A Vector Space Search Engine for Web Services

Christian Platzer and Schahram Dustdar
Vienna University of Technology, Distributed Systems Group,
Information Systems Institute
Argentinierstrasse 8 / 184-1, 1040 Vienna, Austria
cplatzer@infosys.tuwien.ac.at

Abstract

As Web services increasingly become important in distributed computing, some of the flaws and limitations of this technology become more and more obvious. One of this flaws is the discovery of Web services through common methods. Research has been pursued in the field of "Semantic Web services". This research is driven by the idea, to describe the functionality of Web services as accurately as possible and to create programs automatically out of already existing Web services. In this paper we discuss a new method for discovery and analysis of Web services. Our approach uses a Vector Space Search Engine to index descriptions of already composed services. Rather than generating or automatically composing applications, this approach provides developers with a valuable utility to browse repositories based on already existing information. Furthermore, we propose some additional modifications to extract the maximum amount of semantics from existing service definition repositories.

1 Introduction

The basic idea behind our approach is a combination of common information retrieval methods and existing standards for the description of Web services (WS). WSDL and UDDI [5] are today's standards to describe a SOAP-based Web service well enough to place a remote procedure call and therefore invoke the service. Unfortunately, the knowledge how to call a method is not sufficient in many cases. The ongoing research in the field of Semantic Web services aims to describe the functionality behind the methods too[3]. The major drawback here is that with increasing possibilities to describe a method, the complexity of the used ontology or description language rises equally. What we propose is a different approach to solve this issue.

Instead of introducing a new language or ontology to describe a Web service, we first take a look at the existing information and use it as thoroughly as possible. A Web service description, no matter if it is available as a WSDL file or within a UDDI registry, contains a certain amount of information, entered by the programmer. The method names are a very good example. Most programmers tend to name their functions or methods according to their functionality like "+getMaximumInteger()" or "searchByString()". Furthermore, most descriptions contain some sort of comments for human readers. Our vision is, to create a search engine, where all this information is gathered and used to find the best matching method for a specific request. For this purpose we utilize a search mechanism common in modern information retrieval systems: The **Vector Space Model** (VSM) [13]. This approach is mainly used for search engines, based on natural language. Many search engines on the Web utilize this method to search their repositories of Web pages. The underlying concept is quite simple. A document is split up in keywords. Each of this keywords constitutes a dimension in a n-dimensional vector space. Therefore, a document can be seen as a vector within this "term space". The position of this vector to other vectors within the same vector space describes their similarity to each other. The mathematical method to evaluate how similar two documents are to each other and respectively match a given query, varies. A popular method is to calculate a cosine value for them and express the result as a percentage rating. This method produces very good results for natural language but it is not limited to this field alone.
Virtually any document collection can be mapped to a vector space to create an efficient search mechanism. The mapping includes syntactical indices as well as a semantic representation of the underlying structure.

The paper is organized as follows. Section 2 starts by giving the motivation for our work. Section 3 discusses the related work and the developed algorithm for searching distributed repositories. Section 4 introduces the developed prototype search engine and the produced results. Section 5 finally concludes and gives an overview of the future work.

2 Motivation

Our research is driven by the same idea that drives the whole Semantic Web Services community [3]: How is it possible to describe the functionality of a program or a service on the Web? This is not an official definition of the term "Semantic Web Service" of course, but it gives quite a good idea about the problem, today's programmers are confronted with. The ongoing research, especially in the area of Web services, tries to find solutions for the semantic description of services over the Internet [9].
In our particular case, we want to develop a method to retrieve description files by just entering a search query. A programmer for example, who wants to integrate the Google search engine in his own Web site should be able to enter a query like "Google search Service" and get the corresponding WSDL File for the Web Service.

