UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Information Technology

Taniel Põld

Heuristics for Crawling WSDL Descriptions of Web Service Interfaces - the Heritrix Case

Bachelor's thesis (6 EAP)

Supervisor: Peep Küngas, PhD
Supervisor: Meelis Kull, PhD

Author: …… "….." May 2012
Supervisor: …… "….." May 2012
Supervisor: …… "….." May 2012

Admitted to thesis defense
Professor: …… "….." ……2012

Tartu 2012

Contents

1 Introduction

Data found on the Internet can be divided into two varieties: surface web also known as visible or indexed web, and deep web also known as invisible web. Visible web is data on the Internet that has been crawled and indexed by general-purpose search engines whereas deep web can be defined as a portion of Internet that is not part of the visible web. Invisible web consists mainly of pages that do not exist until they are created dynamically or of data that is accessible only via web service method calls. Data behind these web services is stored in searchable databases that only produce results in response to a request. A paper [4] published in 2001 estimates that public information stored in deep web is 400 to 550 times larger than the commonly defined world wide web.

Current size of surface web is estimated to be at least 8 billion pages [9]. Although the data structures in deep web are not directly comparable with general web page definitions, it still gives us a sense of the amount of data that can be found there.

One of the widely used software development paradigms known as Service-oriented Architecture (SOA) - introduced in the second half of previous decade - has given a boost to the number of web services found on the Web. SOA promotes the idea that independent systems and applications should communicate with each other by exposing and using services [8]. Services used often reside on the Internet and on some occasions contain public API access points for third party software. The article [10] describes the theoretical SOA triangle that was meant for publishing public web services. According to that, the service provider would register its service to a registry that would act as a database. A service consumer would search the service registry for a suitable service and if it is found, would start using it.

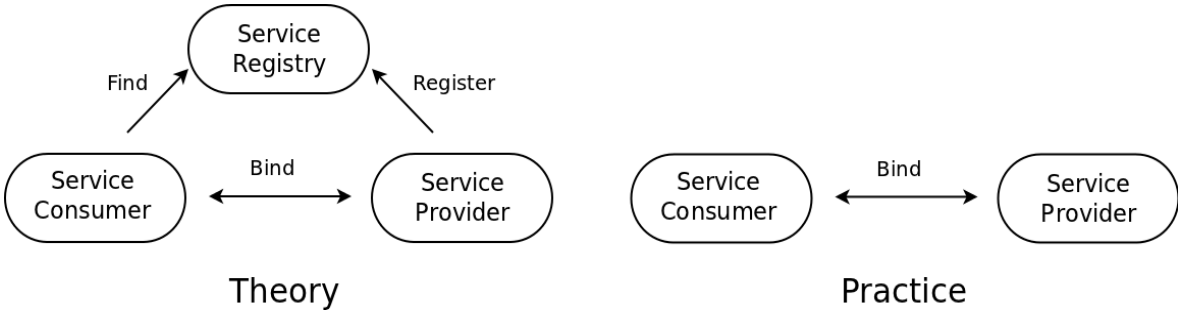

Figure 1. SOA triangle in theory and practice from article [10].

In practice this approach did not work as intended. The model used in most of today's SOA applications consists of only the service consumer and service provider. Service providers do not register their services and therefore consumers cannot rely on the data provided by the service registry.

Considering the estimations we saw earlier, it can safely be assumed that there are at least millions of public web services on the Internet with their descriptions scattered all over the web. Searching for a specific one may prove to be a difficult task as the general-purpose search engines focus on surface web and service registries are often outdated. A possible solution for this could be a global repository or a search engine that would update its database automatically using web crawlers. Crawlers, also known as spiders, are programs designed for conducting large-scale automated search on the web to find desired resources. The process of searching the web for resources is called web crawling or spidering.

In this thesis we are tackling the problem of configuring and modifying existing web crawler to automate the process of finding web services. For the spider, we have chosen an open source web crawler, Heritrix, written in Java programming language. Heritrix has been designed to help Internet Archive [7] store the contents of Internet. It already includes most of the common heuristics used for spidering and it has a modular architecture design which makes it easy to alter.

To fit this task into the scope of a bachelor thesis it was necessary to narrow the scope from finding all the web services to finding services described in Web Services Description Language (WSDL) format. WSDL is a XML based language designed to describe functionality offered by web service. Descriptions contain access point data, information about operations available, and definition of requests and responses. WSDL based services often use Simple Object Access Protocol (SOAP) for communication and are widely used.

The crawler we are altering will not aim to find other web service standards besides WSDL. So the newer Web API standard with representational state transfer (REST) based communication will be excluded from the scope of our work.

2 Background and existing solutions

Starting with the popularization of SOA in the mid-2000s an increasing amount of public web services has come to existence. With service registries containing only some service descriptions, some of which are outdated, several research teams have started to look for a solution. Some of them concentrated on crawling WSDL descriptions from different service repositories to create a large database. Others have used spiders to conduct large scale searches on the Internet.

In 2008 Al-Masri and Mahmoud published a paper *Investigating Web Services on the World Wide Web* [2], which introduced the Web Service Crawler Engine (WSCE). It is a spider that crawls web service repositories and search engines to collect business and web service information. WSCE itself was described in an article a year earlier [3]. The purpose of their solution was to eliminate problems that the centralized service repositories were suffering from, such as single point of failure, bottlenecks and outdated data. To achieve this, they automatically collected web services from other repositories, search engines and service portals to form a new portal. Descriptions of web services that were not entered to repositories and not available in search engines, remained out of reach for WSCE.

In 2010 Duan Dehua [5] defended his master thesis on the topic of *Deep Web Services Crawler* in the Technical University of Dresden. The goal of the thesis was to create the largest annotated service catalogue ever produced. To achieve that, a web crawler was created. It was based on the Pica-Pica web service description crawler that had also been created in the Technical University of Dresden by Anton Caceres and Josef Spillner [14]. Both of these crawlers were designed to search for web services in existing web service repositories and to validate and annotate services found. Crawlers took web service repositories' URLs as seeds and started looking for WSDL services inside them. If a service was found, the crawler would validate it - determined if it was valid or invalid. This check would include testing namespace URI and WSDL definitions. After a web service had been validated, the crawler would try to gather descriptive data for the service. The extracting of service related data was done by implementing algorithms explicit to each registry. This approach created a tight coupling between the existing web service repositories and the crawler, creating the same problem as in WSCE where WSDLs not present in repositories may have been excluded from the crawl.

In 2009 a research team from Innsbruck University in Austria gave a presentation on the topic, titled *Web Service Search on Large Scale* [15]. It concentrated on methods for automated web service crawl and the creation of semantical annotations for found services. The crawler they had created was built on top of the Heritrix spider which is also used in this thesis. In addition to the WSDL crawling strategies featured here, they also focused on finding Web API services. Spidering for WSDLs was done by using the following heuristics: at first they narrowed the crawler's scope down to HTML, XML and text file resources which means that the crawler searches for services from only these types of data, excluding JavaScript, CSS, SWF etc. The second strategy was focusing the crawl to desired resources. It was done by first crawling web sites that were more likely to contain WSDLs. This, in turn, was achieved by assigning cost value to each of the found URIs and sorting the worker queues so that URIs with the lowest cost would be crawled first.

The name for this project was Service-Finder and it was developed as a joint venture with Seekda, a company who owns a search engine for web services [12]. Because of this, the resulting software was of proprietary nature and the source code was not published. Methodology used in the Service-Finder was described in one of the project's deliverables that has been made public [16]. A detailed overview of implementing some of the strategies from there is given in the next chapter.

