler .......... 61
5.3 Throughput Performance Results: Crawl 1 - Crawl 4 ................. 63
5.4 Focused Crawl Performance Results: Crawl 1 - Crawl 4 .......... 64
List of Figures

1.1 Crawler role within the overall Web Service search engine structure 7

3.1 High-level Architecture and Data Flow of the Service Crawler . . 20
3.2 High-level View on Heritrix’ Processor Chain . . . . . . . . . . . 33
3.3 Heritrix Crawl Configuration Web Interface . . . . . . . . . . . . 34
3.4 Ontologies Used for Crawl Data . . . . . . . . . . . . . . . . . . . 49

5.1 Network usage during (a) Crawl 1, (b) Crawl 2, and (c) Crawl 3 66
5.2 CPU usage during (a) Crawl 1, (b) Crawl 2, and (c) Crawl 3 . . 68
5.3 WSDLs/service: (a) Crawl 1, (b) Crawl 2, (c) Crawl 3, (d) Crawl 4 69
5.4 Docs/service: (a) Crawl 1, (b) Crawl 2, (c) Crawl 3, (d) Crawl 4 70

6.1 Recently crawled services at the seekda Web Service search engine 73
6.2 Keyword-based search results on the Service-Finder portal . . . . 74
6.3 Ranking options at the seekda Web Service search engine . . . . 75
## Listings

3.1 PHP server auto-generated WSDLs ........................................ 25  
3.2 Example of a link graph .................................................. 39  
3.3 ARC Archive Header entry ............................................... 40  
3.4 Services to WSDLs index example ...................................... 44  
3.5 WSDLs to services index example ...................................... 44  
3.6 Services to inlinks index example ..................................... 44  
3.7 Archive and index files structure ..................................... 45  
3.8 Services to related documents index ................................... 45  
3.9 Crawl meta-data structure .............................................. 50  
4.1 WSDL service and related documents meta-data used for ranking 53  
4.2 WSDL meta-data used for ranking ...................................... 53  
4.3 Monitoring meta-data used for ranking ................................. 54  
4.4 seekda Ranking Ontology ............................................... 54  
4.5 Calculation of the Related Documents Rank .......................... 56  
4.6 Calculation of the WSDL Metrics Rank ................................. 57  
4.7 Calculation of the Monitoring Rank .................................... 57  
4.8 Global Rank Calculation for WSDL-based services .................. 58  
A.1 Service Crawler launch procedure .................................... 81  
A.2 Service Crawler broad crawl configuration file ...................... 82
Abstract

In the last years we see a clear trend in the Computer Science area to a move towards Service-Oriented Architectures (SOAs). This move can be observed both on the Web - with the Web of documents changing to a Web of services - and in the traditional software domain - where software starts being seen more and more as resource that itself is a service in a cloud. Research in the service domain encompasses its whole life-cycle, including topics as creation, discovery, selection, ranking and composition. This thesis focuses on the discovery and ranking of Web Services. We describe the methodologies that we have developed for enabling Web Service discovery on Web scale, by crawling the Web for Web Services, gathering WSDL service descriptions and related documents, and 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 enrich the resulting services with simple annotations. For Web Service ranking we have developed a novel approach based on non-functional properties of services: information that is available about services by analyzing their description that is available on the Web, their hyperlink relations, monitoring information, etc. We make thereby use of semantic technologies, aggregating the various real-world service aspects as described above in a unified model and providing different rank values based on those aspects.
Publications

This thesis has contributed to the following publications and tutorial.

Publications:

• Nathalie Steinmetz, Holger Lausen and Manuel Brunner. Web Service Search on Large Scale. In the proceedings of the 7th International Conference on Service Oriented Computing (ICSOC), 2009, Springer.


Tutorial:

• Nathalie Steinmetz, Adam Funk and Maria Maleshkova. Web Service Crawling and Annotation. Full-day tutorial at the 2nd Future Internet Symposium, 2009, Berlin, Germany.

Declaration

The work presented here was undertaken within the Department of Computer Science at the University of Innsbruck. I confirm that no part of this work has been previously submitted for a degree at this or any other institution and, unless otherwise stated, it is the original work of the author.

Nathalie Steinmetz, July 2010
Acknowledgements

This thesis has been realized in the scope of my work at seekda GmbH, and I would like to thank both my supervisors and colleagues at the University of Innsbruck and at seekda for enabling this close collaboration. I especially would like to thank Holger Lausen for his productive and unfailing help concerning all technical and non-technical issues related to Web Services and Web Service crawling, as well as Univ.-Prof. Dr. Dieter Fensel for his continuous valuable support.

Moreover the work on this thesis has happened in the scope of the two European Commission FP7 projects Service-Finder and SOA4All and the FFG (Österreichische Forschungsförderungsgesellschaft mbH) FIT-IT project Service-Detective. I am thankful for having had the opportunity for joint work and fruitful discussions with many project partners about the main topics of this thesis, Web Service location and Web Service ranking.

Last but not least I would like to thank my current and former colleagues at seekda and at the University of Innsbruck for inspiring times, as well as my friends and family for all their support.
Introduction

Within the last years we could observe a clear trend both on the Web and in Computer Science in general to a move towards Service-Oriented Architectures (SOAs). While the Web of documents as we used to know it is changing to a Web of services, the traditional software domain is altering as well: software starts being seen more and more as resource that itself is a service in a cloud (following the software model 'Software as a Service', SaaS). Using Web Service technologies all possible functionalities can be exposed and flexibly integrated in all kinds of applications (e.g., traditional software as well as Web pages). This way Web Services provide new means for interoperability of business logics. Another characteristic of Web Services - with high impact potential - is the ability to mash them up to provide new functionalities that may or may not have been intended by the single service providers.

Services can be used both in a fixed and a mobile environment, mobile services becoming a new upcoming trend that increases the importance of services and related technologies even more. The potential of service orientation in the computing area has been recognized both in research and in industry: considerable attention has been given since then to services and related technologies including Web Services and Semantic Web Services technologies.

Various entities, such as companies, public institutions, universities and private persons, are nowadays publishing Web Services, using either the WSDL (Web Service Description Language) standard or following a RESTful (Representational State Transfer) approach. But creating and publishing a service is only the first step in the whole Web Service life-cycle. Following steps include the location of services, exchange of invocation details, agreement on security standards, agreement on terms and licenses, etc. State-of-the-art research in the Web Service domain thus also encompasses the whole service life-cycle, including all relevant topics as creation, discovery, selection, ranking and composition. This thesis focuses on the Web Service discovery and ranking aspects, two crucial steps in making potential users aware of the pure existence of a given service.
In the beginnings of the Web Service era, UDDI[2], a standard describing protocol bindings and message formats required to interact with a Web Service directory, was proposed as solution to publish and search services. This standard has however not prevailed in the domain of publicly available Web Services: all major UDDI Business Registry (UBR) nodes that have once been set up by companies such as IBM, Microsoft and SAP have been discontinued already in 2006. One of the issues UDDI suffered from was the unsatisfactory quality of data; according to [9], e.g., only about 20% of the refereed WSDL files have been valid.

Today the process of publishing Web Services is mainly done in two ways:

- by registering the service at a couple of private or public portals (e.g., ProgrammableWeb[3], StrikeIron[2] or XMethods[4]) that are specialized on listing Web Services.
- by providing access to the service description and at the same time providing some Web pages or other documents describing the features of the service. This information can then be found by the crawlers of standard search engines such as Google[5], Yahoo[6] or Bing[7].

These different ways of publishing Web Services have lead to two main ways how users try to discover the available services:

- Service directories offer a search functionality via a HTML interface to the users. Unfortunately, according to [1] and [11], they only cover small portions of the actually available services and suffer furthermore from the problem of missing or outdated information. Also this approach focuses only on restricted sets of manually registered services.
- Using standard search engines with keyword search it is difficult to restrict a search to Web Services (and exclude unrelated documents, e.g., using filetype:wsdl). [11] and [12] show that this search method provide a big coverage of services, but at the same time has drawbacks and is not highly efficient in terms of recall and precision; it does thus not allow for an efficient service discovery. This is mainly due to the fact that keyword search is done on a pure document level and not on a service level.

[12] provides a more detailed insight and evaluation of the different ways to discover Web Services, presenting quantitative analysis, available information, interface features and accuracy measures of the single approaches.

Another service discovery approach is semantic Web Service discovery [7, 6, 8, 17, 10]. Semantic discovery approaches go beyond keyword based retrieval and
allow to specify functional and non-functional criteria for which suitable services will be found. While semantic discovery achieves a high precision it relies on precise and rather complex descriptions that are presently not available at large scale. Our approach rather focuses on the descriptions which are currently used for the numerous already available services and tries to infer from the existing information some semantic meta-data.

**Contribution of this Thesis**

In the scope of this thesis we provide methodologies for Web Service discovery and ranking on Web scale, with the goal to make potential users aware of the existence of publicly available services and ease their search for specific services. Both approaches are targeted to work over a large number of services - services crawled from the Web.

The service location approach followed in the scope of this thesis allows the discovery of publicly available Web Services published using the WSDL standard. It is based upon active crawling of the Web for service descriptions, collecting public Web Services from all over the world. It does thus not rely on the manual registration of services by service providers. This approach has been developed initially by some of the seekda founders[11] and has been further developed in the scope of this thesis and in the scope of the European projects Service-Finder[9] and SOA4All[9] as well as the Austrian project Service-Detective[10]. The methodology presented within this thesis encompasses steps such as focused crawling for WSDL services and for documents related to these services, building unique service objects, extracting basic service information from the gathered data and enriching the services with simple RDF annotation. Using this approach we collected the largest pool of Web Services known of (June 2010: more than 28,000 services from more than 8,500 providers), presented on the seekda Web Service search engine [http://webservices.seekda.com](http://webservices.seekda.com).

Figure 1.1 illustrates the role of the Web Service crawler within a Web Service search engine environment such as the seekda search engine or the Service-Finder Web Service portal [11]. The Service Crawler is harvesting the Web for WSDL files that describe actual services; once a service has been detected, the crawler starts also looking for any related information (e.g., service documentation, terms and conditions, user provided service ratings or discussions, etc.). The fetched data is then analyzed and semantic annotations to the services - in form of RDF triples - are stored. In order to build a fully-fledged Web Service search engine one can now use the crawled services, the related documents and the extracted meta-data, and provide (a) an index on top of it that allows keyword-based search, and (b) a triple store / ontology repository, as well as a reasoning engine, that allow reasoning on top of the service meta-data. As a last step a user interface needs to build the bridge to the end-user and enable him/her searching

---

for available Web Services, service providers and corresponding documentation.

Concerning service ranking we are proposing a novel approach based on non-functional properties of services: information that is available about services by analyzing their descriptions on the Web, their hyperlink relations, daily monitoring, and similar aspects. After discovery of services, i.e., after users have been made aware of the simple existence of specific services that provide the function they are looking for, ranking is one of the most important steps to support them in selecting the most best-fitting service fulfilling their needs.

Regarding the large size of publicly available services on the Web (e.g., more than 28,000 Web Services found by the seekda Web Service search engine and more than 5,000 Web APIs and Mashups published on the specific Web API portal ProgrammableWeb), it would be a tedious task for a human to filter out the services that are relevant to him.

Structure of this Thesis

The rest of this thesis is structured as follows: in Chapter 2 we introduce some technologies used within this thesis, mainly focusing on Web crawling. In Chapter 3 we describe our approach for focused crawling of Web Services and related documents, and for the analysis of the gathered data. We thereby also provide details on the implementation of the Service Crawler and the Analysis Platform. Chapter 4 describes our service ranking approach, outlining the aggregation of features, based on ontologies, as well as our ranking algorithms. In Chapter 5 we provide evaluations of our Web Service crawling and ranking approaches.
Chapter 6 concludes the thesis and provides some ideas for future work. Finally the Appendix A describes the system requirements and the installation process of the Service Crawler.
2 Background

In the following we introduce the main topics that we are dealing with in the scope of this thesis and present some background and state-of-the-art information on them. Section 2.1 details issues related to Web crawling, the major topic of this thesis, describing basic crawling steps and outlining crawler types and strategies. The sections 2.2 and 2.3 briefly present the current state of research in service discovery and ranking.

2.1 Web Crawling

The big success of search engines today is in part due to innovative and effective solutions for Web crawling. Search engines actually make the most widespread use of Web crawlers when they collect pages on the Web to build their indexes. A web crawler, also called robot or spider, is a software program that starts with a set of URIs, fetches the documents (e.g., HTML pages, service descriptions, images, audio files, etc.) available at those URIs, extract the URIs from the documents fetched in the previous step and start over the process previously described. That is it automatically downloads Web pages and follows links in the pages, this way moving from one Web page to another. This gives us already a hint on how we can modify this classical crawler behavior to focus a crawl on Web Service descriptions. We can, e.g., decide whether we want to follow the links from a specific page or not, depending on various criteria, such as what top level domain this page belongs to, how many links the page contains (it could be a so-called “link farm”), etc. We will investigate such issues more deeply in the coming section on focused crawling.

Now crawling the Web and downloading Web pages sounds rather easy. However, one big issue is the fact that the Web is not static, but quite the contrary, i.e., very dynamic. There are billions of documents available on the Web and crawling all data and furthermore maintaining a good ‘freshness’ of the data becomes almost impossible. The Web changes rapidly, new pages are
added, existing pages are modified and old pages are deleted. To always keep the crawled data up to date we would need to continuously crawl the Web, revisiting all pages we have once crawled. We could do so by over and over again repeating the same crawl and building “snapshots” of the Web or of the part of the Web we are visiting. Whether we need to do this depends a lot on the intention of our crawl. Do we want to archive part of the Web, do we want to crawl one specific part in-depth or do we want to get an idea of how many links lead to a specific document type? [16] describes such possible intentions and corresponding crawling strategies, thereby proposing an adaptive revisiting strategy that is meant to be used for repeated crawls. We will describe two major crawling strategies, incremental and snapshot, in Section 2.1.2.

In general we can say that different crawling strategies are used for different types of crawlers. Crawler types are thus related to the different intentions they pursue when crawling the Web. The main crawl types are:

- **broad or universal crawling**: large crawls with a high bandwidth usage where the crawler fetches a large number of Web sites and goes as well into a high depth on each crawled site.

- **focused or topical crawling**: a number of criteria are defined that limit the scope of a crawl; the crawler fetches similar pages topic-wise, for example

- **continuous crawling**: the crawler continuously visits all URLs in its frontier, i.e., the frontier cannot grow fast and the crawl should be scoped.

Usually crawlers do implement a set of policies that address the issues raised by the different crawling strategies, as, e.g., how to handle the dynamics of the Web, etc. In general we can say that different policies are used for different types of crawlers:

1. a **selection policy** that states which page to download,
2. a **re-visiting policy** that states when to check that a page has changes,
3. a **politeness policy** that states how to avoid overloading Websites, and
4. a **parallelization policy** that states how to parallelize the crawling functionality.

In the following, Section 2.1.1 will describe in more detail the basic crawl steps, while the sections 2.1.2 and 2.1.3 will provide an overview of the above mentioned crawling strategies and crawler types. Section 2.1.4 will then present the major issues that arise around Web crawling.

### 2.1.1 Basic Crawl Steps

In the following we will shortly describe the basic steps that a crawler is executing. What it basically does is executing different specific steps in a sequential
way. The crawler starts by taking a set of seed pages, i.e., the URLs which it starts with. It uses the URLs to build its frontier, i.e., the list of unvisited URLs of the crawler. In the scope of one crawl this frontier is dynamic as it is extended by the URLs extracted from already visited pages. The edge of a frontier will be limited by the number of URLs found in all downloaded documents (and by politeness restrictions that are followed for different servers). If a frontier is not set any limit and if the crawler disposes over unlimited hardware resources, it may grow indefinitely. This can be avoided by limiting the growth of the frontier, either by, e.g., restricting the number of pages the crawler may download from a domain, or by restricting the number of overall visited websites, what would at the same time limit the scope of the crawl.

Whatever frontier strategy is chosen, the crawler proceeds in the same way with the URLs it gets from the frontier. So once a URL is taken from the frontier it traverses the following steps:

1. the crawler checks whether this page is intended to be fetched, i.e., whether there are no rules or policies that exclude this URL
2. the document the URL points to is fetched
3. the crawler extracts links from the downloaded document
4. based on given rules the crawler decides whether he wants to permanently store the downloaded document
5. the crawler feeds the extracted links to the frontier

These steps are executed for all URLs that are crawled by the Web crawler. As it would (a) absolutely minimize the speed of a crawl, and (b) be a waste of resources, a crawler does not proceed the URLs one by one. Although a crawler has only one frontier, the frontier has multiple queues. Queues can be built based on different schemes: e.g., one queue per host. Additionally the queues can be ranked within the frontier which makes then that certain queues are served earlier by the frontier than others. A similar issue is the ranking of the URLs within the single queues. During the setup of a crawl, it must be decided what URLs get what priorities and get thus removed either early or late from a queue to be further processed.

### 2.1.2 Crawling Strategies

When a Web crawl is being designed, this always happens with a specific intention. Often this intention is revealed by the originator of a crawl. Big search engine operators as Google, Yahoo or Microsoft have other intentions than, e.g., national institutions that want to archive one whole country’s Web (as it was done, e.g., by the Nordic Web Archive) or news providers that just crawl the Web for news on one specific topic. So depending on the purpose of a crawl, different crawling strategies may be followed. Two major strategies, as described in [16] are incremental and snapshot crawling.
Snapshot Strategy. In a snapshot strategy the crawler visits a URL only once. If the same URL is discovered again it is considered as duplicate and discarded. Using this strategy the frontier is extended continuously with only new URLs and a crawl can spread quite fast. Using this strategy the crawl operator can take a snapshot of (part of) the Web at one specific moment in time. This snapshot can be done for a broad scope without major problems, just extending the duration of the crawl. The snapshot strategy is though not appropriate for doing crawls that allow to follow the changes on Web sites, as, amongst others, depending on how long a crawl takes it might take a long time until a page is revisited.

