DISSERTATION

Vector Space-driven Service Discovery

ausgeführt zum Zwecke der Erlangung des akademischen Grades eines
Doktors der technischen Wissenschaften

unter der Leitung von

Univ.-Prof. Dr. Schahram Dustdar
Institut für Informationssysteme
Abteilung für verteilte Systeme
Technische Universität Wien

eingereicht an der

Technischen Universität Wien
Fakultät für Informatik

von

Dipl. Ing. Mag. rer. soc. oec. Christian Platzer

christian@infosys.tuwien.ac.at

Matrikelnummer: 9825498
Vogelsanggasse 18/10
A-1050 Wien, Österreich

Wien, Jänner 2008

Kurzfassung

Das zugrundeliegende Themengebiet dieser Dissertation umfasst einen großen Teil des Web service Paradigmas und dessen Anwendung in heutigen verteilten Systemen. Das Hauptaugenmerk liegt dabei auf dem speziellen Bereich der Suche und Auffindung solcher Dienste. Die damit verbundenen Probleme sind vielfältiger Natur. Zum einen gibt es bis dato noch keine gängige Methode oder einen universellen Ansatz zur Auffindung bereits bestehender Dienste, zum anderen steht ein Großteil der Entwickler vor dem Problem nicht zu wissen, an welcher Stelle sie neu verfasste Dienste publizieren sollen. Da es sich bei der Web service-technologie bei weitem nicht um eine neue Facette des Internets bzw. der auf Diensten basierten Infrastruktur handelt, wurden schon etliche Versuche unternommen, diese Diskrepanzen auf einen Nenner zu vereinheitlichen und somit eine Plattform zu schaffen, die sowohl für Entwickler, als auch für die eigentlichen Konsumenten dieser Dienste eine gemeinsame Anlaufstelle bietet. Viele dieser Versuche sind gescheitert, manche zur Gänze, weil es sich um nicht ausgereifte Konzepte handelte, manche nur teilweise, weil trotz eines durchdachten Konzeptes und einer entsprechenden Architektur die Akzeptanz in der Dienst-orientierten Gemeinde fehlte.

Im Rahmen dieser Dissertation wurden diese Probleme analysiert und ein Konzept entwickelt, das für alle Bereiche der oben erwähnten Gruppe eine Lösung bietet. Des weiteren wird VUSE vorgestellt, ein an der technischen Universität entworfener und implementierter Prototyp, der zur Umsetzung dieser Konzepte und schließlich auch der Auswertung der produzierten Ergebnisse dienen soll. VUSE stellt im Kern eine Suchmaschine dar, die sich des Vektorraum-Prinzips zur Ähnlichkeitsbestimmung mittels Winkeldistanz bedient, um performant Ergebnisse auf Suchanfragen zu produzieren und dabei trotzdem die Fähigkeit behält, auf mehrere physisch getrennte Bereiche aufgeteilt zu werden.

Im abschließenden Teil dieser Arbeit wird zudem eine Evaluierung der vorliegenden Forschungsergebnisse, sowie ein Ausblick auf weiterführende und themenverwandte Forschungsgebiete gegeben.

Abstract

The main topic of this thesis includes a large section of the Web service paradigm and its application in todays distributed systems. The main focus lies on the very specific issues of searching and discovering those services. The arising problems are of various nature. On the one hand, there is no universal approach or established method for discovering existing services to date. On the other hand, software developers are confronted with the problem where to publish services they implemented. Due to the fact that Web service technologies are not a new facet of the Internet or the service-oriented infrastructure respectively, many approaches to find a common ground for those issues have already been proposed. They all aim to create a common platform, where both, developers and service consumer can find a common contact point. Many of those approaches failed simply because their concepts where not mature enough, but others failed, because even though the concept was sound and feasible, they lack the acceptance in the service-oriented community.

In the course of this thesis, these problems were analyzed to produce a concept that provides a solution for the problems mentioned above. Furthermore, VUSE is presented, a prototype developed and implemented at the Vienna University of Technology, which is designed to realize these concepts and finally, to evaluate the produced results. The principal conclusion of VUSE can be seen as a search engine which is based on the vector-space model to rate similarities with angle distances and create high-performance results for search queries while still maintaining the ability to be separated into several physical locations.

In the final chapters, an evaluation of the research results is given, together with an outlook on related research fields and ongoing work.

Acknowledgements

For the excellent supervision of this thesis and the mentoring during the different phases of work, my personal thanks apply to Schahram Dustdar. The liberty to choose a research field and direction without restrictions and still receive valuable input is nothing that can automatically be taken for granted. Therefore, it is valued all the more. It was a pleasure to have an advisor who understands the restrictions a family can have on the manageable workload.

Sincere thanks to my family. Without your help, this would not have been possible.

Christian Platzer
Vienna, Austria, January 28, 2008

For Lisi and my family

Contents

List of Figures

List of Tables

Chapter 1

Preface

> Humor can be dissected as a frog can, but the thing dies in the process and the innards are discouraging to any but the pure scientific mind.
>
> E. B. White (1899 - 1985)

1.1 Contribution

Some parts of the work that form the major contribution of this thesis are published in form of articles, conference proceedings or book chapters. Here, the major components and their relations to the publications are introduced. Especially in conference proceedings, the limited space often requires an author to shorten the presented work to an acceptable amount. Fortunately, this limitation only partially applies to this thesis. Therefore, the introduced concepts are explained in greater detail and with more focus on the implementation and evaluation than in the underlying publications.

The first and most important manuscript was published as an IEEE conference proceeding for the *third European conference on Web services (ECOWS)* [49]. This publication introduces the vector space search engine which will be presented in Chapter 2 of this thesis. In addition to the paper, this thesis will present a detailed discussion of the implementation procedure as well as an evaluation of performance related aspects.

The second official publication appeared as a book chapter in *Securing Web Services: Practical Usage of Standards and Specifications* published by *Idea* [50]. The chapter comprises an overview on Quality of Service parameters for Web services and how to use them for service discovery and monitoring. These concepts were extended and are included in Chapter 3 of this thesis.

The statistical approach for service classification, as it is presented in Chapter 4 was submitted for publication to the *ACM Transactions on the Web (TWEB)* Journal in August 2007. With the completion date of this thesis however, no decision about the acceptance of the manuscript was available.

Other publications [1,29], were not used directly within this thesis but deal with related problems and issues. They are cited accordingly.

1.2 Overview

As a general introduction to the topic of this thesis and to give the reader an idea of the involved technologies, this section deals with some of the basic principles of Web services (WS) and the service-oriented architecture as a whole.

Web services in general are basically not a new technology. They essentially grew with the evolution of the World Wide Web and are still strongly related to it in certain respects. The first Web services in a more narrow sense were introduced by the Microsoft Cooperation in July, 2000. Yet, the technology originated from the efforts of many companies that tried to achieve more or less the same goal. Although the philosophy never changed, the evolution of Web services and the involved methods resulted in more sophisticated and convenient technologies. Per definition of the World Wide Web consortium (W3C), Web services are ”software systems designed to support interoperable machine to machine interaction over a network” [8]. This definition is still valid, because it captures two of the most essential properties quite well:

- Web services were designed to be interoperable. This ability is not limited to the mere possibility to connect physically distributed computer systems but also encompasses a certain degree of platform independence. To be more specific, the target was to ensure that Web services can be used, no matter what the utilized programming language is.
- Secondly, the technology was designed to ease the communication between machines rather than interfacing with humans. Nevertheless, the currently established standards aim to provide a human-readable form for both, description of the services and the exchanged messages themselves.

To fulfill all these requirements, the e**X**tensible **M**arkup **L**anguage (XML) was chosen as the enabling technology for Web services. The basic idea behind the XML was to keep various data structures human readable but still provide the flexibility to describe complicated relationships. This flexibility is the main reason why XML became so successful. Listing 1.1 shows a simple example for an XML-based data structure to categorize books.

The result is a markup language that allows to create a bookstore index. The index comprises book titles, authors and even an abstract for the different chapters. To be precise,

```
<?xml version=”1.0”?>
<book title=”Securing Web Services: Practical Usage of Standards
                and Specifications”
       authors=”Platzer, Rosenberg, Dustdar”>
  <chapter title=”Enhancing Web Service Discovery and Monitoring
   with Quality of Service Information”>
     <section title=”QoS Model”>
        This is a description of the Ws−QoS Model.
     </section>
     <section title=”Conclusion”>
        The conclusion for this chapter goes here.
     </section>
      ....
   </chapter>
    ...
</book>
```

Listing 1.1: XML example

XML is not a language in itself but a general-purpose specification to create custom markup languages. This is the point where the XML technology finally intersects with the Web service world. There are two major components for every Web service as they are used today. A description of the service itself and the actual messages. Both are expressed through XML and briefly discussed in the later sections. The elements discussed in this short overview are far from being complete. XML itself consists of a variety of standards and methods for advanced usage. These standards are not mentioned here because of their limited use for this thesis. Should one or more be of particular interest for any of the subtopics, they will be discussed and referenced as detailed as necessary.

1.2.1 WSDL

This section concerns the description of Web services and how it evolved.

The first Web services were nothing more than program functions called over a network. This method is called **R**emote **P**rocedure **C**all (RPC) and is still used as one possible style to invoke Web services as they exist today. Such an RPC can be done by creating an XML message where the name of the function to invoke, along with the necessary parameters are sent to the receiver. Listing 1.2 shows such a simple RPC message.

Even with this very simple example, one of the requirements to Web services becomes clear. The example service is obviously designed to book a ticket for a train to a given destination. The second parameter though could be anything from the number of tickets

```
<?xml version="1.0"?>
 <methodCall>
   <methodName>bookTrainTicket</methodName>
   <params>
     <param>
       <value>Vienna</value>
     </param>
     <param>
       <value><int>1</int></value>
     </param>
   </params>
 </methodCall>
```

Listing 1.2: RPC example

to purchase to the class that should be booked. Furthermore, the user must first know the name of the method that should actually be invoked. To provide the essential information in a convenient and structured way, the **W**eb **s**ervice **D**escription **L**anguage (WSDL) was introduced by Microsoft in September, 2000 [12].

The language is powerful and designed to provide the required capabilities to describe Web Services that are more complicated than simple RPC calls. Listing 1.3 shows a complete WSDL file as it could exist to describe the above example. The most important features and elements are included. The design of this description language is a key element to the work following in this thesis. Therefore, the major elements and conventions are briefly discussed.

Types

The first element begins in line 3 where the `types` element occurs. Just like ordinary type definitions in any program language, they define the structure of the transmitted data. The basic data types in Web services are quite restricted and cover the most important ones like *strings*, *integers* and *boolean* [12]. The possibility to express data types however, is not limited to these types at all. By using custom data types, almost every data type can be expressed, even whole objects. In the WSDL listing, the custom type `TicketOrderRequest` is created as a complex type with two subtypes. This must be done, because the operation to order a ticket takes more than a simple element and, therefore, requires a complex element to be created. In this case the parameters for the operation are a string labeled `destination` and an integer labeled `class`. The labels are created by humans, in most cases, even when they are generated from existing source code. Therefore, they most certainly hold some valuable (quasi-)semantic information about the service. In this case they obviously describe the name of the destination and the desired class to travel.

Messages

To assign the previously defined types to an operation, the `message` element is used. It basically defines which types are contained in either input or output of a single opera-

```
<?xml version="1.0"?>
<definitions name="TicketOrder">
  <types>
    <element name="TicketOrderRequest">
        <complexType>
            <element name="destination" type="string"/>
            <element name="class" type="integer"/>
        </complexType>
    </element>
  </types>
  <message name="OrderTicketInput">
    <part name="body" element="TicketOrderRequest"/>
  </message>
  <message name="OrderTicketOutput">
    <part name="success" type = "boolean"/>
  </message>
  <portType name="TicketOrderPortType">
    <operation name="bookTrainTicket">
      <input message="OrderTicketInput"/>
      <output message="OrderTicketOutput"/>
    </operation>
  </portType>
  <binding name="orderTicketSoap" type="TicketOrderPortType">
    <binding style="rpc" transport="http"/>
    <operation name="bookTrainTicket">
      <operation soapAction=""/>
      <input><body use="encoded"/></input>
      <output><body use="encoded"/></output>
    </operation>
  </binding>
  <service name="TrainTicketService">
    <port name="TicketOrder" binding="orderTicketSoap">
      <address location="http://example.com/ticket"/>
    </port>
  </service>
</definitions>
```

Listing 1.3: WSDL snippet

tion. The messages can include both, simple and complex type definitions. The message element does not define for which purpose the types are used though. Therefore, a message could be used as input and output of an operation at the same time. In line 11 the first message is named `OrderTicketInput` and uses the previously defined complex type `TicketOrderRequest` to define the input message of the operation.

Port types
`Port types` define one or more operations. The example port type in line 17, for instance, defines an operation called `bookTrainTicket` where the previously defined messages are used as input and output parameters. Again, the name of the operation usually reflects its functionality if not intended otherwise. When using the Apache Axis framework [5] for example, the operation names are mapped to the names of the exposed methods whenever the WSDL file is generated. As a result, the names in the description file are bound to hold a certain amount of meaning, which is exploited by the methods used in this thesis.

Bindings
The previous elements would theoretically suffice to describe a service based on the remote procedure call structure. For today's Web services though, WSDL offers two additional description elements. The first is the binding element. Its function is to define which transport and encoding style is used to transmit a message from sender to the destination. Theoretically, a multitude of transport methods are possible, reaching from *http* to email via the *smtp* protocol. In practical usage though, *http* and *soap* are the most established methods. Apart from the transport protocol, the binding also defines the style of the message, being either RPC as in line 24 of the example, or document/literal style. The difference between them is mainly of conceptual nature. RPC style sees Web services as the original remote procedure call, while document/literal just concentrates on the message. Currently, programmers are encouraged to use the doc/lit style because of its better ability to adept to a changing program interfaces.

Services
The last important element of Web service descriptions is the *service* specification itself. It essentially assigns `port` and `binding` to a specific service and gives the exact address location of the service. Line 31 of the example shows that the service is bound to `http://example.com/ticket`. Although it is possible to define more than one texttservice within a single WSDL file, it is uncommon to do so. Multiple service tags could be used to define multiple endpoints for a single service or to classify functionality according to the their URL targets. In practise, however, WSDL files are usually used to process code stubs for the service or dynamically at runtime. In both cases, multiple `service` tags can only be handled by considering them as individual elements. Therefore, the same result can be achieved by defining separate WSDL files.

This completes the short introduction of Web service description files and the most important elements. Some additional notes worth mentioning concern semantic descriptions in WSDL files. The observant reader may have already guessed that the introduced elements are still insufficient to describe what a Web service is exactly capable of. In the end, how can a consumer of the example Web service be sure that the operation called `bookTrainTicket` really books a train ticket? The answer is simple: The consumer can't. And that is the reason why semantic Web services [9, 39] are still a heavily investigated research field. The idea is to add a resource description for each element with a semantic meaning. How to do this, is already defined in the resource description framework or *RDF*, for short [63]. By adding RDF descriptions to an existing WSDL file, it is possible to define what the operation exactly does. Such a description can be created by using SAWSDL (**S**emantically **A**nnotated **W**SDL), for example. Some major drawbacks of semantic descriptions, however, keep them from being used by today's programmers. Some of the reasons are:

- RDF descriptions are always bound to an ontology [64]. This ontology provides a set of hierarchical notions of a specific topic and their relations to each other. The vast amount of possible items forces researches to create a separate ontology for each domain. Therefore, when dealing with a semantic description of any kind, the corresponding ontology must be available. Furthermore, this description certainly reflects the ontology programmer's understanding of a certain topic. If another programmer decides to assign a slightly different meaning to a certain notion, it most often results in different ontologies for the same topic.

- Secondly, the developer creating a Web service not only has to deal with the technical implementation of the service but is also required to assign the right semantic meaning with the right syntax. If the coders decide to omit those descriptions because they are busy with debugging issues or adding new features, the whole concept is bound to fail.

- Another problem concerns the level of detail, a domain-specific ontology can reach. In the previous example, it would be possible to mark the `bookTrainTicket` operation with its corresponding RDF markup. The ontology could possibly be created by an initiative of the traveling-sector. But what if the same operation that takes a string and an integer as input is designed to encrypt the given string with a random encryption algorithm? In this case it is quite impossible to define the exact meaning of the operation by using RDF tags. It would either be impossible because of a limited depth of the ontology or simply result in a description so complicated that it resembles the original source code and would therefore be too hard to read or process automatically. Instead, publishing the original source code would be easier.

Because of this and various other reasons, semantic descriptions are not widely used today. They are more seen as a tool to support semantic-based research dedicated to this particular topic. Nevertheless, the possibility to describe functional semantics of Web services is considered an important issue and, therefore, treated in this thesis as well.

With service descriptions being completed, the last element of the Web service communication stack, the message itself, can finally be discussed.

1.2.2 SOAP

Stemming from XML-RPC as well, SOAP was originally introduced as the **S**imple **O**bject **A**ccess **P**rotocol [24]. The acronym though, is no longer in use since SOAP reached version 1.2 and became a name on its own. This technology is the final link between Web service provider and consumer because it specifies how the messages for Web service requests and responses must look like. To stay true to the original example, listing 1.4 shows the necessary XML code to order a first class ticket to Vienna.

```
 1 POST /ticket HTTP/1.1
 2 Host: example.com
 3 Content-Type: application/soap+xml; charset=utf-8
 4
 5 <?xml version=”1.0”?>
 6 <soap:Envelope
 7 xmlns:soap=”http://www.w3.org/2001/12/soap-envelope”
 8 soap:encodingStyle=”http://www.w3.org/2001/12/soap-encoding”>
 9   <soap:Body xmlns:t=”http://www.example.org/stock”>
10     <t:bookTrainTicket>
11       <t:destination>Vienna</m:destination>
12       <t:class>1</m:class>
13     </t:bookTrainTicket>
14   </soap:Body>
15 </soap:Envelope>
```

Listing 1.4: SOAP request

In contrast to the previous examples, the presented listing is complete and functional with all namespaces and the required HTML header. Beginning with Line 1, the first three lines shows the header elements where the receiving host is defined. This information is derived from the WSDL's `<service>` tag. It basically says that the following message is posted via the HTTP protocol just like any ordinary Web page. The receiver's Web server then decides how to further handle the received XML request. In most cases, the Web server will extract the contained XML message and deliver it to a deployed Web service engine like Apache Axis. The actual content starts with the `<?xml>` tag as usual, followed by the aforementioned namespace declaration in line 6. Namespaces are necessary to precisely define the meaning of an XML tag. Here two namespaces are defined, namely `soap` for the SOAP protocol itself, and `t` for the names used in the WSDL file. In all subsequent lines, the XML tags are marked with their corresponding namespace to guarantee an exact assignment of the tags. The `<destination>` tag for instance, is labeled as an element of the WSDL file but it could have a different meaning in the soap protocol.
The elements belonging to the *t*-namespace are finally responsible for the actual request. They define the operation (`bookTrainTicket`) to call as well as the necessary parameters

to do that. If the request was sent successfully, the receiver sends back an answer in the same style. A possible answer for this request is depicted in Listing 1.5.

```
 1 HTTP/1.1 200 OK
 2 Content-Type: application/soap+xml; charset=utf-8
 3
 4 <?xml version="1.0"?>
 5 <soap:Envelope
 6 xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
 7 soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
 8   <soap:Body xmlns:m="http://example.com/ticket">
 9     <m:bookTrainTicket>
10       <m:success>true</m:success>
11     </m:bookTrainTicket>
12   </soap:Body>
13 </soap:Envelope>
```

Listing 1.5: SOAP response

Just like the request, the response starts with an HTTP header before the actual content starts. According to the definition of the WSDL file, the `bookTrainTicket` operation returns a boolean value named `success` and this is exactly what is done in line 10. Here, a drawback of the Web service technology becomes obvious. The whole response serves the reason to transmit a single boolean value that can be expressed in one bit only. By using HTTP, XML and SOAP for transport and protocol, the actual message contains 459 bytes, which is 3672 times the needed size. This example is, of course, an exaggeration because Web services do not transfer information bit by bit. In some cases though, the used protocols cause an immense traffic overhead to achieve the desired flexibility and ease of use.

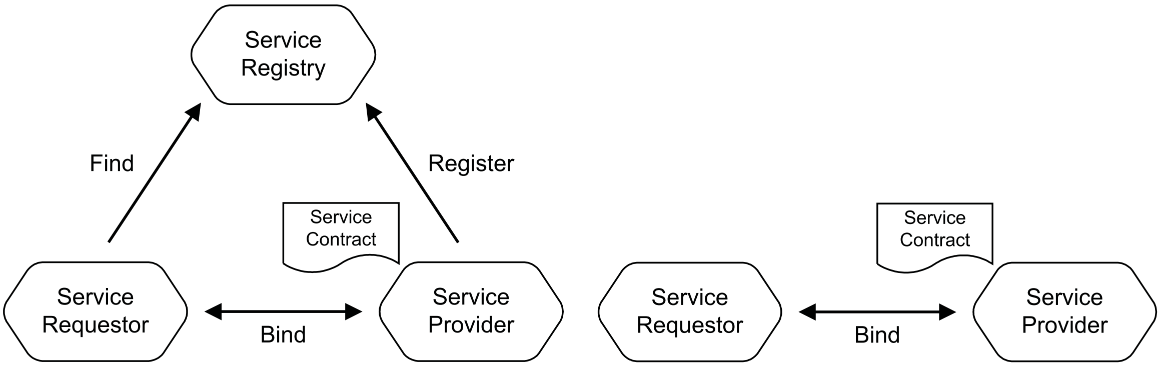
1.3 Motivation and Problem Definition

The overview in the previous section describes the most important functional elements of Web services and their implications of both, caller and provider. To properly describe the problem this thesis tackles, a third non-functional party has to be introduced first. The service registry.

1.3.1 The SOA triangle

During the last few years, **S**ervice-**o**riented **C**omputing (SOC) [46] has become an attracting research area. Not only because of the implied capabilities but also due to some major problems arising with the inital thought to create a loosely coupled infrastructure by using the Web service technology as an enabler. **S**ervice-**o**riented **A**rchitecture (SOA) was meant to be a means to provide an architectural model for developing service-oriented applications. In the last few years, Web services evolved from the RPC-centric model to the previously mentioned messaging-based communication model.

The basic SOA model as it was initially created considers three main elements as shown in Figure 1.1(a). The *service provider* implements a given service like shown in Section 1.2. Furthermore, it publishes the service description in a *service registry*. The *service consumer* searches the registry to find a certain service. If found, it retrieves the location of the service and binds to the service endpoint, where the consumer can finally invoke the operations of the service.



(a) SOA Theory (b) SOA Practice

Figure 1.1: Basic SOA Model – Theory vs. Practice

By implementing the SOA triangle, one could gain flexible solutions with respect to manageability and adaptivity of software systems. In practice, however, software systems hardly ever implement the *publish-find-bind-execute* cycle as proposed by the SOA triangle. Figure 1.1(b) depicts the current model as it is used in most of today's SOA applications. The current model solely consists of *service provider* and *service requestor* and, therefore, comprises just the necessary elements to keep the functional level intact. This implies that the service requestor has to know the exact endpoint address of a service and has to generate a proxy to invoke the service without knowing the location of a possibly changing WSDL description. All this is done in a static way which does not conform to the basic principles of service-orientation. Building systems in such a way does not result in easily adaptable architectures and loosely-coupled systems. On the contrary, service providers and service requestors are tightly coupled and unable to automatically react to changes. Modifying the service endpoint address for instance, results in an unrecoverable application error. And this is a disaster from a software engineers perspective. All these arguments strongly suggest to use the initially forseen *publish-find-bind-execute* cycle instead of the static methods described above. In reality, however, it is not popular at all. There are multiple reasons for this fact but almost every single one emerges from the same problem: UDDI.

1.3.2 UDDI

The **U**niversal **D**escription, **D**iscovery and **I**ntegration specification or UDDI for short [44] which is currently available in Version 3.0, closes the SOA triangle and fulfills the function of the Web service registry. The UDDI architecture is objectively seen sound and meets all necessary requirements to function as a service registry. In reality however, the concept comprises some shortcomings that negatively influences the usability of the existing UDDI implementation.