Increasingly, research currently describes the functionality of Web services by artificial means [4]. Although this approach looks very promising, we think there will arise some additional problems when introducing a solution:

- It is possible to create an ontology for Web services and use it to describe the functionality of the service itself. But what about the already established Web services? There will be no way to assess the functionality of existing services, if the description does not meet the requirements, defined by the ontology. Instead, these service descriptions would have to be reworked or at least a gateway solution had to be introduced before they comply with this new standard.
- The second, and in our opinion even more important issue, concerns the ontology's potential. An ontology, which is able to describe functionality in every detail will raise in its complexity to a point where it is no longer distinguishable from a programming language. A simple ontology may be enough to describe a method which adds two integers but for a function that calculates the hamming-distance for two vectors, the ontology will have to be more powerful and, therefore, more complicated.
- Another well known problem when dealing with ontologies is authenticity. There is no guarantee for a semantic description to actually represent the underlying functionality of a service and not something else.

Given the problems stated above and the current estate for Web services with the involved technologies, we investigated the possibilities to enrich Web service descriptions with information about what the services do. In this first part of our work, we aim to use the available information without adding new restrictions or requirements. Before discussing our new approach in detail, we will take a look on the information that is already provided in today's repositories.

2.1 Web service descriptions

Whenever a Web service is published, a WSDL File will be created to provide all the needed (syntactical) information for other programmers to invoke the service. Even when the description is automatically generated by a development tool, it still holds some valuable information about the data types and their labels, assigned by the programmer. Furthermore, the messages are indicators for the functionality of the underlying methods. A good example is the WSDL File for the Amazon Web service [1].

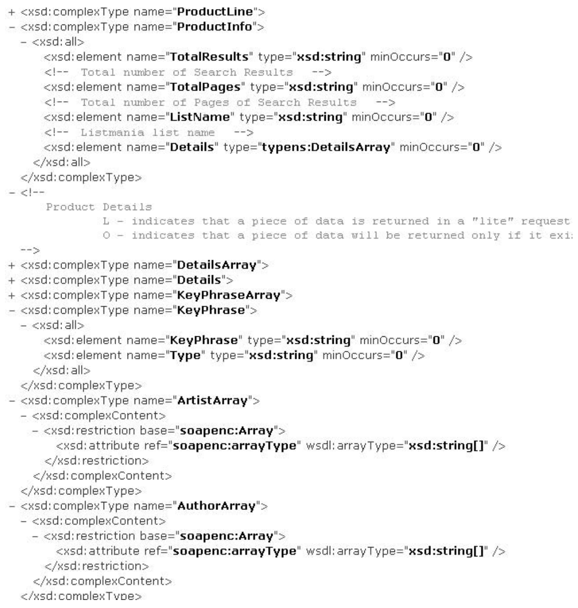

```
+ <xsd:complexType name="ProductLine">
- <xsd:complexType name="ProductInfo">
  - <xsd:all>
      <xsd:element name="TotalResults" type="xsd:string" minOccurs="0" />
      <!--  Total number of Search Results   -->
      <xsd:element name="TotalPages" type="xsd:string" minOccurs="0" />
      <!--  Total number of Pages of Search Results   -->
      <xsd:element name="ListName" type="xsd:string" minOccurs="0" />
      <!--  Listmania list name   -->
      <xsd:element name="Details" type="typens:DetailsArray" minOccurs="0" />
    </xsd:all>
  </xsd:complexType>
- <!--
     Product Details
             L - indicates that a piece of data is returned in a "lite" request
             O - indicates that a piece of data will be returned only if it exi
  -->
+ <xsd:complexType name="DetailsArray">
+ <xsd:complexType name="Details">
+ <xsd:complexType name="KeyPhraseArray">
- <xsd:complexType name="KeyPhrase">
  - <xsd:all>
      <xsd:element name="KeyPhrase" type="xsd:string" minOccurs="0" />
      <xsd:element name="Type" type="xsd:string" minOccurs="0" />
    </xsd:all>
  </xsd:complexType>
- <xsd:complexType name="ArtistArray">
  - <xsd:complexContent>
    - <xsd:restriction base="soapenc:Array">
        <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:string[]" />
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
- <xsd:complexType name="AuthorArray">
  - <xsd:complexContent>
    - <xsd:restriction base="soapenc:Array">
        <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:string[]" />
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
```

Figure 1. Amazon WSDL Snippet

Even without any knowledge about the Web service itself, the names of the elements put across quite a good idea of the underlying function's purpose.
But valuable information is not comprised in the actual tags alone. There are often comments with a human readable description about the elements or the whole service. Our motivation is, to exploit all this information at a maximum level.