Incremental Strategy. The incremental crawling strategy is, as opposed to the snapshot strategy, optimal for doing continuous crawls, i.e., crawls that allow to capture changes on Web sites. This makes that a URL needs to be visited multiple times: when an already visited URL is rediscovered it is not rejected but instead put into the frontier again. Using the incremental strategy the frontier queues will never empty and a crawl could go on for an indefinite long time. If the scope of the crawl is not limited, the frontier will not only revisit known URLs over and over again but it will at the same time continue to expand the crawl field, even though much slower than using the snapshot strategy.

Both crawling strategies can be used with different types of crawls, as will be described in the next section. [16] mentions an example of how both crawling strategies can complement each other: a part of the Web that shall be crawled can be analyzed with regard to how often the pages change. The snapshot strategy can then be used to crawl the given part in a broad and extensive manner once (or on a regular basis), while the incremental strategy can be added to follow the changes on sites that have been identified as changing frequently (e.g., news sites).

2.1.3 Crawler Types

This section provides an overview of three major crawl types. We will explain the purposes that underlie the single types as well as specific issues that appear with these crawler types.

Broad Crawling. Broad crawling (or universal crawling) is the type of crawling that can be used with the purpose of crawling a large part of the Web, if not even the whole Web. Not only the amount of collected Web data is important, but as well the completeness of coverage of single Web sites. Big universal search engines like Google, Yahoo or Bing operate such broad crawls. This crawler type can use both crawling strategies as described in Section 2.1.2. Using the snapshot strategy in a repeated manner over the same seeds, the crawl operator would get single snapshots of the crawled part of the Web that might
though be temporally quite apart. Using the *incremental strategy*, one crawl could lead to incremental updates of a search engine index, or of any other data repository used.

According to [13] the major issues in broad crawls are:

- **performance** - A broad crawler often needs to handle not only thousands, but billions of Web documents in the smallest possible amount of time. That makes that scalability is a crucial factor in large universal crawls. A huge number of documents needs to be fetched from the Web (bandwidth issue), processed (CPU issue) and stored permanently (disk space issue). Single points that can help improving the performance of a broad crawl include the minimization of Domain Name System (DNS) lookups that the crawler makes to resolve host names to IP addresses, the careful distribution of crawl jobs on a large number of crawl machines, etc.

- **trade-off between freshness, importance and coverage** - As we said already before, the Web is very dynamic. That makes that new pages are added constantly while existent pages get modified or removed. To achieve a high coverage a crawler needs to find the new pages (frontier extension) while to achieve a high freshness it needs to frequently revisit pages (constant frontier). A solution to this problem might be to limit frequent revisits to those pages that have been recognized as really changing frequently (e.g., news sites). Also not each broad crawl operator is actually interested in crawling the whole Web, the purpose might as well be to crawl pages on some specific topic, or others. If the intention is to crawl a large number of sites, it might be necessary to limit the coverage of single Web sites, i.e., limit the depth with which a site is crawled. Although this makes that not each site will be crawled completely, this trade-off enlarges the coverage of the crawl with regard to the whole Web or the intended part of it.

**Focused Crawling.** As compared to the broad crawler, the intention of the focused (or topical) crawler is to collect pages from a specific domain, category, topic or similar. There exist several ways to implement focused crawlers and to limit the scope of a crawl: by limiting the URLs to be visited to certain given domains, by doing similarity checks between fetched pages and a given set of example pages, by checking fetched pages for keywords related to a given topic, by using supervised learning mechanism where classifiers work over a set of labeled example pages, etc. An example of a topical crawler is the vertical seekda search engine, that as opposed to universal search engines as Google, Yahoo or MSN, focuses its crawls on Web Services and related information.

Same as for the broad crawl, the focused crawl can be based upon both crawling strategies as described in Section 2.1.2. This depends on the importance of collecting the changes of Web sites accurately. A crawler that is intended to collect news pages on a specific topic will use an *incremental strategy* while a crawler whose purpose it is to collect all pages on one topic in a quite complete
manner, will preferably opt for the snapshot strategy. Depending on the scope limitation of a focused crawl, completeness might be an issue or not, as the crawler will be able to crawl the allowed sites in-depth if the scope is sufficiently limited.

A crucial issue in topical crawling, where the crawl is intended to collect pages related to a specific topic, is prioritization and corresponding cost assignments:

- **URL and queue prioritization** - When we want to focus a crawl on a specific topic, we need to assign priorities or costs to URLs in queues or to the frontier queues themselves. Such priorities need to be distributed by the time the URL is put into a frontier queue, which makes that there need to be heuristics that help decide on how to assign costs or priorities to URLs. When the goal of the topical crawl is to detect pages related to a specific topic, we need to find good ways, be it heuristics or guesses, to determine whether a yet unvisited page might be on that topic or not. Such heuristics can be based on an analysis of the page where the link comes from, on the domain of the link, or on many other points.

**Continuous Crawling.** Continuous crawling is the type of crawling that can be used to accurately follow changes on Web sites, mostly on restricted parts of the Web. This crawler type uses an incremental crawling strategy, as described above in Section 2.1.2. That is after having visited a URL, the crawler enqueues it again, so that it will be revisited.

There are three important issues to mention concerning such continuous crawls:

- **resources usage** - When we crawl large portions of the Web (e.g., one country's part of the Web) and store the data to build an archive, we need a lot of disk storage space. That is we need to seize all possible opportunities to not unnecessarily increase the need of disk space. One such method would be to detect whether a revisited page has changed since the last visit or not, and to only store it in case it has changed.

- **crawl scope** - As well important in continuous crawls is the limitation of the scope of a crawl. If the scope is not limited, the frontier will grow indefinitely and will not allow the revisiting of URLs in a reasonably small time frame. This again will lead to the fact that lots of intermediate changes in frequently changing Web sites will not be crawled and will be lost.

- **politeness policies** - We talked before about a reasonable time frame within which we want a crawler to revisit URLs to not loose too many changes on Web sites. This leads us to the obligation of crawling operators to respect certain politeness policies, whereas different Web servers may set up different politeness restrictions. Such politeness policies are necessary
to prevent crawlers from crawling certain servers too aggressively, what in the worst case could lead to the crawler being blacklisted.

2.1.4 Crawling Issues

In the following we list the major issues that need to be addressed regarding a Web crawler operation. They are valid for all crawling strategies and types, although some issues might be prevalent in some specific crawling configurations, as already pointed out in Section 2.1.3 when describing the single crawler types.

- **performance** - A crawler often needs to handle not only thousands, but billions of Web documents in the smallest possible amount of time. That makes scalability a crucial factor in Web crawls, concerning bandwidth usage (fetching the documents), CPU power (processing the fetched data) and available disk space (permanent storage of the data).

  Single points that help improving the performance and scalability of a broad crawl include the minimization of Domain Name System (DNS) look-ups that the crawler makes to resolve host names to IP addresses, the careful distribution of crawl jobs on a large number of crawl machines, a thorough trade-off between freshness, importance and coverage of the crawled data, for continuous crawls a storage strategy that does not store unchanged pages, etc.

- **crawl scope** - In order to not let a crawl grow indefinitely, it is important to limit the scope of a crawl. Otherwise, with an unlimited scope, the frontier will grow indefinitely and will not allow the revisiting of URLs in a reasonable time frame (important for the freshness of the data).

- **cost assignment** - An important issue in crawling is the assignments of costs to both the frontier queues and the URLs within the queues. Using cost assignments we can assign priorities to queues and to URLs. As the costs are distributed by the time a URL is put into a frontier queue we need to find good ways and heuristics that help decide on how to assign the costs or priorities to URLs.

- **politeness policy** - Different servers may set up different politeness policies. Such politeness policies are necessary to prevent crawlers from crawling certain servers too aggressively, what in the worst case could lead to the crawler being blacklisted.

- **crawler traps** - Describes crawler traps that need to be avoided, as, e.g., Web sites where dynamically created URLs are modified based on the actions the crawler performs. This can lead to an infinite amount of new links that do not lead to any new content.
2.2 Service Discovery

Service discovery deals with the detection of services that are fulfilling a specific user’s concrete needs. Such discovery can be based upon either functional or non-functional aspects of a service, i.e., the functionality that a service provides or aspects like its price, its location, its terms and conditions, its availability, etc. Common approaches in service discovery mostly work on top of restricted sets of services. These can be on the one side the actual Web Services as published on the Web (using WSDL descriptions or RESTful approaches), or on the other side Semantic Web Services, i.e., semantic service descriptions that describe the service, e.g., in terms of its non-functional properties.

Semantic Web Services mostly describe the functionalities and the non-functional properties of the services in a rather complex and complete way. They come along with semantic service discovery approaches\[7, 6, 8, 17, 10\], that go beyond keyword based retrieval. While semantic discovery achieves a high precision it relies on precise and potentially complex descriptions that are presently not available at large scale, but only for restricted sets of services. Semantic service discovery needs often complex reasoning methods in the background, that - although much research is currently going on in this area - are not yet efficient enough to work on top of a real large number of services, e.g., on Web scale.

On the other side we (a) have public and private registries that provide access to restricted sets of publicly available services, and (b) standard universal search engines that index Web Service descriptions when crawled on the Web. As already mentioned in the introduction (Chapter 1), most of the specific service registries work over restricted sets of services that are manually registered by the providers or users of the services. Also both specific service registries and universal search engines suffer from drawbacks in service discovery, as described in \[1, 11\] and \[12\]. One advantage of the service registries (a) over the universal search engines (b), is the information about a service that is presented to the user: while universal search engines center their information around the WSDL document (similar to any HTML document), service registries use the services and their providers as central notions, not the underlying documents.

Another approach, which is followed by the public Web Service search engine of seekda, deals with services on a very large scale, i.e., on Web scale. They actively crawl the Web for service descriptions and do thus collect public Web Services from all over the world, not anticipating that people always register their services with them. In this case the search for services focuses on the descriptions which are currently used for the numerous already publicly available services. This approach reduces Web Service discovery to a special information retrieval (IR) problem. The crawled Web Services and related documents are indexed, i.e., their contained keywords are represented in an inverted index. A query to the search engine consists of one or more keywords that are then matched by the engine against the collected documents. What it returns is a (ranked) list of results. These results are, similar than for the service registries as described above, not centered around the WSDL documents; they have as
central entities the services and their providers, and present moreover information like the service availability (based on daily monitoring), its location, user descriptions and rankings, etc.

2.3 Service Ranking

Many current service ranking approaches work with the assumption that services are either semantically well-described or that we have detailed Quality of Service (QoS) information about them available (e.g., availability and response time of services). [20] addresses the ranking of services based on the semantic description of their non-functional properties, including aspects like locative, temporal, availability, obligation, price, payment, etc. The descriptions need to be provided in WSML [3], a very powerful semantic language, supporting different logical flavors, but with the downside that it is rather hard for non-experts to provide the descriptions. [21] follows an approach for QoS-based ranking with trust and reputation management. This approach assumes that QoS properties such as availability, acceptable response time, through-put, etc. are provided by the service provider.

In contrast to these approaches, that require the active participation of the service providers to describe their services in one way or another, we base our ranking approach on the 'real-world' information that is available about them just by the fact that they are published on the Web, like their descriptions, their hyperlink relations, some monitoring information, etc. We do not assume - unrealistically - that we know how a service behaves on execution, what functionality it delivers, etc. (as would be the case in a 'man-in-the-middle' approach, where we would assume to have such knowledge before doing the ranking).
Within this chapter we will describe the methodologies developed in the scope of this thesis for focused crawling of Web Services and related information, and for the subsequent analysis of the gathered data. Our Service Crawler collects both services and information related to the services (e.g., documentation pages) from the Web. We apply focused crawling techniques to crawl the Web for services, using methods like optimized URL and queue scheduling to scope crawls to the part of the Web that is relevant for services. During a crawl we apply methods to identify the service descriptions; in subsequent analysis steps we build unique service objects out of them and add pointers to all the documents that we deem related to them. The service objects are then enriched semantically by adding simple RDF annotations to them.

Section 3.1 will provide an overview over the Web Service crawling, detailing the Service Crawler’s architecture (Section 3.1.1), the concrete crawling techniques (Section 3.1.2), as well as the Service Crawler implementation (Section 3.1.3). Section 3.2 will then explain how we proceed with the fetched data, outlining the raw crawl data that results from a crawl (Section 3.2.1), presenting the different analysis steps that we perform over the data (Section 3.2.2), and providing an overview of the resulting final crawl data output (Section 3.2.3). Finally we show in Section 3.2.4 how we enrich the initial service descriptions semantically using our analysis results.

3.1 Web Service Crawling

The big success of search engines today is only possible due to efficient crawling solutions. As described in Section 2.1.1 a crawler exploits the fact that Web pages are interlinked with hyperreferences: by following the links found in a set of initial pages (seeds) a crawler discovers more URLs. These (yet) unvisited URLs build the frontier of a crawl. The frontier is dynamic and grows according to the scope of a crawl. A scope defines for which of the newly found URLs will
be disregarded and which will be queued.

For efficiency reasons a frontier is often divided into multiple queues. Queues can be built based on different schemes (e.g., one queue per host or per IP). The frontier can be limited (by a scope) in various ways, such as for example by the number of pages to be fetched from one domain, the number of link hops that shall be followed, etc. Even with restrictions given by a particular scope it is likely that a crawler discovers more pages then it can ever fetch, therefore also the order of URLs in a queue and the precedence between queues is of importance. Scoping and the assignments of weights to queues and URLs are thus the most important aspects in building a focused crawler like our Web Service crawler. In Section 2.1.4 we also identified other issues that need to be taken into account when building an own Web crawler, such as performance, politeness policies and crawl traps.

To be able to focus our crawls on Web Services, we need to start with a good set of seed URLs. These seeds can or shall contain, e.g., Web pages where we know that Web Services are published or that they talk about Web Services. One of the most important aspects when developing a crawling strategy for Web Services is the fact that we do not want to crawl blindly through the World Wide Web, but need to focus our crawl for the Web Service domain. As we thus look for service descriptions and related information, which is mostly stored in textual documents, we can in our focused crawl already reject a lot of content by default, like images, audio or video files, etc. What we want to look at specifically is, e.g., HTML pages, XML files, other text documents, i.e., all types of files that could be either a service description or related information. So far we work with the premise that every service is described with a WSDL interface specification. I. e. during the crawl process we check whether a fetched page is a valid WSDL description and if so, we try to gather more information around the service endpoint (like, e. g., their geographic location or liveliness).

Beside the Web Services themselves, our crawls will as well focus on related information. Such information can be quite divers: documents pointing to the service, the service provider’s service definition, documents pointing to that definition and vice versa. As a first step we consider those resources that include links pointing to the service interface description and vice versa. We expect to gather pages that include general descriptions of the service functionality, FAQs, pricing pages, etc. While the resources located in the same domain (where the service is hosted) will mainly hold descriptive information, pages on different domains will likely include information that talks about and eventually ranks the service. In order to be able to gather as much related information as possible, our crawler will, e. g., crawl the sites of service providers more deeply than other sites.

To realize our focused crawler we base our work upon the Internet Archive open-source crawler Heritrix. Heritrix is an archival crawler for periodic snapshots of a large portion of the Web. It has been designed in a modular way, which allows extensions for all relevant aspects such as scoping of URLs, queue

assignment strategies, URL precedence, content processing as well as content writing [14].

In the following we will first describe the high-level architecture of our Service Crawler (Section 3.1.1). In Section 3.1 we will provide an overview of our targeted service crawling techniques and in Section 3.1.3 we will outline the implementation details of our Service Crawler.

### 3.1.1 High-level Crawler Architecture and Data Flow

Figure 3.1 provides an overview of the Service Crawler’s high-level architecture and data flow. The crawl operator has the task of configuring single crawl iterations, e.g., broad crawls, extended provider domain crawls, etc., and of monitoring the crawls and their run-time behavior. The crawl operator also collects the set of seed URLs that are used to start a crawl and that are a deciding factor for focused crawling (Section 3.1.2.1 will provide more information on seed collection for the Service Crawler).

![High-level Architecture and Data Flow of the Service Crawler](image)

Each crawl iteration as such is started with a new seed collection and is stopped at a certain point of time by the crawl operator. This moment is usually after a large number of services has been identified and some millions of documents have been fetched; it is as well influenced by, e.g., the type of crawl that has been performed (broad crawl, extended domain crawl, etc.).

After a crawl has been stopped, the data post-processing or analysis step starts. During this step, the performance of a crawl is analyzed (both regarding the focus of the crawl and its behavior in terms of throughput) and the service objects that have been identified during the crawl are completed with relations...
to documents that are identified as being related (either by an inlink or outlink
relation or by term vector similarity). During this step, all relevant documents
(service descriptions and related documents) are collected and stored in ARC
archives and corresponding index files are built that allow the direct access to
single documents in the archives. Both, the format of the ARC files and the
index file format, will be described in Section 3.2.1. Next to the document
archives and index, the Service Crawler produces an RDF graph containing all
information that we have been able to extract from the services both during the
crawl and during the data analysis (Section 3.2.4).

3.1.2 Service Crawling Techniques

In the following we describe aspects that are crucial for the focus of our crawls
to the Web Service domain, as well as methods for the identification of Web
Service descriptions and related information.

3.1.2.1 Seeds

As described in Section 2.1.1 on basic crawl steps, a crawl for Web Services
needs to start from a set of seed URLs. These are almost as important as the
crawler itself, as they highly influence the part of the Web that we crawl. The
seeds can or shall contain, e.g., Web pages where we know that Web Services
are published or that they talk about Web Services.