1.3.3 Publishing

The first problem concerns the publishing of Web services. UDDI operates on a data model that supports various capabilities. Since it aims to provide a registry for businesses and corporations in the first place, some elements of the model are specifically tailored to this purpose. The top level element, for instance, contains information about the organization that published the service and is called the *businessEntity*. This information is supposed to be viewed by humans to ease the process of finding a specific company. Furthermore, UDDI provides the possibility to publish a description of a service's business function which is called the *businessService* in the data model. It is hierarchically ordered below the *businessEntity* because a company can provide various services at the same time. A specific service is finally described by *bindingTemplates* where the technical details are defined, including a reference to the service's interface or API. Additionally, UDDI includes a feature to create user-defined taxonomies by using tmodel entries. They are then referenced in the binding template and can be seen as the technical fingerprint of a service.
With all the requirements represented by this architecture, it becomes clear that the problem for publishing Web services in UDDI registries lies mainly in the convenience and simplicity to do so. How this influences real-world registries is shown in 1.3.6 An existing WSDL file cannot just be uploaded to an UDDI directory. Instead, the publisher is required to create a business entity along with the business service for the functionality of the service. Furthermore, the publisher is required to know how and where to enter this information. This required high level of familiarity with the UDDI structure, as well as the discipline needed, produces those unwanted side-effects that lead to the real-world statistics presented in 1.3.6.

- Programmers and software engineers with limited knowledge of Web service technologies are forced to learn at least some parts of the UDDI data model to successfully publish their service descriptions. If they decide the effort not worth the gain, they will simply refrain from publishing their services and send their service description to the designated consumer personally. This results exactly in the unwanted cycle shown in Figure 1.1(b).

- Once a potential Web service provider gets familiar with UDDI, the relative openness of the architecture raises some additional problems. Since WSDL descriptions cannot be published as a whole, tmodel entries are used to provide the necessary information. Unfortunately, the service description in a tmodel key does not necessarily contain the corresponding WSDL file for a service. It could also be a link to a word document or PDF file that describes the service. This fact renders the automatic processing and data extraction from UDDI registries quite complicated.

- Another side-effect is that some programmers will use a trial-and error method to register their services although they don't know exactly how do it. For public registries, they constitute the largest source of disturbance for the registry's content. These "dabblers" are also responsible for some of the problems mentioned below.

1.3.4 Searching and querying

The quite complicated registry structure also poses an obstacle when potential service consumers want to search for a specific service. Depending on where the provider has entered the essential information, the queries can only be submitted on a specific level, e.g., *businessEntities*. Whether the desired service is found depends if the information was entered at the same data structure by the provider. As an example, a company could choose to define a *businessEntity* with `Weather Today` as a caption because they provide some services like temperature and rain forecasts. At the same time there could exist another company that chooses `Meteorology today` because they also provide services for earthquake warnings along with the weather forecast. Searching the business entity for weather service would only reveal the first service while searching at a deeper level would probably also discover the latter.
From the technological point of view, the search and query capabilities are mostly implemented by simple queries, where full text searches are issued upon the local databases. Those queries are quite limited in their capabilities. An additional disadvantage of the UDDI structure is, that WSDL files cannot be queried for their content because they are not available for the registry itself. This means that a query for a specific port type of a service, for instance, is impossible a priori.

1.3.5 Binding

A subject already touched in Section 1.3.3 concerns the binding to services published in UDDI-like registries. With the current setup, a consumer is bound to locate the WSDL file for a service directly at the provider by using the link stored in the corresponding tmodel entry. This method requires the provider to constantly make the service description available without restrictions. On the other hand, it forces the consumer to keep track of the bound services or otherwise loose the endpoint description of the target. If the service provider decides to move the service endpoint or the service as a whole, it is impossible for the consumer to find the new location without querying the registry and hoping to find the right entry again.

1.3.6 Correctness

Another problem that needs to be solved concerns the correctness of the uploaded services. Public UDDI registries as they existed from Microsoft [43] or IBM [27] suffered from an immense inundation of malfunctioning services. During the first phase of this thesis which was conducted in Oct. 2005, the public UDDI registries from Microsoft and IBM comprised about 10000 entries. By using a self-written tool to download links contained in the tmodel section of the UDDI entries, 20% or 2000 entries came along with a valid URL to the supposed description file. From these 2000 links, around 50% or 1000 files where actually reachable and valid and, therefore, downloaded to a local repository. The downloaded entries reached from PDF files to HTML documents that were uploaded as general information about the service but not as a functional description. Out of the remaining 1000 files, only 500 were actually WSDL files which amounts to only 5% of the overall amount of entries in the UDDI directory. The other files where either chunks of existing WSDL files or contained malformed XML content. But it does not stop here. Those 500 files were not always descriptions of functional services. As a matter of fact, the public registries from Microsoft and IBM attracted a large number of unexperienced programmers who published their newly written Web services. Why this poses an enormous problem shall be depicted by the following enumeration of steps that are usually necessary to write a Web service:

1. A Web service capable IDE needs to be installed at a development machine (like, Eclipse, BEA Weblogic, Visual Studio etc..)

2. The code for the Web services has to be written and an entry point defined.

3. The WSDL file has to be generated by defining which methods shall be exposed. In some cases, the WSDL file is generated each time, the corresponding link is retrieved. In other cases, the WSDL file was created before, and a static version is available that has to be updated, each time a change is made.

These steps are straight forward but the third step could easily result in an unusable WSDL file if the IDE is bound to the *localhost* network adapter. As a result, the generated endpoint for the service description looks like `http://localhost:8080/MyWebService`. The problematic thing with this endpoint is that the development machine will never produce an error, because if it tries to bind to the service, it will always find the service. For real users, this is, of course, unusable. They have no means to identify the real location of the published service. In the files extracted from the UDDI registry, about 30% of the URLs pointed to localhost.

Apart from localhost entries, there are various other possibilities, mostly an effect of an improper firewall or network setup. When the Web service engine for instance is not running on the same port as the Web server, it must be ensured that the affected port is forwarded to the right IP address. Another reason of unreachable Web services is of course a simple downtime of the service. Not all implementations are deployed on professional server environments and are, therefore, terminated when the hosting computer is shut down.

1.3.7 QoS and metadata

The last issue that is important in this iteration concerns metadata.
Metadata is a collective term which entitles information about a Web service that does not directly influence its functionality. Some examples for service metadata are

- Performance: How fast can a service respond to a request? What is the service uptime? These categories are entitles as **Q**uality **o**f **S**ervice or short *QoS*.
- Domain specific knowledge: Which domain does the service belong to (e.g., financial, automotive, education, research etc...)
- Location knowledge: Where is the service hosted?
- Technology information: What technology is/was used to implement and deploy the service?

Currently, WS-mex (WS-MetadataExchange) [28] is the only way to attach metadata to a Web service. Some similar approaches propose to attach some information directly to the WSDL file. To do so, they have to extend the existing WSDL specification with some markup to hold the desired information about a service or at least part of it. Such a method, however, bears some disadvantages which are further discussed in Chapter 6 along with other related work.
Another issue besides attaching known metadata is how to retrieve it in the first place. For something like service *cost* it is clear that the service provider needs to define a value. For performance or location information, however, the values are better defined by an independent party to ensure the correctness of the given values an not least the possibility to compare services to each other. This evaluation should, therefore, be performed by the service registry but no standardized way to do so is currently established.

When facing such an enormous amount of unsolved problems it is intelligible why the two largest providers of a public UDDI registry, namely Microsoft and IBM, chose to shut down their sites. In a general FAQ, Microsoft defines the reason for stopping their service[1].

> *Q: Why are IBM, Microsoft and SAP discontinuing the operation of the UDDI Business Registry?*

[1] Source: http://uddi.microsoft.com/about/FAQshutdown.htm, Date: 05.08.2007

> A: The UDDI Business Registry (UBR) was part of the UDDI Project announced in September 2000. The project goals were to define a set of specifications to enable description, discovery and integration and to prove interoperability through a shared implementation of those specifications and provide feedback to refine the specifications through operational experience. The specifications were contributed to the OASIS international standards consortium in 2002. In May of 2003 and February 2005, respectively, the UDDI version 2 and UDDI Version 3 specifications were approved as OASIS standards. The primary goal of the UBR was to prove the interoperability and robustness of the UDDI specifications through a public implementation. This goal was met and far exceeded. The UBR ran for 5 years, demonstrating live, industrial strength UDDI implementations managing over 50,000 replicated entries. The practical demonstration provided by the UBR helped in the ratification of UDDI specifications as OASIS standards and several software vendors now include UDDI support as a key feature in their software products. UDDI registries are being broadly deployed to solve application and service integration challenges.

The basic statement that all goals were achieved and UDDI is primarily seen as a technology for operating in corporate environments can be reckoned as a hint to the shortcomings of the technology when it comes to apply it to the public domain.
Even within businesses, some of the disadvantages cannot be negated and, therefore, demand a solution. The next section discusses how the presented fundamental terms are related to the general notion of service discovery and how they fit into the larger picture.

1.4 Discovery

The term discovery, as far as Web services are concerned, refers to the process of finding those services that match certain computational needs and quality requirements of service users or their software agents. More technically speaking, WS-discovery mechanisms take a specification of certain criteria characterizing a service and try to locate machine-readable descriptions of Web services that meet the search criteria. The services found may have been previously unknown to the requester.

1.4.1 Restrospective

Since Web services were introduced in the new millennium, service oriented architectures had to deal with the discovery problem and it still persists. As already mentioned in the previous sections, the possibilities to describe Web services properly were limited initially. These early services were typically used to achieve a platform independent communication between remote peers, nothing more. This requirement was met with the introduced XML-structured messaging protocol. For the act of discovering those services, however, the actually used protocol made no difference. Services' description files were propagated mostly by artificial means, by sending the file per e-mail, for instance. In some cases, the developer of the Web service also worked on the client-side implementation. For those cases, discovery was not an issue. The required knowledge was directly available.

A proper service description mechanism was only introduced when application developers realized that Web service technology had to be leveraged to a level that obviated the need of service consumer and provider to interact closely with each other prior to using a service. With the definition of WSDL, it was finally possible to describe the interface of a Web service in a standardized manner. The general discovery problem, however, still persisted because no means existed to publish a service description in a widely known index, once the implementation on the provider side was completed. This is where UDDI was introduced. Apart from defining the data models and registry structure presented previously, UDDI was also designed to offer simple search capabilities to help service consumers find Web services. Thus UDDI actually contributed to solve the discovery issue.

As a WSDL description of a Web service interface just lists the operations the service may perform and the messages it accepts and produces, the discovery mechanism of UDDI was constrained to match functionality only. If several candidate services could be found, the service consumer was unable to distinguish between them. Therefore, people felt the need to be able to express semantic properties or quality aspects of requested services as well. But search mechanisms taking into account semantics and quality-of-service properties require richer knowledge about a registered Web service than WSDL can capture.

To complement the expressiveness of WSDL and to facilitate service discovery, DAML-S, an ontology language for Web services, was proposed to associate computer-readable semantic information with service descriptions. Semantic service descriptions are seen as a potential enabler to enhance automated service discovery and matchmaking in various service oriented architectures. For the reasons stated earlier, however, services widely used in practice lack semantic information.

The service discovery process encompasses several steps or layers, by definition. Each step involves specific problems that have to be solved independently. The following list discusses these steps in ascending order, beginning with the most generic one.

1.4.2 Enabling discovery

At a first glance, the act of discovering a service description matching a set of terms characterizing a service, resembles a search processes for Web pages. Well-known search engines like *Google* or *Live* utilize a crawling mechanism to retrieve hyperlinks to Web documents and create an index that can be searched effectively as users enter search terms. A crawler just analyzes a given Web page for hyperlinks and grinds through the tree structure generated by such hyperlinks to find other Web documents. For Web services, or more precisely Web service descriptions, the case is similar except for one major difference:

WSDL files do not contain links to other services. Approaches to write crawlers that search Web pages for possibly published service descriptions will produce very poor results, in general. UDDI registries are just designed to eliminate the need for crawlers or alike. Especially after the two largest UDDI registries from IBM and Microsoft were shut down, the vision of public services suffered immensely. Suddenly the starting point to find a public Web service was lost, leaving the possibility to query common Web search engines for Web services as the only alternative. There are, of course, some other registries but they usually do not implement the UDDI specification, which suggests that UDDI may not be an optimal solution for public service registries.

In a corporate environment, however, the initial discovery step is not a real issue. Web service descriptions can easily be published on an internal Web page or in a UDDI registry and are, therefore, easily accessible from within the institution.

1.4.3 Searching in service registries

Assuming that a comprehensive collection of service descriptions has already be established, the question is how to retrieve the closest match to a user query in an efficient manner.

Today, searching is usually performed by humans and not by software automatically. The challenge is how to create an index of services such that the addition and retrieval of service descriptions can be achieved accurately and fast. Common information retrieval methods are often used for this purpose, ranging from the vector space model for indexing and searching large repositories to graph theoretical approaches for fast processing of a rich data collection.

More or less every registry-based solution encompasses such a facility. Especially UDDI registries often come with the mentioned rudimentary interface to query the database for contained services. Unfortunately, the general structure of UDDI with its tmodel component-layout, which serves as a container to store detailed service information, complicates data retrieval. Furthermore, most UDDI entries do not maintain a complete service description but include links to such descriptions kept elsewhere. As a result, UDDI queries are usually processed on the business data related to a service and not the service description itself. This fact alone limits the usability of UDDI as a discovery mechanisms enormously or more precisely: it leaves most of the index quality in the hand of the users.

This area on the other hand is heavily investigated throughout the research community and several approaches have been presented that aim at improving search capabilities on service collections. Those approaches are mostly designed to handle natural language queries like "USA weather service" and are supposed to provide a user interface for various registry implementations. This particular field is one of the the main targets of the concepts presented in this thesis.

1.4.4 Querying repositories

A more detailed form of search in service descriptions is entitled querying. Unlike direct search, in which a user simply provides a set of search terms, queries are formal expressions using some sort of query language. In the case of relational databases, the Structured Query Language (SQL) is typically used as a syntax. Through a query expression it is possible to search for a specific service signature in a set of descriptions [1,29]. Assume, for example, that a user wants to find a weather service and can provide three bits of information: country, zip_code, and the desired scale for presenting the temperature. Assume further that the user wants to express certain quality requirements. Then, a query expression in a language alike SQL might read as follows:

SELECT description FROM services s WHERE
s.input.COMPOSEDOF(country AND zip_code AND useCelsiusScale)
AND s.response_time < 200ms AND s.downtime < 1%
.

This example also reveals a weakness of service descriptions as their signatures are usually not specified using exact type information such as *city* or *country* but rather basic types like string, integer etc. are used. Hence, it seems more appropriate to search for signatures in terms of basic data types only. But this would likely result in mismatches. Re-considering the query above, a corresponding signature using the basic types [string,integer,boolean] can easily be met by other services. There is no way to distinguish positive matches from unwanted ones without additional information or a richer index. These problems are addressed by introducing semantics and domain knowledge. For the values of response_time and downtime on the other hand, an approach to measure service performance is needed.

1.4.5 Domain-specific knowledge in service descriptions

Another requirement that has to be met by more powerful discovery mechanisms is domain-specific knowledge about a service. To take on the sample above, a discovery mechanism which is able to match the terms city, zip_code and temperature with the semantic categories location and weather would select just the intersection of services dealing with location and temperature. Although domain information is semantic information in certain respects, it does not mean that the information has to be provided upon service registration. A grouping can, for instance, be achieved by using statistical cluster analysis and discover strongly related service descriptions.

On the other hand, domain-knowledge can also be gained by letting the service provider add this information. In practice, however, it proved to be problematic to let users define semantic information for a service. Once, this is due to the fact that a certain amount of domain knowledge is needed by the programmer of the Web service but mostly because the categorization assigned by indexers cannot be validated and could, therefore, be incorrect. This field, just like the following, is still heavily investigated, e.g., under the heading "faceted search". It addresses a broad spectrum of issues but also bears a high potential for innovation.

1.4.6 Quality-of-service properties

The consideration of quality-of-service (QoS) properties in discovery attempts requires the definition of scales of measurements and metrics to qualify the properties per domain. The scales can be of different kinds including nominal, ordinal, interval or ratio. They are used to assign appropriate QoS property values to a service. Here, a service provider has the choice to associate precise values or just value ranges with service property descriptions. The metrics are needed to rank services that match the functional and semantic requirement of a search according to their degree of fulfillment of required QoS properties.

1.5 Requirements

Each of the presented layers of the discovery problem comes with its own set of challenges. This section briefly introduces the three major elements of this thesis and how they contribute to solve these issues. A separate chapter is dedicated to each topic that was treated in detail. To round up the thesis and to provide the means to properly verify the proposed methods, an implementation is provided, that servers as both, a proof of concept as well as a framework to make sure the theoretical designs are feasible and realizable.

1.5.1 Leveraging search and discovery

The first and most important part of the contribution is to come up with an adequate method to index and search service repositories. The desired outcome is a search engine where natural language queries can be processed without any expertise knowledge from the user side. This method must meet several requirements, some of them being performance, scalability and distribution capabilities. These requirements are partially responsible for the failure of the original SOA triangle and must therefore be met to provide a better way for service discovery than it is embodied by UDDI-like registry setups.

Furthermore, the search and indexing concept is a crucial foundation for subsequent tasks and, therefore, have to provide a certain level of openness. A search engine that can be credited with the expected level of contribution must at least provide the following features:

- It must provide the necessary performance to search thousands of services in adequate time.
- It must be able to process natural language queries like "united states weather service".
- The index must be built by using WSDL descriptions only. Any additional information entered by the user must not be obligatory. Should such information be provided, nevertheless, it must be used to enhance the precision rating accordingly.
- The concept must be scalable to offer the chance to react to growing repository sizes. Even if this is not a crucial issue at the moment because publicly available services are limited, any concept not coping with growth issues is bound to fail sooner or later.
- A means to create a federation of distributed instances of such an engine must be available. Otherwise no possibility to link public or private registries and, therefore, access larger amounts of data would exist.

Including all these requirements in a single concept is a challenging task. Out of the already existing methods to actually implement such a search engine, the vector space method proved to be promising and was chosen as the enabling technology. For the requirements that cannot be met by this technology, a solution will be provided in the following chapter, tailoring the vector space method to the particular demands of Web service search engines.

1.5.2 Enhancing index quality

Secondly, this thesis will cope with the problem of missing connections between Web services. After conducting a search, the results are rated according to the relevance to the query. How and if the results are interconnected is a requirement beyond the searching functionality. Part of the contribution lies in a statistical approach that aims to discover relationships between indexed services based on the domain they belong to. To do so, a modified cluster analysis algorithm is applied that is able to operate on the vector space data structure. Again, the approach will be designed to fulfill the requirement to operate with the provided knowledge of the WSDL description only.

1.5.3 Generating metadata

The third part contributing to the issues implied by today's service oriented architectures is the possibility to extract metadata information from service descriptions. The main contribution here is the possibility to do so without directly accessing the machine which hosts the questionable Web service. Three main categories of metadata should be generated.

1. **Quality of Service attributes.** A very challenging part of this thesis but also rewarding in terms of contribution to this scientific field is the generation of QoS values for an existing service. Many approaches, mainly from the area of Web service testing and performance evaluation deal with this issue on a server side approach. This way it is possible to define accurate values especially for performance-related categories like response time, uptime or throughput. This work, however, focuses on a purely client-side approach because for registry-like infrastructures, access to the implementation or the Web service engine is always restricted or denied entirely. Furthermore, this approach features a more realistic view to performance-related aspects of a service since the measures are taken by the consumer and not by the provider.

2. **Domain specific knowledge.** Another distinguishable category of metadata is domain information. The target here is to give information about the domain a given service covers with its functionality. Doing so without requiring a user to give the information on registration time is difficult. An assortment of fuzzy but automatic methods operating on the vector space models should provide a solution to this requirement.

3. **Location data**. The third treated category of metadata is information about the location of the server that hosts the concerned service. For this purpose, the service endpoint is exploited where the corresponding IP address can give hints on the regional settings of the Web service provider. This information can be used to group regional elements based on their location information.

With this section completing the introductory part of this thesis, the following chapters will explain in detail, how all these requirements are met and how the introduced concepts are finally implemented in the research prototype. To round up the contribution and of course to provide the means to properly verify the proposed methods, an implementation is provided, that servers as both, a proof of concept as well as a framework to make sure the theoretical designs are feasible and realizable.

1.6 Structure of the Thesis

The remainder of this thesis is organized as follows: Chapter 2 presents the core element of the vector space model. The first sections deal with the general concepts of search engines based on this technology. Some adoptions to allow an application of this method to data structures emerging from WSDL descriptions, are necessary. In the further course of the chapter, the approach is extended to work with distributed repositories along with the necessary mathematical background for this purpose.
Chapter 3 discusses the possibilities to extract metadata from unknown Web services based on their service descriptions only.

Chapter 4 introduces a modified cluster algorithm to produce the desired clusters for both, a better index quality to enhance query processing and the possibility to categorize Web services based on their respective domain.
Chapter 5 mainly discusses the introduced prototype implementation and how the theoretical approaches were actually put to action.
Chapter 6 positions this work among related approaches and deals with some of the necessary premises to understand how the introduced concepts are connected to already established methods for scientifically related areas.
In the final chapter, a conclusion for this thesis is presented along with an outlook on future research topics.

Chapter 2

A Vector Space based Search Engine for Web Services

Critics search for ages for the wrong word,
which, to give them credit, they eventually find.

Peter Ustinov

2.1 Introduction

The basic idea behind the vector space approach is a combination of common information retrieval methods and existing standards for the description of Web services. As already mentioned in the previous chapter, WSDL and UDDI [15] 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 major drawback when describing Web services on a semantic level is that with increasing possibilities to describe a method, the complexity of the used ontology or description language rises equally.

Instead of introducing a new language or ontology to describe general Web services, a closer look at the existing information and how to use it as thoroughly as possible is provided here. This chapter will deal with real world examples without simplifications. A Web service description in general, contains a certain amount of information, entered by the programmer. This information is some form of natural language in most cases because it comprises non-functional elements of the description or at least those a programmer is free to choose. The method names are a very good example. Most software engineers tend to name their functions or methods according to their functionality like "+getMaximumInteger()" or "searchByString()". Experienced coders will also stick to some form of naming convention when they entitle their methods, like upper-lower case partitioning for *Java* or dashes for *.NET*. Furthermore, most descriptions contain some sort of comments intended for human readers. The 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. As already mentioned, a search mechanism common in modern information retrieval systems: The **V**ector **S**pace **M**odel (VSM) [69] is utilized for this purpose. 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 into keywords. Each of these keywords constitutes a dimension in an 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. These are the driving arguments why the concept was chosen as the enabling technology for this thesis.

2.2 Premises

This research is driven by the same idea that drives the whole Semantic Web services community [9]: 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 researchers 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 [39].
In this particular case however, the target is 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 her own Web site should be able to enter a query like "Google search service" and get the corresponding WSDL file for the Web service. This involves a certain amount of natural language processing.

Increasingly, research currently describes the functionality of WS by artificial means [14]. Although such an approach may look very promising, some additional problems arise with the introduction of an ad-on to an existing standard:

- 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. This fact rules out the possibility to make changes to description files once they are received by the search engine.
- The second, and 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.
- 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. Critics argue that semantic annotations will be used to influence search engines such that they produce higher ratings in cases where it is not even a rudimentary match.