In 2010 AbuJarour, Naumann and Craculeac [1] released an article on the topic of web service discovery. The paper is trying to increase the usability of public web services. It does so by collecting them automatically from their providers' websites with the help of a web crawler. Semantical information is deduced from crawled web pages and from this the application - created by the research team - creates annotations for each service. These annotations are then used to classify each web service into different application domains.

The crawler used in this project was Heritrix configured to crawl for WSDL web services. The heuristics used were the following:

- Narrowing the crawl scope to HTML and XML resources, similar to the Innsbruck University's Service-Finder project we saw earlier.
- Use of crawler trap avoidance mechanisms already implemented in Heritrix. A crawler or a spider trap is a set of web pages that may be used to cause the web crawler to make an infinite number of requests to slow down the crawl's progress.

- Regular expression rule that verified if the element found is WSDL or not.

Heritrix supports all of these methods and they can be implemented through configuration. Unlike the Heritrix based crawler mentioned earlier, the one described in this paper is missing support for focused crawling and does not offer any improvements for WSDL discovery besides the configuration of features already present.

3 Solution

3.1 Heritrix overview

Heritrix is an open-source web crawler that has been written in Java programming language. It was developed jointly by Internet Archive and the Nordic national libraries. The first official release was made in 2004 and it has been developed by employees of the Internet Archive and other interested parties ever since [19]. The main purpose of use for it has been archiving the web in large scale. The license used is GNU Lesser General Public License (LGPL) [18] that allows Heritrix to be used by non LGPL licensed software as it was done in the previously mentioned Service-Finder project.

Source code for Heritrix can be found in a publicly accessible repository hosted in *github.com* [6]. New improvements and bug fixes are being committed there with the current approximate rate of one submission per week. The version discussed in this thesis is Heritrix 3.1.1.

Upon default start up, Heritrix will start a web server bound to your local loopback address. This web server will act as a graphical user interface that can be accessed by using a web browser. From there it is possible to create and edit crawler job files. Use of web server as GUI is not mandatory, but it gives the user a better overview of the crawl process and provides features for creating and editing new crawl jobs.

Each crawler job configuration is saved in a XML format file with *cxml* as file extension type. In job configuration files, users can define bindings for the specific modules with proper parameters that the crawler should use. These job files are in fact application context specifications for the Java Spring framework embedded in Heritrix. During the initiation of the job, modules defined in configuration are being built and coupled together following the Inversion of Control practice from Object Oriented Programming.

During the crawl, the administrative console running on the web server will provide the user with up to date data of current progress. From there it is possible to see the expansion of the crawl frontier or in other words the number of URIs discovered, but waiting to be crawled. Other characteristics that may be of interest include the amount of already

crawled sites, average URI crawl time, average overall download speed, and duration of the crawl.

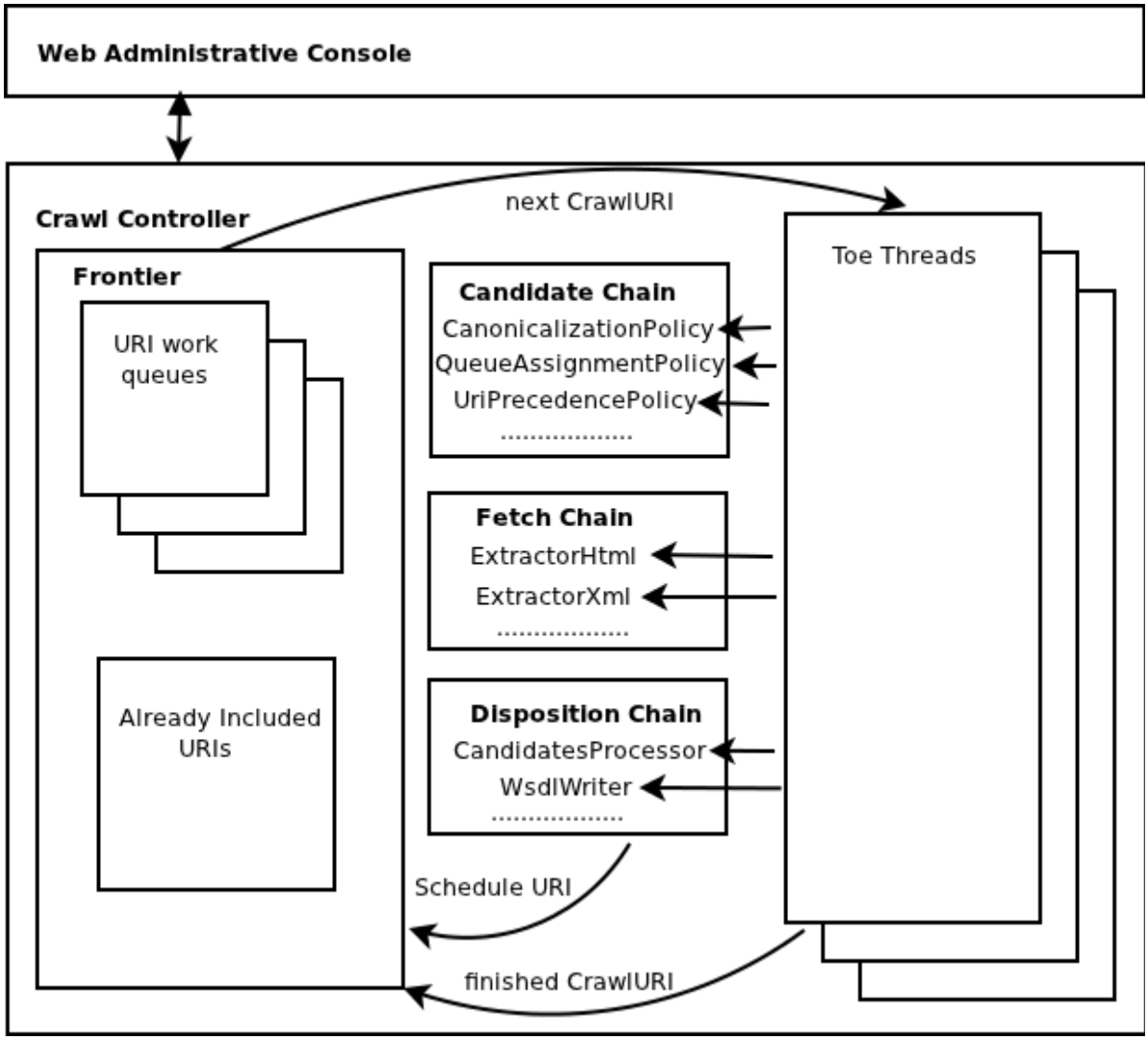

Figure 2. Architectural overview of Heritrix from article [11].

General architecture overview of Heritrix crawl engine can be found in Figure 2. This is not an UML standard diagram, but rather an illustrative scheme showing interactions between different components. The figure was present in article [11] giving insight to Heritrix's structure, written in 2004 and has been altered to suit version 3.1.1.

The Crawl Controller surrounding the components represents Java class *CrawlController* that holds global context for the crawl. All subcomponents can reach each other through it. GUI as the Web Administrative Console in this figure controls crawl process through Crawl Controller.

At the start of the crawl, the frontier will load URIs given as the crawl's starting point.