We started by collecting the seed URLs in a semi-automatic process that in-
volved both screening of well known sites, like the specialized portals mentioned
in Section 1, and utilizing the Alexa Web Search platform. After having iden-
tified those domains that host Web Services or that talk about Web Services
we started a focused crawl in order to obtain as much related information as
possible.

After having run some first crawls, we have enlarged our process of collecting
seed URLs: we use the results of finished crawl iterations. That is we use the
WSDLs we have detected so far, as well as outlinks and inlinks of the WSDLs
(outlinks link from the WSDL to some other page on the Web; inlinks link
from some other page on the Web to the WSDL). Collecting a seed file is a
continuous process that is renewed for each crawl and that takes into account the
results of the crawls finished up until then. In the following we list all the seed
sources that we have explored. We thereby sometimes refer in numbers to one
specific example crawl (“Crawl_A”) in order to illustrate, e.g., the proportion
of WSDLs to related documents.

- **Alexa Web Search** - We run specific small crawls to collect seed URLs from
  Alexa Web Search.

- **Specialized portals** - We run specific small crawls to collect seed URLs from
  specialized portals.

• **WSDLs** - We take all the WSDLs that we have collected during previous crawls as new seed URLs.

During the crawls we write archives that contain the content of the WSDLs and index files that contain an index (URL, archive number and offset) to the single WSDL entries in the archives. To extract all the WSDLs we only need the index files, that means we don’t need to look into all the archives themselves - we just need to cut the offset of the index files. This can be done using a simple bash command like for example:

```
egrep -oh "http[^[:space:]]+" *.arc.idx | sort | uniq > grep_wsdls
```

This way we have, e.g., extracted 125,211 WSDLs from Crawl_A.

• **Outlinks** - We take all the URLs that are mentioned within the WSDL files. The outlinks need to be extracted from the WSDL archives themselves. This can be done using a simple bash command like the one shown above for grepping the WSDL service descriptions. This way we have extracted 528,697 outlinks from Crawl_A.

• **Inlinks** - We can collect both inlinks from first grade and from second grade for our seeds file. In a first step we take all the direct inlinks, i.e., all URLs that link to a WSDL file. To be able to collect these inlinks we need to first run our analysis steps on the link graphs collected during the crawl and assign services to inlinks (see Section 3.2.2). Then we can again use a simple bash command to extract all inlinks from the analysis result:

```
egrep -oh "http[^[:space:]]+" services2inlinks | sort | uniq > seedinlink
```

This way we have extracted 149,785 first grade inlinks from Crawl_A.

After having assembled all inlinks of first grade we collect all the inlinks of second grade, i.e., all URLs that link to the inlinks of first grade. We do this by running another analysis script on the link graphs that result from our crawl. This gives us a file that contains all links and their inlinks. Now we just extract the inlinks of second grade from the file using again a bash command. This way we have extracted 768,709 second grade inlinks from Crawl_A.

• **Add URL** - The seekda Web Service search engine allows users to manually add service descriptions to the search engine if their service is not yet indexed; we use these service URLs as seed URLs in following crawls.

Out of these different sources we build one big seed file, eliminating all duplicate entries. Now we can apply some more checks on it to discard URLs that we deem as not useful as seed URLs, such as:

• **Query Strings** - We check the seeds for URLs that contain query strings, i.e., URLs with ‘?’ and a sometime following ‘=’. We check this by applying a simple bash command on our seed file:

```
egrep -oh "http[^[:space:]]+" services2inlinks | sort | uniq > seedinlink
```
For Crawl A we noted that approximately one third of all the seeds contained query strings.

- **URLs/Host** - We check how many URLs we have per host, as it may not be interesting to have too many seed URLs for single hosts.

Now we do not want to remove all the URLs that contain a query string, as, although this would for sure reduce the number of (often) superfluous links per host, it would on the other side make us loose hold of some hosts that do only publish their services in directory listings that are filtered out by this check.

To avoid that problem we apply the following strategy on finalizing our seed file. We first analyze the initial seed file and build a new file where we collect for each URL (1) the provider, (2) a score and (3) the URL itself. The score is built taking into account the number of path segments of the URL and the fact whether or not the URL contains a query string. Currently the query string fact has a higher weight than the number of path segments. We sort the file we get as result of this analysis and build then our final seed file. We remove all URLs that contain a query string, but to not completely loose that host we add the URL without the query part (i.e., we cut the query part from the URL). Then we keep for each host 100 URLs. Due to the structure of the input seed file, we will first get the shortest paths.

### 3.1.2.2 Web Service Descriptions Identification

After having successfully build a seed file that contains paths to WSDLs and service related pages, we can start actually crawling the Web. This leads us to our next challenge: identifying Web Services on the Web. That is we concentrate our search on service descriptions (and related information), which are mostly stored in textual documents. That said, we can by default reject a lot of content in our crawls, like images, audio or video files. We specifically want to look at pages like HTML, XML, other text documents, i.e., all types of files that could either contain a service description (or related information). We work with the premise that every service is described with a WSDL interface specification. That is during the crawl process we check whether a fetched resource is a valid WSDL description. In addition we analyze some properties of the WSDL files, which allow us to decide whether they are well formed and refer to publicly accessible endpoints.

We have extended the Heritrix crawler by a processor that checks whether a fetched page is valid WSDL and writes it to WSDL archives (see Section 3.1.3). In the scope of this check we first get the content of a page from the Heritrix CrawlURI. This content still contains the HTTP Header, so we discard that one. Next we parse the content with a SAX XML parser and, if the content is actually XML, check whether it is a valid WSDL description.

There are some properties on WSDL files which allow us to decide whether
it is valid WSDL or not, and whether it is a WSDL 1.1 or a WSDL 2.0 description. We extract the XML start element; if the name of the element is definitions we are looking at a WSDL 1.1 description - if the name is description we are looking at a WSDL 2.0 description. Next we check for elements like service, portType and operation. We extract the service name as we will use this to build a unique service object from the service and its related documents (see Section 3.2.2). Next we check the WSDL description for its location (WSDL 1.1) or address (WSDL 2.0). This gives us the endpoint of the service, which we extract and use to define the provider of the service (see Section 3.2.2).

Although we know by now whether we are handling a WSDL file or not, we do not yet know whether this WSDL is a public service; therefore we try to resolve the endpoint address. We do not consider services as public that (a) have a site local address, i.e., an address that is meant to be used for addressing inside of a site, or (b) a loopback address, like, e.g., the IP 127.0.0.1 for IPv4.

As already mentioned above we currently concentrate our efforts on the detection of WSDL service descriptions. We are aware of the fact that not all services are described using WSDL though; there are a large number of Web APIs, RESTful services, JASON, mashups, etc. available on the Web. While WSDL descriptions are well-structured and therefore allow an easy recognition, REST-based services or Web APIs do not follow any strict structural rules and are therefore much harder to detect. Researchers within seekda have started working on the identification of such non-WSDL services, but this topic is too broad to be addressed in the scope of this thesis.

### WSDL Deduplication

As we have mentioned in Section 3.1.2.1 on seeds collection, we gather far more WSDL descriptions from the Web, as we build unique service objects. That is there is no one-to-one relation between service objects and WSDL descriptions. This is because on the one hand there often exist more WSDLs for one single service and, on the other hand, one WSDL can describe several services. In the data analysis step after the crawl we build the unique service objects, by collecting all WSDLs that refer to the same service (see Section 3.2.2). Although this could be interpreted as a kind of ‘deduplication’ of the WSDLs, it is not; we still gather a lot of similar, superfluous WSDL descriptions from the Web. To address this problem we have analyzed services containing large numbers of ‘duplicate’ WSDLs and have developed an improved deduplication approach.

We monitored the average number of WSDL documents per service over a longer time period (ca. 2 years) and could observe a growth from less than 2 WSDLs/service to more than 7 (see details in Chapter 5 on the Service Crawler evaluation). The existence of ‘duplicate’ WSDLs is due to, e.g., the hosting of WSDL services on different servers or even by different providers, different versions of the same service, etc.

http://www.w3.org/TR/wsdl

http://www.w3.org/TR/2003/WD-wsdl20-20031110/

Although different versions of one service might as well be regarded as different unique

24
A manual analysis of a subset of the services that came with an extraordinary big number of associated WSDL files showed that they were to a large extent due to the behavior of PHP servers to auto-generate WSDL files. While PHP is best known as scripting language used to generate dynamic Web pages, it can as well be employed for machine-to-machine communication by exposing class functions from PHP scripts as services. Based upon specific annotations in the script that are examined at run-time, a WSDL is generated for the service. The location of the generated service is always relative to the document root of the Web server and is only determined at run-time. Depending on the URL that requests the WSDL a different endpoint is generated, which is why you can easily get large numbers of WSDL descriptions for one single service. Listing 3.1 shows an example of such auto-generated WSDL description URLs:

```
http://agilepay.ws/ws/server.php///wsdl
http://agilepay.ws/ws/server.php/AddUserPaymentMethod///wsdl
http://agilepay.ws/ws/server.php/AddUserPaymentMethod/
    AddUserPaymentMethod/?wsdl
http://agilepay.ws/ws/server.php/GetRateList?wsdl
http://agilepay.ws/ws/server.php/GetRateList/AddUserPaymentMethod?
    wsdl
```

Listing 3.1: PHP server auto-generated WSDLs

The basic URL for the example shown above is `http://agilepay.ws/ws/server.php`. Behind this basic URL you can add any text and still get a valid WSDL generated, e.g., `http://agilepay.ws/ws/server.php/ANYTEXT/?wsdl`. We have extended our WSDL processor within the Service Crawler to recognize such PHP service URLs and to only keep the shortest possible URL as WSDL URL, i.e., according to the above example, the basic URL ending on `php`, extended with `wsdl`. While this strategy helps filtering out redundant WSDLs that are auto-generated by PHP servers, there might be other cases coming up that need some special handling. The number of WSDLs per service needs thus to be monitored and the process to deduplicate WSDLs needs to be adapted in case of need.

3.1.2.3 Related Information Identification

As already mentioned before, we do not only crawl for service descriptions, i.e. WSDLs, but also for information that is related to these services. This information may consist of provider documentation of the service functionality, provider Web pages, Wikis, Blogs, FAQs, user ratings and many more. Such documents may be pointing to the service, the service provider’s service definition or vice versa.

Detecting this information is important as (a) it allows us to know more about an offered service than only its technical description, and (b) the information can be used for ranking the services (as described in Chapter 4). Con-
cerning (a), the information can help a user understand, e.g., what a service is about and under what conditions he can use it. For a consumer of a service the inputs and outputs, e.g., of a service are in general not as relevant as other information on who is the service provider, its availability and response times, whether it is a commercial service or not, its price, its terms and conditions, what exactly it is offering, how other consumers rate the service, etc.

In the scope of the Service-Finder and the Service-Detective projects for example, we further analyze(d) this documents to find out what category the service belongs to (e.g., Communication) and what information we can find on certain related pages, like documentation, price page, provider contact page, etc. While this information extraction is being handled by other components (not in the scope of this thesis), the Service-Crawler is responsible for gathering useful “raw” data from the Web.

The task of collecting related information is split onto the crawl run-time and the analysis of the data because we consider both resources that include links pointing to the service interface description and vice versa, i.e., the resources that services point to. Those documents that point to the WSDL descriptions (so-called inlinks) cannot be identified during the running crawl. We will collect these documents after each finished crawl iteration in the analysis steps described in Section 3.2.2. In the following we will concentrate on the crawling related part of identifying related information.

After having detected WSDL service descriptions as described in Section 3.1.2.2 we first try to gather more information around the service endpoint. Such information includes the geographic location of the service host (which does not need to be identical with the location of the provider though) and the liveliness, i.e. availability, of the service.

According to the basic crawling behaviour as described in Section 2.1.1 the crawler follows the outlinks that have been extracted from the WSDL service description. We expect, by crawling the direct outlinks, but also by crawling further (e.g., the service host domain and all outlinks) to gather pages as described above, containing, e.g., general descriptions of the service functionality, FAQs, pricing pages, etc. We expect that the resources located in the same domain (where the service is hosted) will mainly hold descriptive information, assuming that the resources are provided by the actual service provider. As opposed to this, pages on different domains will most likely include information that talks about and eventually ranks the service; we assume that this information is provided by users of the service or by the Web Service user community. In order to be able to gather as much related information as possible, our crawler is crawling the sites of service providers more deeply than other sites.

Unfortunately it is not yet sufficient to collect related information only by screening the provider domain more intensively and by relating outlinks and inlinks to the service descriptions. First of all, not all outlinks or inlinks lead to information that is really related to the specific service, i.e., we may as well find useless links. But on the other side, there is a lot of information that stays hidden to us when we only concentrate on outlinks and inlinks. We will illustrate the problem using the following example:

2. We want to find related information to this service; the Web site cdyne.com has approximately 300 pages.

   - We find pages linking to the WSDL (during the analysis described in Section 3.2.2):
     - [http://secure.cdyne.com/developers/default.aspx](http://secure.cdyne.com/developers/default.aspx)
     - [http://www.cise.ufl.edu/~doliver/crawler/](http://www.cise.ufl.edu/~doliver/crawler/)
     - [https://cdyne.com/developers/default.aspx](https://cdyne.com/developers/default.aspx)
     - [https://secure.cdyne.com/developers/default.aspx](https://secure.cdyne.com/developers/default.aspx)
     - ...

   - We miss some important pages that are not related by links (found by manual analysis), such as:

   After a manual check we see that the three above mentioned, missed pages are related and definitely important, but we cannot relate those pages to the service from only looking at the link graph. 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 above pages we would be able to assess the similarity and thus conclude that they are related.

   Concerning the usage of term vectors we need to distinguish three different possible tasks:

   1. Calculate the term vectors of all fetched pages to assess the service category. If, e.g., a page contains many words like “SMS”, “Message”, “send”, etc, we can assume that the page talks about “SMS” and can therefore assign a corresponding service category.

   2. Calculate the term vectors of all fetched pages and compare them. This would allow us to detect “any” page that might be related to a specific service just by seeing a high similarity in their term vectors. This method would go definitely beyond searching related information mainly by outlinks and inlinks to the service description.

   But this kind of term vector analysis cannot be tied into the crawler at run-time. The pure calculation of term vectors can be done during the...
crawling, each time a page is fetched (similar as in the first case above). But the analysis itself needs to be done in a later step in the Service Crawler’s analysis process. Also, clearly, we cannot apply this approach blindly on all fetched documents, as this would require far too much computing power. We restrict the approach to checking the similarity of the term vectors of services to the term vectors of documents fetched from their respective provider’s domains.

3. Calculate the term vector of all fetched pages and compare them to a pre-defined term vector containing terms related to the Web Service domain. This check is meant to find out whether a given page is somehow talking about Web Services. This analysis step can be used to increase the precedence of URLs when certain terms are (more or less frequently) mentioned on a page.

In the scope of this thesis we have added the third of the three functionalities to the Service Crawler (described in the following section 3.1.2.4). Together with other partners in the Service-Finder project we have also developed a solution for the second functionality, that allows us to detect information from a service provider’s Website related to a service (solving the problem as described in the example above). This work is nevertheless not part of this thesis; more information on it can be found in [18].

3.1.2.4 Queue and URL Scheduling

When crawling the Web for Web Services we need to use clever means to prioritize URLs and frontier queues, 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 Web. We want to only fetch those documents from the Web that are relevant for the domain of Web Services. That is we want to scope our crawl to relevant resources and thus enable a so-called focused crawl, as described in Section 2.1.3.

The Service Crawler creates new queues per top-level domain, i.e., each new host gets its own queue. In this queue all the URLs that belong to that host are scheduled. Influencing the URL and queue scheduling in Heritrix means allocating costs to URLs or setting precedences of URLs and/or queues before they are being scheduled by the frontier. The costs and precedences will then in a later stage influence the scheduling in the frontier: URLs with a low cost (or a high precedence) are put more upfront in the queue, such that they are processed earlier than URLs with a higher cost (or lower precedence) that are put more at the back of a queue. The concept of scheduling for queues is similar to the one for URLs: queues with a high precedence are processed earlier than queues with a lower precedence.

Both our approaches for penalizing and privileging URLs are based on experiences that we did during our crawls. Without a proper URL cost assignment it is not possible to focus the crawl to Web Services. Although we can also detect
WSDLs in a 'random', not focused crawl, we will not able to significantly raise the number of both detected services and related information.

URL Cost Assignment

We have developed an approach for URL cost assignment that is targeted on the prioritizing of (assumably) Web Service related documents. We set the cost of each new CrawlURI (i.e., a URI that is going to be scheduled by the frontier) by default to 20. Then we start checking the URL for certain negative features (i.e., features that we identified of belonging to pages often being less important for our purposes) that we penalize as follows:


- **query strings** - We penalize all URLs that have more than one query string. E.g., ?a=b would not be penalized, whereas ?a=b&c=d and ?a=b&c=d&e=f would be penalized by +1 or +2, respectively.

- **path segments** - We penalize all URLs that have more than one path segment string. E.g., http://seekda.com/SOME_PATH/SOME would be penalized by +1. More path segments would accordingly be penalized by the number of path segments in the URL minus one.

- **JavaScript** - We penalize JavaScript files by +3.

After penalizing for the negative features we start privileging positive aspects of the analyzed URLs:

- **WSDLs** - We reduce the costs by 5 for URLs that promise to be WSDL service descriptions, i.e., URLs that contain the string "?wsdl".

- **service related terms** - We reduce the costs by 2 for URLs that seem to be related to Web Services, i.e., URLs that contain one of the following strings: "/ws", "/service" or "/api".

- **provenance page score** - 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 (i.e., when certain terms are (more or less frequently) mentioned on a page); we assume that a page that is talking about services might with a high probability link to other pages that talk about services.

To determine the provenance score of a page we make use of term vector comparisons (as already briefly mentioned in Section 3.1.2.3). We calculate the term vector of all fetched pages during the crawling and compare
them to a pre-defined term vector containing terms related to the Web Service domain.