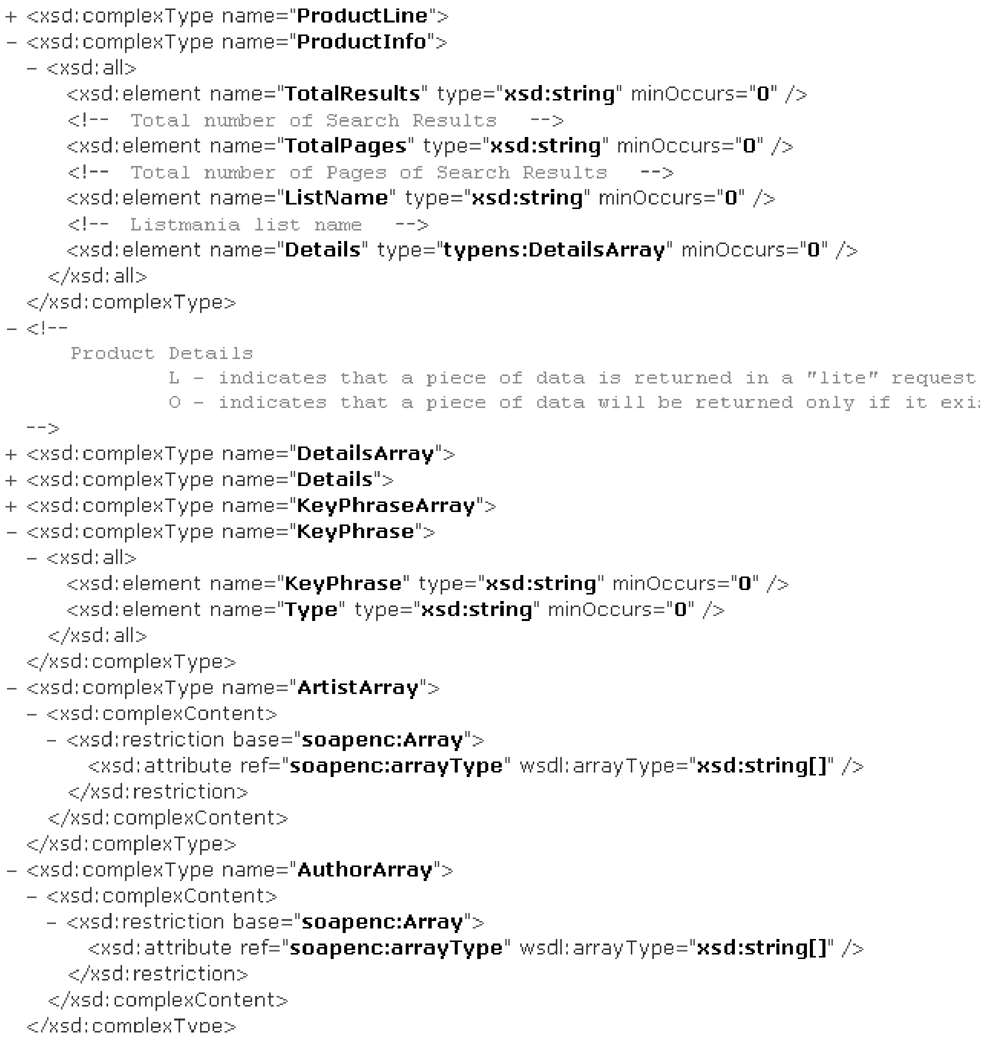
Given the problems stated above and the requirements already given in Chapter 1, an investigation of the possibilities to enrich Web service descriptions with information about what the services do is necessary. In this first part, the goal is to use the available information without adding new restrictions or requirements. Before discussing the approach in detail, a closer look on the information that is already provided in today's repositories is necessary.

2.2.1 WSDL example

Whenever a Web service is published, the 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 schema file for the Amazon Web service[1], which is listed in Figure 2.1.

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. In the above example such a description was added to explain how a certain complex type works and what the parameters mean. The challenge is to exploit all this information at a maximum level.