2.2 Web service Discovery

UDDI registries are designed as the central point to register Web services and to make them publicly available. They

ought to help discovering Web services for a particular operation or from a known publisher.
Unfortunately, UDDI contains some serious flaws which lead to detrimental effects:

- UDDI registries do not contain a proper functionality for limiting the lifetime of a once registered service. Because of that, entries are often out of date and therefore obsolete.

- Everyone can publish WSDL descriptions to a public UDDI registries. As a result, a good deal of the registered services were entered for testing purposes only. When looking at the Microsoft public registry for example, there are even entries where the port address is entered with `http://localhost/MyWebservice/`. Entries like this are useless, of course, but it is obvious why things like this happen. Unexperienced programmes let their development tool create the WSDL description and because the service runs on the local machine, localhost is used for the port description. After publishing the WSDL File to the UDDI registry, the programmer will wonder why his service is not accessible. Availability is the second important issue for those registries.

- The third issue concerns openness. Some of the UDDI registries available today, require some type of subscription, before they accept any files to be published to their database. The IBM UDDI Business registry [8] is a very good example for this. This restriction has the very positive effect that most of the published descriptions are meant serious. Availability, on the other hand is still a problem, because the fact that one has to register or pay before publishing a Web service does not mean it is accessible all the time.

To summarize, there are two possibilities. Either a UDDI registry is made publicly available and contains a good deal of obsolete entries, or it requires registration but only keeps a limited number of available service descriptions.

2.3 Requirements

Before moving to the technical details of our approach, we want to discuss some requirements that have to be met before we can successfully apply our concept to a prototype.

2.3.1 Data resources

Our implementation uses WSDL Files as a source for the data repository because these XML Files carry all the information that is needed to describe a Web service. It would also be possible to use data directly from UDDI registries, but since the data is not required to be in XML but can be only a textual description too, this method is not feasible for our implementation. Retrieving enough WSDL Files from the Internet to form a satisfying repository was a particulary hard task.
We investigated three methods to obtain a large set of WSDL files. Our main prerequisite was to only use "common" means of accumulation and not some research prototype or a single source like woogle [6] to keep the concept as generic as possible.

- **File Sharing:** File sharing platforms like Emule or Kazaa are capable of handling all types of files, including WSDL files. Unfortunately, the amount of shared descriptions is limited. We were able to retrieve 62 valid WSDL files this way.

- **Web Crawler:** Although web crawlers looked very promising at first, we quickly realized that this method is a very poor way to obtain a decent repository. This is because web crawlers need a link that is directed to a WSDL file to successfully retrieve the data. In most cases the file itself is included in an API or some sort of package which makes it impossible to retrieve by a crawler. An enormous amount of traffic was produced by our *wget*-based crawlers, before retrieving a single WSDL file.

- **tModel cross references:** This third method finally produced the desired results. By iterating through a UDDI registry, we were able to retrieve links to service endpoints and description files. We basically extracted the links from UDDI registries and tried to retrieve the files to our local repository. See the Implementation section for a detailed description of the extraction process.

To be accurate, a fourth method to gather Web service description files is provided by the possibility to upload data directly to the server where the search engine is running. This method, which is also implemented in our prototype, is about the same as a UDDI registry, so it was not mentioned in the listing above.

2.3.2 Search engine

The key element is not the data itself. It is the engine, that extracts interesting data and executes queries upon it. The requirements for such an engine are very high. The final product must possess both, good performance and a good precision/recall rating. Designing this engine was the main challenge and required some research in the field of information retrieval. It is useful to take a look on available search engines to get an idea how our solution looks like.

What we expected as an outcome of our research, was an efficient search engine for Web services. This engine has to be capable of handling both WSDL files and UDDI entries at the same time. Furthermore, it should be possible to set up this engine at different UDDI registries and join them to one repository. We utilized state-of-the-art methods from the area of information retrieval to provide a valuable tool for developers.