Frontier is the term used to represent all the URIs that have been found but not yet crawled. In Figure 2, frontier corresponds to a Java class with the same name that is accountable of URIs crawled and URIs to be crawled. The latter are stored in multiple work queues.

The crawl itself is conducted in multiple threads allowing exploration of more than one site simultaneously. Because web crawlers are also known as spiders, worker threads processing URI's data are named Toe Threads after a spider's 8 limbs. During the crawl each Toe Thread will be handling one URI at a time.

Worker thread's tasks for one URI are divided into three processing chains, that are being run sequentially. Each of these chains contains processors configured for the crawl in progress. Processors are Java classes designed as modules holding business logic that can be included in the job's configuration file.

Modules included in Candidate Chain are applied before a URI is enqueued for collection. These include processors determining whether or not URI in question fits the crawl's scope, spider trap avoidance mechanisms, and cost assignment policies; if URIs are wished to be sorted in worker queues so that most prominent elements would be placed up front.

In Fetch Chain the URI is downloaded and examined to find and process data of interest. A processor can be added for each supported data source: e.g. HTML processor, PDF files processor, XML processor and so forth.

Disposition Chain saves the data that has been found. Processors in this chain are applied after a URI is fetched, analyzed, and link-extracted.

Because all of the URIs in the frontier would not fit into one worker queue, they are divided between multiple ones. Usually, most of the URIs gathered from one site also reside in one queue, but depending on the configuration there may be exceptions. URIs to be processed are taken from one queue at a time and if a configurable number of URIs have been taken from one queue, the queues are rotated. Queue rotation means that Toe Threads will start taking the URIs to be crawled from the next worker queue. Heritrix has queue precedence policy classes that are responsible of queue rotation algorithm.

Most of the heuristics used for general-purpose spidering have already been implemented in Heritrix. For example, there exists a module called *FrontierPreparer* - a processor that includes crawler trap avoidance techniques like eluding long URLs, and defining the

maximum depth of a crawl within one site to prevent recursive links. If new features are to be added, then it can be done easily by creating new processor modules.

3.2 Heritrix for WSDL Interface Descriptions

Although Heritrix supports general-purpose web crawling, it is not specifically designed for finding WSDL files.

The first problem we encountered when trying to crawl WSDL URIs with Heritrix, was related to the storing of discovered data. Heritrix 3.1.1 has four different processors for formatting and writing results to the hard drive. All of these will store string based contents of entire web sites by using different format and directory structure. During large scale spidering they will take up enormous amounts of disk space. Because the scope of this thesis was to gather WSDL URLs without arbitrary data, it was necessary to create a new processor. Java class name for this created class became *WsdlWriterProcessor* and all the Toe Threads will add their findings through that into one text file.

The next task was to configure a job for searching WSDL files. AbuJarour, Naumann and Craculeac who had run WSDL search experiments with Heritrix in Potsdam 2010 listed three heuristics that they had used in an article [1]. Job configuration notions taken from there were following:

- Enabling of *PathologicalPath* module in crawler jobs. It will reject URI if path-segment pattern is repeated more than two times. Designed for crawler trap avoidance. e. g. *www.example.com/foo/foo/foo* is rejected.

- Regular expression rule in Candidate Chain's processor that will discard all resources that are not relevant to our crawling task from the scope. This rule was set to include only XML and HTML pages to our crawl.

- Regular expression rule for *WsdlWriterProcessor* that will only accept URLs ending with a case insensitive *wsdl* string.

We had to write regular expressions for the processors ourselves.

As mentioned before, the team working on the Service-Finder project released deliverable [16] providing detailed description of their web service crawling techniques, how they

identify services on the web and an overview of their URL optimization and queue scheduling strategy. As pointed out earlier, their work was not open sourced so no source code of their implementation was released, but guidelines given in the deliverable [16] were the most useful.

As in the article [1], the members of the Service-Finder project team also emphasized the importance of narrowing the spider's scope by limiting the resources from where to look. Unlike regular expression rule defined in the previously mentioned paper, they also included all document file types like PDF, DOC etc. The idea behind it was that service descriptions may be presented as text files. Support for extracting data from PDF and DOC file types has been implemented by the Heritrix's development team so we included this to our configuration. Additional scoping rule provided in Service-Finder project's paper that we used, was following:

- Setting the limit of the maximum number of bytes to download from one document to 2MB. This prevents us from downloading too large documents; neither WSDL service descriptions nor normal Web pages are usually very large documents.

Precise values for the rules above were found from Service-Finder project's deliverable [17].

The most complex spidering strategy introduced in the article [16] was URL and queue scheduling for focusing the crawl to WSDLs and resources related with web services. It was included as cost assignment policy in Heritrix, which had been implemented by creating a new processor module. The idea was to focus the crawl on resources that have more potential for finding WSDLs, i.e. crawling URLs that may contain WSDLs first. To do so, new URIs discovered during the spidering were assigned a cost value to represent the crawler's level of interest in them. The lower the value the better. URI work queues in Heritrix are prioritized so that the URIs with low cost value will be placed up front and will be assigned to Toe Threads before URIs with high cost.

URI's evaluation algorithm described in article [16] was following: the default URI cost value is 20. Penalties are added if URI has a lot of subdomains, has more than one query string, or has more than one path segment. Cost is decreased if URI contains the following strings *?wsdl*, */ws*, */service* or *api*. During the extraction of a web page the content is being semantically analyzed to determine whether the site contains information related to web

services. Based on this data, additional cost reward is added to the outgoing links if the site in question is web service related. The algorithm presumes that sites containing service related information link to other sites accommodating web services.

In this thesis we implemented a similar cost assignment method as the one described above. For this purpose we created a Java class called *WsdlCostAssignmentPolicy* that contains the algorithm's business logic. This class extends abstract *CostAssignmentPolicy* class from Heritrix's engine module and overrides methods from there. Because W*sdlCostAssignmentPolicy* descends from *CostAssignmentPolicy* it can easily be bound with Frontier Preparer in crawler job configuration.

The algorithm will start by adding the default value of 20 to the URI. It will continue by looking trough several cost element conditions to see if there is a match. In case of a match, the element's cost value is added to the URI's cost. Cost element values and condition descriptions are represented in Figure 3.

Cost Element Identifier	**Value**	**Cost Element Condition Description**
costElement_0	20	URI's default cost value.
costElement_1	x*1	Penalty for every subdomain different than *www*. x = count of subdomains. e.g. *ws.example.com* would have a penalty of 1, *test.ws.example.com* would have a penalty of 2 etc.
costElement_2	x*1	Penalty for more than one path segment. x = number of path segments minus one. e.g. *example.com/foo/bar* penalty is 1, *example.com/foo/bar/segm* penalty is 2 etc.
costElement_3	x*1	Penalty for more than one query attribute. x = number of attributes minus one. e.g. *?a=b* no penalty, *?a=b&c=d* penalty is 1 etc.
costElement_4	x*3	Penalty for recurring elements in URI. x = number of recurring elements. Domain elements and path segments are compared. e.g. *example.com/foo/foo/foo* has 2 recurring elements.
costElement_5	-5	Reward for URI ending with *?wsdl*.
costElement_6	-2	Reward for URI containing one of the following strings: */webservice*, */service*, *api*, */ws*
costElement_7	N/A	This value is initialized after cost calculations. Holds the value for URI's content reward which is used in the next cost element.
costElement_8	-x*2	Reward if parent URI's content contains following keywords: *wsdl*, *web service*, *soap*. x = number of keyword occurrences with max value of 5.

Figure 3. URI cost calculation values and condition descriptions.