[1] Source: `http://soap.amazon.com/schemas2/AmazonWebServices.wsdl`, Date: 22.09.2005



```
+ <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 2.1: Amazon Schema Snippet

2.2.2 Web service Discovery

UDDI registries were already partially discussed in Section 1.3.2 where it was stated that they are designed as the central point to register Web services and to make them publicly available.
The list of flaws that limit the usability of UDDI registries can even be extended. From the search engine's point of view, the following problems are added to the list:

- Current 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 registry. As a result, a good deal of the registered services are entered for testing purposes only. This issue is strongly related to the one presented in 1.3.3. When looking at the Microsoft public registry for example, there was a multitude of entries where the port address is entered with a localhost endpoint like `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 do not care if the generated code is correct.

- The third issue not discussed before concerns openness. Some of the UDDI registries available in the past, required some type of subscription, before they accept any files to be published to their database. The IBM UDDI Business registry [27] 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 automatically accessible all the time. In corporate environments however, it can be assumed that most entries are correct or updated with correct values because it negatively influences the productivity if services are non-responsive or malfunctioning.

To summarize, there are two possibilities. Either a Web service 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 Data accumulation

Before discussing the technical details of this approach, a proper repository is necessary to give an appropriate idea of what real word Web services offer in their descriptions.
Retrieving enough WSDL files from the Internet to form a satisfying repository is still a particulary hard task. An investigation upon the possibilities to obtain a set of description files showed three possibilities. The main prerequisite was to only use "common" means of accumulation and not some research prototype or a single source like woogle [16] 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. It was possible to retrieve 62 valid WSDL files this way. This method is of course not intentionally used by users of Web services. It can be assumed that their appearance is merely a coincidence because the files happen to be located in a shared folder.

- **Web Crawler:** Although Web crawlers look very promising at first, it quickly becomes obvious that this method is a very poor way to obtain a repository of decent size. 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. Even for those services that offer the possibility to retrieve a generated version of the description online, there has to be a link somewhere in order to get discovered by a Web crawler. Over 2 Gigabyte of Web traffic was produced by a *wget*-based crawler that processed various domains, before a single WSDL file could be retrieved.

- **tModel cross references:** The third method produced good results compared to the first two. By iterating through a UDDI registry, it is possible to retrieve links to service endpoints and description files. Basically the links from UDDI registries are extracted and afterwards downloaded to a local repository if possible. See Section 2.6 for a detailed description of the extraction process.

- **Direct Web Service Interface** The fourth method is increasingly accepted among today's registries. By providing a SOAP interface to query the underlying database, those implementations of Web service registries offer a trivial but convenient way to retrieve the desired data. An example is represented by the XMethods [2] registry. A small drawback of such a method is that the XML service descriptions must either be serialized to fit a single string type which, in turn, often results in wrong character encodings, or they require the implementation of an additional WS-standard like WS-attachment. The reason why character encodings might get mixed up is based on the fact that the encoding of the whole XML file depends on the SOAP implementation of the engine. Hence, it is possible that the XML envelope is encoded with the latin-1 character set, while the transferred string itself complies to UTF-16 and contains Chinese characters which cannot be properly transferred by the XML envelope.
  Therefore, the best method when relying to a Web service interface is to transfer just URLs of the original service descriptions like it is done at Xmethods.

To be accurate, a fifth 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 the research prototype, is about the same as a UDDI registry and poses a trivial solution, so it was not mentioned in the listing above.

2.4 Search engine

The key element is not the data itself, but rather 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 and natural language processing. It is useful to take a look at available

[2] http://www.xmethods.net/

search engines to get an idea how a possible solution looks like.

As an outcome of this research, an efficient search engine for Web services will be created. This engine has to be capable of handling existing WSDL files and convert UDDI entries to the local structure at the same time. Furthermore, it must be possible to set up multiple engines at different locations and join them to one repository.

In terms of coverage, this part deals with the first two parts of the contribution mentioned in Section 1.5. An algorithm which allows to join detached document repositories to a single one and execute queries upon the resulting vector space will be introduced. Furthermore, the first element of the proof-of-concept prototype implementation will be presented.

2.4.1 Architecture

The basic use of the Vector Space Model presented in the following sections 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 2.2 for a visualization of the concept. It shows how a service description is routed through the engine and how a response to user queries is created.

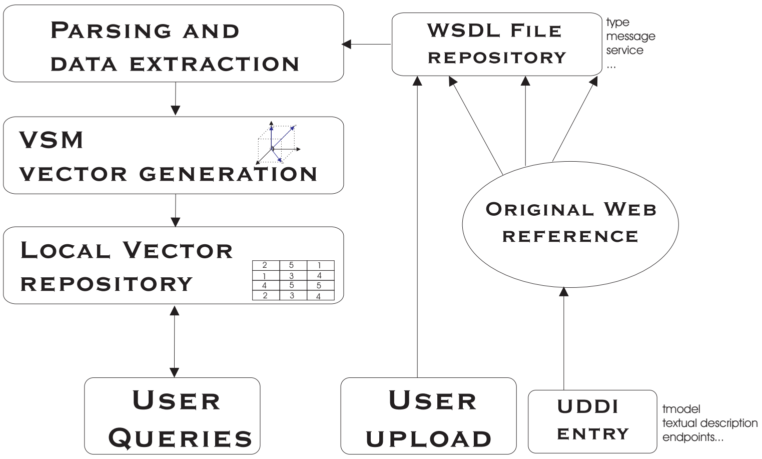


Figure 2.2: Basic architecture

This architecture allows the creation of a localized search engine. To take the concept one step further, it is made possible to 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, the existing approaches had to be extended, while a new algorithm for the processing phase had to be introduced.

2.5 The vector space concept

The **V**ector **S**pace **M**odel (VSM), as proposed by Salton [54] was basically designed for various applications where a fast search method is needed. This section tackles some important elements of the concept but a thorough explanation of the topic is given in Section 6.1.

2.5.1 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 i is represented by a term t_i [65]. This space can grow in dimension every time a new keyword is added. Shrinking it is only possible by deleting keywords from the vector space and, therefore, reducing the dimensional size.
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 [65]. The importance can be expressed in several ways. For ordinary documents it will most probably be the number of occurrences of a term in one document. 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 [61]:

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

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. They form a trivial n-dimensional vector space with two values for each characteristic. This fact alone limits the field of application enormously, because most data is not processable in binary form. However, if it is possible to represent the underlying data structure with binary weights, it positively effects retrieval speed and should therefore be used.

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

The approach discussed in the previous section and more thoroughly in Section 6 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, some of them complicating the concept.
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_4
google	1	1	1
search	1	0	1
result	0	0	1

When 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	0	1
search	1	0	1	1
result	0	0	0	1
service	0	0	1	0

This operation can be done in one access to the remote vector space without any drawbacks because of the boolean nature of the terms. Therefore, binary weighting and vector creation is fit for a distributed environment and thus capable of handling multiple service registries.

2.5.2 Weighting

Binary weighting like presented in the in the previous section will not be sufficient for a sophisticated search engine, especially when the goal is to create a powerful index for WSDL files. 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 [54]. To do so, the values expressing a single dimension are extended to real numbers.

2.5.2.1 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 [25]. Its purpose is to weight terms highly if they are frequent in relevant documents, but infrequent in a collection as a whole. This way, more important terms will produce a higher rating, because they occur in few documents. According to Salton [54], the inverse document frequency is calculated as:

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

Therefore the term weight *tf* x *idf* for any position in the matrix 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 registry 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. This change is performed for *every* vector in the whole term space. After the update, the values for each vector reflect the overall number of documents and the particular weights for each term. This way, inserting new vectors becomes more expensive but the positive effect on queries, which are assumed to build the majority of accesses, is predominating. Unfortunately, this is only possible in a centralized model, because the values for N and n_k are not known for the whole collection. 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 that approach. The following steps have to be carried out, when a new document is added to the repository:

- The raw term frequencies are calculated for the document. For general data applications, this value must reflect the importance of the current characteristic without any weighting schemes applied. 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 as part of a vector.
- The values for N and n_k are updated for the collection.

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, thus forming a matrix. 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

$$n_k = n_{k_1} + n_{k_2} \Rightarrow n_{service} = 2 \text{ and } n_{search} = 3.$$

These values reflect the influence of the local term space on the remote term space as far as term frequencies are concerned. To leverage this procedure to the general case of 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

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

Should any document be present in more than one collection, a slightly reduced term weight would be the result. This behavior might not be wanted under certain circumstances, although it is formally correct. Such a problem can easily be solved by introducing hash values for the content of each documents acting as the primary identifier on each instance of the vector space.
The presented distribution scheme does not apply to the comments of service descriptions only. It is used for every weighted keyword.

2.5.2.2 TF X IDF NORMALIZATION:

Moreover, the bare *tf* x *idf* value is not enough, because it rates longer documents higher than shorter ones [72]. 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 [54]:

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

Distribution capabilities are the same as mentioned in 2.5.2.1. There are no additional values required to normalize the weights according to this formula.

2.5.3 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 [69] [16].

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 [72]. In a n-dimensional space, the cosine value between two vectors p and q is calculated as

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

whereas $p \cdot q$ entitles the dot product and is calculated by multiplying term weights of the query- and document vector together [65]. 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 $\sum_{i=1}^{n} p_i q_i$. 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 becoming 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, good results for those components of a service description can still be expected.

One aspect of the chosen method is its linearity. As a result, the form of distribution, presented in 2.5.2 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 this case.

2.6 Implementation

To demonstrate the reliability of the concept and to show how an application for the presented architecture may look like, a prototype search engine was implemented. The application was designed with a Web frontend and made publicly available to offer the possibility to try some of its functions and evaluate the produced results. The Web application can be accessed via the VitaLab implementation[3].

2.6.1 Set-up

To follow the initial idea of a service-oriented approach, the application consists of several parts which are connected via Web services. The implementation of the Web frontend was done with Visual Studio 8.0.50727 using the .NET Framework Version 2.0.50727 SP1 and later expanded to SP2. 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 the client as long as it is capable of handling Internet sites. The frontend was deployed on a Server Blade with 4 logical CPU's (Dual Xeon), 2 GB of main memory and Windows IIS under Windows Server 2003 to deliver ASP.NET Web pages.
The second part of the application was written in JAVA 1.5 and deployed on a Apache Tomcat 5.0 environment with Apache Axis 1.4 as the Web service container. This part contains the search engine itself including distribution capabilities and persistent Memory option. This core component is designed to be deployed on any machine capable of running JAVA and Tomcat to ensure platform independence. The exposed services can

[3] http://copenhagen.vitalab.tuwien.ac.at/VSMWeb/Main.aspx

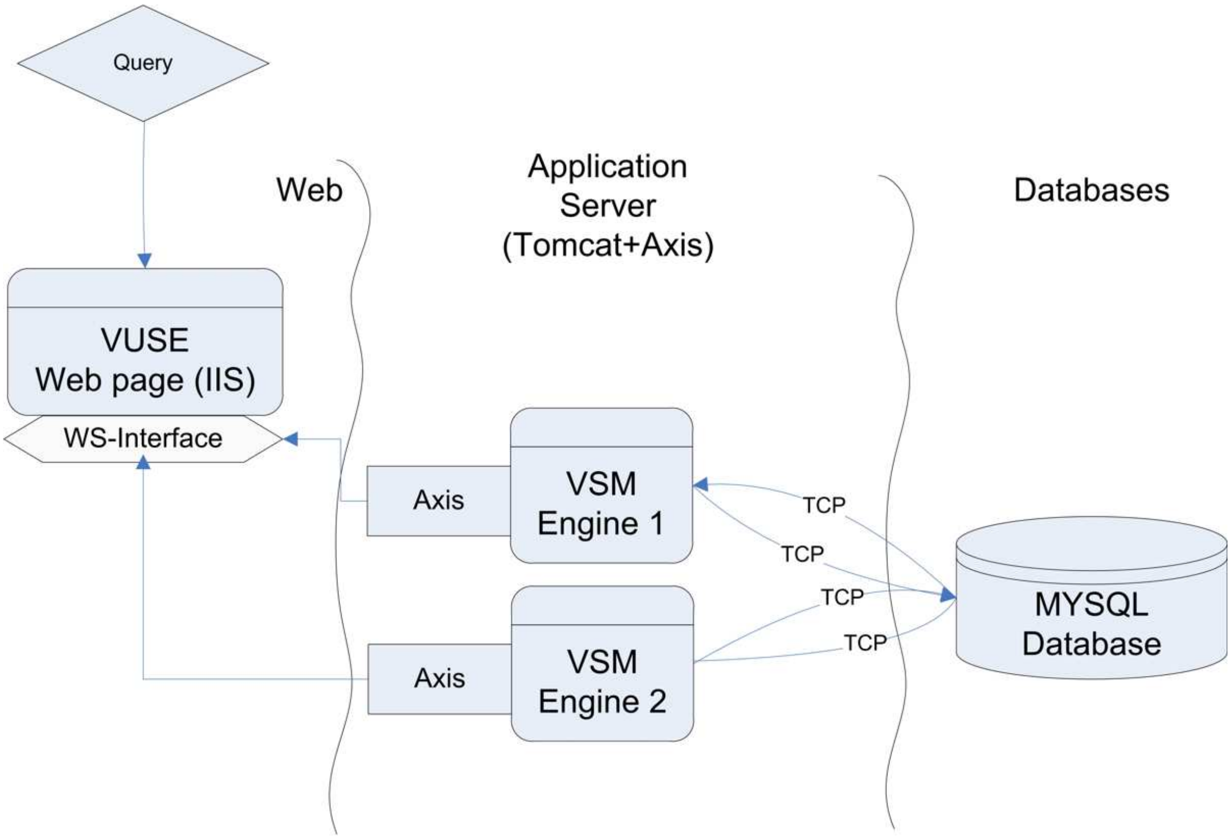


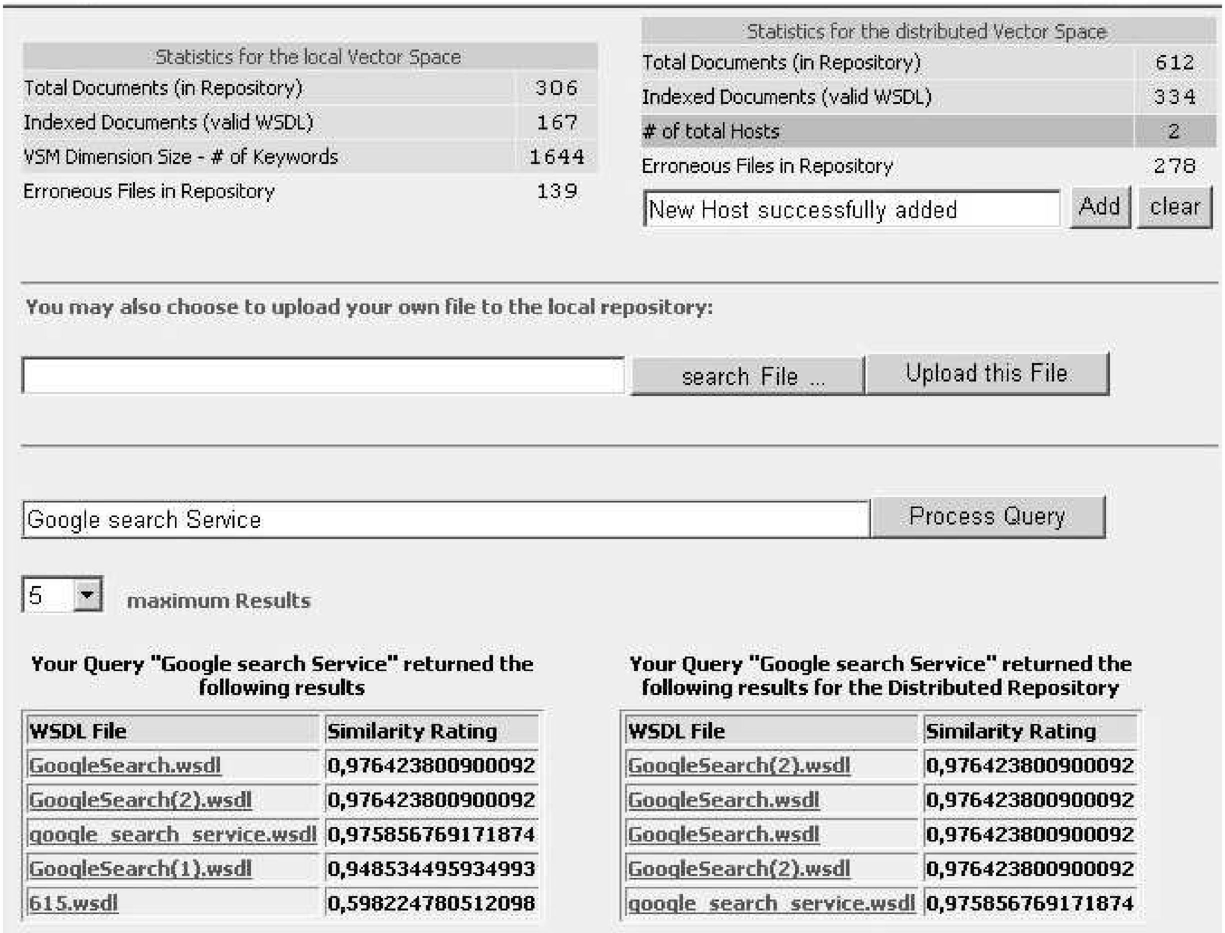
Figure 2.3: Application Overview

furthermore be used to supply other applications with search and index capabilities. The WSDL-related data extraction and processing is done by the front-end part.
Connected to the back-end is a MYSQL 5 Database which is responsible for persisting the vector space information. The search engine can be called with two options. Either volatile, which means that the data is stored in hash tables and, therefore, lost upon server termination or restart, or in persistent mode, where the vectors are mapped to a database. The persistent version is of lower performance than the volatile but it is basically easier to handle. From an algorithmic view, the two versions are equal and follow the same structure.

The application layout is depicted in Figure 2.3 where the interaction between the component is shown. Web service traffic occurs between the Web front-end and the application server where the engine is hosted. The database for the persistent version is connected through a much faster TCP interface facilitated by the JDBC database driver.

2.6.2 Frontend

The frontend and its user interface is deliberately kept simple. It supports file uploads, local queries including partial searches, statistic analysis of the local repository and remote query processing.
To join the local repository with a remote repository, the endpoint of the remote Web service 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 its 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 locally and remotely. The screen shot in Figure 2.4 shows how the result for the joined repository looks like.



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 2.4: Query Results - Screenshot

The original repository contains about 250 WSDL files from the UDDI extraction process along with the files retrieved by other registries like XMethods. This amount varies according to the amount of discovered descriptions during a retrieval phase. Furthermore, the prototype is open for file uploads to test the application with user-defined WSDL files.

2.6.2.1 UDDI downloads

Strictly spoken, the UDDI extraction mechanism is part of the front-end and, therefore, implemented as part of the Web page. Access to it is restricted, since it is not part of the functionality needed by the consumers. As already mentioned earlier, the Microsoft UDDI SDK Version 2.0 Beta was used 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. As already touched in Section 1.3.1, the result was quite poor. The query that was conducted at the Microsoft Public UDDI registry [43] in March 2005 resulted in 6438 parsable URLs.
- In the final step, 30 threads were generated to retrieve the WSDL descriptions contained in the URL and add them to the local repository.

A complete listing of the involved code can be found in Appendix A.1. The first interesting result showed when the URLs from the initial query were iterated. Out of 6711 possible entries, 1272 were actually downloadable files from their original site, which means that 19% of the entries are actually functional. A far lower ratio was expected because the registry is public and free to use for everyone.
546 of the downloaded files were valid XML files and parsed by the keyword extractor, which means that 9% of the original entries are actually WSDL descriptions. The other files were either pdf descriptions or plain HTML files. These percentages kept quite constant for every registry that was parsed. For security reasons, no frontend for the UDDI extractor was implemented. It would be possible to misuse the provided extraction procedures to initiate a denial of service attack for public UDDI registries.

2.6.2.2 Keyword extraction

The first key functionality of the front-end is the keyword extractor. 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 annotations 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 [51]. On startup, the keyword extractor iterates through all files in the repository and isolates as many keywords as possible. Some of the handled elements are:

- 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.

All query words are normalized to lower letters and tailing spaces are removed. After this extraction, the vector for this document is transferred to the back-end by a single Web service call. The transmitted data consists of an ID, in this case the endpoint URL, and a pairwise list of all extracted terms and their raw frequencies in the document. All further processing is handled by the back-end.

2.6.2.3 Query Processor

The query processor takes a query string, in this case ”Google Search Service”, and splits it up to a list of keywords. The splitting is done by the same algorithm that is used in the keyword extractor. If processed locally, a query-vector is generated and directly submitted to the back-end where the result list is generated and delivered as the return value of the original call. The result is a list of documents, sorted by their similarity rating. See Figure 2.4 for a screenshot where a sample result list is shown.

2.6.2.4 Joiner

To finally join one or more of these search engines, the required functionality needed to be exposed as a Web service. Two methods of the front-end are visible from the outside:
RemoteQueryStats: This method takes a query string and returns all necessary values to invoke a distributed query at another front-end, namely n_k, N and a list of involved query words.
processDistributedQuery: This method finally processes a distributed query on the local back-end with the values retrieved by the above function.

For a distributed query, 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.

2.6.3 Back-end

The most essential part of the search engine consists of a single java package which contains the necessary methods to add, delete and, of course, to query the vector space. The package is deployed under the Axis framework which exposes two interfaces with basically the same signature as a Web services[4]. The interfaces are named `VolatileVSM` and `PersistentVSM`.

Function name	description
`cleanup`	Performs a cleanup of the vector space (clears unused terms)
`deleteDocument`	deletes a vector with the given ID
`getDocumentIDS`	retrieves a list of all indexed document ID's
`addVector`	adds a new vector with a list of term-frequency pairs
`getAllSorted`	retrieves a sorted list of all matches to a given query
`getAllSortedDistributed`	retrieves a sorted list of all matches to a given query with modified values for N and n_k
`getBestMatching`	retrieves a list of the m best matches to a given query
`getBestMatchingDistributed`	retrieves a list of the m best matches to a given query with modified values for N and n_k
`getDocCount`	retrieves the number of contained documents (N)
`getDimensionCount`	retrieves the number of total dimensions
`getStatistics`	retrieves the necessary statistics to perform a distributed invocation for a given query

Table 2.1: Interface elements

As the name suggests, they both provide the same set of operations differing only in their persistency mode. The volatile version operates on hash tables to store the vectors while the persistent version transforms the queries into SQL statements and operates on the connected MYSQL database to fulfill the request. Table 2.1 shows all provided methods and a short description of the functionality. The following subsections will depict how the actual implementation looks like and how an estimation for the computational expense can be given based on the used data structure.

2.6.3.1 Table generation

When using the interface for the persistent engine, the engine automatically generates the needed tables to hold the necessary data. In each call, the name of the repository is used to identify those tables. Appendix A.2 shows a listing of the actually involved java code, where the tables are created. Three tables are necessary to build a working repository and they are created automatically if they do not exist already. Database wrappers like Hibernate

[4] http://copenhagen.vitalab.tuwien.ac.at:8080/axis/servlet/AxisServlet

for Java provide a similar functionality but are more complex to use and, therefore, not necessary for smaller applications. The tables are checked whenever a new call is made to the Web service to guarantee that a working repository exists.

2.6.3.2 Query processing

The created statements are best explained by discussing how a query is processed with the weighting functionality enabled. For this purpose, we assume a 3 × 3 matrix with the following structure:

$$\begin{array}{c c} & \begin{array}{ccc} d_1 & d_2 & d_3 \end{array} \\ \begin{array}{c} t_1 \\ t_2 \\ t_3 \end{array} & \begin{pmatrix} x_{11} & x_{21} & x_{31} \\ x_{12} & x_{22} & x_{32} \\ x_{13} & x_{23} & x_{33} \end{pmatrix} \end{array}$$

The first requirement is to be able to weight one matrix element according to the previously discussed weighting formula:

$$x'_{ij} = x_{ij} * ld(\frac{N}{n_j} + 1)$$

To do so, queries for N as a whole and n_j in each iteration have to be created. The first element N or the cardinality of all documents is trivial:

$$N = |D| \Rightarrow N = \text{SELECT count(*) FROM Documents D}$$

Appendix A.2 shows the corresponding JAVA code where the statement is executed in line 41. Next, the term frequency n_j has to be defined for the current term. One iteration is necessary for each term in the query. The value is calculated as:

$$n_j = \sum_{i=1}^{N} b_{ij} \text{ with } b_i = \begin{cases} 0 & \text{if } x_{ij} = 0 \\ 1 & \text{otherwise.} \end{cases}$$

$$\Rightarrow \text{SELECT count(*) FROM Relations r where r.term\_id = currentTermID}$$

Again, the JAVA implementation is shown in A.2 line 63. Finally, the concrete value x_{ij} of a single term has to be processed in each iteration. The query is the same as before with the only difference that the selection now retrieves the frequency, resulting in the following statement:

$$\Rightarrow \text{SELECT frequency FROM Relations r where r.term\_id = currentTermID}$$

This query is essentially the same as the one before. In the implementation they are executed as one query to save time. That completes all necessary queries to weight a single element according to the *tf* x *idf* algorithm. The computation expense is linear to the query size, because an iteration of every contained term is necessary. On the other hand, dimensional reduction has a positive impact here because it causes less iterations on the vector space. A term that does not occur in the collection automatically drops out of the algorithm and, therefore, reduces the iteration size.

Now that the values can be weighted at runtime, the similarity measure can be processed following the formula:

$$cos(d_m, d_n) = \frac{\sum_{i=1}^{N} x_{mi} x_{ni}}{\sqrt{\sum_{i=1}^{N} x_{mi}^2 \sum_{i=1}^{N} x_{ni}^2}}$$

or in the more concrete case, d_2 and d_3 shall be processed:

$$cos(d_2, d_3) = \frac{(x_{21}x_{31}) + (x_{22}x_{32}) + (x_{23}x_{33})}{\sqrt{(x_{21}^2 + x_{22}^2 + x_{23}^2)(x_{31}^2 + x_{32}^2 + x_{33}^2)}}$$

The goal is to still use one iteration for the whole query and calculate all necessary values in the same step. This can be done by splitting numerator and denominator and complete them during each iteration step. The temporal values are stored in three hash tables. Table 2.2 shows each iteration step and the content of the hash table.

term	content of z-Hash	content of N1-hash	content of N2-hash
t_1	$(x_{21}x_{31})$	(x_{21}^2)	(x_{31}^2)
t_2	$(x_{21}x_{31}) + (x_{22}x_{32})$	$(x_{21}^2 + x_{22}^2)$	$(x_{31}^2 + x_{32}^2)$
t_3	$(x_{21}x_{31}) + (x_{22}x_{32}) + (x_{23}x_{33})$	$(x_{21}^2 + x_{22}^2 + x_{23}^2)$	$(x_{31}^2 + x_{32}^2 + x_{33}^2)$

Table 2.2: iteration steps

After the execution the final value can, therefore, be calculated with:

$$cos(d_3, d_2) = \frac{\text{z-Hash}}{\sqrt{\text{N1-hash} * \text{N2-hash}}}$$

The hash tables contain all relevance ratings for all documents relevant to the query. The key is defined by the documentID and line 51 of the implementation shows the method where the relevance rating is produced according to the above method.