We calculate the score based on the number and position of the occurrence of Web Service related terms. We proceed like follows: First we extract all visible plain text from parsed HTML documents, skipping comments, JavaScript, etc. We distinguish between normal text and title text, as we deem words appearing in the title more important than others. Now we retrieve all unique tokens from the plain text and normalize it in that we transform them to lowercase and eliminate all words smaller than three letters (as for example: in, of, at), i.e., our so-called stopwords. After this step we build term vectors of all unique tokens, their frequency (i.e., how often they appear on a page) and their weight (based on the position of the token in the text and/or the title).

Now to assess whether a page is relevant to the Web Service domain we compare the term vectors with a pre-defined list of weighted terms from the Web Service domain, as shown in Table 3.1.

<table>
<thead>
<tr>
<th>Term</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsdl</td>
<td>1.0</td>
</tr>
<tr>
<td>soap</td>
<td>1.0</td>
</tr>
<tr>
<td>ws</td>
<td>0.7</td>
</tr>
<tr>
<td>api</td>
<td>0.5</td>
</tr>
<tr>
<td>rest</td>
<td>0.1</td>
</tr>
<tr>
<td>service</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 3.1: Web Service Domain Term Vector

We set the weight from the Web Service Domain Term Vector off against the term vectors calculated from the single documents. That is we calculate a final score for a specific Web page by matching the Web Service domain term list with the term vector resulting from our Web page and multiply the Web Service term weights with the Web page term vector scores.

Finally we reduce the costs of the outgoing links by the score of the provenance page.

Let us take as example a Web page that contains the terms WSDL, SOAP and service. Table 3.2 summarizes the example page, only showing those terms from the term vector that match the terms in our Web Service Term Vector. The score that we get for each term takes into account both the frequency of a term in the document and the position of the term. A term is worth more if it appears in the title of a page than if it appears in the normal text: we assign 5.0 for each time a term appears in the title and 1.0 for each time a term appears in the remaining text.

To compute the final score of the specific Web page we match the Web
<table>
<thead>
<tr>
<th>Term</th>
<th>Score (Title + Text)</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsdl</td>
<td>7.0 (5.0 + 2.0)</td>
</tr>
<tr>
<td>soap</td>
<td>2.0 (0.0 + 2.0)</td>
</tr>
<tr>
<td>service</td>
<td>9.0 (5.0 + 4.0)</td>
</tr>
</tbody>
</table>

Table 3.2: Example Term Vector

Service Domain Term Vector with the term vector resulting from our Web page and multiply the Web Service term weights with the Web page term vector scores.

\[1.0 \times 7.0 + 1.0 \times 2.0 + 0.1 \times 9.0 = 9.9\]

We have so far provided a simple implementation of this above described term vector comparison. The performance, the features and the flexibility of our implementation can probably be improved by using an external library for fast tokenizing and word stemming (the tokenizer to be used must be configurable, e.g., to also include camel casing, i.e., compound words or phrases in which the words are joined without spaces and capitalized, like “sendSMS” or “getAvailabilityGraph”), an optimized structure for representing term vectors with frequencies and weights, as well as some more generic algorithms to calculate similarity scores between two term vectors. Such an improvement can be seen as future work in the URL and queue scheduling methodology.

Queue Scheduling

The aforementioned strategies to set, increase or reduce the costs of URIs are restricted to the URIs: we cannot apply the same strategies to queues. Concerning the queue scheduling we follow another approach: we use the Heritrix pre-defined `HighestUriQueuePrecedencePolicy`, i.e., 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 are always processed first.

Besides allocating costs or precedences to URLs and queues we can influence the scheduling behaviour of the Heritrix frontier by configuring the number of URLs to be processed by one queue before it is laid back to wait for a new run, the so-called “balance-replenish-amount”. By default this replenishment amount is set to 3,000, which makes that the crawler stays quite long with one queue before changing to others. This has the drawback that some queues simply never get served and that they respond very slowly to changes in the precedences. We have set the replenishment amount to 500. We assume that setting it too low might entail different drawbacks, as, e.g., more I/O activities on the crawler machine.
3.1.3 Crawler Implementation

As already mentioned above we build upon the open-source Java crawler Heritrix for the actual implementation of the Web Service crawler. Heritrix is an archival crawler which was developed with the intention to be used for producing archived periodic snapshots of a large portion of the Web. To be able to scope the crawls according to our specific needs as described in the sections before, we had to (a) configure Heritrix accordingly where possible and (b) write extensions to Heritrix where the needed functionality is not available. In the following we will provide an overview of Heritrix’ internal structure (Section 3.1.3.1), and of the major configurations, changes and extensions that we did within Heritrix to adapt it to our WSDL and related documents crawl approach (Section 3.1.3.2).

3.1.3.1 Heritrix Structure

As we have already mentioned before, one of the main components of a Web crawler, and also of Heritrix in specific, is the frontier that manages the state of an active crawl. When a URL is taken from the Heritrix frontier for being processed, it is being sent through a set of processors which are grouped in chains. Every URL needs to pass through these chains before being finished. The processors on different chains perform different tasks: pre-selecting URLs (e.g., scope check), enforcing preconditions (e.g., checking the robots.txt file), fetching the actual data, extracting links from the data, writing the data to archive files (ARC file format), checking the extracted links with regard to the crawl scope and scheduling the newly detected URLs to the frontier.

Figure 3.2 provides a high-level overview of Heritrix’ processor chain. Our extensions to Heritrix are mostly situated within the marked area, containing mostly data handling Processors and Writers (see Section 3.1.3.2).

3.1.3.2 Service Crawler Configuration and Extension

As we have already described before, the intention of our crawls is the detection of Web Service descriptions and of their related information. To adopted the Heritrix crawler to our needs, we have added special scoping rules that help limiting the crawl to the relevant portions of the Web and have added additional processors to the default processor chains. While the additional processors represent real extensions to Heritrix, the scoping rules are added via the standard Heritrix crawl configuration Web interface. This interface, as partly shown in Figure 3.3, provides a large number of crawl properties that can be customized by a crawl operator depending on his goals and needs. The following scoping rules and performance helpers do only represent a part of our specific Service Crawler configuration; an example configuration can be found in Appendix A.3.

In the following we present scoping rules, performance helpers and new processors that we have added to Heritrix:

http://crawler.archive.org/
Scoping rules and performance helpers:

- **On domain rule** - Depending on whether we want to do a broad crawl or a more extensive domain crawl, we can limit the scope of our frontier to only visit URLs that are on the domains from our initial seed set. This rule is, e.g., appropriate if we want to find related information to already known service descriptions. It is not appropriate if we want to extend our frontier to search for completely new Web Service descriptions.

- **Useless content** - We reject all useless content, that is we do not visit URLs that lead to, e.g., images, video files, audio files. What we want to look at specifically are, e.g., HTML pages, XML files, other text documents, that is all types of files that could be either a service description or related information. This rule helps us in not downloading unnecessary and unwanted data from the Web and thus burden both our bandwidth and our disk space.

- **Quota enforcer** - We add a quota enforcer to limit the amount of data that we crawl from one host. This enables us to keep a certain balance between the hosts, so that we do not crawl one host in complete depth. It prevents us from downloading huge amounts of data from just one host, as often the supplementary data is not that valuable any more.
Figure 3.3: Heritrix Crawl Configuration Web Interface

- **Download size** - We limit the maximum number of bytes to download from one document. This prevents us from downloading too large documents; neither WSDL service descriptions nor normal Web pages are usually very large documents.

- **Processors:**
  - **URL Cost Assignment** - We added a processor that sets the costs of CrawlURIs, that is of URLs that are meant to be submitted to the frontier for being scheduled. The goal of this processor is to distribute costs to the URLs that shall influence the scheduling in the frontier later: URLs with a low cost will be put more upfront in a frontier queue, such that they are processed earlier than URLs with a higher cost that are being put more in the back of the queue. Our URL Cost Assignment strategy has been described in more detail in Section 3.1.2.3.
Queue Precedence Policy - We added a processor that does a similar precedence assignment to queues than the URL Cost Assignment does to URLs. It consults the URI precedence values and makes that queues that contain many URLs with a low cost are getting a better precedence than queues that contain less valuable URLs.

Link graph - We added a processor that, during the crawl, writes link graphs to files. These graphs contain information about which page(s) link to which other page(s) and are useful for the analysis steps that we run after finishing a crawl (see Section 3.2.2).

PDF to HTML Converter - We added a processor that allows better and faster handling of PDF documents by transforming them to HTML and then letting them be processed as other original HTML documents by our fast and efficient HTML extractor. This processor will be described in more detail in the following of this section.

HTML Extractor - We replaced the standard Heritrix HTML Extractor with a new one that (a) provides more features, and (b) enables us to reach higher performance in terms of HTML document processing. Same as the PDF to HTML Converter, this processor will be described in more detail in the following of this section.

WSDL writer - We added a processor that handles all WSDL files that we find. After checking whether the WSDL descriptions are valid, publicly available and conform to the WSDL 1.0 or WSDL 2.0 specification, the processor identifies the unique identifier of the corresponding service (see Section 3.1.2.2 for details on these issues) and writes the WSDLs into archive files (ARC format), also adding index files that help accessing the records in the archive files in a random order.

ARC writer - We have extended the ARC writer that is shipped with Heritrix to also write index files corresponding to the archive files. This writer is used both by the above introduced WSDL writer and as normal writer that stores all fetched documents permanently on disk. The archives aggregate the fetched data in approximately 100MB large files, the file starts with a special ARC header and concatenates then single records that represent the Web pages. With each archive we produce an index file during the crawl that allows us to quickly jump to a specific offset in a specific ARC file and extract the corresponding archive record (see Section 3.2.1 for details).

RDF Writer - We have added a processor that writes service meta-data - RDF triples - into a triple store during the crawl run-time. We will describe in Section 3.2.4 what meta-data we extract from the crawl data.

During each crawl iteration Heritrix writes both logs and reports. While the reports are meant as final reports and are only written on specific user
demand or on termination of a crawl, the log files are extended continuously. For each processor that we have added we have as well extended the reports such that data statistics from the processors are written to disk at the end of a crawl. Furthermore we have extended the log files and created new ones. This is important as it allows us to monitor the progress of our crawls. We have extended the log files by, e.g., the URL costs, the page score (if available), the number of new services and new providers, both in total and per day.

In the following we will describe in more detail two of the processors listed above, the PDF to HTML Converter and the HTML Extractor. These two processors have enabled major improvements in terms of performance in processing related documents, i.e., mostly HTML and PDF documents. Performance is an important aspect for the Service Crawler, as we have an average throughput of around 20 documents per second (see Chapter 5 for details) and process millions of documents per crawl. Each of these documents (in case of HTML for example) is parsed, links are extracted, marked up data and specific page features are extracted, term vectors are built for the proper cost assignment (see Section 3.1.2.4), and finally the documents are stored on disk. Improving performance is in this scenario related to both the document processing time and the document storage place.

PDF to HTML Converter

Although the related documents that we collect do mainly consist of HTML documents, up to 20% of PDF documents may be part of an average crawl. Processing the PDF documents is unlike harder than processing the HTML files. The standard PDFExtractor that comes with Heritrix does provide support for parsing the PDF and extracting links from it, but the process is admitted to be quite slow. Moreover, following this approach, we are not able to extract any other features from the documents or to apply our term vector-based cost assignment approach upon them.

In order to address these issues we decided to transform native PDF documents to HTML before they are further processed within the crawler or the analysis steps afterwards. We have extended the Service Crawler with a new processor, PDF2HTMLConverter, that is situated in the processor chain in front of the HTMLExtractor. Within this processor we make use of an external open-source tool that converts PDF to HTML, running under Linux: ‘pdftohtml’

In order to call the tool we get a run-time instance of the environment in which the crawler is running (a Linux Debian system) and execute a simple Linux exec bash command.

The content of a fetched document within the crawler is recorded with the URI that is being processed by one crawler thread and can be accessed and retrieved as InputStream. Within the PDF2HTMLConverter we though do not only retrieve the content of the crawled document, we do replace it with the converted HTML. Together with the pure content we also change the content-type

in the HTTP Header to \texttt{\textquoteleft text/html\textquoteright}. The main reason for changing the original header corresponding to the modification we did with the content is that in later analysis steps (both in the Service Crawler scope and in other external Information Extraction components) some analysis method application choices are made based on the HTTP header of the delivered data. In order to keep track of the content modification we additionally add a new specific field to the HTTP Header: \texttt{X-seekda-Filter:pdf2html}. During the later performed analysis steps this addition allows us to easily recognize the transformed former PDF (now HTML) documents.

After the document leaves the PDF2HTMLConverter processor to be further processed within other processors, the document is treated like a native HTML document. Within the HTML processor its links are extracted and the term vectors for the cost assignment are built for it. Concerning the features we apply and the optimizations we did in the scope of HTML processing (see following of this section), as well as the bad performance of the original Heritrix PDFExtractor processor, the transformation from PDF to HTML has positive effects in both processing performance and general data quality (better focus due to the enabled cost assignment, easier identification of related documents).

### HTML Extractor - Optimized HTML Processing

The main documents being processed within the Service Crawler are HTML documents, which is the reason why performance and enhanced features within the HTML Extractor are important issues for us. The standard Heritrix HTMLExtractor processor is mainly responsible for the link extraction from HTML documents. It does not parse the HTML but performs Pattern Matching upon a simple char sequence gathered from the document’s content InputStream. This approach turned out to be unable to adequately fulfill our requirements on an HTML processor.

To be able to (a) get the ability to make enhanced use of the HTML structure of a document, and (b) allow a leaner storage of HTML documents in order to save disk storage place, we have replaced the standard Heritrix HTML processor with a new customized HTMLExtractor.

Regarding (a) we have integrated the simple term vector building method (as described in Section 3.1.2.4) into the HTML processor; this allows us to calculate a provenance page score and in the following do a better cost assignment to the extracted outlinks. This add-on makes use of the HTML structure, in that it uses the knowledge of where certain tokens appear in an HTML document (such as title, headings, normal text). While this extension to the HTMLExtractor does focus on better usage of the HTML structure, we present in the following improvements that we provided in terms of document processing and required storage place:

- **HTML mark-up filtering** - We remove all tags and tag attributes from HTML documents that do not provide any useful information regarding the type or importance of the tag content or that do provide pure format-
ting information (e.g., `<img>` tag, `align` attribute, etc.). What we keep are mainly the tags that provide the ground structure of the document (as, e.g., `<body>` tag, `<head>` tag, `<title>` tag, headings and tables), tags that provide link information (e.g., `<a>`, `<link>`, `<address>`) and tags that reveal information on the importance of specific Strings (as, e.g., `<b>`, `<i>`, `<u>`, `<strong>`, `<th>`).

- **Document shortening** - We shorten documents that are extensively long as our experience has shown that either the information contained within very long documents does not contain much concrete useful information or the documents are even spam documents. The maximal length that we allow for original HTML documents is 100,000 characters. As PDF documents often tend to contain longer descriptions or documentations we allow 300,000 characters maximum length for documents that have been transformed to HTML from PDF.

The results of the performance improvements have been very positive in that the processing performance is very fast despite the extended analyses we do on the document structure and content and in that we could reduce the required storage place by around 20%. Similar as we do it in the PDF2HTMLConverter and as we have described above, we replace the original content of the fetched document that is attached to the URI with the new lightened and shortened content. Again we do as well change the HTTP Headers: in case we only lightened up the mark-up we add the following header field: `X-seekda-Filter:lightened HtmlMarkUp`. In case we shortened the document we update the existing `Content-Length` field to the new document length and additionally add the fields `X-seekda-Filter:strippedHTML` and `X-seekda-Original-Content-Length` that allow us to keep track of the fact that this is a shortened document and to know its former size.

To create some sort of conformity for the new header structure we decided to introduce the `X-seekda-Filter:xxx` field, which in the future could be extended in a similar manner by new features.

### 3.2 Analysis and Output of the Crawl Data

The Service Crawler produces snapshots of that part of the Web that is relevant for Web Services. In the following we will describe the raw data that initially results from a crawl (Section 3.2.1), what analysis steps we run on this data (Section 3.2.2) and what we get as final output from the Service Crawler after the analysis (Section 3.2.3). In Section 3.2.4 we then provide an overview on how we semantically enrich the services using the analysis output.

#### 3.2.1 Raw Crawl Data

After having harvested the Web for Web Service descriptions and related information we remain with big amounts of raw data which can be categorized into
four different kinds of information:

- **WSDLs** - We first of all have a large amount of WSDL service descriptions, stored in the ARC file format. With each archive we produce an index file during the crawl that allows us to quickly jump to a specific offset in a specific ARC file and extract the corresponding archive record. Both the ARC file format and the index files are described in more detail in the following of this section.

- **Link Graphs** - During the crawl we build link graphs that contain information about which page(s) links to which other page(s). This information is useful to help denoting information that is related to services. Such related information may, e.g., be Web pages that link directly to Web Service descriptions (what we call an inlink of first grade).

Listing 3.2 shows a short example link graph:

```plaintext
http://example/services/wsdl
http://example/services/documentation.html
http://abc.com/.../
http://tests.at/.../
http://servicesToTest/services/wsdl
http://.../...
```

Listing 3.2: Example of a link graph

- **Related Information** - The hugest amount of data we remain with after a crawl iteration is the (possibly related) data (HTML pages, other text files, etc.) that we collected. The fetched documents are stored as ARC files (as described below) and come together with an index file.

- **RDF graph** - During the crawl we build an RDF graph containing basic statements about the identified services, their WSDLs and their providers.

As already mentioned before, the Heritrix writers store all data on disk in the ARC file format (Internet Archive): these archives aggregate data in approximately 100MB large files, the file starts with a special ARC header and concatenates the single WSDLs/Web pages. We use gzip compressed ARC files to further reduce the storage space requirements. In a gzip ARC file every single record is gzipped separately, which allows random access to the single records using offset information.