Most of the cost element conditions and their values in Figure 3 have been taken from Service-Finder project's descriptions. The cost element not taken from there is the *penalty for recurring elements*. During the crawling experiments we encountered multiple identical

WSDL descriptions generated recursively by the same web page where URI contained recurring elements. This rule was introduced to avoid depth-first crawling on these web pages. Reward for keywords found in parent URI's content was a simplified take on the complex semantical web page analysis implemented in Service-Finder project.

The *WsdlWriterProcess* we have implemented writes down cost elements of each WSDL URI that has been found. With this data it is possible to conduct an analysis for finding correlations between cost element values and WSDL URIs. This kind of analysis could provide information for calibrating weight multipliers for the cost elements. First column in Figure 3 contains identifiers for cost elements that are used in the URI logging process.

To allow queue scheduling so that worker queues containing URIs with the lowest cost would be processed first. We enable *HighestUriQueuePrecedencePolicy* class as queue precedence policy in our job configuration. This policy will set the queue's precedence value to the lowest cost that the URIs within this queue contain. So during the worker queue rotation, the queue containing the URI that has the lowest cost will be processed next. The aforementioned policy has already been implemented in Heritrix.

Heritrix's sourcecode with the changes we made can be found in the appendix section. In there is also an installation guide for the improved crawler.

4 Experiments

To see if the focused crawl strategy centered on WSDL descriptions - described in the previous chapter - introduced any increase in the crawler's speed or accuracy, we conducted an experiment where we compared crawl results of two Heritrix instances. One of the instances was running a configuration for finding WSDL URIs without the focused crawl improvements, and the other instance had the same settings with focused crawling being the only addition. The experiment was repeated three times to ensure accurate results.

4.1 Input Data

For bootstrapping crawling, Heritrix needs seed URLs. During the crawl's initiation, these seeds are placed in the worker queues and they will be the starting frontier. For the seeds in our experiments, supervisor Peep Küngas provided approximately 60 000 WSDL URLs. From these more than half were offline and proved no value as WSDLs descriptions, but were suitable as a starting point for Heritrix. Because even if the WSDL description has been removed, there remains a possibility of finding it from another location on the same site or the site could contain other web service descriptions. All three experiments described in here used the same collection of seed URLs. The seed URLs used are the property of *soatrader.com* [13] and were provided for research in this thesis only.

4.2 Crawling

To support the discovery of WSDL descriptions and to improve the crawler's speed, we had to make changes in the default job configuration files Heritrix had created. For this reason we made the following modifications:

- *TextSeedModule* was employed to include reading of the seeds from external text file.

```
<bean id="seeds" class="org.archive.modules.seeds.TextSeedModule">
    <property name="textSource">
        <bean class="org.archive.spring.ConfigFile">
            <property name="path" value="seeds.txt" />
        </bean>
    </property>
    <property name='sourceTagSeeds' value='false'/>
    <property name='blockAwaitingSeedLines' value='-1'/>
</bean>
```

Figure 4. TextSeedModule configuration.

- To support writing of the WSDL URLs found during the spidering we had to enable *WsdlWriterProcessor* and *WsdlFileWriter*. Both of these classes have been created for this thesis.

  To write only WSDL resources we add a regular expression rule to our writer processor that will only accept URIs ending with *?wsdl* or *wsdl*. This rule is case insensitive.

```
<bean id="wsdlWriter" class="org.archive.modules.writer.WsdlWriterProcessor">
    <property name="shouldProcessRule">
        <bean class="org.archive.modules.deciderules.MatchesRegexDecideRule">
            <property name="decision" value="ACCEPT" />
            <property name="regex" value=".*(?i)wsdl$*"/>
        </bean>
    </property>
</bean>

<bean id="wsdlFileWriter" class="org.archive.modules.writer.WsdlFileWriter" scope="singleton">
    <property name="fileName" value="wsdls.txt" />
    <property name="directory" value=""/>
</bean>
```

Figure 5. WsdlWriterProcessor configuration.

- To exclude the resources that we are not interested in from the crawl's scope, we introduce a regular expression rule.

```
<bean class="org.archive.modules.deciderules.MatchesListRegexDecideRule">
    <property name="decision" value="REJECT"/>
    <property name="listLogicalOr" value="true" />
    <property name="regexList">
        <!--We are only intrested in HTML, XML and document data types.
        Exclude all others-->
        <list>
            <value>.*(gif|GIF|jpg|JPG|jpeg|JPEG|tif|TIF|tiff|TIFF|png|PNG|bmp|BMP)</value>
            <value>.*(mp3|wav|wma|mpeg|avi|dvix|mov|mp4|wmv|xls|ppt|css|swf)</value>
            <value>.*(js|JS|css|CSS|rss|RSS)</value>
            <value>.*(zip|gzip|tar|tar.gz|rar)</value>
        </list>
    </property>
</bean>
```

Figure 6. Rule to narrow the crawl's scope.

- To improve the crawl's speed, several modifications were made. In *FetchHTTP* class that is responsible for downloading resources we reduced the timeout length and set the maximum bytes to be downloaded per file to 2MB. The number of *ToeThreads* used was increased to 200. In frontier we decreased *snoozeLongMs*, *retryDelaySeconds* and *maxRetries* values to reduce the amount of time spent on crawling one URI.

For the job that includes improvements for focused crawling we reduced *balanceReplenishAmount* from 3000 to 500. This ensures replacing the active worker queue after 500 URIs have been processed from there. We also enabled *HighestUriQueuePrecedencePolicy* which was explained in chapter 3.

```
<bean id="fetchHttp" class="org.archive.modules.fetcher.FetchHTTP">
    <property name="maxLengthBytes" value="2097152" />
    <property name="soTimeoutMs" value="3000" />
    ...

<bean id="crawlController" class="org.archive.crawler.framework.CrawlController">
    <property name="maxToeThreads" value="200" />
    ...
<bean id="frontier" class="org.archive.crawler.frontier.BdbFrontier">
    <property name="snoozeLongMs" value="300000" />
    <property name="retryDelaySeconds" value="120" />
    <property name="maxRetries" value="5" />
    <!-- following values are set for focused WSDL crawling -->
    <property name="balanceReplenishAmount" value="500" />
    <property name="queuePrecedencePolicy">
        <bean class="org.archive.crawler.frontier.precedence.HighestUriQueuePrecedencePolicy"/>
    </property>
    ...
```

Figure 7. Performance improvements.

- *WsdlCostAssignmentPolicy* class that we created, was added to one of the jobs to enable focused crawling of WSDL descriptions.

```
<bean id="preparer" class="org.archive.crawler.prefetch.FrontierPreparer">
    <property name="costAssignmentPolicy">
        <ref bean="costAssignmentPolicy" />
    </property>
    ...

<!-- COST ASSIGNMENT POLICY -->
<bean id="costAssignmentPolicy" class="org.archive.crawler.frontier.WsdlCostAssignmentPolicy">
    <property name="defaultCost" value="20" />
    <property name="subDomainPenaltyWeight" value="1" />
    <property name="queryStringPenaltyWeight" value="1" />
    <property name="pathSegmentPenaltyWeight" value="1" />
    <property name="recurringElementsPenaltyWeight" value="3" />
    <property name="parentKeywordsRewardWeight" value="2" />
    <property name="endsWithWsdlReward" value="-5" />
    <property name="urlKeywordsReward" value="-2" />
    <property name="urlKeywords" value="/webservice;/service;api;/ws" />
    <property name="contentKeywords" value="wsdl;web service;soap" />
</bean>
```

Figure 8. WsdlCostAssignmentPolicy.