A very positive side effect of this iterative method to process the cosine value is the possibility to apply an early termination mechanism. After the first iteration, the result will always be valid. Each subsequent iteration adjusts the multidimensional angle more accurate of course, but it is possible to define a time constraint and break the iteration when the maximum time for query processing is met. The result will not be complete but if the most important terms were processed, the result shifts just marginally. In the implementation, the early termination feature is enabled and visible in the iteration loop in line 54 where `maximumTimeoutMS` is used for the upper bound of milliseconds, the search is allowed to take. By adjusting this value, the performance of the search engine can be set to a maximum time, which comes handy when large vectors with a lot of matching keywords must be processed on a high-dimensional vector space.

2.6.4 Implementation experience

Summarized in this section is a list of implementation experiences that came up when writing the code of the search engine. Although they are only interesting from an engineers perspective, they explain why some concepts could not be implemented as initially planned.

- Platform dependency: To demonstrate platform independency, the back-end was written in JAVA and deployed under tomcat. The MYSQL database is connected via JDBC using TCP connections. The actual database in the test environment was set up in a linux environment along with other databases. Upon migration of the database to a Windows machine, the implementation ceased to work, although the connection was established correctly. The reason was, that MYSQL uses the folder structure on operating system level to build tables. In Linux environments, those entries are case sensitive but on Windows machines, the folder names are automatically translated to lower case. That caused the engine to re-create the tables with each query. This problem was easy to solve programmatically once it was discovered.

- Tables and indices: The first version, which operated on plain tables, took an enormous time span to process tuples of search terms. This time was decreased by introducing indices for the primary keys of the tables upon creation.

- Parallel queries: Upon startup of the front-end, all documents have to be indexed first. The original idea was to do that in parallel tasks with the asynchronous callback functionality of .NET. The approach seemed possible because tomcat is able to handle parallel requests and the .NET wrapper for the service provides the needed methods to do so. After finishing the implementation, however, the Axis framework fired unknown exceptions. Because it was not possible to determine which part of the implementation caused the error, the synchronous method to add vectors was used subsequently.

This completes the development of a vector space-driven Web service search engine that works in distributed environments. The following chapters continue this concept by enriching the concept and providing additional information about Web services. This information is needed to build a better index for search and discovery on one hand and also allows to provide the necessary information that enables users to decide which service will be able to fulfill the required operation most accurately.

Chapter 3

Bootstrapping and Exploiting Web Service Metadata

> Now that we have all this useful information,
> it would be nice to do something with it.
> Actually, it can be emotionally fulfilling just to get the information.
> This is usually only true, however, if you have the social life of a kumquat.
>
> Unix Programmer's Manual

3.1 Metadata in Web services

This chapter deals with a fundamental problem that applies to almost every Web service. How is it possible to retrieve a maximum amount of meta information about a Web service with only a WSDL description to start with?
To answer this question, it is necessary to define, what metadata is. Simply put, metadata is information about data. For example, source, server location, production date, etc. When it comes to Web service descriptions, certain classes of information are more interesting than others. For this thesis, the range is narrowed down to three categories, being Quality of Service (QoS), Location information, and Domain knowledge. Each section will deal with the challenges to bootstrap this information with only WSDL files as a starting point and no additional knowledge about implementation, hosting environment or deployment.

3.2 QoS

The availability of a QoS description for a set of services is considered as an *enabler* for solving many problems currently heavily investigated by different research groups. Such problems include composition, especially dynamic composition [10,17,22,60,74] as well as service discovery, search and selection of Web services [34,38,49] which is the goal of this thesis.

Currently, Web services support Quality of Service (QoS) attributes [37,40] for service descriptions by implementing the WS-mex specification [28]. Such attributes define non-functional attributes of a service. This metadata includes availability, latency, response time, authentication, authorization, cost, etc. Primarily performance-related aspects of Web services are required by various researchers because they provide valuable information. For the time being, those very specific measurements are not available a priori.

In this thesis, a framework is introduced, which provides the possibility to assess certain QoS attributes for a given Web service of the search engine's repository. The main contribution lies in the automatic approach for bootstrapping and constantly monitoring QoS parameters for existing services that are currently lacking such valuable descriptions. The goal is to achieve both, a maximized dynamic sampling and a broad spectrum of usable Web services. Sometimes, a service is very important and needs a high availability rating, while in other cases the response time is of more interest. With this information available, a better categorization of Web services is possible.

This section mainly deals with performance and availability related QoS attributes by using a flexible Web service invocation mechanism combined with aspect-oriented programming which allows to weave performance measurement aspects directly into the byte-code of the Web service stubs [32]. Business related values, such as cost, payment, etc. are omitted on purpose. They cannot be determined automatically but are provider-specific and defined by the business and cost model implemented by the service provider.

The result is a basic set of QoS attributes for a given service. To further use them, the concrete QoS attribute attachment mechanism is abstracted and supports two possibilities, (a) publish the QoS attributes together with the service description in UDDI (proposed by [62] or [52]) or (b) add them to the WSDL file by using WS-Policy [66].

3.2.1 Monitoring Approaches

Especially for performance-related attributes, several ways exist to assess those values in the real world. Each one has a certain level of advantages and disadvantages which are discussed here.

3.2.1.1 Provider-side instrumentation

The first and easiest way to bootstrap QoS parameters for Web Services is to directly instrument the service at the provider side. This way also allows to appoint values for security or implementation-related attributes. It bears the enormous advantage of a known service implementation. The service provider has to choose, if the dynamic attributes are calculated directly within the service code (invasive instrumentation), or by utilizing an artificial monitoring device (non-invasive instrumentation). Either case has two major drawbacks:

- All monitoring is done from the provider side, which means that network latency cannot be taken into account for a connecting client
- A service consumer has to trust the provider, to publish correct values.

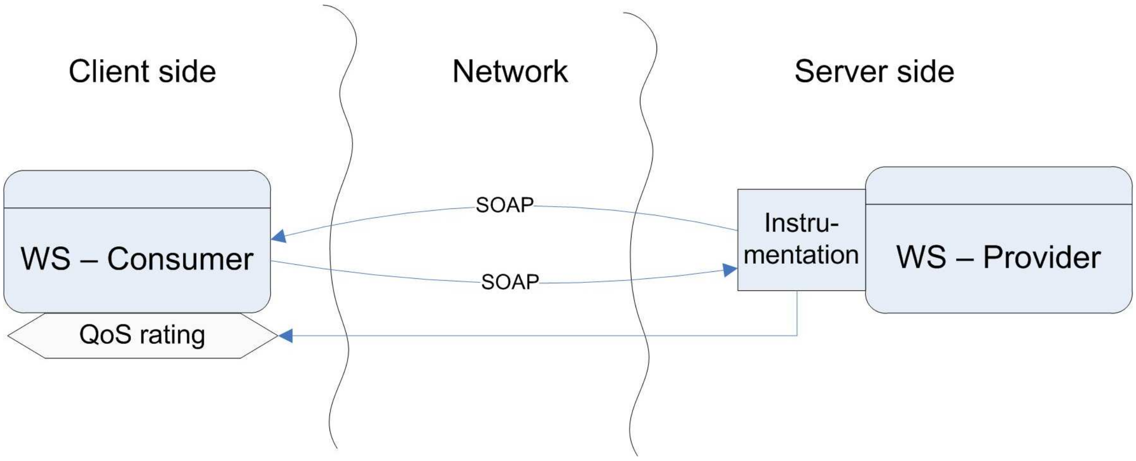


Figure 3.1: Provider side instrumentation

The approach, visualized in Figure 3.1, allows a very accurate measurement of all provider-specific values, even those not measurable like cost and security.

3.2.1.2 SOAP Intermediaries

By using an intermediary party, the traffic is not directly routed from client to server but to an intermediate party that is responsible for maintaining QoS related data. The big advantage with this approach lies in the trustworthiness of the received data. This third party is effectively a proxy that is able to handle incoming requests and forward them to the original destination. Although this approach solves the problem of trustworthiness and partially the latency issue, it still has an enormous drawback. Because of the proxy function, the third party will always be the bottleneck that limits scalability. Furthermore, the influence on the measured performance caused by the proxy is not negligible. High-performance Web service implementations would certainly experience a performance reduction when using this infrastructure.

In Figure 3.2, a graphical representation of this approach is depicted. It is obvious, that this method is unable to assess non-functional data like cost, because these values are explicitly created by the providing party.

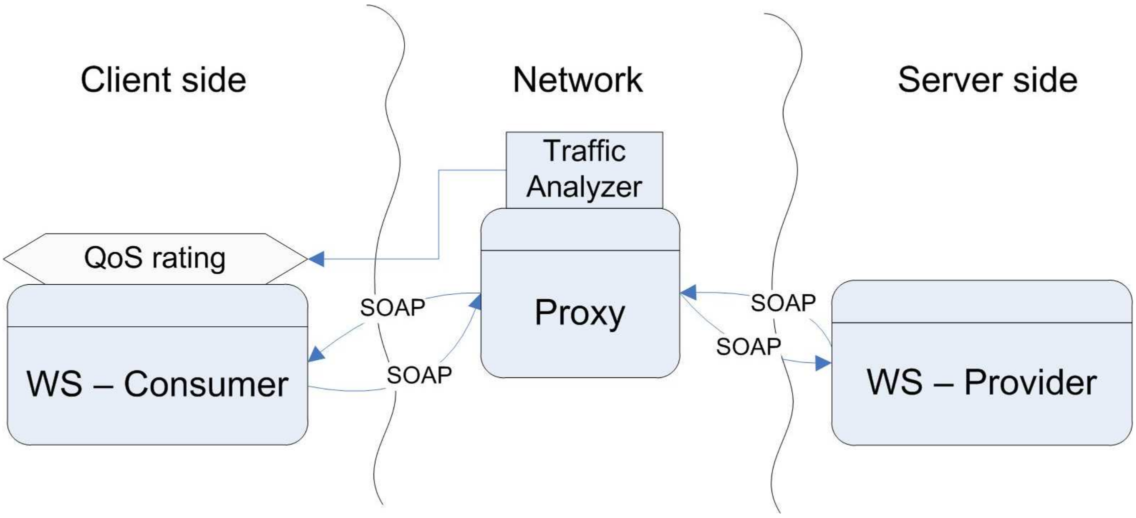


Figure 3.2: Intermediary

3.2.1.3 Probing

Just like bots for search engines and Web crawlers, the possibility to implement probes to collect data about Web service endpoints must be considered. This method bears a lower runtime overhead compared to SOAP intermediaries but still has the disadvantage of not being consumer-specific.

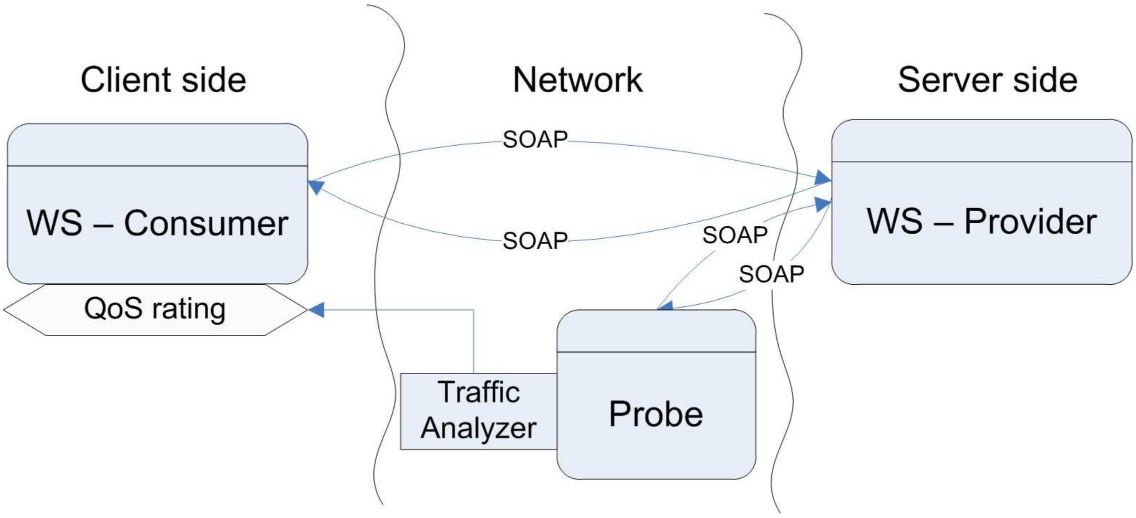


Figure 3.3: Probing

When invoking a service that is monitored by a probe as shown in Figure 3.3, the original message is not altered in any way and the metadata is provided by the probing party which invokes the original services on a scheduled basis. The problem is that providers might recognize when they are being probed and react differently than to real clients and manipulate the performance measurement the probe produces.

3.2.1.4 Sniffing

The last method introduced here, makes use of a *sniffer* that captures all outgoing packets from the client side. This way, the real traffic can be monitored and used to produce the required QoS data. Other than all the methods presented above, the values produced here are really consumer-specific. The performance rating strongly depends on the used client and its location, which is the desired behavior. Two major disadvantages come with this method:

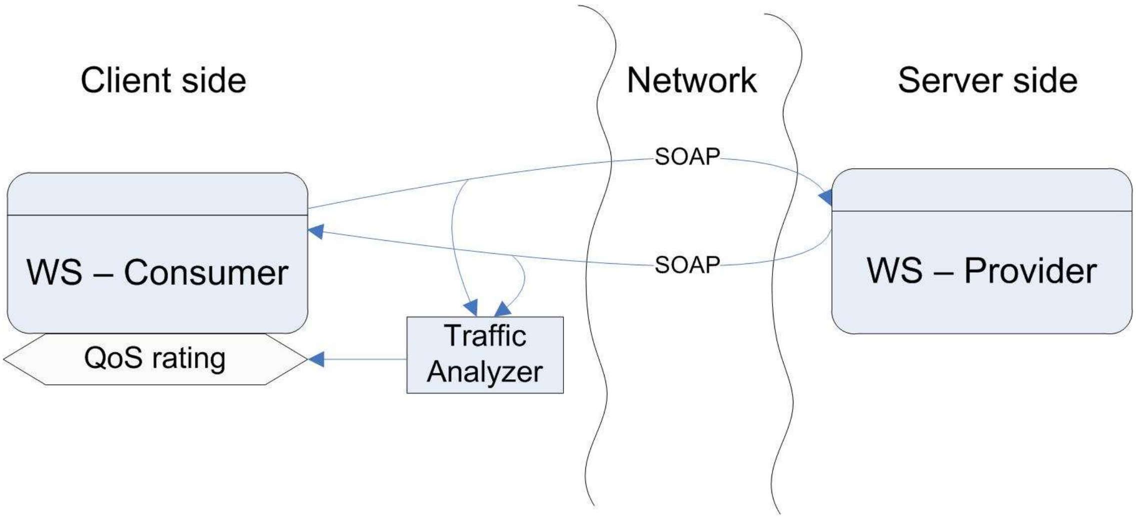


Figure 3.4: Sniffing

- QoS parameters of unknown services are not available until they are invoked the first time.
- The capturing process can only measure the time from the moment the packet leaves until an answer is returned. This timeframe usually includes a subset of other timeframes, such as processing time, wrapping time, etc. that have to be distinguished.

However, these problems can be solved. The following sections will explain how.

3.2.2 QoS Model

The first requirement is a basic QoS model which can be used to express QoS attributes for Web services. The most important point to realize here is that many of the evaluated attributes are dynamic and site-dependent. The service response time, for example, will experience a significant variation, depending on the type of connection used to evaluate it (e.g., Modem, DSL, T1 etc.). As a result, the produced values cannot be seen as global attributes, but as site-specific statistics with a strong local context. This is an intended behavior, because the parameters, influenced by the local conditions, increase the significance of the whole value. Two Web services are assumed for example, named A and B with the same implementation and the same hardware. The Web service consumer is located at a remote place where the routing of the actual IP packets is the only difference between the two services. Therefore, A may respond faster than B in this case while B could be the faster service when queried from another place. With this framework, a developer can choose the currently best suitable service depending on the provided QoS attributes.

The model can be categorized into several groups with each group containing related QoS attributes. Four main QoS groups, namely *Performance*, *Dependability*, *Security* and *Cost and Payment* can be identified for non-functional attributes. As already mentioned before, the focus lies on bootstrapping, evaluating and constantly monitoring the QoS attributes of the first two groups. The other ones cannot be estimated automatically. The first category and its elements concerns performance.

3.2.2.1 Performance

Processing time: Given a service S and an operation o, the processing time $t_p(S, o)$ defines the time needed to actually carry out the operation for a specific request R. The processing of the operation o does not include any network communication time and is, therefore, an atomic attribute with the smallest granularity. Its value is determined by the implementation of the service and the corresponding operation. To take Google as an example, t_p entitles the actual search time that is also displayed for search-requests sent by the Web interface.

Wrapping Time: The wrapping time $t_w(S, o)$ is a measure for the time that is needed to unwrap the XML structure of a received request or wrap a request and send it to the destination. The actual value is heavily influenced by the used Web service framework and even the operating system itself. In [68], the authors even split this time into three sub-values where receiving, (re-)construction and sending of a message are distinguished. For this purpose it does not matter if the delay is caused by the XML-Parser or maybe the implementation of the socket connection, because it can assumed to be constant for the server.

Execution Time: The execution time $t_e(S, o)$ simply is the sum of two wrapping times and the processing time: $t_e = t_p + 2 * t_w$. It represents the time that the provider needs to finish processing the request. It starts with unwrapping the XML structure, processing the result and wrapping the answer into a SOAP envelope that can be sent back to the requester.

Latency: The time that the SOAP message needs to reach its destination is depicted as latency or network latency time $t_l(S)$. It is influenced by the type of the network connection the request is sent over. Furthermore, routing, network utilization and request-size play a significant role for the latency.

Response Time: The response time of a service S is the time needed for sending a message M from a given client to S until the response R for message M returns back to the client. The response time is provider-specific, therefore, it is not possible to specify a globally valid value for each client. The response time $t_r(S, o)$ is calculated by the following formula: $t_r(S, o) = t_e(S, o) + 2 * t_l(S)$.

Round Trip Time: The last time-related attribute is the round trip time t_{rt}. It gives the overall time that is consumed from the moment a request is issued to the moment the answer is received and successfully processed. It comprises all values on both, requester and consumer side. Considering the formulae above it can be calculated as:

$$t_{rt} = (2 * t_w)_{con.} + t_l + (t_p + 2 * t_w)_{provider} + t_l + (2 * t_w)_{con.}$$

See Figure 3.5 for a graphical representation of all involved time frames.

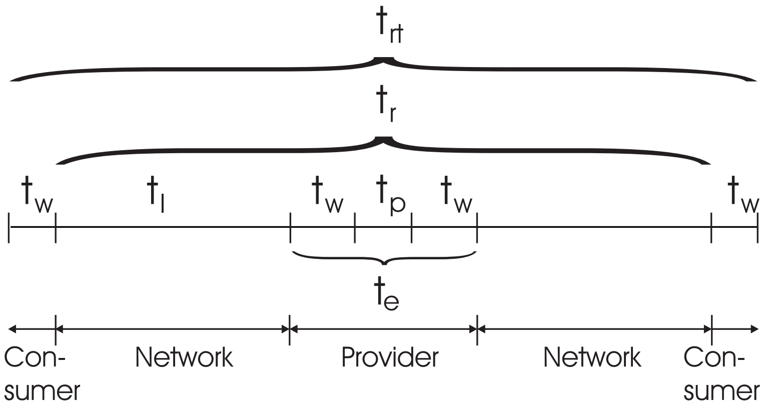


Figure 3.5: Service Invocation Time Frames

Throughput: The number of Web service requests R for an operation o that can be processed by a service S within a given period of time is referred to as throughput $tp(S, o)$. It can be calculated by the following formula:

$$tp(S, o) = \frac{\#R}{time\ period\ (in\ sec)}$$

This parameter depends mainly on the hardware power and service engine stability of the service provider and is measured by sending many requests in parallel for a given period of time (e.g., one minute) and count how many request come back to the requester. The evaluation process showed that most Web service engines have a maximum of parallel requests they can process before they cease to function and throw exceptions or shut down entirely.

Scalability: A Web service that is scalable, has the ability to not get overloaded by a massive number of parallel request. A high scalability value states the probability for the requester of receiving the response in the evaluated response time t_r.

$$sc(S) = \frac{t_{rt}}{t_{rt(Throughput)}},$$

where $t_{rt(Throughput)}$ is the round trip time which is evaluated during the throughput test.

3.2.2.2 Dependability

Availability: The probability that a service S is up and running and producing correct results. The availability can be calculated the following way:

$$av(S) = 1 - \frac{downtime}{uptime + downtime}$$

The downtime and uptime are measured in minutes.

Accuracy: The accuracy $ac(S)$ of a service S is defined as the success rate produced by S. It can be calculated by evaluating all invocations starting from a given point in time and examining their results. The following formula expresses this relationship:

$$ac(S) = 1 - \frac{\#failed\ requests}{\#total\ requests}$$

Robustness: It is the probability that a system can react properly to invalid, incomplete or conflicting input messages. It can be measured by tracking all the incorrect input messages and put it in relation with all valid responses from a given point in time:

$$ro(S) = \frac{\sum_{i}^{n} f(resp_i(req_i(S)))}{\#total\ requests}$$

The part $resp_i(req_i(S))$ represents the i^{th} response to the i^{th} request to the service S, where n is the number of total requests to S. The utility function f is calculated as:

$$f = \begin{cases} 1, & isValid(resp_i) \\ 0, & \neg isValid(resp_i) \end{cases}$$

and is used to evaluate, if the response was correct for a given input.

3.2.3 Bootstrapping, Evaluating and Monitoring QoS

The used bootstrapping and evaluation approach for the different QoS parameters from Section 3.2.2 is such a *client-side technique* which works completely Web service and provider independent. Many different steps are necessary to successfully bootstrap and evaluate QoS attributes for arbitrary Web services. An overview of the main blocks of the system architecture and the three different phases of the evaluation process are depicted in Figure 3.6.

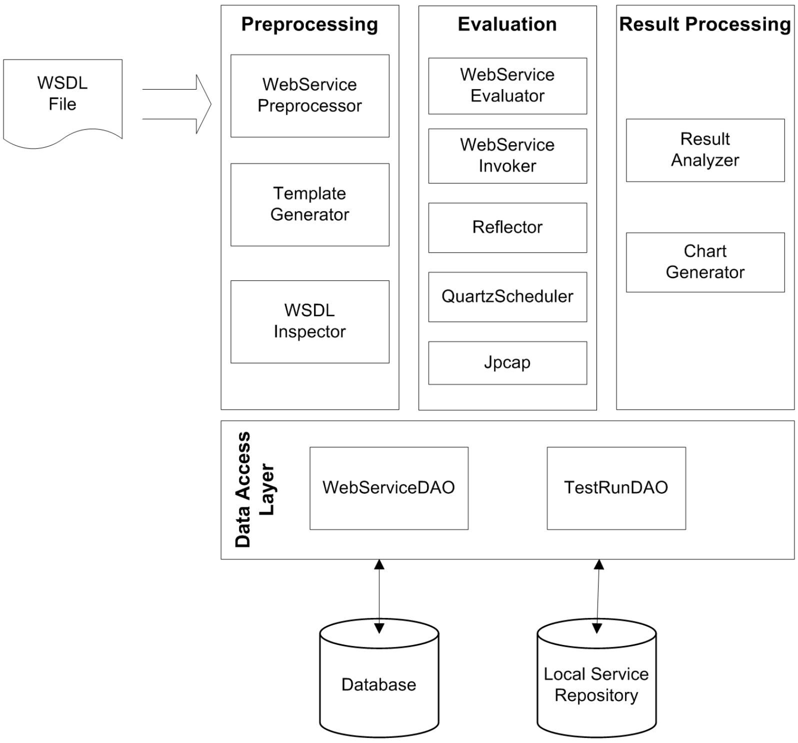


Figure 3.6: System Architecture

Preprocessing Phase: In this initial phase, the `WSDL Inspector` takes the URL of one or more WSDL files as an input and fetches it into the *local service repository*. Then, the WSDL file is parsed and analyzed to determine the SOAP binding name (only SOAP bindings are supported. HTTP GET or POST is not supported). From the name of the `binding`, the corresponding `portType` element can be retrieved, thus, all operations which have to be evaluated can be determined. Furthermore, all XSD data types defined within the `types` tag have to be parsed to know all the available types needed for invoking the service operations. The information gathered by analyzing the WSDL is used in the evaluation phase to dynamically invoke different operations of a service. As a next step, the Web service stubs are generated as Java files by using the WSDL2Java tool from Axis [5]. The performance measurement code itself is implemented by using **a**spect-**o**riented

programming (AOP), thus, an aspect which captures the evaluation information where it occurs is created. Aspect oriented programming is needed because the code generated by WSDL2Java is not known a priori but has to hold the needed methods for performance measurement. The aspect is discussed in detail in Section 3.2.5.
The Java source files together with the aspect are compiled with the AspectJ compiler to generate the Java classes. All the aforementioned steps are fully automated by the `WebServicePreprocessor` component and do not need to be executed every time a specific service has to be evaluated. It has to be done only once, then the generated code is stored in the local service repository and can be reused for further re-executions of the evaluation process itself.

Evaluation Phase: During the evaluation phase, the information from the WSDL analysis in the preprocessing phase is used to map the XSD complex types to Java classes created by the WSDL2Java tool. Furthermore, Java Reflection is heavily used, encapsulated in the `Reflector` component, to dynamically instantiate these complex helper classes and Web service stubs. The `WebServiceInvoker` component tries to invoke a service operation just by "probing" arbitrary values for the input parameters for an operation. If this is not possible, e.g., because an authentication key is required, a template-based mechanism is supported that allows to specify certain parameters or define ranges or collections to be used for different parameters of a service operation. Such a template is also generated by the `TemplateGenerator` component during the preprocessing phase based on the `portType` element information in the WSDL file. The main part of the evaluation is handled by the `WebServiceEvaluator` and the `EvaluationAspect`. The aspect defines a pointcut for measuring the performance related QoS attributes from Section 3.2.2. For example, the response time t_r is measured by defining a pointcut which timestamps before and after the `invoke(..)` method of the Axis `Call` class. The `Call` class handles the actual invocation to the Web service within the previously generated stub code. The response time itself is then calculated by subtracting the timestamp after the `invoke(..)` call from the timestamp before the call. Due to the client-side mechanism, the QoS parameter such as the latency t_l cannot be measured as comfortable as t_r. Therefore, the packet capturing library Jpcap [21] is used within the `EvaluationAspect` to measure the latency and the processing time on the server by using information from the captured TCP packets. Details are discussed in the following sections.

Result Analysis Phase: In this phase, the results generated from the `WebServiceEvaluator` are collected by using the `ResultCollector` which represents a singleton instance. It collects all results generated by the `WebServiceEvaluator` and the `EvaluationAspect` and stores it in a database. Afterwards, the `ResultAnalyzer` iterates over these collected results and generates the necessary statistics and QoS attributes. Moreover, these resulting QoS attributes can be attached directly to the evaluated service as mentioned in Section 3.2. It is relatively straightforward to do so, and, therefore, not explained in detail.

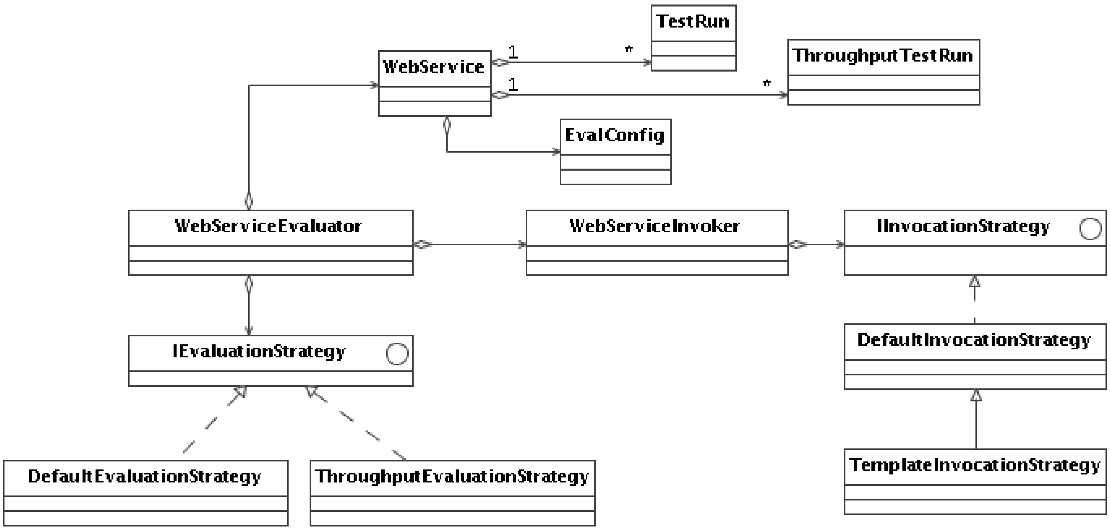


Figure 3.7: Architectural Approach

3.2.4 Architectural Approach

The architecture for evaluating and thus invoking arbitrary Web services is quite flexible because many design patterns from [23] were applied. In Figure 3.7, some parts of the architecture are depicted as UML class diagrams. The core class is the `WebServiceEvaluator` which encapsulates all the evaluation specific parts. A `WebService` class encapsulates all information about a Web service (endpoint, reference to WSDL, location in repository, port type, binding information, etc). An evaluation is either performed at the operation level or the service level. The operation level denotes that only one given operation of a service is evaluated by using the `evaluate(String operationName, Object[] parameters)` method. The service level means that all operations of a service are evaluate by using the `evaluate()` methods, which implicitly invokes the aforementioned method for every operation. Furthermore, different invocation mechanisms for a Web service with respect to the way to generate reasonable input parameters for the different operations are available.

The architecture encapsulates the algorithms to actually evaluate a service or invoke certain operations of a service by using the strategy pattern [23]. Two strategies to evaluate an operation are implemented. The `DefaultEvaluationStrategy` simply calls a service operation once with a given implementation of the `IInvocationStrategy` interface. The QoS attributes are encapsulated in the `EvaluationAspect`, which is woven into the byte code of the application and the stub code of the service. By contrast `ThroughputEvaluationStrategy` allows to measure the throughput of a Web service by sending multiple requests in concurrent threads to the service, according to the formula given in Section 3.2.2.

Service Invocation Strategies. The invocation strategy defines how an invocation of a service operation is done. Again, two different choices are available. The `DefaultInvocationStrategy` implements a default behavior by iterating over all parts of the input messages of a service and instantiating the corresponding input type as generated by the WSDL2Java tool. The instantiation of complex types (even nested ones) is handled by the `Reflector` component. The `TemplateInvocationStrategy` uses the invocation template generated during the service preprocessing to invoke the service operation. The template can be edited by the user to add various pre-defined values for the different input parameters. The template functionality is important for services that require some sort of structured data to be called correctly. An example is a used ID or a password that has to be provided for accessing the service. Generated input data would result in an erroneous invocation. In Listing 3.1, the main algorithm for invoking a service operation with a previously generated template is depicted. The algorithm uses the stubs for the Web service which are generated during the preprocessing phase and tries to find a value for each parameter in the XML template file. If no parameters can be found in the template or the template is not available, the `Reflector` tries to instantiate the required parameter type with a default value (handled by the `initalizeParameter()` method).

The main advantage of this flexible architecture is the possibility to add new evaluation and invocation mechanisms or even selecting them at runtime without changing the structure of the system and the aspect.