Each archive entry, or so-called URL record, contains the following information:

```plaintext
URL-record-v1 ==
<url><sp><ip-address><sp><archive-date><sp><content-type><sp><length><nl>
```

An example of such an archive entry - including customized seekda HTTP header changes as described in Section 3.1.3 - is presented in Listing 3.3.
http://abn.business.gov.au/(psb5wx45prykuuvnI5qqaf)/downloads/
UsingABNLookupwebsevices.pdf  210.193.176.124  20090702104819
text/html 92725
HTTP/1.1 200 OK
X−seekda−Original−Content−Length: 441695
Content−Type: text/html
Last−Modified: Wed, 22 Apr 2009 22:00:42 GMT
Accept−Ranges: bytes
ETag: "0216bc795c3c91:7 bf
Server: Microsoft−IIS /6.0
X−Powered−By: ASP.NET
Date: Thu, 02 Jul 2009 12:48:19 GMT
Connection: close
X−seekda−Filter: pdf2html, lightenedHtmlMarkUp
X−seekda−New−Content−Length: 91715

<html><title>Using ABN Lookup web services</title>
<meta content="text/html; charset=UTF−8"
><meta content="pdf2html 0.36" name="generator"
><meta content="Theresa Macgregor" name="author"
><meta content="2009−04−23T08:00:41+00:00" name="date"
><meta content="Revision: 0" name="subject"/>
</meta></meta></meta></meta></meta></body>

Listing 3.3: ARC Archive Header entry

To facilitate random access of ARC archives, every archive is accompanied
by an index file, having the same file name extended with an “.idx” suffix. Every
line within this index file corresponds to one record in the ARC archive. The
format of the line is <URL><tab><offset>. This allows an application to access
the contents of a URL by scanning the index file and then using the offset to
directly retrieve the desired document, instead of sequentially iterating through
the complete ARC file.

The raw data that we gather during one crawl, as described above, can take
very large dimensions. Table 3.3 shows as example the extent of raw data that
we have gathered in one example crawl - CrawlA. This crawl was an extended
crawl that intensively crawled the Web sites of known service related pages. The
links row shows the amount of links that we extracted and the size that the link
graphs took on disk. The logs and state rows describe the size of the internal
Heritrix state and log files. The two arcs rows show how big the collections of
overall fetched data became, both in a compressed and an uncompressed way. Finally WSDLs describes the comparatively small size of the WSDL archives.

This large amount of raw data that we gather from the Web is being reduced
in the data analysis step, described in the following section.

3.2.2 Data Analysis

After having harvested the Web for Web Service descriptions we remain with
a large amount of service descriptions and related documents. In the following
we describe our approach for analyzing the raw crawl data. We show in Section 3.2.2.1 what additional information we gather from the data and how we select
the information that we deem important. Section 3.2.2.2 will go into details on how we actually run our analysis steps.

### 3.2.2.1 Analysis steps

We run two different kinds of analysis steps on the gathered data. While the analysis steps that build unique service objects works only over the service endpoint, we require the overall raw crawl data for collecting the related information corresponding to the single service objects.

#### Unique Services

As we saw above we remain with a large amount of WSDLs after the termination of a crawl iteration. But not all of these WSDLs correspond to exactly one unique service (as already mentioned in Section 3.1.2.2 when talking about WSDL 'deduplication'): First, one WSDL can contain more than one single service, each bound to different endpoints. Even more usual is though the case that multiple WSDLs are out there that resume to one single service. Often service descriptions are hosted on more than one server, even sometimes from more than one provider. Our goal in providing service discovery means is though not only to return a user a set of WSDL descriptions that comply to his search, but a set of unique service objects that do not contain duplicates any more. Now the question is how we define uniqueness of Web Service descriptions or how we filter out the duplicate or similar service descriptions.

An obvious approach would maybe be to compare checksums or hash values of two service description files. But this approach would not be appropriate for checking the uniqueness of Web Service descriptions as the actual files may differ in minor points without though altering the meaning of the service descriptions. That is we need to find a deduplication approach which is specifically tailored to Web Service descriptions: we therefore have developed a deduplication method that is centered around the endpoint of a service. In the following we will describe our approach to deduplicate WSDLs by assembling all duplicate descriptions into one unique service description that represents a single unique service.

### Table 3.3: Raw Crawl Data - Crawl_A

<table>
<thead>
<tr>
<th></th>
<th>amount</th>
<th>size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>links</td>
<td>30.180.758 (from) - 1.310.455.874 (to)</td>
<td>56</td>
</tr>
<tr>
<td>logs</td>
<td>n/a</td>
<td>19</td>
</tr>
<tr>
<td>state</td>
<td>n/a</td>
<td>134</td>
</tr>
<tr>
<td>arcs (compressed)</td>
<td>34.028.621</td>
<td>374</td>
</tr>
<tr>
<td>arcs (uncompressed)</td>
<td>34.028.621</td>
<td>974</td>
</tr>
<tr>
<td>WSDLs</td>
<td>106.812</td>
<td>2</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td><strong>585</strong></td>
</tr>
</tbody>
</table>
One option to solve the problem could be to assign a unique service object to all unique service endpoints. Unfortunately this way we could have services that escape our deduplication method, as there may be single services that are bound to multiple endpoints. This is why we have adopted another approach: we build a mapping from all WSDLs to unique, newly defined, service identifiers. Each unique service is now defined by its provider and the provided service. In a first step we extract the provider name 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\(^9\) instead. An example would be the URL \texttt{http://www.library.uibk.ac.at/test.wsdl}, where the provider resolves to uibk.ac.at, the domain of the University of Innsbruck.

Having the provider extracted, we build a new unique (seekda) URL for the service. This URL contains first the provider's name. Then in a second step we add the service name to it (e.g., \texttt{http://seekda.com/providers/cdyne.com/IP2Geo}). The fact that we build these unique service objects based on provider and service names means that the service objects that we present to the user as result to his search do not necessarily consist of only one WSDL, but might as well refer to a set of WSDLs. If one service assembles a set of WSDLs under his 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 least 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.

After a crawl, as described in Section 3.1, where we have a resulting set of more than 200,000 WSDL service descriptions, and after applying our algorithm for service deduplication on it, we remain with more than 28,000 unique Web Services. These services belong to more than 8,000 service providers.

### Service Related Information

Detecting information on the Web that is related to specific services is an important issue if we want to increase the knowledge that we have about a service both in terms of its functionality and its non-functional properties as price, terms and conditions, etc. This is why we do not restrict our Web Service crawl to Web Service descriptions, but do as well gather related information from the Web. The information that we could extracted about the service descriptions during a crawl was either related to the service endpoint or to the service’s outlinks. To collect the service inlinks, and thus to collect all available related information that belongs to the single unique services, we need the raw crawl data that results from a crawl iteration, specifically the link graphs.

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 descrip-

\(^9\)\url{http://publicsuffix.org/}
tion and vice versa. We can gather this information from the link graphs that are being written during a crawl iteration (see Section 3.2.1); the crawler follows the outlinks in a given page and writes the from-link and its outlinks into a link graph. Now we combine the information of what service object belongs to what WSDLs with the information contained in the link graphs to build an index that maps the single service objects to all their related information.

To do so, we go through the link graphs and collect all information that is related to our WSDL service descriptions. Related means in this case either related by an “outlink”, that is a link from the WSDL file to a Web document, or related by an “inlink”, that is a link from a Web page showing to the WSDL description. We can collect inlinks of first grade (that is, directly linking to the WSDL) or of higher grade (that is linking to a document that links to a document ... that links to a WSDL), this only requires multiple passes of our analysis algorithm. We collect all these documents that we identify as related to the WSDL descriptions, relate them back to the new service objects (using the WSDL to service identifier mappings defined when building the unique service objects) and build a corresponding index.

As mentioned in Section 3.1.2.3 we have in the scope of the Service-Finder project also developed a term vector based methodology to identify service related information [18]. This work has been done together with project partners and is not part of this thesis.

3.2.2.2 Running the Analysis

In the following we describe how we run our analysis steps, concentrating on the building of unique service objects and the collection of service related information, as described in Section 3.2.2. These two analysis steps are currently done by executing a number of Java scripts in a given order. These scripts need to be run on the machine where the crawling engine is running and where the raw crawl data is stored.

The whole analysis task can be broken down to the following steps:

1. Manually start a specific analysis script.
2. After the job is finished, check whether the expected output file is written.
3. Check the output file(s) for syntactical correctness.
4. Repeat steps 1-3 in a specific order with other scripts.
5. Check samples of the final output for semantic correctness.

In the following we describe the different analysis scripts that need to be run, in the right order how they need to be executed:

1. `createService2SimilarWsdlsIdx` - We build a file, `services2wsdls.idx`, that contains a mapping from the newly built unique services to all similar WSDLs. For the WSDLs it contains an index that tells us where in the
WSDL archives we find the specific entry. The script iterates over the crawl job’s WSDL archives. The output looks as shown in Listing 3.4:

http://seekda.com/providers/100percentlove.com/Service
http://100percentlove.com/cartoogen/service/service.asmx?wsdl 19 74336660
http://seekda.com/providers/101webdesigns.com/BlogImporter
http://www.101webdesigns.com/api/BlogImporter.asmx?WSDL 10 67841419

Listing 3.4: Services to WSDLs index example

2. createWsdls2ServicesIdx - We build a file, wsdls2services.idx, that contains the mapping from the single WSDLs to their unique service object id. This script needs as input the services2wsdls.idx file that we build in Step 1. The output looks as shown in Listing 3.5:

http://seekda.com/providers/oclc.org/SRWSampleService
http://ald.yellowmap.de/MTE/MessagingService.asmx?WSDL
http://seekda.com/providers/yellowmap.de/MessagingService

Listing 3.5: WSDLs to services index example

3. createService2InlinksIdx - This script writes a file, services2inlinks.idx, that maps unique service objects to their relevant links, i.e., to all documents that have linked directly to one of the WSDLs that belong to that service object. For the related links it builds an index that tells us in what ARC archive and at what offset we find the specific entry. Furthermore it adds an information on the reason why a link is relevant for the service (e.g., 'f' for an inlink of first grade).

This script iterates over the link graphs that we have built during the crawl, as well as over the ARC archives containing all fetched documents to build the index (see Section 3.2.1). The output looks as shown in Listing 3.6:

http://seekda.com/providers/cdyne.com/PhoneNotify
http://wsoogle.com/directory.do?category=Communications 502 17797994 f
http://ws.cdyne.com/NotifyWS/phononotify.asmx 408 77268226 f
http://www.xmethods.org/ve2/ViewListing.po?key=420987 55 55096702 f

Listing 3.6: Services to inlinks index example

4. createRelatedDocArc - This script uses the services2wsdls.idx and the services2inlinks.idx built in the steps before to build new archives that contain the WSDLs and all the documents that have been identified as relevant. That is it iterates over both the WSDL and document archives we have
collected during the crawl. The output are archive files and corresponding indexes. The resulting structure looks as shown in Listing 3.7:

```
seekda−20100609111354−00001.arc.gz
seekda−20100609111354−00001.arc.gz.idx
seekda−20100609111759−00002.arc.gz
seekda−20100609111759−00002.arc.gz.idx
...```

Listing 3.7: Archive and index files structure

5. `createService2RelatedInfo` - This script uses the `services2wsdls.idx` and the `services2inlinks.idx` files written in the steps before and writes one new file, `services2related.IDX`, containing the mapping from the unique service objects to their corresponding WSDLs and relevant documents as they are saved in the new archives that we have built in Step 4. With the mapping they also contain the new corresponding index. The outlook looks as shown in Listing 3.8:

```
http://seekda.com/prov iders/cdyne.com/PhoneNotify 1 151 a
http://ws.cdyne.com/notifyws/phon enotify. asmx?wsdl 1 32738 w
http://ws.cdyne.com/NotifyWS/LM_GetListIDsByLicensekey 1 5713 o f
http://ws.cdyne.com/NotifyWS/GetIncomingCallScript 1 6215 o f
http://wiki.cdyne.com/index.php/Notify_TextToSay_Commands 1 7203 l f
http://www.cdyne.com/downloads/SPECS−CDYNEPhoneNotify.pdf 1 8641 l f p
```

Listing 3.8: Services to related documents index

The format of this resulting final index file will be explained in more detail in Section 3.2.3.

Although the analysis scripts can be quite flexibly adapted to new needs, they are not optimal in that they need too much manual intervention to run them in the correct order and to check whether the outputs are correct. Depending on the extent of raw crawl data that results from a crawl this analysis period might well last up to a whole day and requires then repeatedly attention. Looking into new means to run these analysis jobs, by, e.g., using specific job schedulers, is part of potential future work.

Next to assembling the above described data archives and writing a corresponding index file, we store the same information as RDF triples. This allows us to later on (a) analyze the data by querying it using SPARQL, and (b) use the semantic meta-data for our ranking approach described in Chapter 4.

### 3.2.3 Crawl Data Output

When all our analysis steps have run through, we have assembled the following final data:

- **Service identifiers** - unique service objects
• Archives - WSDL service descriptions
• Archives - WSDL abstracts
• Archives - related information
• Index - index file for the collected archives
• RDF graph - lightweight semantic service annotations in RDF

The output from the Service Crawler is stored in ARC archives (as described in Section 3.2.1), and comes together with an index file that provide two kinds of information: (a) where in the archives a specific document is stored (in terms of archive number and offset), and (b) the relation between unique services and their corresponding WSDL files or other related documents (as described in Step 5 in Section 3.2.2.2).

The bullet list outlining the final output of the Service Crawler contains one item, WSDL abstracts, that we have not mentioned so far. To ease further analysis work on top of the WSDL service descriptions we have developed a WSDL to HTML translator. That is for each service we consolidate the important data (title, port type, operations, etc.) from a WSDL into an HTML document. This HTML document can then be used for further analysis, as, e.g., a term vector analysis (on WSDLs and related documents).

Translating the WSDL description to HTML means to create some sort of WSDL 'abstract' (in HTML format) for each service. In this abstract we consolidate the relevant data from the WSDL: service name, port type, operations, endpoint, documentation, etc. As we create only one WSDL abstract per service we first choose the 'main' WSDL for each service (as explained in Section 3.1.2.2 one service object may contain more than one WSDL document). We consider the WSDL document with the URL that has the shortest path and the least subdomains, and that contains the provider name as main WSDL description of a service (as we assume that this WSDL is published by the provider itself). In case none of the WSDL documents of a service contains its provider name in the URL we choose the document having the URL with the shortest path and the least subdomains as main one.

The index file that results for the final collected data archives has already been provided above in Listing 3.8. Behind each URL some additional information is provided behind the archive offset. Depending on the document type the index contains the following fields:

• Unique service identifier - Behind the URL of the unique service identifier we have: (1) archive number of this service’s WSDL abstract, (2) corresponding archive offset and (3) type of document, i.e., a for WSDL abstract.

• WSDL description - Behind the WSDL URL we have: (1) archive number, (2) archive offset and (3) type of document, i.e., w for WSDL 1.1 documents or W for WSDL 2.0 documents.
3.2.4 Automatically Enriching Service Descriptions

As already mentioned before, we build next to the archives and index, both during the crawl and during the analysis process, an RDF graph containing semantic annotations of the service objects. That is to each new, unique service identifiers we 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 and seekda Crawl Ontology.

These ontologies are meant for a bottom-up service annotation approach, which allow to describe (a) what we know about the service from analysing its structure and/or link relations and (b) simple categorisations. They can be expressed in RDF and do not require expensive reasoning; thus they allow for large scale service annotations. Moreover these lightweight annotations can happen in a fully or semi-automated way. Both ontologies will be further described in the following of this section.

Service Meta-data Content

Part of the RDF meta-data that we store corresponds to the information that we have collected in the afore mentioned archives and index file: the relation between services, their providers and their related documents. Other meta-data that we collect refers to basic information that we can extract around the fetched Web Services and goes beyond the data in the archives and index file. One of these is the geographic location of the service, that is the country where the service is located. This is an information that we gather around the service endpoint. Unfortunately this information bit does not yet tell us where the actual provider of the service is located, as it might be that a provider hosts his services in another country. That is, it could be that a service is provided by a German company and is hosted in South-Africa. In this case the service object would be listed as African service.

Other 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 this information as RDF triples on request. When we check the liveliness of a service, we check at the same time whether the servers are correctly implementing the SOAP protocol, what does nevertheless not mean that we check the actual functionality of the services.
Moreover we store for each service-related document additional meta-data, as, e.g., the number of external links on a page, the number of internal links (linking to the same site), the number of camel-cased tokens, etc. This information can - in future work - be used to refine the focused crawl approach and to extend the service identification beyond WSDL service descriptions (e.g., RESTful services).

All the meta-data that we store so far about the service objects is related to the service endpoints, the servers they are hosted on, or the crawl link graphs, and can thus be concluded by the crawler or a direct 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 (as, e.g., on the seekda Web Service search engine, the Service-Finder Portal and the SOA4All studio).

Service Meta-data Structure

In the following we briefly describe the ontologies that we use in the scope of this thesis to describe service related meta-data:

**Service-Finder Service Ontology.** All information about the services, such as their WSDL description(s), their related documents, their provider, their operations, etc. is structured according a generic Service Ontology. This ontology has been developed in the scope of the European project Service-Finder, which is building a platform for Web Service discovery embedded in a Web 2.0 environment. The [Service-Finder Service Ontology](http://www.service-finder.eu/ontologies/ServiceOntology) is available at [http://www.service-finder.eu/ontologies/ServiceOntology](http://www.service-finder.eu/ontologies/ServiceOntology) (respectively the RDFS version at [http://www.service-finder.eu/ontologies/sfo.rdfs](http://www.service-finder.eu/ontologies/sfo.rdfs)) under the Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License.10

**seekda Crawl Ontology.** To be able to store supplementary data about services, we have - within seekda and in collaboration with other seekda researchers - extended the Service-Finder Service Ontology by developing the **seekda Crawl Ontology**. Such meta-data contains mainly statistics about the fetched documents: the number of textual links (Website internal links), the number of external links and the number of camel-case tokens. Also we can use this ontology to specify how to retrieve an archived document, that is by means of the name of the ARC file along with the offset of the record. Moreover this ontology allows the description of RESTful service scores and classifications, a topic that is though not covered within this thesis. The **seekda Crawl Ontology** is available at [http://seekda.com/ontologies/CrawlOntology.rdfs](http://seekda.com/ontologies/CrawlOntology.rdfs).

Figure 3.4 provides an overview of the most important concepts of the

---

10[http://creativecommons.org/licenses/by-nc-sa/3.0/]

11Figure by Manuel Brunner, Nathalie Steimmetz and Holger Lausen, seekda GmbH.
There are four types of document annotations that can be applied to WSDL-based service objects: WSDL annotations, outlink annotations (i.e., an annotation to a document to which one of the services’ WSDL files links), inlink annotations (i.e., an annotation to a document that itself links to one of the services’ WSDL files) or term vector similarity annotations (i.e., an annotation to a document that is not related by a direct link to or from the WSDL files, but that is classified as related by the term vector similarity analysis). Finally we can use the seekda Crawl Ontology to store document statistics, as well as the location of the single documents within the crawl archives.

In the remaining of this section we provide an overview of the structure that we use to store the meta-data for WSDL-based services and their related information. We will use the following namespaces and prefixes:

- **Service-Finder Service Ontology:**
  sf: http://www.service-finder.eu/ontologies/ServiceOntology#

- **seekda Crawl Ontology:**
  sco: http://seekda.com/ontologies/CrawlOntology#

- **XML Schema:**
  xsd: http://www.w3.org/2001/XMLSchema#
The following Listing 3.9 provides the meta-data structure overview:

Listing 3.9: Crawl meta-data structure