- The last configuration change we made was to include *WsdlCrawlExperimentLogger* class. We created this class to log results of our experiment in every 30 minutes.

```
<!-- Temporary class created for logging experiment results in every 30 min -->
<bean id="experimentLogger" class="org.archive.crawler.reporting.WsdlCrawlExperimentLogger">
</bean>
```

Figure 9. WsdlCrawlExperimentLogger.

Job configuration files for the experiments can be found in the appendix section.

For the crawling experiment we installed two Heritrix instances to one server located in the server park of the University of Tartu. Both of the instances were given 3584MB of memory. The server had a broadband connection with the maximum download speed of more than 100Mb/s. A job configuration was created on each of the crawlers so that one of them included URL optimization and queue scheduling strategy to support focused crawling, and the other one was configuration for baseline test. The two jobs were started at the same time and ran in parallel.

The first experiment lasted for 27 hours, the second and third experiments lasted for 23 hours and 30 minutes. All the experiments were conducted in May 2012.

4.3 Analysis

The first task after the experiments had been completed, was to clean up the crawled data because the WSDL URIs we had found contained recursively recurring elements. The aforementioned URIs were caused by crawler traps into which our spider sometimes tumbled. The traps generated thousands of URIs for the same WSDL description recursively. For example, site *www.niceties-token.com* created an URL *http://www.niceties-token.com/ /NiceLogbookWebServices-WS.php?wsdl* from where it was possible to end up in URL *http://www.niceties-token.com/NiceLogbookWebServices-WS.php/getNiceMessages/ getNiceMessages/postNiceMessage/postNiceMessage/postNiceMessage?wsdl*. We created a script to remove these entries from the result files. The relation between found URIs and unique URIs is represented in Figure 10 .

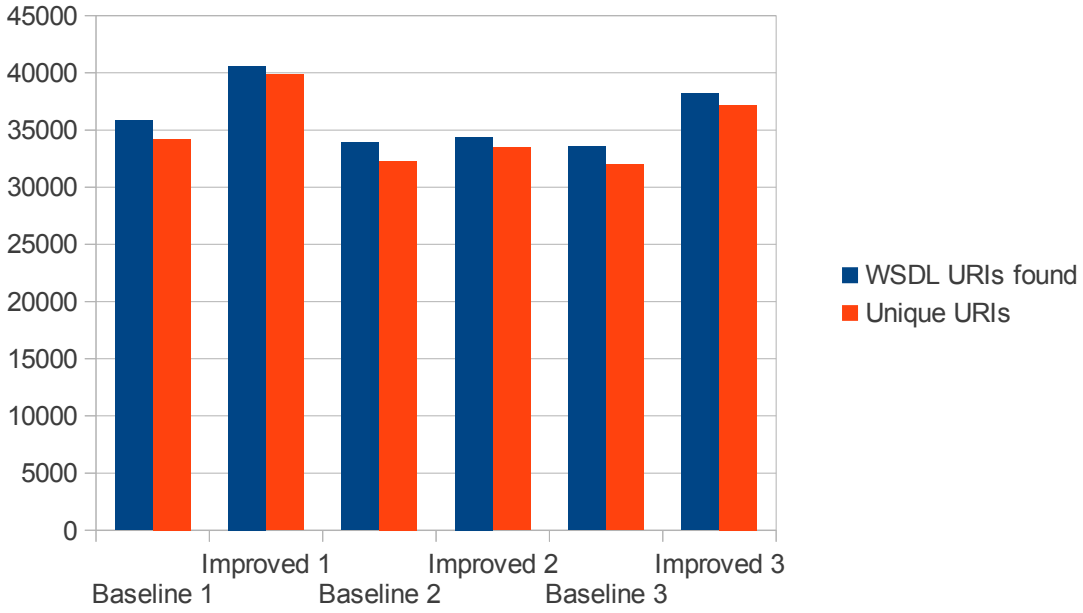

Figure 10. Recurring elements in the crawl's results.

When we received the purged data, we wanted to see how many of the URIs found originated from the seeds. In addition to this, we also wanted to know how big is the overlap of URIs found in the baseline run and improved run. Figure 11 shows the amount of URIs that had come from seeds, and Figure 12 displays the overlap of URIs found in the baseline run and improved run.

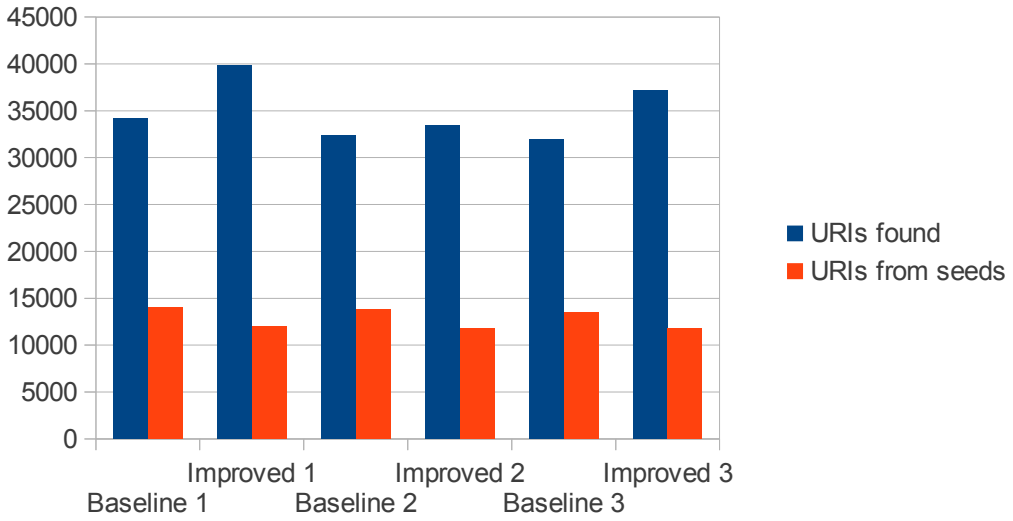

Figure 11. WSDL URIs that originated from the crawl's seeds.

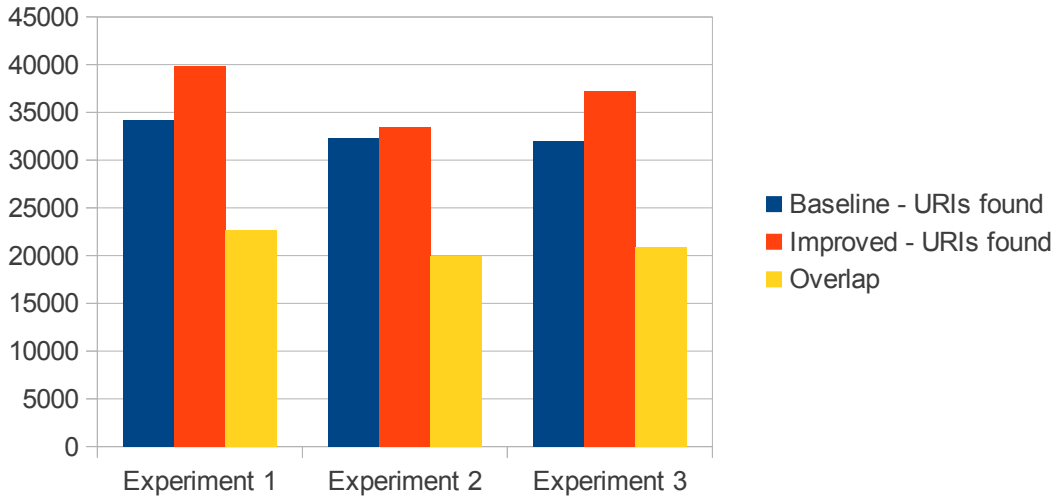

Figure 12. Overlapping of URIs found in the baseline run and improved run.