2.4 Contribution

Our contribution is twofold. We will introduce an algorithm which allows to join detached document repositories to a single one and execute queries upon the resulting vector space. Furthermore, we will present our proof-of-concept prototype implementation.

We already mentioned in the introduction, that our approach to solving the problems of Web service discovery is novel. Our main inspiration came from Web search engines like Google or Fireball. Even with all those irrelevant pages around the Web, they manage to create very valuable results for a large range of queries [11]. Our task was not much different. We developed a search engine for Web services where developers are able to find what they need for their applications amongst the available repository, even if the repository contains a good amount of improper entries.

2.5 Architecture

The basic use of the Vector Space Model presented in the following section does not differ from applications for natural language.

Service descriptions will be parsed for relevant data like type definitions, elements, and service names. The extracted keywords are then used to create a vector space where every document represents a vector within it. See Figure 2 for a visualization of the concept.

This architecture allows the creation of a localized search engine that can be installed at a UDDI registry, for example. We wanted to go one step further and allow distributed search engines to interact with each other and process queries as if they operate on one single document repository. To to this with a vector space model, we needed to extend existing approaches and introduce a new algorithm.

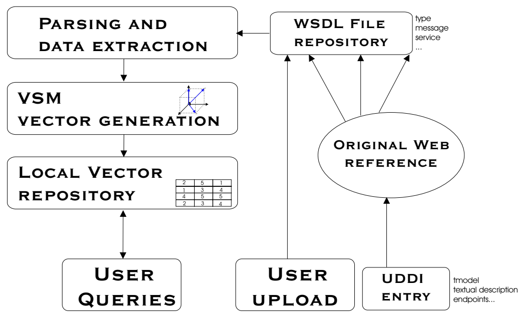


Figure 2. Basic architecture

3 Related Work

3.1 The VSM Concept

The VSM, as proposed by Salton [7] was basically designed for various applications. This section provides a bottom-up analysis with a discussion on the capabilities which are needed for the above mentioned UDDI join.

3.1.1 The Term Space

The core of a Vector Space Engine is the Term Space itself. The idea behind it is to create a Vector Space where each dimension is represented by a term [15]. This space can grow in dimension every time a new keyword is added.
In a vector-based retrieval model, each document is represented by a vector $d = (d_1, d_2, ..., d_n)$ where each component d_i is a real number indicating the degree of importance of term t_i in describing document d [15]. How this weighting is done, has a major impact on the overall performance and behavior of the system. The easiest method is a boolean weight [14]:

$$d_i = 1 \; \forall \; t_i \in C \text{ with C being the Term Collection}$$

which means, if term i of the collection C occurs in the document, its corresponding value in the vector is 1 and 0 otherwise.
Binary values are the simplest form of a document representation based on vectors.

Once, all documents (e.g. WSDL files) are represented within the common term space, the relevance between them can be rated according to various rating procedures. Before moving to document rating and term weighting, an evaluation of the presented model in respect to its distribution capabilities is necessary.

The approach, discussed in the previous section is based on the assumption that the term space is available at a

centralized point. When it becomes necessary to create a distributed form of this model, certain additional points have to be considered.
In the presented binary form, distribution is not a big issue. Keeping vectors valid on different spots can simply be achieved by transporting relevant vectors with their corresponding keywords. At the destination space, the vector is then treated as a new document, like in the following example. We assume that there are two different term spaces C_1 and C_2:

C_1=

Dimension / Document	d_1	d_2	d_3
google	1	1	0
search	1	0	1
service	0	0	1

C_2=

Dimension / Document	d_1	d_2	d_3
google	1	1	1
search	1	0	1
result	0	0	1

Now, if it is necessary to evaluate how relevant document d_3 from C_1 is in C_2, the simplest method is to transfer the whole vector with all keywords and treat it like a new document in C_2. As a result, C_2 is expanded by one dimension resulting in the following term space:

C_2=

Dimension /Document	d_1	d_2	d_3	d_4
google	1	1	1	0
search	1	0	1	1
result	0	0	1	0
service	0	0	0	1

In general, binary weighting and vector creation is fit for a distributed environment and therefore capable of handling multiple UDDI registries, for example.