```
public Object invokeOperation(String operationName,
                              Object[] paramValues) {
  Operation op = service.getOperation(operationName);
  Message inputMsg = op.getInput().getMessage();
  Class[] parameters = new Class[parts.size()];
  Object[] paramInstances = new Object[parts.size()];

  // go through each part and try to find Java class
  // or simple type and initialize it
  for(Part p : inputMsg.getParts()) {
    QName type = p.getTypeName();
    Class param = convertXSDTypeToJavaType(type);
    if (param == null) { // it is not a simple type
      String javaName = convertQNameToPackageName(type);
      param = Class.forName(javaName, false, classloader);
    }
    parameters[i] = param;
    paramInstances[i] = initalizeParameter(operationName,
                                    p.getName(), param);
  }
  // use reflection to invoke the operation
  Method m = service.getStubClass().getMethod(
                          operationName, parameters);
  return m.invoke(service.getStub(), paramInstances);
}

public Object initalizeParameter(String operationName,
                      String paramName, Class paramType) {
  // try to find param value from template
  String value = findParamValue(operationName, paramName);
  if (value != null) {
    return convertToObject(paramType, value);
  }
  return Reflector.instantiate(paramType);
}
```

Listing 3.1: invokeOperation Algorithm

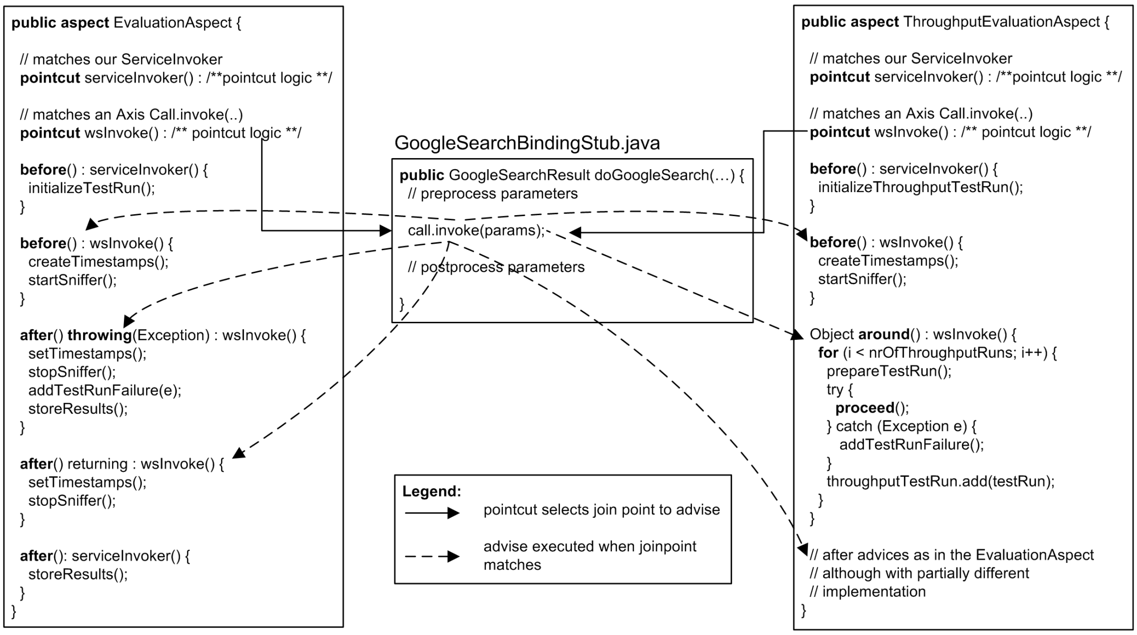
3.2.5 Evaluating QoS Attributes using AOP

This approach measures the performance-related QoS values which is achieved by using aspect-oriented programming (AOP). It is an ideal technique for modeling cross-cutting concerns. The evaluation part is such a cross-cutting concern since it spans over each service that needs to be invoked during the evaluation. The basic idea of the approach is described in Figure 3.8. During the preprocessing phase, the stubs for the service which should be evaluated by using the WSDL2Java tool are generated. For the Google Web service, as one example, the main stub class that is generated is called `GoogleSearchBindingStub`. Each stub method looks similar, first is the wrapping phase, where the input parameters are encoded in XML. Secondly, the actual invocation is carried out by using the `invoke(..)` method of the `Call` class from the Axis distribution. At last, the response from the service is unwrapped and encoded as Java arguments and returned to the caller.

Therefore, the `EvaluationAspect` defines the following pointcut to measure the response time t_r:

```
pointcut wsInvoke(): target(org.apache.axis.client.Call)
                  && ( call( Object invoke(..)) ||
                       call( void invokeOneWay(..)));
```

Whenever a service operation is invoked by using the `WebServiceEvaluator`, the `wsInvoke()` pointcut defined for this join point is matched. Before the actual service invocation the *before* advise is executed. This is where the actual evaluation has to be carried out. It mainly consists of a timestamp and the generation of an EvaluationResult, as well as starting the packet-sniffer to actually trace the TCP traffic caused by the following request. After the `wsInvoke()` pointcut the execution of the corresponding *after* advise is triggered, depending whether the service invocation was successful or not. At this point, packet capturing can be stopped and the collected data can be extracted. The timestamps taken before and after the invocation can directly be used to calculate the response time. To distinguish between latency and execution time, the TCP level has to be investigated.



```
public aspect EvaluationAspect {

  // matches our ServiceInvoker
  pointcut serviceInvoker() : /**pointcut logic **/

  // matches an Axis Call.invoke(..)
  pointcut wsInvoke() : /** pointcut logic **/

  before() : serviceInvoker() {
    initializeTestRun();
  }

  before() : wsInvoke() {
    createTimestamps();
    startSniffer();
  }

  after() throwing(Exception) : wsInvoke() {
    setTimestamps();
    stopSniffer();
    addTestRunFailure(e);
    storeResults();
  }

  after() returning : wsInvoke() {
    setTimestamps();
    stopSniffer();
  }

  after(): serviceInvoker() {
    storeResults();
  }
}
```

GoogleSearchBindingStub.java

```
public GoogleSearchResult doGoogleSearch(...) {
  // preprocess parameters

  call.invoke(params);

  // postprocess parameters

}
```

```
public aspect ThroughputEvaluationAspect {

  // matches our ServiceInvoker
  pointcut serviceInvoker() : /**pointcut logic **/

  // matches an Axis Call.invoke(..)
  pointcut wsInvoke() : /** pointcut logic **/

  before() : serviceInvoker() {
    initializeThroughputTestRun();
  }

  before() : wsInvoke() {
    createTimestamps();
    startSniffer();
  }

  Object around() : wsInvoke() {
    for (i < nrOfThroughputRuns; i++) {
      prepareTestRun();
      try {
        proceed();
      } catch (Exception e) {
        addTestRunFailure();
      }
      throughputTestRun.add(testRun);
    }
  }

  // after advices as in the EvaluationAspect
  // although with partially different
  // implementation
}
```

Legend:
pointcut selects join point to advise
advise executed when joinpoint matches

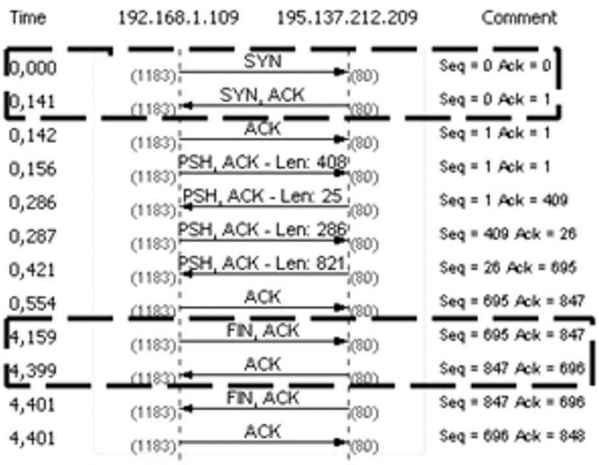
Figure 3.8: Aspect for Service Invocations (simplified)

3.2.5.1 TCP Reassembly and Evaluation Algorithm

The core element of the stub-based approach is the TCP sniffer and traffic analyzer. These elements finally make it possible to perform the service evaluation at the client side. A service invocation, which is a TCP communication after all, actually consists of at least three sub-messages if the TCP level is observed. The first and the last are always handshake messages, with no payload attached. More precisely, the TCP handshake consists of two parts:

- The connection setup via a SYN packet, issued by the Web service client, and a following SYN/ACK packet by the Web service provider to confirm the established connection.
- The connection termination, again issued by the client to signal the end of the transmission via a FIN, plus an optional ACK flag for previously received traffic followed by the servers ACK and (FIN, ACK) to confirm the connection termination.

Each of those handshake messages only needs the time to overcome the network latency, plus some negligible time span the operating system uses to create an acknowledge packet and send it back. Therefore, at least two meaningful values for the network latency can be gathered from one single request. A complete trace visualized by the packet capturing tool Ethereal(Wireshark) is illustrated in Figure 3.9. The dashed lines highlight the handshake and connection termination message exchange.



Time	192.168.1.109		195.137.212.209	Comment
0,000	(1183)	SYN →	(80)	Seq = 0 Ack = 0
0,141	(1183)	← SYN, ACK	(80)	Seq = 0 Ack = 1
0,142	(1183)	ACK →	(80)	Seq = 1 Ack = 1
0,156	(1183)	PSH, ACK - Len: 408 →	(80)	Seq = 1 Ack = 1
0,286	(1183)	← PSH, ACK - Len: 25	(80)	Seq = 1 Ack = 409
0,287	(1183)	PSH, ACK - Len: 286 →	(80)	Seq = 409 Ack = 26
0,421	(1183)	← PSH, ACK - Len: 821	(80)	Seq = 26 Ack = 695
0,554	(1183)	ACK →	(80)	Seq = 695 Ack = 847
4,159	(1183)	FIN, ACK →	(80)	Seq = 695 Ack = 847
4,399	(1183)	← ACK	(80)	Seq = 847 Ack = 696
4,401	(1183)	← FIN, ACK	(80)	Seq = 847 Ack = 696
4,401	(1183)	ACK →	(80)	Seq = 696 Ack = 848

Figure 3.9: TCP Handshake Traffic

Unfortunately, exploiting the traffic of the actual message transfer is not easily achieved because it is harder to predict how the traffic eventually looks like. In a standard scenario, the client sends a single HTTP message with one POST and receives an acknowledgement packet that contains the SOAP encoded answer. Even in this very fundamental case, many variations are possible:

- The Web server may or may not return a result value for the request. Therefore, it is not clear a priori, how large the payload of the server answer is.
- The original request or even the answer could be of a rather large size, exceeding the maximum TCP frame length, making either a multi-frame transmission or a TCP frame length update necessary. The additional messages make it hard to isolate the packet where the Web Server executes the operation and, therefore, consumes the execution time that needs to be evaluated.
- The packet transmission may be disturbed at some point, forcing the sending partner to retransmit the lost packet. Again, the obsolete packets must not be used to calculate network latencies.

To overcome these obstacles, the following algorithm (see Listing 3.2) can be used to analyze the message flow:

```
TCPPacketList sourceList, destinationList;

foreach (TCPPacket p in TCPTrace) {
   if (p is outgoing) {
      if (p.sequenceNumber NOT IN sourceList) OR
          sourceList.getPacket(SequenceNumber) has no payload OR
          p has no payload)) {
                add p to sourceList with key p.SequenceNumber
      }
    } else if (p is incoming) {
      add p to destinationList  with key p.AcknowledgeNumber;
    }
}
```

Listing 3.2: TCP Message Flow Algorithm

What this sequence basically does is to put the packages in two hash tables, indexed by the packet's sequence number for the client's packages and acknowledge numbers for the server's packages. Packets from repeated transmissions or frame updates are omitted because they are mapped to the same position in the tables and will be overwritten. After all packets are received from a single request, the analyzing procedure iterates through the list of source packages and tries to find a packet in the destination list with a matching acknowledge number. Each match can then be used to calculate a latency and is added to the list of latencies for the whole request. The largest latency in this list is assumed to include the processing time and is not added to it. Finally, the arithmetic mean of all collected latencies is subtracted from the largest time to calculate the execution time without trailing latencies.

For massive amounts of requests at the same time, as it happens in a throughput test for example, TCP packets sometimes interleave. This problem was met in two ways: Firstly, dynamic filter rules for the TCP sniffer were used. This way, the captured packets can be limited to a single endpoint IP and a port (which is 80 for HTTP in most cases). Secondly, captured packets are sorted according to the outgoing port, which is determined by the operating system's port allocation algorithm. This way, it is possible to distinguish between multiple requests to the same endpoint.

3.2.5.2 Implementation Details

The system is called QuATSCH which is short for ”**Qu**ality **A**ssessment **T**ool for Web **S**ervice **C**onsidering network **H**eterogeneity” and is implemented with the Java 5 platform and AspectJ 1.5 [19] for implementing the evaluation part. For parsing and analyzing the WSDL files the WSDL4J library from SourceForge[1] was used. The transformation from WSDL to Java classes is handled by the Axis WSDL2Java tool [5]. The calculation of the latency and the execution time is done by using Jpcap [21], a Java wrapper for libpcap, which allows to switch to the promiscuous mode to receive all network packets from the NIC not only these addressed to a specific MAC address. These raw evaluation results are collected and stored in a MySQL database.

3.2.6 Integration

One side effect of the implementation has a negative influence on integration capabilities of the QoS assessment tool. The fact that network analyzers of any kind need direct

[1] http://sourceforge.net/projects/wsdl4j

access to the network interface card of the executing machine requires to run it with root or Administrator permissions. Especially for software that is exposed through a Web interface, this is an enormous problem. In case of a security breach, a possible attacker could easily gain root access to the hosting machine.
The other problem comes with the functionality of the software itself. Making a tool publicly available, which can deliver a massive amount of parallel requests at the same time is a potential way to initiate denial of service attacks to Web service providers. Especially when the target service is hosted under an environment that does not cope with lots of parallel requests like Axis, for example. Furthermore, some of today's public Web services are not designed for high loads and a constant polling interval like it is needed for this purpose, would most probably cause more traffic to the target service than actual requests. Therefore the QoS measuring tool is not integrated in the official Web site.

3.3 Location Data

Another form of service metadata is represented by the location of the server where the Web service is hosted. Other than the QoS parameters, location data does not automatically hold qualitative information. Even when a server is located far away, it may perform better than a local service. Besides, performance ratings are already measured by the QoS tool.
The data is used mainly to correlate other information. By tracking response times of various services and comparing them to their destinations for example, a performance estimation of newly added services could be given. Other possibilities to exploit the gathered information exist and are briefly discussed below.

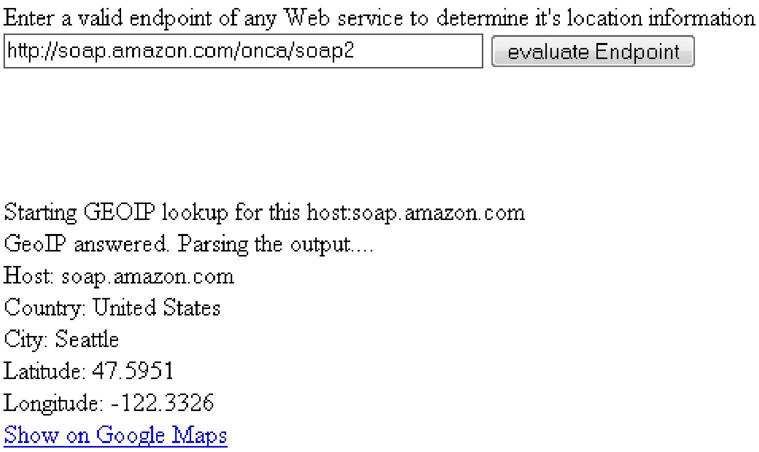
3.3.1 Concept

The starting point to evaluate the position of a Web server hosting a Web service is always the endpoint address. This bit of information is the only source that can be used to gather metadata without actually invoking the service. In contrast to user comments, message names or port types, the endpoint is the only connection to the network layer. Furthermore, it cannot be assigned at will.
This endpoint address provides the necessary host name which in turn maps to an IP address. Determining the IP address of the server hosting the service in question is straight forward and can be achieved by a simple DNS lookup. Determining the location of this address, however, is a challenging task. It is possible to distinguish two classes of IP addresses:

- Static IPs: Static IP addresses require their holders to register location information of some sort when the address is requested. An example would be a company that reserves an 8-Bit Subnet to provide the necessary addresses for its machines. The problem here is how to get the data entered by the consumer and relate it to world map coordinates. For Europe, the Middle East and parts of Asia, the RIPE NCC service[2] is a central point of registration where lots of information about an IP address is stored and can be queried. Comparable databases exist for other regions. For a global IP lookup tool however, all these databases must be collected and combined correctly to ensure usable readings for any possible IP.

- Dynamic IPs: Determining the exact location of a dynamic IP address is next to impossible. Dynamic IPs are issued by **i**nternet **s**ervice **p**roviders ISPs to their costumers, for example. To find out where the corresponding IP is located, the ISP would have to provide a realtime list of the issued IPs and the costumer data. Doing so would be a breach of consumer privacy. Besides, the high fluctuation rate of dynamic IP addresses causes the collected data to be outdated almost instantly. Fortunately, dynamic IP addresses are almost never used by Web service providers if the service is designed with a high availability. Furthermore, even if a dynamic address is used for a provider it can be assumed that the location is not very far from the ISPs own location.

The implementation of such a database requires a high amount of resources and is in itself a pure engineering task. One of the biggest providers of location information for IP addresses is called *maxmind*. Their tool **GeoIP** essentially provides all the tasks presented above in a Web application[3] which is available for 25 daily lookups to demonstrate how it works. For this thesis, a wrapper was created, that uses those daily queries to assess the necessary endpoint information. The location evaluation tool itself can be accessed directly from the VitaLab implementation[4].



Enter a valid endpoint of any Web service to determine it's location information

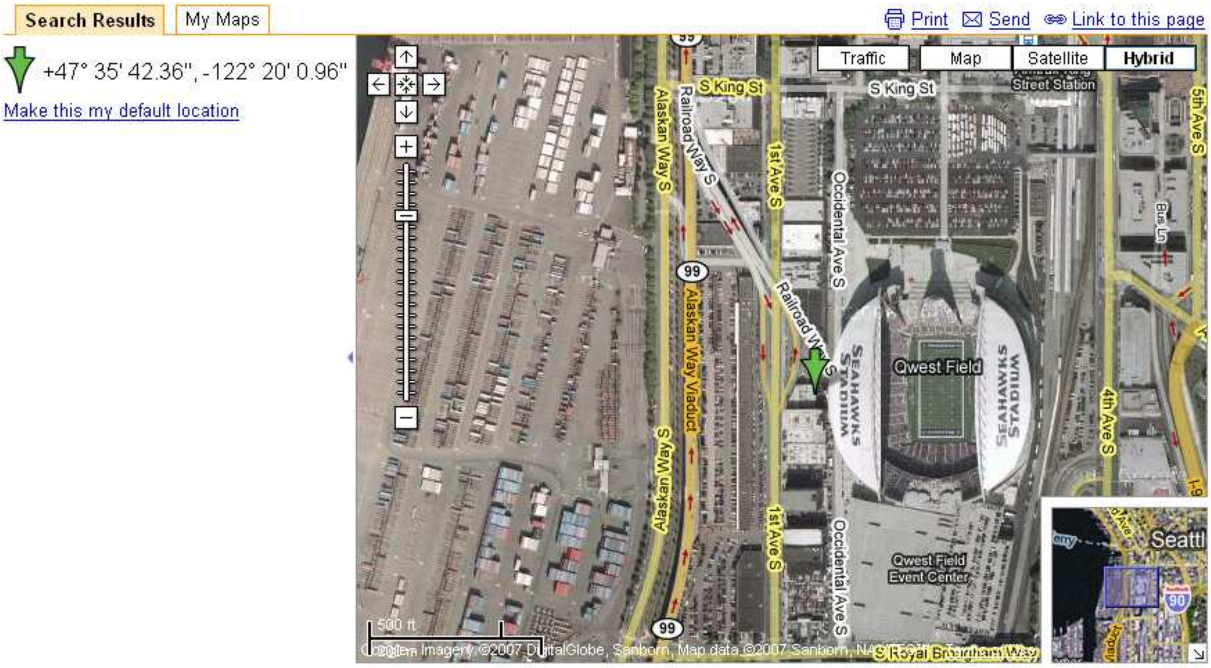
http://soap.amazon.com/onca/soap2

Starting GEOIP lookup for this host:soap.amazon.com
GeoIP answered. Parsing the output....
Host: soap.amazon.com
Country: United States
City: Seattle
Latitude: 47.5951
Longitude: -122.3326
Show on Google Maps

Figure 3.10: Location lookup of soap.amazon.com

[2] http://www.ripe.net/whois
[3] http://www.maxmind.com/app/locate_ip
[4] http://copenhagen.vitalab.tuwien.ac.at/locator/locator.aspx

Looking up the endpoint of the Amazon Web service, produces the output shown in Figure 3.10. The most essential values are latitude and longitude. Along with the coordinates, the lookup also returns the state and city name of the position.



+47° 35' 42.36", -122° 20' 0.96"

Figure 3.11: Google Maps location view

To get a visual view of the suggested location, a link to Google Maps is provided. The exact endpoint URL that was used for the query is `http://soap.amazon.com/onca/soap2`. It is reported to be located in Seattle, USA. The screenshot presented in Figure 3.11 shows the coordinates in a graphically more appealing form. Nevertheless, the world coordinates are the most important results of the lookup and all other operations will take them for further use.

3.3.2 Exploitation

Some possibilities to exploit the gathered information were already mentioned above. Some of them are more feasible than others. It is possible for example, to calculate metric distances between the locations of two Web services (A and B), using the following formula:

$$d = r_{earth} * \arccos(\sin(A_{lat}) * \sin(B_{lat}) + \cos(A_{lat}) * \cos(B_{lat}) * \cos(B_{long} - A_{long}))$$

Although this formula is mathematically correct, it produces quite poor values for small distances due to the imprecise calculations of $\cos(\alpha)$ for small angles ($\alpha < 1$ arcsecond). Furthermore, the earth radius of 6350km is also just an approximated value. For most applications though, the imprecision does not really matter because it still gives a good

idea of the involved distances.
With the distance values between Web service providers it is now possible to search for geographically near Web services. It would even be possible to create a cluster analysis based on these values, but the result is more than questionable. In the end, the fact that two Web services are on near locations does not influence their provided functionality. Service settings on the other hand are a different matter. Web services located in China have a good chance of using the UTF-16 or at least UTF-8 character set for the encoding and probably Chinese as the description language. This could be valuable information to rule out certain service implementations.

The most important way to use this location information though is for service performance prediction. In Section 3.2.3, the problem how to handle new Web services was mentioned. An estimation of the service QoS can only be given after the first service invocation, which could be delayed because of scheduling or other reasons. For network-based values however, is is possible to overcome these restrictions. Some of the values from the QoS model presented in Section 3.2.2 are based on the restrictions the network topology constitutes. High latency ratings can have several reasons ranging from bad routing over packet losses to slow ISP uplinks. Predicting the latency value for an unknown host is particulary hard when the only known thing is an endpoint IP or host name. Nevertheless, an estimated value may be needed even for newly added services. To provide a very rough estimation of the minimum service response time, the gathered location information is utilized.

In most cases, network latency is caused by an over-utilization of the network route. An estimation of the consumed time can be based on the assumption, that geographically close hosts use similar routes to transport data packets to very distant locations. When the same ISP is used, the only difference in the route will be the last few hops. Even when different ISPs are used, the traffic will most likely take the same route for the majority of the distance. Therefore, already bootstrapped surrounding Web services can be used as a reference point for the expected network latency. To ensure that the estimated values at least give an idea what to expect, the following points have to be considered.

- When assessing an endpoint host, the distance from the consumer d_{target} must first be calculated. This can be done by using the distance formula provided earlier in this section.

- Of all the hosts already evaluated, only those relatively close to the target host must be considered. As a rule of thumb, the distance from the target service to an adjacent host should be less than 5% of the original distance: $d_{adjacent} < 0.05 * d_{target}$

- From the remaining candidates, only the 90% percentile is considered. This way, statistical outliers can be cut off. Such outliers can be caused by extremely fast uplinks or modem connections, which result in an extraordinary latency value.

- The latency values for all candidates are averaged and the standard deviation is calculated and used as an initial estimation of the network latency to expect. At this point, the precision/recall rating can be adjusted. By lowering the bound for the standard deviation, the estimation becomes more precise. On the other hand it means that less results can be produced.

Summed up, an estimation of the service latency depends on the size of the already evaluated repository. The more hosts near to the original target were already evaluated, the better the estimation will be. Nevertheless, the estimated value will never be as precise as an actually measured one. It merely serves as a straightedge of what to expect from the invocation process.

The evaluation of the location information completes the assessment of the most important tangible Web service metadata. To gather further meta information, automatic approaches are not enough. Some sort of user input is required where WSDL files cannot provide enough information.

3.4 Semi automatic domain classification

Some information often required when searching service repositories is the class or domain, a service belongs to. This information is next to impossible to automatically derive from a WSDL file. Unless the description file uses an additional set of markup to describe the domain the service or operation belongs to, no means to directly describe the service's domain affiliation in the WSDL file exists. This section introduces a classification system for the service domain which is able to give a recommendation based on already entered domain information.

3.4.1 Domain Tree

When classifying a service, the possible classes are usually represented in tree form. Each possible category is entered in a node. Each node can contain sub-nodes which are of a smaller granularity, giving the possibility to exactly define the best fitting domain. The sample tree in Figure 3.12 visualizes some of the categories of the ACM computing classification system[5]. In a trivial system, a user that provides a Web service description is simply asked to categorize the service with the available domains. Even though the user is willing to do so, it may happen that the right sub-category is missing. Therefore, the tree has to be extendable and let users add their custom subcategories to the available structure. Furthermore, it must be possible to categorize a service with a node that is not a leaf of the tree, if no exact match can be found. A Web service, for example, that takes

[5] http://www.acm.org/class/1998/overview.html

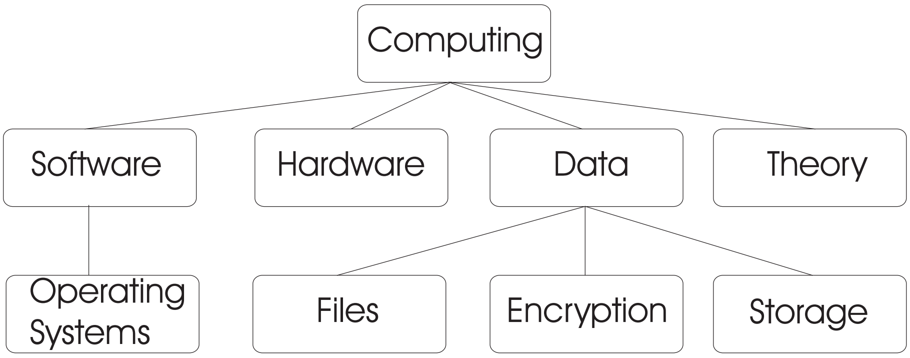


Figure 3.12: Domain Tree example

a file, encrypts it and sends it back will best fit in the *Data* category of the domain tree since more than one of the leafs fit.

Such an extensible domain tree is basically nothing new, and the gathered metadata is not automatically created, but entered by the user. To guarantee, that each service comes with this domain information, each user would have to be forced to enter that data upon service registration. Such a method however, is a limiting fact for the usability of a service registry. The more obligatory input a user has to provide, the more likely it is for the user to skip the registration process entirely. Therefore, a system that can give a suggestion for the most probable domain is needed. When the user decides to skip this part of the service specification, a classification can still be performed.

3.4.2 Recommendation System

The emerging issue can be seen as an information retrieval problem. The critical part when giving a recommendation about the topic that most likely fits a description is to find the most closely related entries done by humans. When a list of similar entries was generated, a categorization algorithm can be utilized to find the right topic. The following iteration briefly explains how such an "average" domain recommendation is built for a new entry.

1. An initial search is executed. Either the best n matches are used for further processing, or a lower boundary (e.g. 0,8) for the similarity rating is defined. For an example, three elements (A, B and C) are assumed to be closely related to the initial query. The corresponding topics are (A: Operating Systems), (B: Files), (C: Storage).

2. Each element of the result set is processed. Every node has a weight assigned. Whenever a node is visited by an element, this weight is increased by one. Element A for instance, visits the nodes *Computing, Software* and *Operating Systems.*

3. After all weights are assigned, the tree is processed from the lowest level up. In each level, values are decreased until only one or no node with an assigned value is left in this level.

4. The recommended topic for the query item Q lies along the longest path of the tree with a weight still assigned.

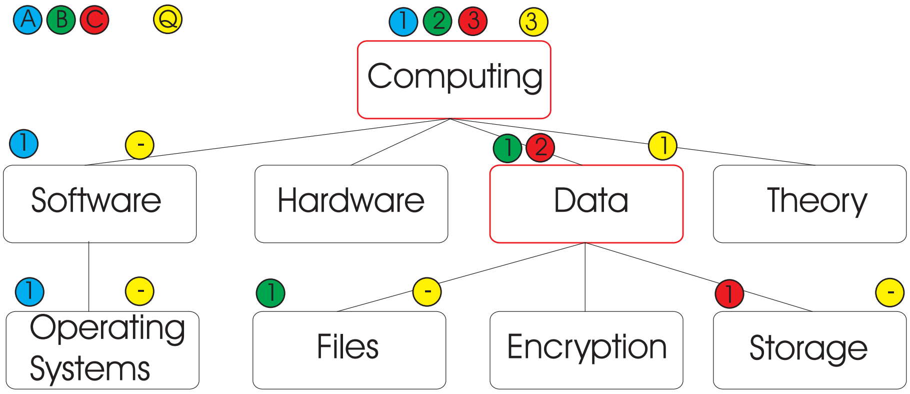


Figure 3.13: Recommendation example

The example presented in the iteration is visualized in Figure 3.13, where the query element is mapped to the *Data* domain. A positive side effect of this algorithm is that the remaining weights (presented in yellow) give an idea how certain a specific topic was hit. In the example, the query is definitely a member of the *Computing* area, whereas the *Data* sub-domain is not guaranteed to apply. The result is finally presented to the user, who can alter the suggestion if needed.

3.4.3 Vector generation

The quality of the suggestion depends on two criteria. First, a certain level of user input is required to actually produce results at all. It is not possible to give a good recommendation without this knowledge base. Second, and even more important is the initial search mentioned in the first step of the enumeration. It suggests itself that the search is performed by the engine introduced in Chapter 2. There is no argument against using the vector space principle to implement the search. The indexing method, however, is a different case. Recapitulating the search functionality and the vector generation for the WSDL search engine, the problem becomes obvious. These vectors mainly consist of WSDL information like operation names, parameter names, service identification, and the like. Using such a vector to search entries from similar domains is not guaranteed to succeed. The only part of a WSDL file that would be adequate for this purpose is the comment section. These comments can be assumed to describe the purpose of the service. In practise, however, comments are not always usable. From over 270 real world WSDL files, only 28 even contain a comment section. And from these files, 25 were generated, holding a line like

WSDL created by Apache Axis version: 1.3 as the only comment. The remaining 3 files, with a textual description of the service inside the description file were Amazon, Google and a service called ws4lsql.wsdl. All other files contained virtually unusable comments, at least where matchmaking for domain suggestions are concerned.

A more experimental approach uses a completely different source to build up the necessary vectors for the initial search. The idea behind it is, that with every Web service comes a Web page. When analyzing the endpoint URL of an arbitrary Web service, the corresponding Web page can be derived by cutting the URL address until a valid page is returned. The endpoint address for the service described by *xCharts.wsdl* for example is `http://www.xignite.com/xChart.asmx`. By cutting the URL to its domain name `http://www.xignite.com/`, the company's Web page can be retrieved. The page clearly describes the service to be of the financial sector.
The problem here is the vector generation itself. Today's Web pages are generally not composed of simple markup where it is enough to delete the tags to retrieve the actual content. Instead, they are overloaded with scripts, tables and other elements which cause an automatic retrieval of the actual content to become a very challenging task. Data mining and information extraction from Web sites are heavily investigated fields. Frameworks [6] exist, that allow users defining certain criteria for data extraction. Thus, it would be possible to generate the desired vectors from these Web pages. The problem, however, is not limited to the extraction process alone.

- It already starts with the location of the Web page itself. The Web service mentioned above is a good example. When created with the .NET framework, a Web service endpoint may also deliver a custom Web page when accessed via a Browser. In most cases though, the returned page will be a default site with just a listing of the exposed methods, which do not hold any valuable information. Furthermore, the need to distinguish between standard error pages (HTTP error code 404) and custom pages arises. It can easily happen that custom error pages are mistaken for actual content.

- Even when using an advanced tool for data extraction, the extracted vector is not guaranteed to really represent information about the Web service in question. Possible reasons are shared domain names among companies or a multitude of services from various domains, provided by a single company.

- Other than WSDL files, where certain elements are obligatory, Web pages are not bound to a certain structure. Therefore, common methods from natural language processing must be applied.

- Language differences in Web pages will cause strongly related pages to show as very different unless some sort of translation methods is facilitated. The required resources are usually quite expensive and limited to a single language. This is the reason why such methods are usually implemented in natural language search engines. Furthermore, the processing capability is always limited to the used resources.

- A complete new indexing infrastructure is needed to make the approach work.

3.4.4 Remarks

Summed up, the process to relate Web services based upon their Web content is not very promising because of the arising problems and requirements. Instead, the already generated vectors are used to search for services to recommend. This way, the recommendation will maybe not be as detailed as it might be based on a complete Web page index, but the recommendation concept is not designed as an exact system anyway. The goal is to give an idea of the domain a service may fit into. For this purpose the already established data structure is sufficient.

For the future however, possible alternatives for the recommendation system will be an interesting field of investigation.

Chapter 4

Result Classification using Statistical Cluster Analysis

Inanimate objects are classified scientifically into three major categories -
- those that don't work, those that break down and those that get lost.

Russell Baker, 1925

This chapter addresses the last issue mentioned in Section 1.5. With the functionality provided by the search engine and the additional metadata being available, searching repositories for WSDL files is now possible. An additional need that arises with the growing number of entries is to automatically create a list of related services that match a given entry. The basic idea is the same that drove the development of the recommendation system from Section 3.4. The big difference is the need to enrich the search-engine's hits with a number of related services that fulfill a similar task, or are of a similar domain without any user input. Apart from the direct relation to the issued query, there is currently no established method to relate these possible matches to each other. Such a functionality would assist in browsing the content of the repository enormously. It is important to understand though that the original search result is unaffected by those relations. The clusters are built for each element of the search result and aim to provide a set of alternatives for each entry in a result list.

In this part of the thesis, an approach is presented that uses statistical cluster analysis to create the desired containment for the most significant matches of a given query. The focus is laid on an efficient algorithm that is scalable to very large service repositories and still supports distributed processing of queries for the generated matches. For this purpose, the usual Euclidean distance for proximity measurement is substituted with the multi-dimensional angle produced by the vector space search engine. Furthermore, the very complex runtime creation of the matrix for the distance measurements is discussed and changed to a more effective method. This change is necessary, because large service repositories otherwise result in cubic runtime complexity and are therefore limited in their processing capacities.

4.1 Prerequisites

This section gives an overview of the most important issues related to general clustering problems and Web service clustering in particular. The principal method is a modified version of the common statistical cluster analysis. Because some of these methods are not directly applicable here, certain adjustments have to be made to ensure they can still be used.

4.1.1 Requirements

Statistical cluster analysis can be used for a broad spectrum of input data, ranging from Boolean values or even nominal scales to relational scales. The more unambiguous the data for the different variables can be set, the more significant the clustering result will be. Therefore, it has to be dealt with the requirement to have float or integer values for a numerical representation of a service description. Furthermore, the cluster algorithm must not be limited to a specific number of variables and/or a maximum size for the stored entries to allow it to be executed on a multidimensional term space. With these requirements in mind, the indexing method can be examined.

4.1.2 General Indexing

When processing a WSDL description in general, the desired result is an index where each characteristic is represented as a dimension of an n-dimensional vector space. This requirement is generally the same as for the vector space engine. An example is shown in Figure 6.1 of the related work section, where three fictional vectors are represented in a three dimensional space. It depends on the used indexing procedure what the dimensions mean. If domain-specific information is characterized by a dimension for example, the cluster analysis will produce a result, where elements of similar domains are grouped and identified as related to each other. How well the desired outcome matches the expectations of a query therefore depends on the quality of the index as well as the used algorithms. The following two types of index structures are possible when dealing with XML based service descriptions.

4.1.2.1 Syntactic Indices

A syntactic index has the advantage of being able to process any valid WSDL file from any source simply because input data is not restricted by any means. In Chapter 2, such an index is used to create vectors for WSDL files and process queries upon them by calculating document similarities. This method is a common approach used in all fields of natural language processing. The list of keywords is then mapped to a vector space, where every dimension represents a characteristic or in this case a keyword. All dimensions together then span the n-dimensional vector space, where n is the number of characteristics that have been quantified.
Per definition, this data structure is enough to describe any document as a single point within the space with the possibility to expand the dimensionality when needed. Therefore, also a cluster algorithm can be applied.

4.1.2.2 Rich Indices

Other than purely syntactic indices, a rich index deals with some sort of specialized information contained in the original input data. The information can be of various structure. The four most important values are listed below.

- **Semantic descriptions:** When a WSDL description is enriched with some sort of semantic descriptions like RDF [63], this data can be parsed and stored for further usage. The information can be mapped to a domain-specific ontology in most cases and processed accordingly. For this purpose however, this possibility is not an option because no real-world service really implements semantic descriptions in the form of RDF tags. The same applies for semantic annotated Web service descriptions (SAWSDL).

- **Domain information:** Other than semantic descriptions that might be entered directly into the WSDL file it is possible to use domain information about a specific service. Unfortunately, the location data, like it was generated in the previous chapter, cannot directly be used to span a vector space. The data itself is of Boolean nature. Either a service belongs to a specific domain or it does not. Although theoretically possible, it renders the output quite useless. For this reason, domain information is used merely to provide additional information about the service in question rather than influencing vector dimensionality.

- **Location information:** It can be gathered from the endpoint of a given service. There are services like GeoIP that allow to determine the location and additional information using the endpoint IP of a service. Other than domain information, location data can be processed by cluster algorithms. To do so, the distance between two services has to be calculated from the location information that was extracted. This data forms a two dimensional plane where clustering can be applied again.

- **QoS descriptions:** Similar to domain and location knowledge, QoS descriptions for performance related aspects of Web services are a type of meta-data that can be gathered for Web services as proposed in Chapter 3. When the evaluated QoS is used to build indices, the clustering algorithm will produce groups of similar performance values. This can be an intended behavior or an unwanted side-effect depending on the desired clusters that should be produced as a result.

Although it is possible to merge information from the above categories with syntactical information it is not recommended to do so. This is because vector spaces built from service descriptions tend to span across a large number of dimensions. Even when additional information like domain information for instance produce an exact match, the multitude of other vectors that have to be considered is likely to move such vectors apart from each other and, therefore, reduces the efficiency of the clustering algorithm. Thus, rich indices should always span a decoupled vector space or a sub-space of the original vector space. Sub-spaces cause the same effect as a decoupled vector space with the only difference, that a complete and combined version could be used when needed. In this particular case however, this possibility cannot be used. An index which is based on QoS will not benefit from an index based on natural language processing, even when the numerical structure is identical. A search which is issued on a combined space would result in a broken search semantic. An example shall clarify this situation. A cluster analysis based on a search string like "credit card verification" expects to find clusters of the same domain. With a merged QoS index however, the result would possibly contain a service that verifies lotto numbers but has the same QoS attributes and list it as very strongly related. Therefore, vector spaces should not be merged across different index strategies.
In this particular case the most feasible and also most depictive indexing method is the syntactic clustering. First, the produced data cloud is of the highest density and on the other hand it forces to make the changes that allow the algorithm to operate on spaces with unbounded dimensionality.

4.2 Basic Concepts of Statistical Clustering

With an already established vector space for an existing WSDL repository, the clustering approach can be discussed in detail. As already mentioned in the introduction, traditional statistical cluster analysis does apply in a Web service environment with certain limitations. Those limitations and how to overcome them is discussed in this section.

4.2.1 Proximity measure

When dealing with metric scale levels as in this approach, some ways exist to measure the distance between two elements.

4.2.1.1 City-Block distance

In an m-dimensional vector space, the City-Block distance between two elements j and k is calculated with

$$d_{jk} = \sum_{i=1}^{m} |x_{ij} - x_{ik}|.$$

This distance measurement is designed to be used for a data cloud where elements are not very different from each other. The absolute value for the difference on each axis is added pairwise. As a result, one dimension with a large deviance results in a large distance for the tuple as a whole.

4.2.1.2 Euclidean distance

Although the City-Block distance is a valid measurement for this purpose, the Euclidean distance bears the advantage of considering the direct distance between two points in the vector space, no matter how large the value for a particular dimension is. The Euclidean distance is calculated as follows:

$$d_{jk} = \sqrt{\sum_{i=1}^{m} (x_{ij} - x_{ik})^2}.$$

A variation of this measurement is the squared Euclidean distance. Its only difference to the standard method lies in omitting the square root for the final value.

Both of the methods mentioned above are theoretically applicable in the current environment. Nevertheless, the practical use shows the limitations and restrictions. The first step when splitting a data cloud in the clusters is to calculate distances for every element in the space. For statistical use, the number of elements is usually limited to amounts of around 100 elements. The number of dimensions rarely exceeds 100 as well. For that amount of data, the values can be processed relatively fast and precisely. For WSDL repositories however, 10^4 entries are considered medium size. Furthermore, the dimensional size of the vector space is also considerably larger. Depending on the input data 10^3 to 10^4 dimensions is a reasonable amount for documents that are of different structure as WSDL descriptions. With those assumptions in mind, the expense to process standard distances can be calculated:

1. All distances for all elements within the vector space have to be processed. Permuting n elements without repeating, results in a Gaussian progression for the number of needed iterations:

$$\#Iterations = \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

So the number of iterations alone results in upper bound of $\mathcal{O}(n^2)$ for the problem complexity. Additionally, each iteration needs to process as many elements, as there are dimensions present. It is possible to pre-compute the distances on a local repository. With a distributed vector space, however, access to all vectors is not guaranteed.

2. In the next step, a cluster algorithm has to be applied, where near elements are grouped by either hierarchical or agglomerative procedures. When assuming the worst case scenario, each application of the cluster algorithm results in a pair of two elements, leaving $n-1$ elements for the next iteration of the algorithm. As a result, the cluster algorithm consumes an additional $(n-1) \Rightarrow \mathcal{O}(n)$ iterations in the worst case.

3. Finally the results, and possibly results of remote vector spaces can be combined and displayed.

Besides these performance-related drawbacks, there also arises a problem where documents of different length are not considered in one cluster because they are represented by a different cardinality. As an example, two descriptions are taken which are both of the financial sector. After the indexing phase, both documents are represented by the keywords "interest" and "investment". Because one document is longer than the other, the cardinality is different which means that the Euclidean distance puts those two documents in different clusters even though they are strongly related.

With all these restrictions it is obvious that a better way to compute distance values has to be found; One that copes with performance and precision at the same time.

4.2.1.3 Multidimensional Angle

An elegant solution for similarity or distance ratings in an n-dimensional vector space is to use the multi-dimensional angles introduced in Section 2.5.3. In this approach, it is not the absolute position of two points (p, q) in space and the Euclidean distance between them but the cosine of the angle between two vectors reaching from the origin to p, q. With this method, the imbalanced rating of longer documents compared to shorter ones is not an issue anymore. In the example presented above, the vectors produce the same angle and are, therefore, considered as close to each other.

In terms of performance, this approach also produces better results. When taking a sample vector space with 10^4 dimensions, it can be assumed that a single document does not incorporate all different dimensions. Therefore, dimensional reduction can be applied while computing angles for two vectors. In this particular case, a dimension is only considered when it is present in both vectors. Otherwise, it would drop out of the equation anyway. The results show the saved computing time for a single vector query.

This approach also provides the possibility, to produce every angle for a single document in the same iteration by storing denominators and reusing them for other elements as shown in Section 2.6.3.2. With this method, the expense to produce the distance matrix can be reduced to $\mathcal{O}(n)$ which is an acceptable but still improvable growth rate. For high-performance search-engines where results are created on the fly, this leaves just two options.