To draw any conclusions from our experiments, we would need to look at the progress of both of the jobs during the crawl's process. To display this information on a graph we have created Figures 13-15 that present the increase of WSDL URIs found in the experiments' timespan. Besides the numbers of WSDL descriptions found, we also logged the total number of URIs that the crawler had explored. From this data we can deduce the effectiveness of our crawl jobs. Figures 16-18 show the relations between found WSDL URIs and total number of URIs crawled.

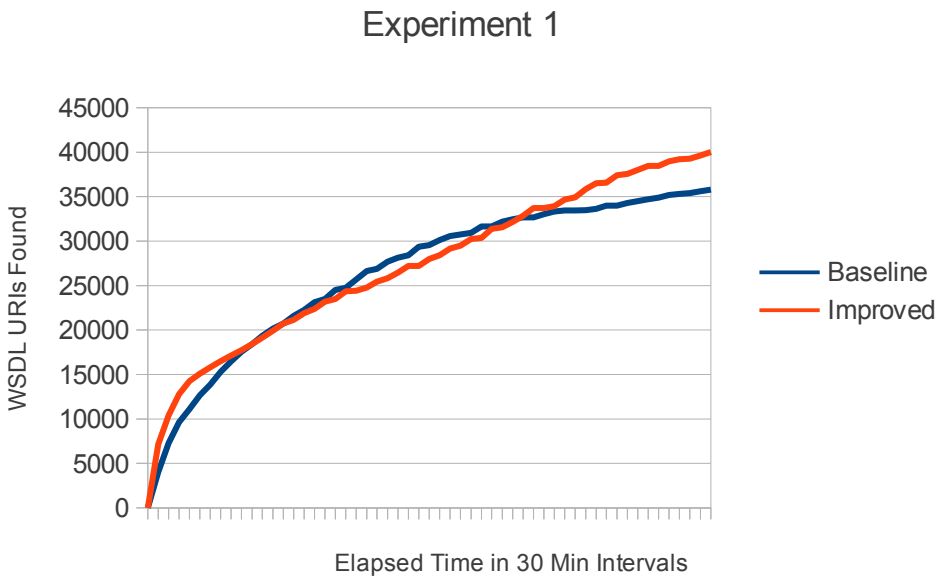

Figure 13. Progress of the first experiment.

Figure 14. Progress of the second experiment.

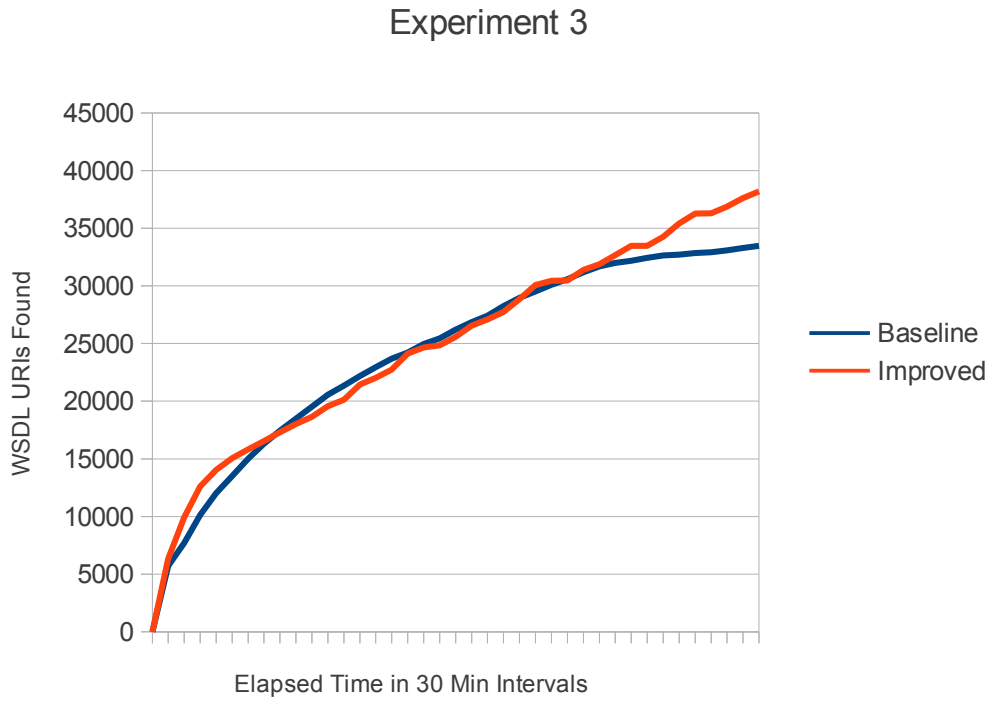

Figure 15. Progress of the third experiment.

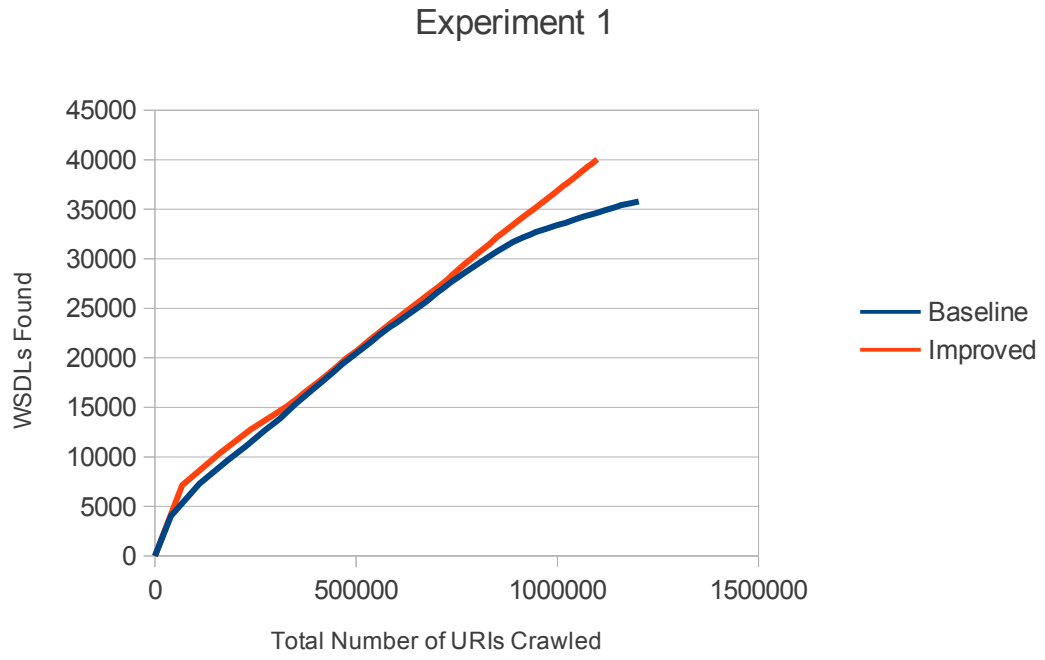

Figure 16. Ratio of WSDLs found and the total number of URIs crawled in the first experiment.