```
<serviceURI> <rdf:type> <sf:Service>
<serviceURI> <sf:hasProvider> <providerURI>
<providerURI> <rdf:type> <sf:Provider>
<providerURI> <sf:hasDomain> <providerDomain>
<providerDomain> <rdf:type> <sf:Domain>
<documentURI> <rdf:type> <sf:Document>
<documentURI> <sf:hasSize> <xsd:number>
<documentURI> <sf:retrievedAt> <xsd:date>
<documentURI> <sf:hasTitle> <xsd:string>
<documentURI> <sco:numberOfCamelCaseTokens> <xsd:number>
<documentURI> <sco:numberOfTextualLinks> <xsd:number>
<documentURI> <sco:numberOfExternalLinks> <xsd:number>
<wsdlURI> <rdf:type> <sf:WSDLDocument>
<wsdlAnnotationID> <rdf:type> <sf:DocumentAnnotation>
<wsdlAnnotationID> <sf:belongsToDocument> <wsdlURI>
<wsdlAnnotationID> <sf:isAboutEntity> <serviceURI>
<outlinkURI> <rdf:type> <sf:Document>
<outlinkAnnotation> <rdf:type> <sf:DirectOutLink>
<outlinkAnnotation> <sf:belongsToDocument> <outlinkURI>
<outlinkAnnotation> <sf:isAboutEntity> <serviceURI>
<inlinkURI> <rdf:type> <sf:Document>
<inlinkAnnotation> <rdf:type> <sf:DirectInLink>
<inlinkAnnotation> <sf:belongsToDocument> <inlinkURI>
<inlinkAnnotation> <sf:isAboutEntity> <serviceURI>
<tvURI> <rdf:type> <sf:Document>
<tvAnnotation> <rdf:type> <sf:TermVectorSimilarityAssociation>
<tvAnnotation> <sf:belongsToDocument> <tvURI>
<tvAnnotation> <sf:isAboutEntity> <serviceURI>
<wsdlURI> <sco:contentLocationArc> <xsd:string>
<outlinkURI> <sco:contentLocationArc> <xsd:string>
<inlinkURI> <sco:contentLocationArc> <xsd:string>
<tvURI> <sco:contentLocationArc> <xsd:string>
```
4

Web Service Ranking

In the previous chapter we have described the methodologies that we have developed for Web Service Location, based upon Web crawling and subsequent data analysis. After having located publicly available Web Services on the Web, we are able to provide potential customers with the possibility to discover publicly available services that fit their needs, based upon both the technical service descriptions and information that is related to the services. Now after users have been made aware of the simple existence of services that provide the functionality they are looking for, ranking is one of the most important steps to support them in selecting the most best-fitting service: it would be a tedious task for a human to filter out the services that are relevant to him, regarding the large size of publicly available services on the Web.

We propose in the following of this chapter a novel service ranking approach that is based on non-functional properties of services: information about them that is available on the Web, their availability, their hyperlink relation, etc. Section 4.1 will focus on the aggregation of ranking-relevant data into a semantic model using ontologies, while we present the ranking model that is based upon this data in Section 4.2.

4.1 Feature Aggregation Based on Ontologies

This section describes the data aggregation for our ranking approach that is based on feature aggregation using ontologies. We use semantic technologies to aggregate various aspects related to Web Services in a unified model, aspects that encompass information that is available about services by analyzing their description and their hyperlink relations, by talking to their hosting server, etc. We do not rely on handcrafted, manually added, information, but only take into account real-world information that is anyway available.

As described in Chapter 3 the services are gathered by crawling the Web. Together with the services the Web is fostered for related documents, e.g., ser-
vice descriptions, help pages, etc. The data that is resulting from the crawler comes together with RDF meta-data that describes amongst others the relation from services and their related documents. The service meta-data can be used for a multi-valued ranking approach, taking into account aspects like the number and the quality of related documents, “live” monitoring data and metrics from the WSDL descriptions, as, e.g., how much documentation is provided for a service. Based on our crawling experience and on our work on the seekda Web Service search engine (http://webservices.seekda.com) we see that the aspects upon which we base the ranking approach are realistically available: looking at only one specific crawl produced for the seekda Web Service search engine, we see that from more than 28,000 publicly available services approximately 20,000 relate to one or more other Web resources; around one fourth of all WSDL descriptions contain some documentation on the service or operation level; all available services are monitored on a daily basis by the seekda search engine.

In the following we will first outline what RDF meta-data we use for ranking; next we will describe the WSDL metrics and the monitoring data that we build upon and will in a last step provide an overview on existing and new ontologies that we use for modeling the ranking. The data described in this section is used for the Multi-valued ranking approach as described in Section 4.2 Each of the following subsections will introduce the crawl meta-data that it relies upon, if any. Section 4.1.4 will then describe in detail the new ontology elements that we use to describe the ranking.

We use the following namespaces and prefixes in the above mentioned subsections:

- **Service-Finder Service Ontology:**
  
sf: http://www.service-finder.eu/ontologies/ServiceOntology#

- **seekda Crawl Ontology:**
  
sco: http://seekda.com/ontologies/CrawlOntology#

- **seekda Ranking Ontology:**
  
sro: http://seekda.com/ontologies/RankingOntology#

- **XML Schema:**
  
xsd: http://www.w3.org/2001/XMLSchema#

### 4.1.1 Crawl Meta-data

For WSDL services and their related resources the meta-data delivered by the crawler consists mainly of annotations to the single Web documents, tying them on the one side to a service and describing on the other side of what kind the relation to the service is. We will use the following information for ranking:

- Number of related documents per service
- Kind of relation from document to service
The meta-data is stored using elements of the Service-Finder Service Ontology, as shown in Listing 4.1 as RDF triples. DirectInLink, DirectOutLink and TermVectorSimilarityAssociation are sub-classes of a DocumentAnnotation.

```
<sf:DirectInLink> <sf:isAboutEntity> <sf:Service>
<sf:DirectInLink> <sf:belongsToDocument> <sf:Document>
<sf:DirectOutLink> <sf:isAboutEntity> <sf:Service>
<sf:DirectInLink> <sf:belongsToDocument> <sf:Document>
<sf:TermVectorSimilarityAssociation> <sf:isAboutEntity>
  <sf:Service>
<sf:TermVectorSimilarityAssociation> <sf:belongsToDocument>
  <sf:Document>
```

Listing 4.1: WSDL service and related documents meta-data used for ranking

### 4.1.2 WSDL Metrics

A WSDL describes a Web Service from an “operational” point of view: services, their operations, messages, message formats, endpoints, network bindings, etc. While this information as such is not useful for ranking the services, the documentation of the single elements is worth being taken into account (<documentation> tag within both WSDL 1.1 and WSDL 2.0): a well documented WSDL improves the ranking of the corresponding service.

We take into account the documentation of the service and of the operations. The data is stored using elements of the Service-Finder Service Ontology, as shown in Listing 4.2 as RDF triples.

```
<sf:Service> <sf:hasDescription> <xsd:string>
<sf:Operation> <sf:hasDescription> <xsd:string>
<sf:Service> <sf:implementsInterface> <sf:Interface>
<sf:Interface> <sf:hasOperation> <sf:Operation>
```

Listing 4.2: WSDL meta-data used for ranking

### 4.1.3 Monitoring Information

Interesting criteria for service ranking are related to Quality of Service information. One such information is the availability of services, i.e., their liveliness. This data is monitored and stored by seekda (Web Service search engine at http://webservices.seekda.com/) on a daily basis. The availability is based upon the endpoint of a service. Monitoring the liveliness of a service does not mean that the functionality of the service is tested in any kind: it expresses whether the server where the service is hosted is reachable or not, checks at the same time whether the server is correctly implementing the SOAP protocol, whether the page needs an authentication, and more, based on the HTTP response codes.

The availability data is calculated on a weekly basis. It contains the average (percentage) availability of a service over the last 6 months, the last month and the last week (if possible). Listing 4.3 shows the elements that we use from the Service-Finder Service Ontology to store this data.

```xml
<sf:Service> <sf:hasDescription> <xsd:string>
<sf:Operation> <sf:hasDescription> <xsd:string>
<sf:Service> <sf:implementsInterface> <sf:Interface>
<sf:Interface> <sf:hasOperation> <sf:Operation>
```
4.1.4 Ranking Ontologies

To structure and store the data as described above we rely, as already mentioned, on ontologies. Where possible, we reuse the Service-Finder Service Ontology and the seekda Crawl Ontology (see Section 3.2.4 for more details). Furthermore we have developed a new seekda Ranking Ontology that allows us to express the new ranking specific information that is not yet expressible within the other two ontologies.

seekda Ranking Ontology. To be able to store ranking specific data about services, we have developed the seekda Ranking Ontology (shown in Listing 4.4). This ontology allows the description of meta-data that is required to calculate the ranking, and that is not covered by the two other ontologies, and provides a possibility to store each of the single ranking values that are calculated with the help of the aggregated meta-data described in the earlier sections. One of the ranking values is meant for Web API ranking; we do nevertheless not describe this ranking in more detail, as neither Web API location nor ranking are covered within this thesis. The seekda Ranking Ontology is available at http://seekda.com/ontologies/RankingOntology.rdfs.

4.2 Multi-valued Ranking

In Section 4.1 we have outlined what aspects of services we aggregate to build our ontology-based feature aggregation for multi-valued ranking approach. The data is aggregated into a unified model using mainly three ontologies, the Service-Finder Service Ontology, the seekda Crawl Ontology and the seekda Ranking Ontology. We do not only use the ontologies for the meta-data that is needed
to calculate the rankings but as well to structure and store the rank values. The fact that we have all the service meta-data that we need for our ranking available as semantic data allows each semantic-aware client to build its own ranking based on the same service meta-data creating individual rules (e.g., using SPARQL).

In the following we will present the way we combine the data gathered as described in Section 4.1 to get a global ranking value (Section 4.2.1). In Section 4.2.2 we will briefly describe the implementation of our approach, before we conclude this chapter in Section 4.2.3 by describing the advantages of our multi-valued ranking approach over other state-of-the-art approaches (as described briefly in Section 2.3).

4.2.1 Rules For A Global Multi-valued Rank

When applying our ontology-based feature aggregation for multi-valued ranking approach to WSDL services, we distinguish two steps: we first calculate three independent ranking values (based on crawl meta-data like info on related documents, WSDL metrics and monitoring data), which are then in a next step combined to one global rank. The following subsections present the rules that we apply to calculate the single rank values, described in a procedural pseudo-code.

4.2.1.1 Related Documents Rank

This rank is based on the crawl meta-data that is delivered by the crawler, as shown in Listing 4.1, and will be calculated based on the following information and assumptions:

- How many related documents does a service have? We need to check the document annotations that belong to a service and then count the unique documents that are tied to the annotations (as more annotations can refer to the same document).
  
  We improve the ranking of a service that is related to other Web resources as in that case the probability is higher that the service contains some relevant documentation, pricing information, etc. (e.g., on the service provider’s homepage).

- How is the document related to a specific service? Documents can be direct inlinks or direct outlinks of the WSDLs that belong to a service, or the connection to the service can be coming from a term vector analysis of the documents and the service.

  After manual analysis of a number of services and their related documents we got to the conclusion that important information bits are mostly contained in the services’ inlinks (i.e., pages that link to the WSDL document), as well as in pages that are related via term vector similarity (e.g., pages that speak about the service but are not related to it via a direct link).
In a first step we thus need to calculate the number of related documents per service. To do so it is not enough to just take the number of document annotations as one document might have several annotations (e.g., a document that has a DirectOutLink annotation and a TermVectorSimilarityAssociation). We need to first extract all annotations of type DirectInLink, DirectOutLink and TermVectorSimilarityAssociation. This way we get the identifiers of all documents that correspond to the annotations. Now we count the documents, counting multiple occurrences of the same document only once. This value is then stored using the hasNumberOfRelatedDocuments relation of the seekda Ranking Ontology (see Section 4.1.4).

Now the related documents rank is calculated as follows (described in pseudo-code) in Listing 4.5:

```plaintext
// get the average number of related docs per service
average = totalNumberOfRelatedDocs / numberOfServices;
// get the root mean square deviation of the distribution of
// related docs per service
sumDeviationFromAverage = 0;
for (Service s : allServices) {
    sumDeviationFromAverage += s.numberOfRelatedDocs - average;
}
variance = power(sumDeviationFromAverage, 2) / numberOfServices - 1;
rootMeanSquareDeviation = positiveSquareRoot(variance);
// get max outliers values
maxOutlier = average + (2.5 * rootMeanSquareDeviation);
// take into account the kind of relation from document to
// service. If a service has a number of related documents
// that is outside of the max outlier value we set the number
// to the average of related documents per service in order
// to not allow spam to influence the ranking value
for (Service s : allServices) {
    temporaryRank = 0;
    if (s.numberOfRelatedDocs > maxOutlier) {
        s.numberOfRelatedDocs = maxOutlier;
    }
    if (s.hasInlink) {
        temporaryRank = s.numberOfRelatedDocs * 5;
    }
    if (s.hasTermVectorAssociatedDoc) {
        temporaryRank += s.numberOfRelatedDocs * 4;
    }
    if (s.hasOutlink) {
        temporaryRank += s.numberOfRelatedDocs * 2;
    }
    s.finalRank = temporaryRank / maxOutlier / 11;
}
```

Listing 4.5: Calculation of the Related Documents Rank

The single values that are used for the single kinds of related documents to calculate the temporary rank are experimental: they can be changed and adapted, in order to discover the values that seem optimal. The final rank is stored for each service using the hasRelatedDocsRank relation of the seekda Ranking Ontology.
4.2.1.2 WSDL Metrics Rank

This rank is based on metrics that we extract from the WSDL descriptions. We currently take into account the documentation of (a) the service element, and (b) the operations. As we mentioned already in Section 4.1, approximately a fourth of all service descriptions contain some documentation on the service or operation level. The rank is calculated as follows in Listing 4.6:

```java
for (Service s : allServices) {
    finalRank = 0;
    if (s.hasServiceDocumentation) { s.finalRank = 1; }
    if (s.hasOperationDocumentation) { s.finalRank += 3; }
    s.finalRank = finalRank /4;
}
```

Listing 4.6: Calculation of the WSDL Metrics Rank

We put more importance on the documentation of the single operations than of the service documentation, as we think that the operation might contain useful information regarding the functionality provided by the operation and regarding its invocation. We currently do not differentiate between whether all operations of a service are documented or only one or some. The final rank is stored for each service using the hasWSDLMetricRank relation of the seekda Ranking Ontology.

4.2.1.3 Monitoring Rank

This rank is based on the liveliness information of a service, e.g., is the server reachable, does it correctly implement the SOAP protocol, etc. This liveliness information is delivered on a weekly basis as shown in Listing 4.3. The availability score is a number between 0 and 1 that is set depending on the endpoint check result. The score is, e.g., 0 for read time-outs or errors and 1 if, based on the resulting payload (e.g., XML fault), we are rather sure to be talking to a WSDL over SOAP. In-between different scores are set to express pages that are not found, pages that require a login or an authentication, etc., mostly based on the HTTP response code.

We get the average service availability score for different time periods: last week, last month and last 6 months. We assume that the long-time availability of a service is more relevant than only the short-time availability over one week. It is important to note that this rank does not state anything about whether the functionality that the service announces is correctly implemented or not. The rank is calculated as follows in Listing 4.7 and is stored for each service using the hasMonitoringRank relation of the seekda Ranking Ontology:

```java
for (Service s : allServices) {
    finalRank = ((s.availabilityLastWeek * 1.5) +
                 (s.availabilityLastMonth * 2.5) +
                 (s.availabilityLast6Months * 6)) / 10;
}
```

Listing 4.7: Calculation of the Monitoring Rank
4.2.1.4 Global Rank

We calculate the global rank based on the Related Documents Rank, the WSDL Metrics Rank and the Monitoring Rank. The single ranks are numbers between 0 and 1, and from these we calculate the global rank as follows in Listing 4.8, putting equal relevance on the availability of documentation (related documents being estimated more important than the documentation within the WSDL) and on the liveliness of a service. This way services that are on the side reliable and on the other side (assumably) well documented are ranked best. The global rank is stored for each service using the hasGlobalRank relation of the seekda Ranking Ontology.

for (Service s : allServices) {
    s.globalRank = (s.hasRelatedDocsRank * 0.35) +
                   (s.hasWSDLMetricRank * 0.15) +
                   (s.hasMonitoringRank * 0.5);
}

Listing 4.8: Global Rank Calculation for WSDL-based services

4.2.2 Implementation

The service ranks, produced as described above, take as input meta-data in RDF triples format and return the ranks in the same way. We use the seekda Ranking Ontology (as introduced in Section 4.1.4) to store and distribute the service ranks. In-between we have a Java component that calculates the ranks. Together with the single ranks the meta-data triples that the ranks are based upon can be distributed. As both the single ranks and the global rank are values between 0 and 1, all reasoners that can do ordering on numbers are able to work with the ranks.

4.2.3 Advantages and Novelties

The approach that we follow with our multi-valued ranking in general and specifically with the export of the ranking related data in a semantic format has three major advantages (and novelties):

1. Each RDF aware client can understand the rational of a ranking, i.e., it can work with the final ranks, but can as well analyze the data on which the ranking is based. This is a transparent approach that is very unlike most of the currently available approaches. It allows in the end each client to perform, if desired, its own ranking calculation with the service meta-data that is provided.

2. Our ranking approach covers and combines many aspects of a service, like its documentation, the existence of related information that is available on the Web and the liveliness data (which is a strong QoS criteria). Most existing ranking approaches concentrate rather on individual ranking criteria like availability, reputation, etc.
3. We get one part of our service meta-data from the Service Crawler, which extracts this information in its analysis process (as described in Section 3.2). The other part relies on a daily monitoring of the connectivity of the services; valuable data which is provided by seekda and which is, up to our knowledge, only monitored in that detailed way by the seekda Web Service search engine.
In the scope of this thesis we have developed a Web Service Location and a Web Service Ranking approach that rely on crawling the Web for services and related information. In the following we will present the evaluation of these approaches: Section 5.1 describes the criteria that we use to evaluate the Web Service crawling, and provides the corresponding evaluation results, while Section 5.2 informs about our multi-valued ranking evaluation approach.

5.1 Service Crawling

The development of the Service Crawler as described in this thesis has happened over a period of approximately two years. Already at the beginning of this period we have defined some evaluation criteria for the Service Crawler that enabled us to continuously improve and check the quality of all data releases. In the following we first describe the service crawling evaluation criteria (Section 5.1.1), and then explain and compare in Section 5.1.2 the concrete evaluation results for four different crawl iterations, listed in the following table.

<table>
<thead>
<tr>
<th>Crawl #</th>
<th>Date of Web crawl</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawl 1</td>
<td>September 2008</td>
</tr>
<tr>
<td>Crawl 2</td>
<td>December 2008</td>
</tr>
<tr>
<td>Crawl 3</td>
<td>June 2009</td>
</tr>
<tr>
<td>Crawl 4</td>
<td>November 2009</td>
</tr>
</tbody>
</table>

5.1.1 Evaluation Criteria

The evaluation criteria of the Service Crawler reflect several indicators. Some of the indicators represent pure performance measures as how much content could be obtained in a particular amount of time (e.g., number of documents crawled per second, kB crawled per second). Others refer to the quality of the resulting
data, i.e., to how much relevant information could be found (e.g., number of WSDL documents, number of extracted service identifiers).

In addition to this numeric evaluation every crawl has been analysed using varying metrics in order to understand the nature of the relevant part of the Web and to better adjust the crawling framework to fetch relevant information.

### 5.1.1.1 Throughput Performance Metrics

Table 5.1 gives an overview of the relevant metrics that are used to assess how well we have utilised the available resources.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>days</td>
</tr>
<tr>
<td>avg. throughput</td>
<td>kb/sec</td>
</tr>
<tr>
<td>avg. document throughput</td>
<td>docs/sec</td>
</tr>
<tr>
<td>hosts visited</td>
<td>#</td>
</tr>
<tr>
<td>hosts found</td>
<td>#</td>
</tr>
<tr>
<td>URIs extracted</td>
<td>#</td>
</tr>
<tr>
<td>URIs queued</td>
<td>#</td>
</tr>
</tbody>
</table>

Table 5.1: Service Crawler Throughput Performance Indicators

### 5.1.1.2 Focused Crawl Performance Metrics

In addition to the throughput criteria that are applicable for every crawl we devise a set of criteria that measures how well the crawling performs with respect to finding Web Services and their related information. The criteria envisioned are depicted in Table 5.2.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of relevant archives</td>
<td>GB</td>
</tr>
<tr>
<td>number of WSDL files</td>
<td>#</td>
</tr>
<tr>
<td>number of related documents</td>
<td>#</td>
</tr>
<tr>
<td>hosts with related content</td>
<td>#</td>
</tr>
<tr>
<td>number of providers</td>
<td>#</td>
</tr>
<tr>
<td>number of services</td>
<td>#</td>
</tr>
</tbody>
</table>

Table 5.2: Service Crawler Focus Performance Indicators Crawler

### 5.1.1.3 Additional Metrics

Additional metrics help to analyze and evaluate the overall resource usage and the resulting performance.
Resource Usage. We have continuously measured the load of various components such as CPU, disk I/O, network I/O and so on. For a longer running crawl it is also important to have an overview of how the resource usage varies over time. It might for example be that a certain component within the crawler performs well at the beginning but causes significant disk I/O at the end of the crawl.

WSDLs and Services. Another evaluation that we perform is the analysis of the distribution of WSDLs and services. That is, we do not only look at the total numbers of WSDLs and services, but as well at how many services do provide how many WSDLs and what information can we gain from this.

Services and Related Documents. Similar as for WSDLs and services as described above, we analyze the distribution of services and related documents.

5.1.2 Evaluation Results

In the following we present the evaluation results of four large broad crawl iterations as outlined in the beginning of this section; these four crawl jobs have been running using different versions of the Software Crawler. Between these crawls we have run a number of crawls that are not taken into account for this main evaluation, lots of them aiming at testing new features and improving the Service Crawler performance.

Crawl 4 is a particular case, as although it has been aggregated to one big crawl output, it is based upon two different crawl jobs: one broad crawl with the goal to find as many WSDLs as possible and one crawl that had been started at a later moment, after we had already found a large number of services, with the goal to crawl the domains of service providers more extensively in order to detect as many as possible related documents. In the following we will present the final evaluation numbers of Crawl 4, the numbers corresponding to the final data output and thus to the two unified crawls.

5.1.2.1 Throughput Performance Metrics - Results

Table 5.3 shows the results of the relevant throughput performance metrics. We can see that the number of found hosts and of queued URIs was significantly larger in Crawl 1 than in the other crawls. One reason for this is: (a) Crawl 2, Crawl 3 and Crawl 4 jobs did not run as long (in terms of days) as Crawl 1. Another important reason is the improvement in terms of focus in the Service Crawler. Between Crawl 1 and Crawl 2 and between Crawl 3 and Crawl 4 we improved the cost assignment for URIs, a deciding factor in what URIs are queued. This results in more profound checks whether a URI is to be scheduled or not and thus to a smaller number of queued URIs (for more details see Section 3.1.2.4). Also some performance improvements have happened during this period, such as the extension of Heritrix with our custom, fast HTML Extractor (i.e., the crawler processor that analyzes HTML files to see whether
they are Web Service related and extracts links to build the crawl link graph. See Section 3.1.3 for more details). Crawl 4 shows the so far best results in terms of focus where we detected the largest number of services and related documents with the least “effort” in terms of visited hosts and fetched URIs so far. The results of the improved focus are also well visible in Table 5.4.

Table 5.4: Throughput Performance Results: Crawl 1 - Crawl 4

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Value</th>
<th>Crawl 1</th>
<th>Crawl 2</th>
<th>Crawl 3</th>
<th>Crawl 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>duration</td>
<td>days</td>
<td>18</td>
<td>15</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>avg. throughput</td>
<td>kb/sec</td>
<td>251</td>
<td>360</td>
<td>246</td>
<td>417</td>
</tr>
<tr>
<td>avg. document throughput</td>
<td>docs/sec</td>
<td>13.7</td>
<td>24.05</td>
<td>18.74</td>
<td>20.25</td>
</tr>
<tr>
<td>hosts visited</td>
<td>#</td>
<td>1.402.444</td>
<td>1.024.796</td>
<td>1.104.685</td>
<td>832.794</td>
</tr>
<tr>
<td>hosts found</td>
<td>#</td>
<td>40.361.093</td>
<td>3.060.475</td>
<td>3.542.753</td>
<td>1.549.776</td>
</tr>
<tr>
<td>URIs extracted</td>
<td>#</td>
<td>20.979.387</td>
<td>21.962.352</td>
<td>15.765.437</td>
<td>10.691.908</td>
</tr>
<tr>
<td>URIs queued</td>
<td>#</td>
<td>87.178.246</td>
<td>49.869.479</td>
<td>34.894.326</td>
<td>15.284.878</td>
</tr>
</tbody>
</table>

5.1.2.2 Focused Crawl Performance Metrics - Results

Table 5.4 shows the results of the relevant focused crawl performance metrics. The most significant changes can be seen in the number of found WSDLs and related documents. Between Crawl 1 and Crawl 2 we had improved, as already described above, the cost assignment strategy of the Service Crawler by adding a Term Vector similarity approach in order to detect pages that talk about services. Due to these improvements we collected a significantly larger number of WSDLs and related documents in Crawl 2 than in Crawl 1. In Crawl 3 and Crawl 4 we have improved the collection of related documents further. Between Crawl 2 and Crawl 3 we had included an improved WSDL deduplication methodology in the Service Crawler (e.g., discarding WSDLs that are produced by “buggy” server implementations, like PHP auto-generated WSDLs, as described in Section 3.1.2.2), which made that the number of WSDLs dropped significantly for Crawl 3 (it raised again in Crawl 4, which is partly due to the fact that the crawler needs to be continuously adopted to changes on the Web and thus to changed styles of duplicate WSDLs).

The fact that we found less services in total in Crawl 3 than in Crawl 2 is mainly due to the shorter run-time of the crawl (8 days for Crawl 3 compared to 15 days for Crawl 2). Crawl 4 again shows the further improvements in the Service Crawler since the Crawl 3 software: although the job was only running for 12 days, we have detected the largest number of services so far in one crawl iteration. With our approach to simultaneously start a second crawl job (Crawl 4B) that focuses on the domains of the detected services we also saw the the number of related documents grow further, as opposed to former crawls.
<table>
<thead>
<tr>
<th>Criteria</th>
<th>Value</th>
<th>Crawl 1</th>
<th>Crawl 2</th>
<th>Crawl 3</th>
<th>Crawl 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>size of relevant archives</td>
<td>MB</td>
<td>266.5</td>
<td>725.4</td>
<td>561</td>
<td>441</td>
</tr>
<tr>
<td>nr. of WSDL files</td>
<td>#</td>
<td>40.257</td>
<td>191.831</td>
<td>47.080</td>
<td>130.859</td>
</tr>
<tr>
<td>nr. of related documents</td>
<td>#</td>
<td>74.389</td>
<td>201.196</td>
<td>369.564</td>
<td>448.057</td>
</tr>
<tr>
<td>hosts with related content</td>
<td>#</td>
<td>8.435</td>
<td>8.028</td>
<td>11.640</td>
<td>12.669</td>
</tr>
<tr>
<td>nr. of providers</td>
<td>#</td>
<td>6.893</td>
<td>7.906</td>
<td>8.243</td>
<td>8.829</td>
</tr>
<tr>
<td>nr. of services</td>
<td>#</td>
<td>22.497</td>
<td>26.579</td>
<td>24.741</td>
<td>26.939</td>
</tr>
</tbody>
</table>

Table 5.4: Focused Crawl Performance Results: Crawl 1 - Crawl 4

5.1.2.3 Additional Metrics - Results

In the following we provide an overview of the results of some additional metrics evaluation approaches. They help us in analyzing the overall crawl resource usage and the resulting performance.

Resource Usage. For comparing the resource usage over different crawls we depict two graphs obtained by the Munin monitoring framework. We use Munin to identify performance related problems, by monitoring our network and resource usage, as well as our focused crawl related metrics. [19] provides information on how we integrated Munin in our server infrastructure and on how we adapted and now use it to monitor our crawling performance.

Figure 5.1 shows the network usage during the three crawls Crawl 1 - Crawl 3. Crawl 4 is based on two internal crawls that have been running simultaneously on two different machines; we omit showing these graphs here, as we have already provided the corresponding numbers in Table 5.3. We can see that the network usage varies mainly between 3 to 6 MBit per second, with short peaks over 10 MBit per second at the beginnings of the crawls. In general we can observe that for all crawls there is a strong increase in the initial period and an overall decline from the mid to the last phase of a crawl. To understand this behaviour we did some benchmarks on different machines (with more or less CPU power and memory) and with different crawl settings. Also we analyzed additional metrics like the number of downloaded URLs per second, the ratio between discovered and downloaded URLs, etc. (see [5,3], and discussed the issue on the open discussion group of Heritrix [2], the open-source crawler underlying the Service Crawler. We got to the conclusion that the behaviour is dependant on two criteria: (a) the RAM available to the crawler, as two large data structures - the collection of all pending-to-crawl URI queues and the set of already discovered URIs - are implemented and partly cached by the crawler, and (b) the crawl...
time, as according to the Heritrix developers any broad crawls tends to get slower over time as the record of already discovered URIs gets larger and thus checking whether newly discovered URIs need to be crawled becomes more disk I/O expensive.

Figure 5.2 shows the CPU usage during the three crawls (again we omit Crawl 4 here, which has been running simultaneously on two machines). The blue area indicates how much percentage of CPU time was used by the crawler process itself, the purple area indicates how much CPU time was lost due to I/O wait. The increase of I/O wait can be explained by the increasing amount of meta-data the crawler has to manage, e.g., which pages are visited or how far the various quotas have been reached. The spike at the end of the crawls in I/O wait time is due to a complete analyse of the log files performed by the crawler when terminating a crawl. The CPU usage stayed stable over the different crawl iterations that we performed.

WSDLs and Services. Figure 5.3 provides an overview of how many WSDL documents we have found per service per crawl. For Crawl 1 and Crawl 2 the bottom diagram has the Y-axis cut at 50 to allow to better see the distribution on the lower end of the scale. From the top diagram we see that a small number of services have several thousand of WSDLs. A closer examination revealed that some Web Service middle-ware generates endless URI sequences resulting all in similar WSDL documents. On the other hand nearly 90% of the services have less WSDL documents then the average indicates (1.8 WSDLs per service in Crawl 1 and 7.2 WSDLs per service in Crawl 2). As already mentioned in the explanatory text to Table 5.4 we have added a WSDL deduplicator between Crawl 2 and Crawl 3, which makes that in Crawl 3 the number of services with a large number of WSDLs has significantly decreased. In Crawl 4 the distribution of WSDLs per services is similar to Crawl 3, the main difference being that there are again a small number of services with a higher number of WSDLs in total.

Services and Related Documents. Figure 5.4 provides an overview of how many related documents we have found per service per crawl. The bottom diagrams cuts the Y-axis at 50 or 70, to better show the distribution on the lower end of the scale. From the top diagrams we can see that for some services we detected a lot of related documents, while for a large number of services we found only few documents (or even no documents, as not all services do have related documents at all). A big step forward has been reached with the large increase of crawled related documents in Crawl 2, after the integration of a better cost assignment policy (as described already above). At the same time we have in Crawl 2 some services that contain up to 12,000 documents; we have in the following adapted the crawler to not crawl too many documents related to one service, as we do not expect any more useful information in these pages (based upon manual checks of part of this particular cases). In Crawl 3 the max number of related documents

Figure 5.1: Network usage during (a) Crawl 1, (b) Crawl 2, and (c) Crawl 3
has thus decreased significantly again. Crawl 3 and Crawl 4 show also real improvements in the terms of collected related documents as the number of services that contain more than five related documents has increased to over 22,000/23,000 and the large number of related documents shows a significantly better distribution than in the previous crawls.

5.2 Service Ranking

The multi-valued ranking approach based on ontology-based feature aggregation, which we have presented in the scope of this thesis, is especially valuable in service search in an open, large scale environment, i.e., search over a large number of services whose degree of documentation is unknown in advance and where no agreements with the service providers have been made beforehand concerning the service documentation. The rank does not take into account specific user requirements, as the features we include in our rank calculation are of such nature that we assume they are always relevant for users (e.g., we assume that it is always important for a user to know whether a service is available or not) and are thus adequate for a generic ranking (as all list of services that results from a search request needs to be ranked in some way before being presented to a user). Although our final global rank does not take into account specific user requirements, this rank can be easily modified and composed in a different way by any RDF aware client as we return all single rank values as simple RDF triples (and also provide the underlying crawl meta-data that could be used to build a new rank value that has not been anticipated by us).

As our ranking approach is still very new, especially concerning the inclusion of related Web resources for services, we cannot yet provide a fully-fledged evaluation for it. Parts of the ranking approach have though been evaluated in the scope of the seekda Web Service search engine: the WSDL metrics and the monitoring meta-data (as described in the sections 4.1.2 and 4.1.3) are used as basis to rank the services as presented in Section 4.2. The search engine ranking has been evaluated by experts in the Web Service domain; they have manually assessed the relevance of the discovered services with different ranking aspects (such as availability or documentation) taken into account and different combinations of the weight of these aspects being applied. The approach as we describe it in this thesis, concerning the WSDL Metric Rank and the Monitoring Rank, is the result of this assessment and corresponds to the approach that was best evaluated by the participating experts.

Providing a complete evaluation of our ranking approach is part of potential future work. Approaches to evaluate the service ranking include the conduction of an empirical evaluation, employing techniques as, e.g., user interviews and questionnaires to assess the importance and relevance of our ranking value(s). Also the approach would be evaluated with regard to other ranking approaches (such as provided in [15], [21] or [22]).
Figure 5.2: CPU usage during (a) Crawl 1, (b) Crawl 2, and (c) Crawl 3
Figure 5.3: WSDLs/service: (a) Crawl 1, (b) Crawl 2, (c) Crawl 3, (d) Crawl 4
Figure 5.4: Docs/service: (a) Crawl 1, (b) Crawl 2, (c) Crawl 3, (d) Crawl 4
In the scope of this master thesis we have developed methodologies to enable the discovery of publicly available Web Services on large scale, as well as to use 'real-world' information about the services to provide a transparent, customizable ranking of the discovered services. In the following we summarize the contributions and achievements of this thesis and provide some insights to potential future work in the addressed area.

Service Crawling

We have started this document by providing a motivation for our work, i.e., explaining why we see the need for a new approach for Web Service discovery on large scale and why we want to use focused crawling techniques for realizing this goal. We introduced the Web Crawling, Service Discovery and Service Ranking domains to the reader, focusing on the crawling part, as this represents the main topic of this thesis. Hence we have presented the basic crawl steps and important crawling strategies and crawl types, and we have pointed out some issues that need to be respected while crawling the Web.

After describing this background to the thesis topic, we provided a detailed overview on the Web Service Crawling techniques that we use to focus our crawls for Web Services and related information. We explained how we collect seed URLs to start crawl jobs and what methods we use to identify Web Service descriptions and related information on the Web. We have presented our URL and queue scheduling approaches and have depicted the major extensions we have implemented to the Heritrix crawler to adapt it to our needs.

Crawling the Web results in large amounts of data. We have described in detail the raw crawl data that remains after a crawl has finished and have shown what analysis steps and further data assembling steps we apply to it. 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. Finally we have provided an overview of what other meta-data we extract from during and after a crawl iteration and have outlined the ontologies that we use to structure and store this information.

During the work on the Service Crawler described within this thesis, we have experimented with both extended crawls, where we crawled known domains in-depth, and broad crawls, where we tried to detect as many as possible new service descriptions. We have continuously adapted our Web Service Crawling techniques, especially focusing on the URL and queue scheduling strategies which, as our experiences so far show, seem to be highly relevant for (a) the success of the crawls in terms of amount of detected service descriptions and related information and (b) the performance of the crawler. We have outlined the different evaluation criteria that we apply to the Service Crawler and have presented the corresponding evaluation results.

Our work has been used and deployed in the scope of the seekda Web Service search engine, and the Service-Finder project Web Service portal. Figure 6.1 shows how the seekda search engine enables users to look for recently crawled services, while Figure 6.2 shows a service list resulting from a keyword search on the Service-Finder portal. For both deployments it becomes obvious that not only the technical service descriptions are returned as search results, but as well service meta-data, like, e.g., the provider, its location, service descriptions, user ratings, etc.

**Service Ranking**

We have described our approach for multi-valued ranking of Web Services, based on ontology-based feature aggregation. We therefore outlined what real-world data our ranking values rely upon, such as WSDL document metrics, monitoring information and related document statistics, and provided an overview on how the different rank values are calculated. Our ranking approach covers and combines thus many aspects of a service: its documentation, the existence of related information that is available on the Web and the liveliness data (which is a strong QoS criteria).

Moreover we have presented the seekda Ranking Ontology that we have developed for the structuring of the ranking values. The approach that we follow with our multi-valued ranking in general and specifically with the export of the ranking related data in a semantic format has some major advantages: The approach is very transparent, in that each RDF aware client can understand the rational of a ranking and can as well analyze the meta-data on which the ranking is based. This allows each RDF-aware client to perform, if desired, its own ranking calculation with the service meta-data that is provided.

Figure 6.3 shows how the ranking is applied in the scope of the seekda Web Service search engine. The search results are ranked by default according to the global rank; the user can modify the ranking of the search results by choosing to
see services sorted according to, e.g., their availability or their documentation.

**Future Work**

In the following we will give a short outline of major issues that can be tackled in the future to improve the Web Service crawling and analysis, as well as Web Service ranking.

**Deduplication of related documents.** We can improve the crawl output by filtering out duplicate Web documents. This is, same as for WSDLs (as described in Section 3.1.2.2), not only related to exact duplicates that can be detected by doing checksum or MD5 hash comparison, but mainly to the detection of similar documents.

**Related Information.** We could try out additional means to detect information related to services (e.g., by following deeper grades of inlink/outlinks from the link graph or by further enhancing the term vector similarity approach). Improving the gathering and analysis of related information is a good mean to improve the crawl process. The knowledge about where we find the most related
information can help focusing the crawler even more on finding information that is related to services.

**URL and queue scheduling.** We could further investigate and experiment the issue of URL and queue scheduling. This is important in that it can improve the performance of the crawler and can lead us to find still more service descriptions and more related information. On the way we could tackle some minor issues like better recognizing spam pages, directory listings, etc.

**Web API crawling.** In the scope of projects like SOA4All and Service-Detective we have started research in the area of identification of non-WSDL services on the Web, i.e., Web APIs, RESTful services, JASON services, etc. We are aware of the high importance of these services on the Web, and therefore this work, which has however not been part of this thesis, needs to be continued, further
improved and evaluated.

**Ranking evaluation.** The ranking approach should be evaluated by conducting an empirical evaluation, employing techniques as, e.g., user interviews and questionnaires to assess the importance and relevance of our ranking value(s). Also the approach should be evaluated with regard to other ranking approaches (such as provided in [15], [21] or [22]).

**Ranking criteria.** In order to further enhance our ranking approach we could enlarge the number of criteria that we take into account for the single ranking values.

**Community data.** Our ranking approach could be extended with the usage of community data. That is we could use statistical data coming from, e.g., the seekda Web Service search engine, to improve the ranking. Such data could include views of services, edits of service descriptions, try-outs of single services, etc. Also we can image to actively taking user feedback and requests into
account, i.e., providing the user with a possibility to select what features are most important to him.
In the following we will explain the installation of the Service Crawler and will describe the launch procedure and an example crawler configuration.

A.1 System Requirements

Due to the CPU, memory and disk space requirements of the Service Crawler, we recommend as minimal setting the usage of a server with the following components:

- **CPU** - AMD Athlon 64 X2 5600+ Dual Core (or similar)
- **Memory** - 4 GB RAM (a minimum of 256 MB RAM is indicated by Heritrix as being sufficient for crawls with hundreds of hosts. Clearly this is insufficient for larger crawls).
- **Disk** - 2 500GB SATA drives (for smaller crawls a single smaller disk is sufficient but for large crawls separate disks for state information and archive files is recommended)
- **Network** - Depending on the crawl settings a machine with resources as indicated above will be able to download between 2-15 Mbit/sec.

The Heritrix crawler, that is the open-source crawler upon which our Service Crawler is built, is implemented purely in Java. The same counts for all extensions that we made to Heritrix in the scope of our Service Crawler. This means that a real requirement for running the crawler is an installed JAVA JRE. The Java version to be used is Java SE 6.

With regard to the operating system that the crawler runs on, we use a Linux Debian distribution.

\[1http://crawler.archive.org/\]
A.2 Installation and Launch

The Service Crawler release contains four directories and three files:

- **bin** directory - contains the Heritrix start script
- **conf** directory - contains some default configuration files needed by Heritrix
- **jobs** directory - contains (a) two default Heritrix job profiles, `profile-basic_seed_sites` and `profile-broad_but_shallow`, and (b) two specific seekda job profiles, namely `profile-seekda-very-broad-crawl` and `profile-seekda-rest-and-rdf-broad-crawl`
- **lib** directory - contains all the libraries that are needed to run the Service Crawler and the Heritrix Web User Interface
- **HOWTO-Launch-ServiceCrawler.txt** - a short description for how to launch the Service Crawler and where to find more detailed help
- **ServiceCrawler-demo-license.txt** - the seekda demo-license for the Service Crawler binaries
- **log4j.properties** - the configuration file for log4j

The libraries that we use within the Service Crawler project contain the Heritrix libraries, three libraries concerning our extensions, some third-party libraries delivered with Heritrix and some third-party libraries needed by the Service Crawler extensions. The libraries are outlined in the following. Together with the libraries themselves, the Service Crawler release contains the license agreements for the different libraries.

The Heritrix libraries:

- `engine-2.0.0.jar`
- `modules-2.0.0.jar`
- `commons-2.0.0-fixed-nullpointerinarcreader.jar`
- `webui-2.0.0.war`

The Heritrix third-party libraries:

- `ant-1.6.5.jar`
- `archive-overlay-commons-httpclient-3.1.jar`
- `archive-overlay-commons-pool-1.3.jar`
- `bsh-2.0b4.jar`
- `cmdline-jmxclient-0.10.5.jar`
• commons-cli-1.0.jar
• commons-codec-1.3.jar
• commons-collections-3.1.jar
• commons-el-1.0.jar
• commons-io-1.3.1.jar
• commons-lang-2.3.jar
• commons-logging-1.0.3.jar
• commons-net-1.4.1.jar
• dnsjava-2.0.3.jar
• fastutil-5.0.7.jar
• itext-1.3.jar
• jasper-compiler-5.5.15.jar
• jasper-runtime-5.5.15.jar
• javaswf-CVS-SNAPSHOT-1.jar
• je-3.2.44.jar
• jericho-html-2.3.jar
• jets3t-0.5.0.jar
• jetty-6.0.2.jar
• jetty-util-6.0.2.jar
• jsp-api-2.0.jar
• libidn-0.6.5.jar
• mg4j-1.0.1.jar
• oro-2.0.8.jar
• poi-2.5.1-final-20040804.jar
• poi-scratchpad-2.5.1-final-20040804.jar
• servlet-4.1.34.jar
• servlet-api-2.4.jar
The Service Crawler extension libraries:

- crawler.jar
- RegisterWebService-R5551.jar
- rdf.jar

The Service Crawler third-party libraries:

- axiom-api-1.2.7.jar
- axiom-impl-1.2.7.jar
- axis2-adb-1.4.1.jar
- axis2-kernel-1.4.1.jar
- backport-util-concurrent-3.1.jar
- collections-generic-4.01.jar
- colt-1.2.0.jar
- concurrent-1.3.4.jar
- document_comparation.jar
- junit-3.8.2.jar
- juniversalchardet-1.0.jar
- log4j-1.2.8.jar
- filterbuilder.jar
- htmllexer.jar
- htmlparser.jar
- jaxen-1.1-beta-7.jar
- junit.jar
- jyaml-1.3.jar
- neethi-2.0.4.jar
- opennlp-tools-1.4.3.jar
- openrdf-sesame-2.2.1-onejar.jar
- rabin-hash-function-2.0.jar
- rapidminer-text-4.2.jar
The installation and launch procedures for the Service Crawler are generally the same than for Heritrix. The process as it is described in the following applies for running the Service Crawler on Linux only.

First of all the classpath needs to be set to the path where the Service Crawler package was put to:

```bash
export HERITRIX_HOME=/PATH/TO/THE/HERITRIX_LIBRARIES_AND_BIN
```

Then you can start the crawler from the Linux bash console (please note that you need to set the Heritrix start script as executable file before). The crawler with the Heritrix Web User Interface can then be started as shown in Listing A.1:

```bash
cd \$HERITRIX_HOME
chmod u+x \$HERITRIX_HOME/bin/heritrix
\$HERITRIX_HOME/bin/heritrix -b IP -a PASSWORD
```

Listing A.1: Service Crawler launch procedure

The whole launching process is described in detail on the Heritrix installation Web page [http://crawler.archive.org/articles/user_manual/install.html](http://crawler.archive.org/articles/user_manual/install.html) or in the HOWTO-Launch-Heritrix.txt file within the lib folder. Please refer to one of those sources for more help or information on launching the crawler.

A.3 Configuration

Together with two default Heritrix job profiles, we deliver two specific Service-Finder job profiles, namely `profile-seekda-very-broad-crawl` and `profile-seekda-rest-and-rdf-broad-crawl`. The job profiles contain a directory called `sheets`. Within this directory we find the main Heritrix configuration file, namely `global.sheet`.

The following Listing A.2 provides the configuration file for a broad crawl, that is a crawl that is used to detect as many as possible new WSDL files on the Web.
root = map, java.lang.Object
root: metadata = primary, org.archive.modules.writer.
  DefaultMetadataProvider
root: metadata: description = string, Broad domain crawl
root: metadata: robots = honoring - policy = primary, org.archive.modules.net.RobotsHonoringPolicy
root: metadata: robots = honoring - policy: user = agents = list, java.lang.String
root: loggerModule = primary, org.archive.crawler.framework.
  CrawlerLoggerModule
root: seeds = primary, org.archive.modules.seeds.SeedModuleImp
root: seeds: seedsfile = file, /usr/local/tarantula/seeds/seeds20080905
root: scope = object, org.archive.modules.deciderrules.
  DecideRuleSequence
root: scope: rules = list, org.archive.modules.deciderrules.DecideRule
    AcceptDecideRule
    PrerequisiteAcceptDecideRule
    MatchesRegExpDecideRule
    AcceptDecideRule
    DecideResult = REJECT
root: scopes: rules: 3: comment = string, no useless content
    DecideResult = REJECT
root: scope: rules: 3: regexp = pattern, .* (gif | GIF | jpg | JPG | jpeg | JPEG | tif | TIF | tiff | TIFF | png | PNG | bmp | BMP | gif | gzip | gnzip | mp3 | wav | wma | mpeg | avi | divx | mov | mpeg | xvi | xis | xis | cs | swf)
root: uriUniqFilter = primary, org.archive.crawler.util.BloomUriUniqFilter
  TopmostAssignedSurtQueueAssignmentPolicy
root: server = cache = primary, org.archive.modules.net.BdbServerCache
  CredentialStore
root: controller = primary, org.archive.crawler.framework.
  CrawlControllerImpl
  BdbFrontier
root: controller: frontier: balance = replenish - amount = int, 500
root: controller: frontier: max - retries = int, 3
root: controller: frontier: preference = embed - hops = int, 0
  TopmostAssignedSurtQueueAssignmentPolicy
  HighestUriQueuePrecedencePolicy
  canonicalize.CannotalizationRule
  canonicalize.LowercaseRule
  canonicalize.StripUserinfoRule

82
canonicalize.StripWWWNRule
canonicalize.StripSessionIDs
canonicalize.StripSessionCFIDs
baserule.RemoveDirectorySortings
canonicalize.FixupQueryStr
root:controller:maxtoethreads=int, 200
root:controller:processors=map, org.archive.modules.Processor
prefetch.Presselector
root:controller:processors:Preprocessor=object, org.archive.crawler
.prefetch.PreconditionEnforcer
heritrix.QuotaEnforcerFIX
root:controller:processors:QuotaEnforcerFIX:group=max=all-kb=long,
51200
FetchDNS
.FetchHTTP
String
modules.deciderules.DecideRuleSequence
root:controller:processors:HTTP:decide-rules:comment=string, only
fetch documents with http codes 200, 301, 302
archive.modules.deciderules.DecideRule
archive.modules.deciderules.AcceptDecideRule
seekda.tarantula.deciderule.HTTPStatusDecideRuleTarantula
reject all docs with HTTP status codes except 200, 301, 302
org.archive.modules.deciderules.DecideResult–REJECT
root:controller:processors:HTTP:max-length–bytes=long, 2000000
modules.deciderules.DecideRuleSequence
archive.modules.deciderules.DecideRule
heritrix.ContentLengthDecideRuleFIX
string, allowing only 2MB max
heritrix.ContentTypeMatchesRegExpDecideRuleFIX
string, no images, videos, ms office files or zips
root: controller: processors: PDF2HTMLConverter=object, com.seekda.tarantula.processor.PDF2HTMLConverter
root: controller: processors: FileBasedUrlGraph=object, com.seekda.tarantula.processor.FileBasedUrlGraph
root: controller: processors: WSDLWriter=object, com.seekda.tarantula.processor.WSDLWriter
root: controller: processors: LinksScoper=object, org.archive.crawler.postprocessor.LinksScoper
root: controller: processors: FrontierScheduler
root: controller: processors: EmptyDataMap=object, com.seekda.tarantula.processor.EmptyCuriDataMap
root: controller: processors: LowDiskPauseProcessor=object, org.archive.crawler.postprocessor.LowDiskPauseProcessor

84
Listing A.2: Service Crawler broad crawl configuration file

Within the Heritrix Web User Interface this profile will show up and can be easily copied to a new job. As such the job configuration is ready-to-use, the only missing thing are the seed URLs. Within this fact-sheet we do not explain our approaches to gather seeds, as these have already been described in Section 3.1.2.1 To be able to start any crawl you will need to either upload a new seeds.txt file to Heritrix or you simply edit the file: root:seeds:seedsfile=file, /usr/local/tarantula/jobs/seeds/seeds.txt

For more detailed explanations on the job processors and decide rules, please refer to Chapter 3

A.4 Crawling Output

The output of the crawls, that is the link graphs, WSDL archives and general document archives will be stored on disk during the crawl. Within the configuration sheet the user can determine the disk location where to put the crawled data, as, e.g., in the above configuration:


After termination of a job the link graphs can be found in the job’s subdirectory links, the WSDL archives and indexes can be found in the job’s subdirectory wsdl and the location of the data archives can be chosen, but corresponds to job_arcs in the above example.

The reports and log files of Heritrix and the Service Crawler give indications on the behavior of the crawl. The logs can be found in the job’s subdirectory logs, while the reports are directly written into the main job directory.
Bibliography