3.2 Weighting

Binary weighting like presented in the in the previous section will not be sufficient for a sophisticated search engine. It completely ignores important information like term frequency or document length. For this reason, term weights are assigned to the vector-elements. A common method to assign term weights is to store the inverse document frequency of a document as the vector element [7].

3.2.1 Term frequency and inverse document frequency:

The inverse document frequency *idf* of a term is a function of the frequency f of the term in the collection and the number N of documents in the collection [11]. Its purpose is to weight terms highly if they are frequent in relevant documents, but infrequent in a collection as a whole. According to Salton [7], the inverse document frequency is calculated as:

$$idf_k = ld(\frac{N}{n_k} + 1).$$

Therefore the term weight *tf* x *idf* is calculated as:

$$w_{ik} = tf_{ik} * ld(\frac{N}{n_k} + 1).$$

with T_k = term k in Document D_i
tf_{ik} = frequency of term T_k in document D_i
idf_k = inverse document frequency of term T_k in collection C
N = total number of documents in the collection C
n_k = the number of documents in C that contain T_k

When it comes to using this weighting scheme for distributed UDDI joins, the solution is not a trivial one any more. Because the vector components are now dependent on each other, the method used to add new documents and store the vector values must be taken into account.
In common Information Retrieval systems, term weights are updated, once a new document is added to a repository. After the update, the values for each vector reflect the overall number of documents and the particular weights for each term. Unfortunately, this is only possible in a centralized model, because the values for N and n_k are known. When two separated vector spaces must be combined, this knowledge is not available a priori. Thus, sending an already weighted document vector to another term space is not possible for a successful comparison in separated vector spaces. Instead, all necessary data has to be stored individually, to enable weighting at runtime. This, of course, means an additional overhead for query processing. Adding new documents on the other hand, is extremely fast with our approach. The following steps are carried out, when a new document is added to the repository:

- The raw term frequencies are calculated for the document. If one or more terms are not present in the term space, the space is expanded, by adding a new entry to the list of known terms.
- The raw term frequencies are stored for each term that occurs in the new document.
- The values for N and n_k are updated.

The data structure to store the term frequency will be a hash table or indexed list in most cases. Every keyword represents an entry in this hash table. The following example shows how a document or query from one collection is used to create a relevance rating at another collection.

We start with the collection C_1, indexed with the raw term frequencies:

C_1=

n_{k_1}	Dimension	d_1	d_2	d_3
2	google	5	3	0
2	service	4	0	8
1	search	0	0	9

,$N_1 = 3$

C_2=

n_{k_2}	Dimension	d_1	d_2	d_3
2	google	8	0	2
3	result	3	2	6
2	search	2	0	1

,$N_2 = 3$

Now, document d_3 of C_1 shall be rated at Collection C_2.
For this purpose, document d_3 is merged with C_2 to a temporary term space with $N = N_1 + N_2 = 6$. For all terms occurring in d_3 the temporary term count is calculated as $n_k = n_{k_1} + n_{k_2} \Rightarrow n_{service} = 2$ and $n_{search} = 3$.

Thus, in a distributed environment with m diverse term spaces, the values for N and n_k of a single document vector, located at any term collection C_j are:

$$N = \sum_{i=1}^{m} N_i$$

and

$$n_k = \sum_{i=1}^{m} n_{k_i} \{k | t_k \neq 0\}.$$

If any document is present in more than one collection, a slightly reduced term weight would be the result. This weighting scheme does not apply to the comments of service descriptions only. It is used for every weighted keyword.

3.2.2 tf x idf normalization

Moreover, the bare *tf* x *idf* value is not enough, because it rates longer documents higher than shorter ones [2]. Out of this reason, term weights are usually normalized to an interval between 0 and 1, so the total number of occurrences within one document does not matter anymore. The following formula is used, to normalize the weight of term k in document i [7]:

$$w_{ik} = \frac{tf_{ik} * ld(\frac{N}{n_k})}{\sqrt{\sum_{k=1}^{t}(tf_{ik})^2[ld(\frac{N}{n_k})]^2}}$$

Distribution capabilities are the same as mentioned in 3.2.1. There are no additional values required to normalize the weights according to our formula.