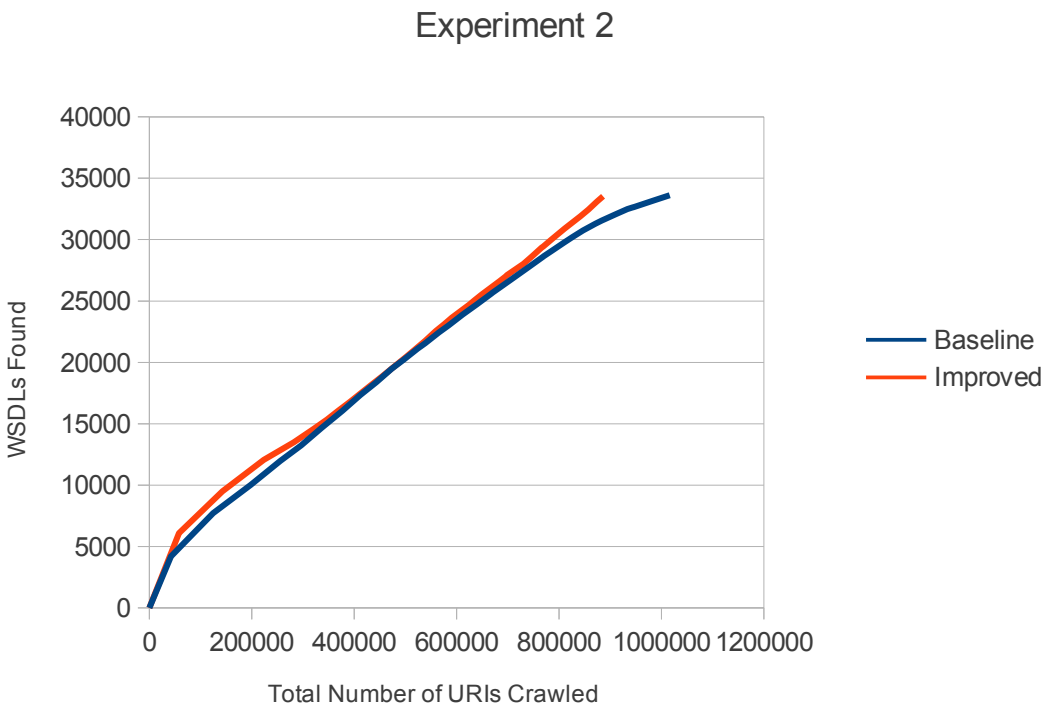

Figure 17. Ratio of WSDLs found and the total number of URIs crawled in the second experiment.

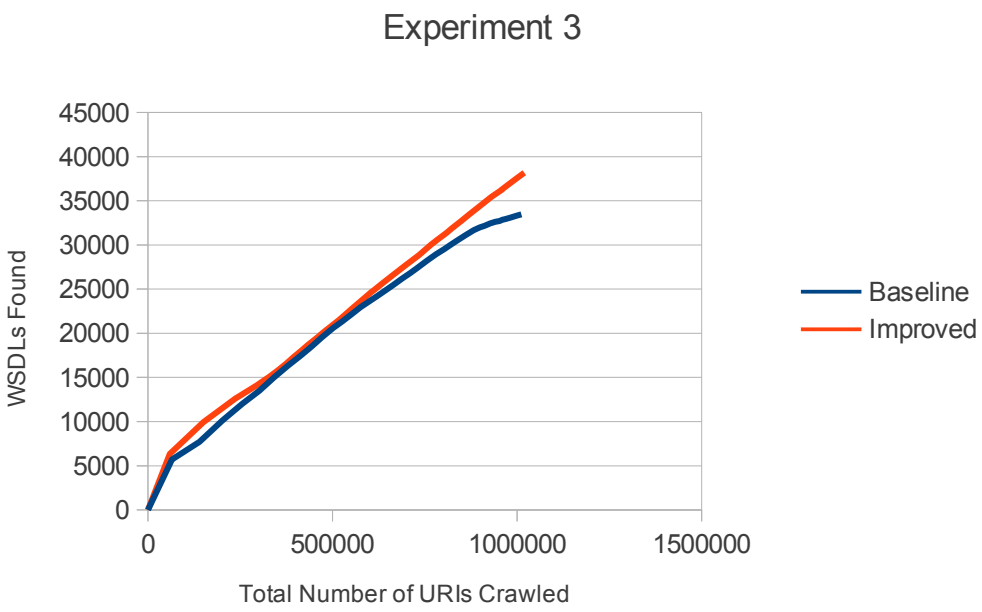

Figure 18. Ratio of WSDLs found and the total number of URIs crawled in the third experiment.

From the above graphs we can draw some conclusions. By looking at the Figures 11 and 12 we can see that, although approximately one third of the WSDL URIs that were found during the crawl originated from the seeds, there remained another third - that the baseline and improved crawler had found separately - which contained URIs with no overlap. This means that our focused crawling heuristic had at least some effect on the paths the crawlers took.

It seems that our experiment's timespan should have been longer. Because in Figures 13-18 the graph lines are taking interesting turns on the last third of the graph. But still some conclusions can be derived from these diagrams.

From Figures 13-15 it can be seen that the rates of WSDL descriptions found in time drops faster in baseline experiments. At first, when the crawl is near the seed URI sites, the baseline run performs better than the improved run, but as the spiders wander further, the job without focused crawling additions finds it harder to find new services. So eventually our improved crawler is discovering more WSDL URIs.

The reason why our baseline job finds less web service descriptions in the experiment's time frame is the ratio of WSDL URIs found in the total number of crawled URIs. In Figures 16-18 we can see that while the crawler with focused crawling support is looking

through a constant amount of URIs to find a service description, the baseline spider ratio is dropping. On the first two attempts our reference point job explored a greater number of URIs than the improved job, but discovered an equal amount or less services (Figure 16 and 17). Smaller crawl speed on the focused spider may be due to the overhead that our additions are creating. In the future it would be wise to performance tune the Java code created to enable focused crawling.

5 Conclusion

The Heritrix spider provides support for most of the heuristics used in large-scale web crawling. There are means for multithreaded crawling, spider trap avoidance, and defining the crawl's scope. Adding basic functionality for WSDL descriptions discovery to Heritrix can be done by altering the crawler's job configuration - provided that you are familiar with Heritrix's jobs and strategies for finding web services.

To help out users who are trying to map service access points in World Wide Web with Heritrix, we gathered a collection of strategies and crawler job configuration options to be used in this case. These originated from the published works that the other teams had done on the topic. In addition to it, we created a new module to the crawler's source code, that allows logging of search results without any excessive data.

With the job configuration changes mentioned, it was possible to spider the web for WSDL description URIs, but as Heritrix does not support focused crawling, the spider would explore all the web sites it happens to stumble upon. Most of these sites would accommodate no information relevant to finding web services. To guide the course of the spider's job to the resources potentially containing "interesting" data, we implemented support for focused crawling of WSDL URIs. The change required the creation of a new module in Heritrix's source code, the algorithm used as basis for our solution was described in the article [16]. Heritrix's source code with the changes we made can be found in the appendix section.

To see if our enhancement provided any improvement in the crawl's process, a series of experiments were conducted. In them we compared performance and accuracy of two crawlers. Both of which were configured for WSDL descriptions crawling, but one of them was also fitted with module providing support for focused crawling. From the analysis of the experiments' results we deduced that although the crawler job set for the experiments' baseline processed URIs a bit faster, the spider with the improvements found WSDL descriptions more accurately and was able to find more of them.

In the future, a new, longer, experiment should be conducted to see the crawler's progress after 24 hour run. It would also be wise to performance test and tune the Java code created for the focused web service discovery.

Heuristikud WSDL standardil veebiteenuste otsimiseks roomaja Heritrix näitel

Taniel Põld
Bakalaureusetöö (6 EAP)
Resümee

Käesoleva bakalureuse töö eesmärgiks on seadistada ja täiustada avatud lähtekoodil baseeruvat Heritrix veebiussi. Tehtud muudatuste tulemina peab Heritrix suutma leida veebiteenuseid märkivaid WSDL faile. Veebiuss ehk *web crawler* on programm, mis otsib automatiseeritult mööda Interneti avarusi ringi liikudes soovitud veebidokumente. WSDL on XML formaadis keel, mis sätestab veebiteenuse asukoha ja protokolli ning kirjeldab pakutavad meetodid ja funktsioonid.

Eesmärgi saavutamiseks uuriti avaldatud artikleid, mis kirjeldasid erinevaid strateegiaid Internetist veebiteenuste otsimiseks kasutades veebiussi. Mainitud tööde põhjal loodi Heritrix'i seadistus, mis võimaldas WSDL teenuse kirjeldusi otsida. Lisaks kirjutati programmeerimis keeles Java Heritrixi täiendav klass, mis võimaldab lihtsustatud kujul salvestada veebi roomamise tulemusi.

Ühes leitud artiklites kirjeldati suunatud otsingu (*focused crawling*) toe lisamist veebiteenuseid otsivale Heritrix veebiussile. Suunatud otsing võimaldab ussil hinnata uusi avastatud veebilehti ning lubab keskenduda lehtedele, mis suurema tõenäosusega sisaldavad otsitavaid ressursse. Kuna vaadeldavas programmis puudub tugi suunatud otsingu funktsionaalsusele, lisati see käesoleva töö käigus täiendava mooduli loomisega. Algoritmi aluseks võeti mainitud artiklis kirjeldatud lahendus.

Selleks, et kontrollida kas lisatud täiendus muutis roomamise protsessi täpsemaks või kiiremaks teostati eksperiment kolme katsega. Käivitati kaks Heritrixi exemplari, millest mõlemad seadistati WSDL teenuse kirjeldusi ostima, kuid ainult ühele neist lisati suunatud otsingu tugi. Katse käigus vaadeldi leitud teenuste arvu ja kogu läbi kammitud veebilehtede kogust.

Eksperimendi tulemuste analüüsist võis järeldada, et suunatud otsingu funktsionaalsus muudab roomamise protsessi täpsemaks ning võimaldab seeläbi WSDL teenuse kirjeldusi kiiremini leida.

References

[1] Mohammed AbuJarour, Felix Naumann , Mircea Craculeac. Collecting, Annotating, and Classifying Public Web Services. In *Hasso-Plattner-Institut, University of Potsdam, Germany* , 2010.

[2] Eyhab Al-Masri and Qusay H. Mahmoud. Investigating Web Services on the World Wide Web. In *University of Guelph*, 2008.

[3] Eyhab Al-Masri, and Qusay H. Mahmoud. WSCE: A crawler engine for large-scale discovery of web services. In *International Conference on Web Services*, 2007.

[4] Michael K. Bergman. White Paper: The Deep Web: Surfacing Hidden Value Volume 7, Issue 1, 2001.

[5] Duan Dehua . Deep Web Services Crawler. In *Dresden University of Technology*, 2010.

[6] GitHub. internetarchive/heritrix3 – GitHub. https://github.com/internetarchive/heritrix3. Cited: 27 March 2012.

[7] Internet Archive. Internet Archive: Digital Library of Free Books, Movies, Music & Wayback Machine. http://archive.org/index.php. Cited: 10 May 2012.

[8] Nicolai Josuttis. SOA in Practice: The Art of Distributed System Design. O'Reilly Media, Inc., 2007.

[9] Maurice de Kunder. World Wide Web Size. www.worldwidewebsize.com. Cited: 3 April 2012.

[10] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards recovering the broken SOA triangle: a software engineering perspective. In *W-SOSWE '07: 2nd International Workshop on Service-oriented Software Engineering*, pages 22–28, New York, 2007.

[11] Gordon Mohr, Michele Kimpton, Michael Stack, Igor Ranitovic and Dan . An Introduction to Heritrix, an archival quality web crawler. In *Proceedings of the 4th International Web Archiving Workshop (IWAW '04)*, Sept 2004.

[12] Seekda. Web Services Search Engine. http://webservices.seekda.com, 2009. Cited: April 7, 2012.

[13] SOATrader. Home. http://soatraders.com. Cited: 14 May 2012.

[14] Josef Spillner. Deep Web Services Exploration. http://beta.crowdserving.com:3000/josef/10-x-scalability-goals/deep-web-services-exploration, April 6, 2010. Cited: May 15, 2012.

[15] Nathalie Steinmetz, Holger Lausen, Manuel Brunner. Web Service Search on Large Scale. In *International Conference on Service Oriented Computing (ICSOC)*, Stockholm 2009 .

[16] Nathalie Steinmetz, Holger Lausen, Martin Kammerlander. D2.1 Crawling Research Report - Version 1, 31 Oct 2008. Available from: http://www.service-finder.eu/attachments/D2.1.pdf. Cited: May 8, 2012.

[17] Nathalie Steinmetz, Holger Lausen. D1.3 Final Crawling Prototype , April 15, 2011. Available from: http://service-detective.sti2.at/deliverables/D1.3.pdf. Cited: May 8, 2012.

[18] Wikipedia. GNU Lesser General Public Licence – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/GNU_Lesser_General_Public_License, 13 May 2012. Cited: 14 May 2012.

[19] Wikipedia. Heritrix – Wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Heritrix, 18 November 2011. Cited: May 8, 2012.

Appendices

Improved Heritrix's Sourcecode and Installation Guide

Source code for the Heritrix improvements can be found in the following public repository: https://bitbucket.org/tanielp/heritrix-for-wsdl-crawl

Installation guide for Ubuntu Linux 11.04 with Java JDK installed:

1. Get software:
   sudo apt-get install mercurial maven2

2. Clone Mercurial repository:
   cd ~/workspace
   hg clone https://bitbucket.org/tanielp/heritrix-for-wsdl-crawl

3. Compile and package sourcecode:
   cd ~/workspace/heritrix3
   mvn -Dmaven.test.skip=true install

4. Copy and extract packaged Heritrix:
   cp ~/workspace/heritrix3/dist/target/heritrix-3.1.1-SNAPSHOT-dist.tar.gz ~/
   tar -zxvf ~/heritrix-3.1.1-SNAPSHOT-dist.tar.gz

5. Run Heritrix:
   cd ~/heritrix-3.1.1-SNAPSHOT-dist/bin
   ./heritrix

Experiment's Job Configuration Files

Job configuration files used in experiments can be found in the following public repository: https://bitbucket.org/tanielp/heritrix-for-wsdl-crawl-experiment-jobs

Download guide for Ubuntu Linux 11.04:

1. Get software:
   sudo apt-get install mercurial

2. Clone Mercurial repository:
   cd ~/directory
   hg clone https://bitbucket.org/tanielp/heritrix-for-wsdl-crawl-experiment-jobs