3.3 Rating Algorithms

This section discusses the most important rating algorithm and its distribution capabilities for common vector space models. Once the term weights for a document or a query are properly assigned, the similarity to other documents within the same term space can be rated and compiled to a final ranking of the most relevant results. One method is quasi-state of the art for data repositories based on natural language [13][6].

3.3.1 The Cosine Coefficient

The cosine value is the most commonly used rating algorithm. It takes two vectors of the term space and generates the cosine value for the angle between them [2]. In a n-dimensional space, the cosine value between two vectors p and q is calculated as

$$cos(p, q) = \frac{p \cdot q}{p \times q},$$

whereas $p \cdot q$ entitles the dot product and is calculated by multiplying term weights of the query- and document vector together[15]. Therefore the cosine value can also be written as

$$cos(p, q) = \frac{\sum_{i=1}^{n} p_i q_i}{\sqrt{\sum_{i=1}^{n} p_i^2 \sum_{i=1}^{n} q_i^2}}.$$

If the values for p and q are already normalized to the Euclidean norm, the cosine value can also be written as $\sum_{i=1}^{n} q_i d_i$. The idea behind this approach is, that two documents with a small angle between their vector representations are related to each other. Documents with no terms in common will have a cosine of 0 while identical documents will produce a cosine of 1. This is where the whole concept is getting a little fuzzy. The assumption that semantics are primarily expressed by term frequencies is not equally valid for every field, especially within natural language processing. Therefore, the results of rating functions can produce outputs of varying quality. Because of the few words, method names consist of, and therefore the small dimensionality of the resulting vector, we expected good results for those components of a service description.

One aspect of the chosen method is its linearity. As a result, the form of distribution, presented in 3.2 is also applicable. The ad-hoc generation of term weights in addition with the transported values for term- and document counts are sufficient to create a coherent term space where the resulting relevance rating is valid, even when term spaces are split and distributed like in our case.

4 Implementation

To demonstrate the reliability our concept and to show how an application for the presented architecture may look like, we implemented a prototype search engine. We designed our application with a Web frontend and made it publicly available to offer the possibility to try some of its functions and evaluate the produced results.f The Web application can be accessed via `http://www.vuse.de.vu`.

4.1 Set-up

We implemented the search engine and the Web frontend in Visual Studio 7.1.3088 using the .NET Framework Version 1.1.4322 SP1. This IDE is a good tool to create a fast Web-based implementation without worrying about circumferential problems like deployment or compatibility. Memory requirements and Processor speed are negligible for our sample repository of 4096 WSDL files. The test machine is a 800 Mhz Pentium III Machine with 512 MB of main memory.

4.2 UDDI extraction

We used the Microsoft UDDI SDK Version 2.0 Beta for interaction with UDDI V2.0 conform services to communicate with public UDDI registries. The extraction took place in the following sequence.

- A public UDDI registry is entered and the application extracts an alphabetical list of available TModels.
- For every entry in the retrieved list, the TModelInfo has to be retrieved, consisting of a representation of every entry, including the TModelKey.
- Finally, every TModelKey has to be retrieved in a single request, and for each Key, the DocumentURL is stored in a list. Our query at the Microsoft Public UDDI regsitry [10] gave a result of 6438 parsable URLs.
- In the final step, we generated 30 Threads to retrieve the WSDL descriptions contained in the URL and added them to our local repository.

The first interesting result showed when we iterated through the URLs we received from our initial query. Out of 6711 entries, we were able to actually download 1272 files from their original site, which means that 19% of the entries are actually functional. We expected a far lower ratio because the registry is public and free to use for everyone.
546 of the downloaded files where valid XML files and were parsed by our keyword extractor, which means that 9% of the original entries are actually WSDL descriptions. The other files where either pdf descriptions or plain HTML files. For security reasons, we did not implement a frontend for our UDDI extraction functionality. It would be easy to misuse the provided extraction procedure to initiate a DOS-attack for public UDDI registries.

4.3 Keyword extraction

A service description, no matter if it is a WSDL file, or data from a UDDI registry, consists of two different types of data, as far as the Vector Space Model is concerned. First, there are user comments written in plain text. This information is voluntarily and not every developer will enter comments as descriptions. If comments were entered, the extracted data is treated as natural language. Therefore, familiar methods for vector space engines like stop word lists or normalization procedures can be applied [12]. On startup, the keyword extractor iterates through all files in the repository and isolates as many keywords as possible. Our current version handles the following elements:

- Endpoint URLs: The endpoint, which is present in every WSDL file, is split up to multiple keywords containing domain names and suffixes.
- Types and their attribute names are parsed and split up if possible.
- Messages are parsed for their names and split up to single words.
- XML comments are parsed and treated as natural language. Like other elements, the words are split if possible and fed to the search engine.

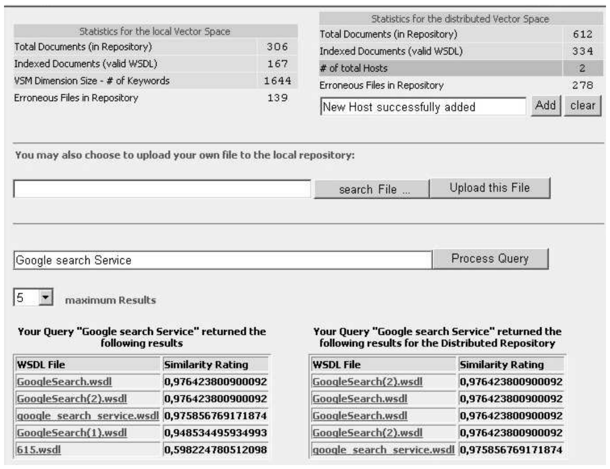
After extracting the keywords, the vector for this document is built according to the keyword frequencies, and added to the local vector space.

4.4 Query Processor

The query processor takes a query string, in our case "Google Search Service", and splits it up to a list of keywords. After building the corresponding query-vector, it is projected into the local term space where the cosine-value is evaluated for all other documents. The result is a list of documents, sorted by their similarity rating. See Figure 3 for a screenshot of the results for the query.

4.5 Joiner

To finally join one or more of the search engines, we implemented the query interface as a Web service. Two methods are visible from the outside:
RemoteQueryStats: This method takes a query string and



Statistics for the local Vector Space	
Total Documents (in Repository)	306
Indexed Documents (valid WSDL)	167
VSM Dimension Size - # of Keywords	1644
Erroneous Files in Repository	139

Statistics for the distributed Vector Space	
Total Documents (in Repository)	612
Indexed Documents (valid WSDL)	334
# of total Hosts	2
Erroneous Files in Repository	278

New Host successfully added

You may also choose to upload your own file to the local repository:

Google search Service

5 maximum Results

Your Query "Google search Service" returned the following results

WSDL File	Similarity Rating
GoogleSearch.wsdl	0,976423800900092
GoogleSearch(2).wsdl	0,976423800900092
google_search_service.wsdl	0,975856769171874
GoogleSearch(1).wsdl	0,948534495934993
615.wsdl	0,598224780512098

Your Query "Google search Service" returned the following results for the Distributed Repository

WSDL File	Similarity Rating
GoogleSearch(2).wsdl	0,976423800900092
GoogleSearch.wsdl	0,976423800900092
GoogleSearch.wsdl	0,976423800900092
GoogleSearch(2).wsdl	0,976423800900092
google_search_service.wsdl	0,975856769171874

Figure 3. Query Results - Screenshot

returns all necessary values to invoke a distributed query, namely n_k, N and a list of Query words.
processDistributedQuery: This method finally processes a distributed query on the local repository with the Values retrieved by the above function.

The invoking host first gathers all required information from all peers and then invokes the distributed query with an assembly of all statistics. The result must be displayed by the invoking host, of course.

4.6 Frontend

The frontend is deliberately kept simple. It supports file uploads, local queries, statistic analysis of the local repository and remote query processing.
To join the local repository with a remote web service, the service endpoint can simply be added by typing it in the designated field and pressing "Add". Because the sample application will most probably be the only one running at the time, it is possible to test it with it's own endpoint at `http://{domainname}/VSMWeb/VSMJoiner.asmx`. In this case, the search result will of course display the same file twice, since it is processed localy and remotely. The screenshot in figure 3 shows how the result for the joined repository looks like.
The original repository contains about 250 WSDL files from the UDDI extraction process along with the files retrieved by file sharing systems. We left erroneous files in the repository on purpose, to show the percentage of valid files that were retrieved. The prototype is open for file uploads to test the application with user-defined WSDL files.

5 Conclusion and future work

In this paper we presented a novel distributed Web service search engine based on the Vector Space Model for information retrieval. We gave an overview of the underlying technologies and extended the existing methods with a technique to make the concept work for distributed environments. We finally discussed our prototype implementation and showed that the presented approach works even for large WSDL repositories.

Unlike other search engines, no template document collection exists to evaluate the final precision/recall rating. To formally evaluate and optimize the search engine's performance parameters, a test-collection with predefined results has to be established. Furthermore, the vector matrix is currently uncompressed. By erasing zero entries in the matrix and therefore compressing the vector space, we think the performance can be increased significantly.

We think that it is very hard to automatically generate working applications out of Web services without human judgement. Creating ontologies may help to a limited degree. For the future, we plan to extend the indexing procedure from purely syntactical data to a semantic level. For this purpose we will utilize a domain-specific ontology to describe the functionality of a service endpoint and integrate the result in a BPEL-process. The major problem here is, to find a fitting indexing method for the ontology itself. Furthermore, by using a domain-specific resource, the application domain is limited equally, which is quite the opposite of what we want to achieve. A possible tradeoff could be achieved by combining syntactical analysis and ontology-supported weight adjustment. It remains to be seen how beneficial the application of ontologies is to leverage the search mechanism to a semantic level.

Acknowledgements

We would like to thank the reviewers of this paper for their constructive remarks. This paper is partially funded by the EU Project Nedine.

References

[1] Amazon. Amazon Web service definition File. `soap.amazon.com/schemas2/AmazonWebServices.wsdl`, 2005.
[2] D. L. L. Budi Yuwono. Search and ranking algorithms for locating resources on the world wide web. *IEEE*, 1996.
[3] A. M. Christoph Bussler, Dieter Fensel. A conceptual architecture for semantic web enabled web services. *htpp://swws.semanticweg.org*, 2003.

[4] O. Conlan, D. Lewis, S. Higel, D. O'Sullivan, and V. Wade. Applying adaptive hypermedia techniques to semantic web service composition, 2003.

[5] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, 2002.

[6] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. *VLDB*, 2004.

[7] M. J. M. Gerard Salton. *Introduction to Modern Information Retrieval*, volume 1. McGraw-Hill, Inc., 1983.

[8] IBM. IBM Business registry. `https://uddi.ibm.com/ubr/registry.html`, 2005.

[9] S. McIlraith, T. Son, and H. Zeng. Semantic web services, 2001.

[10] Microsoft. Microsoft public UDDI registry. `http://uddi.microsoft.com/inquire`, 2005.

[11] B. C. Monika Henzinger, Brian Milch and S. Bin. Query-free news search. *ACM/WWW*, 2003.

[12] M. Porter. Porter stemming algorithm, 10 2004. http://www.tartarus.org/ martin/PorterStemmer/.

[13] W. Z. SKM Wong and patrick Wong. Generlized vector space model in information retrieval. *ACM*, 1985.

[14] M. S. Xiaoying Tai and Y. Tanaka. Improvement of vector space information retrieval model based on supervised learning. *ACM*, 2000.

[15] S. W. Z.W.Wang and Y. Yao. An analysis of vector space models based on computational geometry. *ACM/SIGIR*, 1992.