- All distances are processed in advance and re-calculated as soon as a new entry is added to the vector space. This results in a huge amount of processing time for adding elements. Furthermore, it strongly affects distribution capabilities of the whole approach.
- Clusters are built for results of a search query only. That limits the amount of elements to a reasonable size and besides improves visibility of the result.

In this implementation the latter solution is preferable because of the distribution capabilities. With the limited size of the repository, the runtime overhead is tolerable.

4.2.2 Cluster algorithm

For the final cluster algorithm, quite a large range of possibilities exists but it is out of scope of this chapter to discuss all advantages and disadvantages.

In general, partitioning and hierarchical methods can be distinguished. Depending on the underlying data structure, various different algorithms to create clusters in data clouds exist. The k-means method [31] is among the most popular of them and, therefore, posed the first choice for a possible application. Structurally, k-means is a partitional cluster method. The algorithm assigns each point in the data cloud to the cluster whose center is nearest. The center is simply the average of all points in the cluster. For multiple dimensions that means that each coordinate is the arithmetic mean of this dimension for all points belonging to this cluster. The original k-means algorithm as proposed by MacQueen [35] consists of the following steps:

1. Choose the number of clusters k.
2. Randomly generate k clusters and determine the cluster centers, or directly generate k random points as cluster centers.
3. Assign each point in the data cloud to the nearest cluster center.
4. Shift the new cluster centers according to the added points.
5. Repeat steps 3 and 4 until some convergence criterion is met.

With some minor adoptions and performance optimizations this algorithm is widely used to generate clusters for all different kinds of quantified data. For this purpose though, there are two drawbacks. First, the number of clusters has to be predefined. Unfortunately, there is no way to estimate how many items of the original search result are strongly related to each other. Therefore, a rule of thumb – like the elbow criterion for example – would have to be applied to appraise the number of clusters that make sense in the final result. The second drawback of this method is that it does not yield the same result with each run, since the resulting clusters depend on the initial random assignments. For these reasons a very similar but hierarchical method with an agglomerative approach was used for this problem. Unlike the k-means method, each element is considered as a single cluster at startup. With each iteration, new clusters are built and contained elements are grouped to the new layout before the algorithm is started all over. Just like the k-means algorithm, all elements are finally distributed to a cluster with the only difference that the number is denoted by the iteration step and not defined at the start. For a better visualization, however, a hierarchical method with a centroid fusion algorithm is preferable to the partitional approach. It provides a good performance for the fusion process and is limited in the necessary iteration steps. The algorithm is applied as follows:

1. Search for the pair in the distance matrix with the minimum distance $d_{min}(a, b)$.
2. Create a new distance matrix where distances between clusters are calculated by their mean value $d(\bar{a}, \bar{b})$
3. Save the distances and cluster partitions for later visualization.
4. Proceed with step 1 until the matrix is of size $n = 1$ which means that only one cluster remains.

To give a better understanding of the involved algorithm, an example with a matrix of size 5 is provided, including all necessary steps for the matrix reduction. The algorithm starts to build the initial matrix by querying all relations to item A. If a relation exists, it is entered into the matrix with its corresponding value and zero otherwise. A sample initial matrix is shown in Table 4.1(a). Issuing a query with the vector of item A for example would result in a rating of 0.8 for item E, 0.55 for item D and so forth. Each element is processed this way until the necessary elements are entered into the compressed matrix. Then the matrix is decompressed, to ease the following iteration steps. To do so, the main diagonal has to be filled with the element which represents the strongest relation (in this case 1). All other values can simply be mirrored due to the bijective nature of the document relations.

Then the algorithm is processed in the above mentioned order. The highlighted element is the minimum distance in the current reduction step and, therefore, determines which elements will be combined for the next step. Higher values mean a smaller distance or put differently, higher similarity, because they represent the cosine between two vectors angles. The values on the main diagonal are not considered here. Furthermore, this value denotes

Table 4.1: Matrix reduction example

(a) Compressed initial Matrix

	A	B	C	D	E
A	-	0.3	0.5	0.55	0.8
B	-	-	0.7	0.6	0.85
C	-	-	-	0.9	0.4
D	-	-	-	-	0.1
E	-	-	-	-	-

(b) Decompressed initial Matrix

	A	B	C	D	E
A	1	0.3	0.5	0.55	0.8
B	0.3	1	0.7	0.6	0.85
C	0.5	0.7	1	**0.9**	0.4
D	0.55	0.6	**0.9**	1	0.1
E	0.8	0.85	0.4	0.1	1

(c) Reduction step 1 (4 Elements)

	A	B	$\overline{CD}$	E
A	1	0.3	0.525	0.8
B	0.3	1	0.65	**0.85**
$\overline{CD}$	0.525	0.65	1	0.25
E	0.8	**0.85**	0.25	1

(d) Reduction step 2 (3 Elements)

	A	$\overline{BE}$	$\overline{CD}$
A	1	**0.55**	0.525
$\overline{BE}$	**0.55**	1	0.45
$\overline{CD}$	0.525	0.45	1

(e) Reduction step 3(2 Elements)

	$\overline{ABE}$	$\overline{CD}$
$\overline{ABE}$	1	**0.4875**
$\overline{CD}$	**0.4875**	1

(f) Termination step

	$\overline{ABCDE}$
$\overline{ABCDE}$	**1**

the coefficient that enables the visualization of the cluster. In each reduction step, the new matrix is shrunk by one element and the new matrix elements are calculated as an arithmetic mean value until the matrix reaches its trivial state of two remaining elements as shown in Table 4.1(e), where the last distance is showed and the matrix reaches the termination state 4.1(f). The last remaining value, which is $0,4875$ in this case, denotes the distance of the centers of the two remaining clusters before they are fused. An agglomerative algorithm always processes the whole data cloud until one single cluster remains. How the reduction steps are used can be decided later.

4.2.3 Results

In the result phase, the clusters can either be visualized in an elbow-diagram or as a dendrogram. The advantage with already normalized values is that they are always in a range $[0,1]$. Additionally, the distance matrix gives a good idea of the different steps the algorithm went through. With all distances calculated in the above example it is finally possible to visualize the cluster distances in such a dendrogram. Figure 4.1 shows the strong relation of the items *C,D* and *B,E*. Although the layout might suggest otherwise, the dendrogram is not to be mistaken with a hierarchical organization because the cluster elements are equal. It merely describes the distances of the produced clusters.

With this example it also becomes clearer why it is so difficult to use predefined numbers

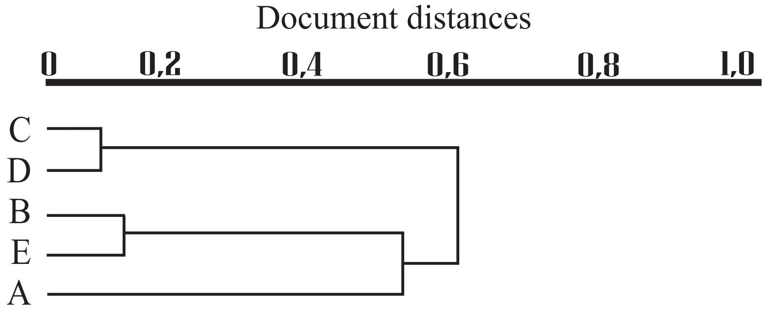


Figure 4.1: Dendrogram visualization

of clusters or a preset termination target. With the clusters set to 2 for example, the result would just show the elements *C,D* and *A,B,E* as part of a cluster but not how strong they are related to each other. Furthermore, when setting a termination distance of $0,75$, the algorithm would end up with just one cluster which contains all elements. By looking at the dendrogram, however, it immediately becomes clear which elements are grouped tightly.

In the next section, an explanation of how to implement this rather theoretical approach in the existing search engine will give further insight into the involved methods.

4.3 Implementation

To evaluate the efficiency and usability of the proposed approach, an implementation was embedded into the existing search engine which was presented in Section 2.6. A discussion of the involved performance measures and scalability issues based upon that implementation forms the main element of this section.

One criteria that was casting for the decision to implement the prototype with a Web based interface was that it requires no additional features or installations to run it and test the underlying functionality. The environment of this implementation is provided by the VitaLab[1] laboratory, just like for the implementation presented in Chapter 2. Supporting various kinds of frameworks like ASP.NET on IIS, Apache Axis and the like, this environment gives the freedom to choose the best solution to implement a particular research prototype. Each of the integrated machines encompasses 2 Dual Core Intel Xeon CPU's with 3,2 GHz, 1GBit network interface, 2 GB of main memory and 10krpm RAID 1 HDs. The machine where V-USE is deployed runs on Microsoft Windows Server 2003 Enterprise Edition with Service Pack 1.

[1] http://www.vitalab.tuwien.ac.at/

4.3.1 Plug-in location

The major elements were already introduced in the previous sections. Figure 2.3 gives an overview of the most important elements. This structure allows a decoupled development of the search engine and the application for the user interface. The layout supports a possible usage of the functionality by other means. Because of this structure it is also possible to plug in the clustering functionality at two different points.

- The back-end where the vector space itself is handled. The advantage of this location is simply a better debugging capability and a increased performance. The performance gain is a result of the adjacency of the search engine. Processing n queries for a n-element cluster means an equal amount of Web service invocations if handled by the front-end application. Furthermore, the execution time for each single query as well as the overall times can easily be extracted this way by simply logging them to the Java runtime environment.

- The front-end on the other side, also bears one big advantage. The original vector space engine is designed to allow repositories to be split into several smaller repositories while queries can still be executed upon them as if on one single repository. The already implemented method for runtime weighting and normalization is able to map invalid vectors from spaces with different dimensional layout to any vector space. The gained benefit is the possibility to split repositories in several sub-repositories whenever the performance is not satisfactory and still keep the dimensional structure and relations intact. For the clustering approach it would, therefore, make sense to execute the queries to fill the cluster matrix at the client side where the distributed spaces are joint in the first place. Nevertheless, it was decided to implement the prototype with the method mentioned first, because distribution capabilities are no issues upon development.

As a result, the clustering functionality is realized as an extension of the original Web service and can be seen as a plug-in for the vector space engine on the server where it is deployed. The system layout as depicted in Figure 4.2 shows the general architecture, including the cluster plug-in as it is implemented in the back-end based approach.

4.3.2 Process flow

With this structure, a completed request causes the following order of events:

1. A request to the Web service endpoint reaches the axis servlet. Whether the request was sent by the front-end or another VSM implementation is not important at this point.

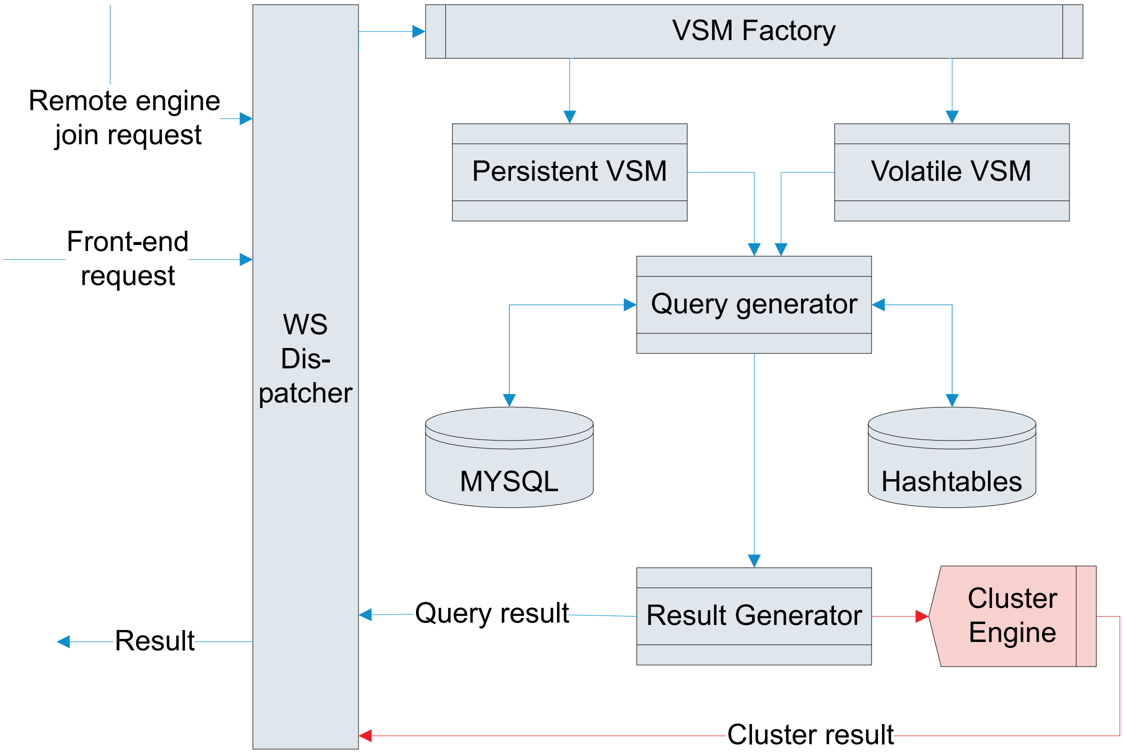


Figure 4.2: System Architecture with cluster plugin

2. Depending on the request signature, the VSM Factory decides whether to instantiate the persistent or volatile version of the search engine. The volatile engine is blanked every time the tomcat servlet engine is restarted, because the vector matrix is implemented with hash tables. In the persistent version, this limitation does not apply naturally. Instead, the constructor makes sure that the requested database structure is valid. The code for the table generation of the persistent engine is listed in A.2. If not all tables exist they are created on the fly, thus providing an empty vector space on startup.

3. In case of a normal query, the query generator decides whether to directly access the local hash tables, or create the corresponding SQL statements to fetch the same data from the database. It is important to remember that those queries effectively implement a dimensional reduction of the original query vector. This means that even for vector spaces with a large dimensional count, the processed data can be kept low. Search queries normally do not exceed a size of 10 words, and, therefore, dimensions. Documents that do not contain any of the query words at all, are automatically omitted (see Section 2.6) and, therefore, do not cost any processing time. This advantage does not apply to the cluster engine because when building the matrix, every element of the original vector must be taken into account and results

in large query vectors of 100 and more dimensions. Furthermore, it is almost certain that every document keeps at least one of the vector elements and, therefore, will also produce a query rating. The actual numbers are discussed in Chapter 5.

4. After the query is executed, the result generator takes over and either produces a sorted list of results that can be handed to the Web service dispatcher, or fill the matrix of the cluster engine.

5. The cluster engine decompresses the matrix, and reduces it step by step until the cluster algorithm is successfully finished. This reduction can be done in constant time, because once the matrix is filled properly, the contained elements are already of top priority.

6. The Web service dispatcher can now deliver the results to the caller and free all used resources.

The above scenario describes how the back-end reacts to a request and executes the corresponding operations. A more precise evaluation of the involved time-frames and the delay caused by the cluster algorithm can be created by utilizing the prototype implementation with the Web interface [13]. The resulting values are strongly coupled to the performance of the original implementation since it is used to find inter-document distances. A complete evaluation is given in Chapter 5, where all elements of the application are discussed.

Chapter 5

Evaluation

I think that God in creating Man somewhat overestimated his ability.

Oscar Wilde (1854 - 1900)

In this chapter, the introduced methods and implementation are evaluated. The first sections will focus on the implementation that is available for use via the Web client. This encompasses the Chapters 2 and 4. The other section deals with the implementation which is available as a prototype only, without any public access. The included methods are mainly of Chapter 3. The used case study deals with the involved issues from the perspective of a public registry or public service because they tend to be more heterogeneously in their structure and, therefore, bring up more problems than in a corporate environment.

5.1 Web implementation

An exact and significant evaluation of prototypes and implementations for Web service technology is always a very challenging undertaking. Partially because the usefulness of a new method is hard to proof in a prototype but basically because of a lack of sufficient real-world services. A restricted amount of descriptions for actually working services is not big an issue for fundamental implementations that deal with services at a functional level. For search engines and metadata related research though, this fact poses enormous problems. Repositories with around 1000 distinct services are often too small to present a real problem for the involved algorithms. The only possibilities left are to either produce copies of already existing services, or create dummy services with no actual implementation to simulate larger repositories [36]. In either case, the drawbacks are quite obvious, being either duplicate entries and, therefore, insufficient search capabilities, or biased results because of the generation algorithms to produce the services. In most cases, the best method is to base an evaluation on a real repository and assess performance and scalability capabilities by extrapolation.

5.1.1 Prototype execution

The file base is a critical issue for the numerical evaluation of such an approach. The repository used for testing purposes contains a set of 275 distinct WSDL files from different sources. Part of them were extracted by the UDDI cross-reference download, which is no longer possible for the biggest ones (Microsoft and IBM) since they were shut down. Therefore, UDDI is considered as a secondary option to gather public service descriptions for current implementations.

The second source was the public Web service registry from xmethods[1]. Other than UDDI registries, xmethods provides a multitude of possibilities to access the underlying services reaching from standard Web pages to RSS feeds and even a Web service interface to query the database. Additionally, the functionality to populate Web services with a set of input parameters is provided. This way, the services can be tested for their availability directly from the Web site.

With this repository, the VSM engine can be populated and afterwards readied for a cluster analysis. To produce demonstrative results, two different queries for the initial search were chosen. One aims to find the description of the popular Amazon Web service. The corresponding query string is ”Amazon web service”. The other one tries to find a service for verifying credit card numbers. Here the query is ”Credit card verification”. These strings are simply entered into the search field of the Web site, while background activities can be monitored at the server.

Both of the initial searches took 16 milliseconds to finish upon the 275-Element repository, using the persistent VSM engine. Looking at the queries one will see that both of them encompass three dimensions. Starting from this point, the Files ”wsCreditVerify.wsdl” and ”AmazonWebService.wsdl” were the best and also desired results found by the search engine. After completing the search, the clustering functionality is available for each result by clicking the ”cluster” button for the selected result element. Completing this initial search as fast as possible is an important matter for the search engine. Otherwise the Web page would respond slowly which is a very limiting fact for such a facility. The most important steps when processing the query are as follows:

1. The query word is taken and normalized by the same algorithm that filters the WSDL files. With the first string, the result is ”amazon web service”. This normalization filters spaces, eliminates alphanumeric signs or cuts spaces where they are not needed.

2. Now the result can be put into a vector. To do so, a weight has to be applied for each element. Balanced search is achieved with a weight of 1 for each dimension. So

[1] http://www.xmethods.org/

the final vector is (1,1,1) with (amazon,web,service) as the corresponding terms. It is also possible to weight terms otherwise, based on linguistic resources for example.

3. In the next step, all remote repositories are queried for their statistics with the given vector as input. This is technically like an ordinary query with the difference that results don't have to be sorted, which saves computing time. Also it can happen simultaneously on each remote host.

4. All statistics are merged and finally each vector space can be queried with the vector and the accumulated statistics. The results are again merged according to their relevance and displayed afterwards. When processing local vectors only, step 3 can be omitted. The example from above produces a relevance rating of 0.953 for the amazon query on the local repository.

To proceed with the clustering, the next step in the execution chain is to re-create the original vector of the initial root Element. The Amazon Web service will serve as an example. Here, the WSDL description file reflects with 210 dimensions/keywords in the vector space. The list of all indexed keywords is listed in Appendix B.1. This screenshot is taken from the original output of the search engine's clustering implementation. The high dimensionality means, the query that has to be executed on the vector space for the matrix creation will cause a significantly larger load than a normal search query.

The settings of the cluster engine are such, that the initial matrix is of size 15x15. This value can freely be chosen in the Web application. It defines, how much results of the query with the root element are taken into the matrix. It's important to remember, that the queries are executed the same way an ordinary query gets processed. Therefore, the matrix is filled with elements from the whole repository. It also means, the same amount of queries has to be executed to fill the matrix. The number was chosen because it provides a fair tradeoff between good performance and a meaningful result. The 15 elements that build the matrix of the Amazon-Cluster took 165,5 ms average, plus an additional 65 ms for the sorting algorithm of the result. Compared to the 16ms of the initial query, the impact of the dimensional reduction algorithm becomes clear. That means, a matrix of size 15 takes about 3,5 seconds to fill. This time does not directly depend on the size of the underlying repository but on the speed of the executed queries. Therefore, with a growing size of the desired elements comprised by the final cluster, the time to fill the matrix grows linear, independent from the repository size.

After the cluster matrix is filled, the cluster algorithm itself can proceed in its execution and reduce the initial matrix step by step until it is of size 1. The matrix reduction for a 15-elements matrix takes a constant time and finishes in approximately 150 ms on the test machine. The result of the overall procedure is a list of n elements and the clusters they belong to. The complete matrix from its compressed form that is the result of the query until the algorithm reaches the trivial state is depicted in Appendix B.2. The processing steps are exactly as described in Section 4.2.1.3:

1. The vector resulting from the initial search is used as a normal query. The best 15 results are the elements of the compressed matrix. Alternatively, it is also possible to use the original result of the query and fill the matrix with these elements. Doing so would result in a matrix that encompasses a ”thinned” space population, because only elements of the result are considered. For the time being, the formally complete method with a whole vector is used. The other possibility is discussed later.

2. A query with the vector of each element in the matrix must be processed to fill the matrix fr all other elements.

3. The result is decompressed to a 15x15 matrix. To do so, the main diagonal is filled with 1. All other elements are mirrored.

4. In each step, just like in Table 4.1, the biggest element (or smallest distance) except the main diagonal is searched. These elements are combined. The distance is memorized and shown in the output as the cluster coefficient for each step. See Figure B.4 for a screenshot of all coefficients for the amazon cluster in particular.

5. The elements in the previous step are combined by shrinking the matrix by one dimension. All elements in the corresponding line and column are fused to represent the average angle of both elements. Now the previous step is repeated until the matrix is of size 1.

To correctly read the Appendix and all values, Figure B.3 shows all elements of the initial matrix in a numbered order. After combining two elements, however, say element 2 and 6, the new element 2 is already a combination of 2 and 6 ($\overline{26}$) and element 6 is deleted, while all successive elements move forward by one. Therefore, after step 1, the elements do not match the numbers of the cluster coefficients any more. They rather entitle the line and column of the reduction matrix, which is marked red besides. The similarity values of the clusters after each reduction step can be visualized in a graphical representation.

Figure 5.1 shows how close the combined elements were after each reduction step that was carried out. This graphic shows a complete cluster analysis, with no termination before the last cluster is built. It shows very tight relations between the first four combinations of the amazon cluster and the first three of the credit card service. Medium distances of 0.4 - 0.7 can be seen as moderately relevant, while those below are of low significance. Alternatively, it is possible to define a termination point either by setting a maximum distance a cluster may reach or a maximum number of clusters that should be built. When using a maximum distance, all elements in a more or less tight cluster are considered to be strongly related to each other, while the individual distances are not viewed with special concern. This allows quite easily to visualize cluster results, because individual distance values don’t have to be displayed. On the other hand, the result looses a lot of its value when omitting these elements. The decision which method to use, or if a complete analysis is preferable to one with an early termination depends purely on how the result should be presented to the user.

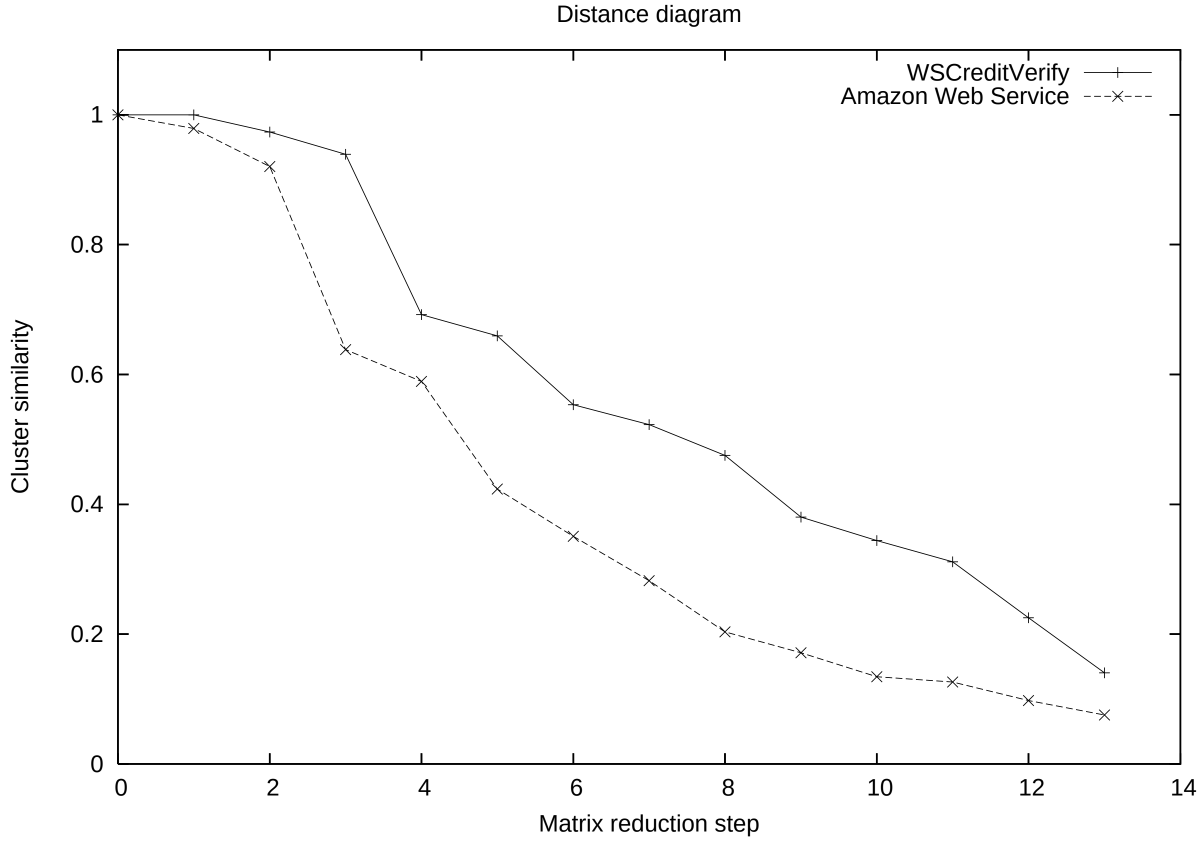


Figure 5.1: Cluster similarity diagram

In this case, the complete result is used. The previously presented diagram however, is not enough to know which elements are grouped. For this purpose, the distances are visualized by using a dendrogram. See Figure 5.2 for the example of the Amazon-Cluster with 15 elements.

When examining the graphic, the four tightly clustered elements mentioned above can be identified. Note that the elements for the numbers in the dendrogram can be found in Table 5.1 or Figure B.3 of the appendix. First is a very tight cluster of Amazon and "ECOWS WS sample" which essentially is a copy of the Amazon WSDL file. This file was injected to ensure that the cluster algorithm puts them in the same cluster with the minimum distance. The next cluster (3 and 4) consists of variations of the Google Web service. Services 2 to 5 are various versions of the same description either gathered from different sources or being different versions, changed by the provider. This cluster is later joined by element 2 at a distance of about 0.4. The first moderately tight cluster is built by "xExchanges" and "Exchanges", both services to query worldwide exchange rates, joined by "InsiderTransactionInfo", also a service for the financial sector.

Altogether, the search and cluster algorithms produce the desired results. A search

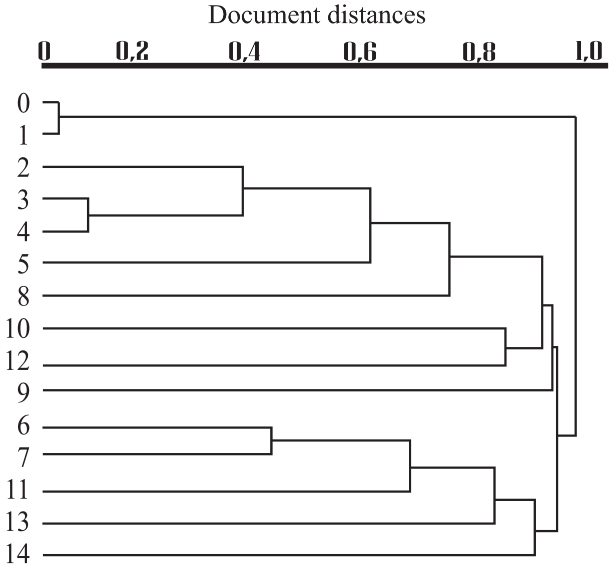


Figure 5.2: Dendrogram for ”Amazon.wsdl”

query can easily be enhanced by creating a cluster analysis of the result contents. For the Amazon service a search engine like Google produces a close cluster because they comprise similar functionality like search and query execution. Other elements, like the above mentioned from the financial sector, are not strongly related to Amazon directly, but their close relation to each other is recognized by the algorithm. From a usability point of view, there are still two possibilities left that have to be considered.

When creating the initial matrix, there are essentially two ways to do so.

1. The method used in the example is to execute a query, then select one of the results and do a subsequent query with this vector as the search element. The best n matches are then used to produce the initial matrix.

2. Another possibility would be, to use the result of the initial query to populate the matrix elements without an additional query process. The difference would be that all elements contained in the matrix are directly related to the query string. On the other hand, it would also mean that resulting clusters are not complete but may comprise additional elements.

Again, both methods are theoretically possible and sound. It is simply a matter of user preferences what to actually use.

5.1.2 Performance

As mentioned in the earlier chapters, it is quite difficult to measure performance without a decent set of elements in the repository. Therefore, the performance evaluation is based on

Document Number	Service Name
0	AmazonWebServices
1	ECOWS WS sample
2	GoogleSearch
3	GoogleSearch(2)
4	google_search_service
5	GoogleSearch(1)
6	Exchanges
7	xExchanges
8	ElmarSearchServices
9	ws4lsql
10	WolframSearch2
11	InsiderTransactionInfo
12	JetFoldersService
13	xHoldings
14	ZacksCompany

Table 5.1: Document names

an extrapolation of existing elements to demonstrate scalability issues. Furthermore, it has to be kept in mind that the presented implementation is still a research prototype where the time to optimize performance is limited. The matrix generation process for example was accelerated to 40% of the original time by introducing proper database indices and adjusting the queries accordingly. There are still possibilities to enhance further, but those enhancements are not essential for the approach itself. The actual numbers are shown in Table 5.2.

The measures where taken with the same query used in all the above examples, with the query string being ”amazon web service”. For the cluster creation, the matrix size is again set to 15 elements.

	Repository size [Files]		
Performance Element	274	549	1096
Cross-query average [ms/element]	167,2	278,1	580,3
Sort time[ms/element]	62	140	282
Keyword retrieval time [ms]	16	15	18
Original Query time [ms]	31	53	92
Matrix reduction time[ms]	110	110	110
Overall time for size 15 Matrix [ms]	3750	6112	12322

Table 5.2: Performance comparison

The cross-query time entitles the time used to execute a query that fills one line of the initial matrix and is an average of all 15 queries. It can be seen, that it grows linear each time the repository size doubles. Directly related to it is the overall time used to execute the whole cluster algorithm. This proves that the theoretical assumption to be able to execute a cluster algorithm with less than $\mathcal{O}(n^2)$ effort is correct. However, a linear growth is not the best result for the cluster algorithm. When looking at the original query time in the table, the impact of the dimensional reduction becomes obvious. Instead of doubling the time to process a query, the necessary time barely triples for a repository of four times its original size. And there is still room for further enhancements in this direction, by optimizing the generated database queries for instance. At the moment, the query generator is optimized for normal search queries, because the search engine is still the main focus. Because of the heightened possibility of term occurrences in a cluster query, new ways to reduce the query time for whole vectors have to be found. The times for retrieving the keywords of a single vector are remaining more or less constant, because they can be handled in a simple query. Small fluctuations in the exact values are caused by the java timestamp functionality which is limited in its precision and varying speed in the database connection.

Another performance-relevant issue is the persistence mode, the search engine is set to. The volatile form shows a much better performance, of course. Here the queries are not executed on a database but directly on the memory-based hash tables. Nevertheless, this does not affect scalability issues in the first place. The measurements and estimations of effort presented in this chapter basically still apply. The change merely effects the concrete execution times. That means, when using the volatile form of the search engine, queries are executed much faster but the grow rate for the computation expense is the same as for the database.

An important aspect of this research is implied by the general structure of the search engine itself. As already mentioned in the previous chapters, the search engine is designed to work on distributed repositories. That means, queries can be processed on the local repository with a single request, or on a compound repository that acts like a large one as explained in Section 2.6. Because the steps needed to fill the cluster matrix are nothing more than large queries, it is possible to process them the same way as an ordinary distributed query. To do so, the cluster processing has to be moved to the front-end side, because this is where the final result list is generated. The current implementation uses the server-side method because it is easier to debug and implement, but when the query processor is optimized for the clustering requests, it can also provide a client-side version for the cluster engine that is able to take advantage of the original distribution capabilities. As a result, it would be possible to define a maximum time, a search or cluster request can take before it has to be split in two separate repositories [49]. The split parts could then be processed in parallel which increases scalability and of course performance. When looking at the values from the server side evaluation in Table 5.2, it becomes obvious that the cross-query time acts as a bottleneck for the whole approach. Figure 5.3 visualizes the performance gain for the cross-query times when the distribution capabilities are enabled.

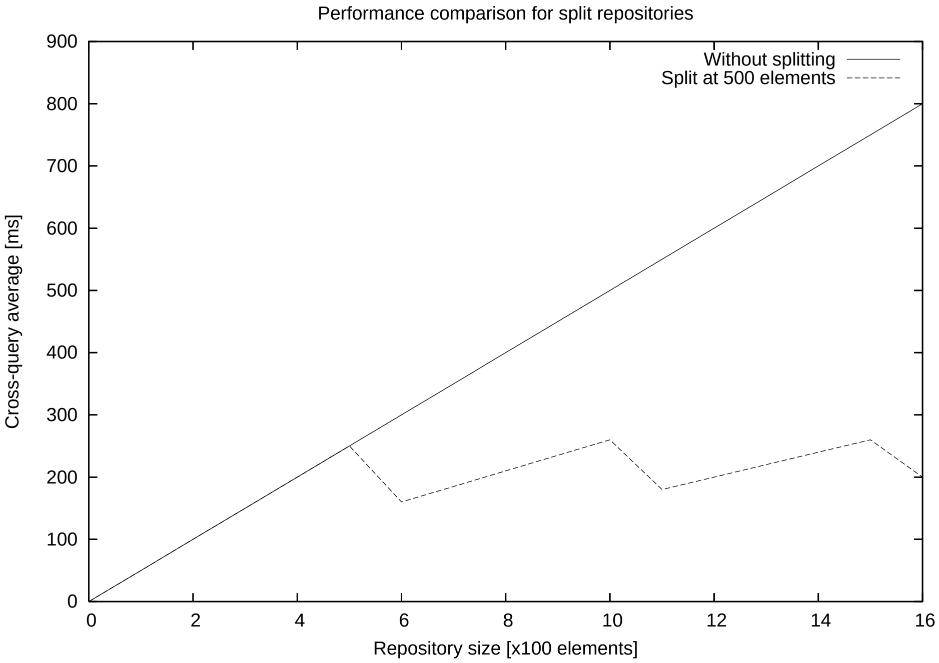


Figure 5.3: Performance gain with split repositories

The partitioning was such, that repositories are split once they reach 500 elements. Then, an empty vector space is created and all elements are evenly distributed. The resulting federation is connected via the *VSMJoiner* Web service of the front end. This Web service allows to process distributed vector spaces as described in Section 2.6. The queries for the matrix generation were run on the federation which produces the result in a fraction of the original time because of the parallel processing capabilities. A small overhead of 10 ms in the processing time occurs because of the additional effort to merge the results once they return from each peer. Without the complete implementation of the clustering approach on the client side, it is only possible to run the matrix generation queries on the distributed version but since it showed that those queries take most of the time, the speed gain is considerably big. This method to speed up the whole process allows to define an upper bound for the execution time of both, the original queries and the cluster algorithm as a whole. It was not necessary to activate any of these performance restrictions while evaluating the approach. The processing speed is very satisfying for the repository at hand. Even for very large repositories, splitting them will be the last way to increase performance. First, the implemented methods for dimensional reduction and the early termination facilities will be exploited as far as possible. The ability to join split repositories is of much greater importance where registry federations are needed. It removes the constraint to decide where to actually search for a Web service, as long as the largest ones are joint.

In a nutshell, the performance of the implemented approach lives up to the expectations and the original concept. At the same time, there is still the possibility for performance enhancements and tweaks, especially where the database access and query generation is concerned.

5.2 QoS prototype

The second part that requires a thorough evaluation is the concept to bootstrap QoS information for unknown service implementations. Other than the search and cluster engine of the previous section, where it was possible to access every point throughout the whole implementation, the QuATSCH tool has to cope with unknown services and servers which sometimes complicates evaluation procedures. To minimize these influences, a simple Web service with a pre-defined behavior was developed that allows to demonstrate the accuracy of the proposed method. The `QoSTestingService`[2] consists of the following operations:

- waitTenMs
- waitHundretMs
- waitTenSec
- waitOneMin

The operations of the Web service just wait the specific period of time (as encoded in the operation name) before they return. By using these operations for evaluating the approach, it is possible to clearly determine the measurement accuracy of the different QoS attributes introduced in Section 3.2.2.

The *QoSTestingService* is deployed in the VitaLab[3] environment on a Dual Pentium Xeon with 3.2 GHz and 2 GB RAM. The client machine where the QuATSCH tool is installed and invoked is a Laptop with an Intel Core Duo T2400 (1.83 GHz) with 2 GB RAM. This way it was possible to test the tool in both, a real-world world environment and the high-speed environment of the local lab which is similar to a corporate network environment. For the real-world simulation discussed below, the client machine is connected to the Internet using DSL with an approximate downstream of 4096 kbps and 512 kbps upstream.

Figure 5.7 to 5.6 show the measurements where the waitHundresMs operation of the QoSTestingService Web service was accessed from outside the lab through the WAN connection. The focus lies on this operation because it reflects usual execution times around the Web quite well. Figure 5.6 shows the evaluated execution times along with an average value as the dotted line. The average execution time was estimated to 119 ms (milliseconds) for a total of 100 requests. The variation to the real execution time of 100 ms is caused by

[2] `http://copenhagen.vitalab.tuwien.ac.at:8080/axis/services/QoSTestingService?wsdl`

[3] `http://www.vitalab.tuwien.ac.at`

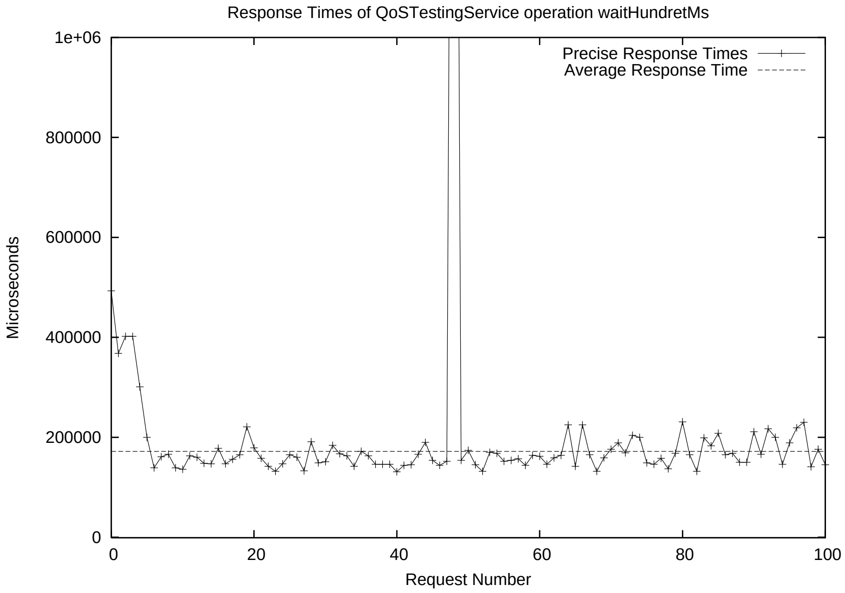


Figure 5.4: Response times

the varying network latencies visualized in Figure 5.5 on the one hand and the spike in one of the response times on the other. Nevertheless, it was possible to figure out the service execution time with a precision of ±20% for the 100ms range in a real-world environment. In the lab environment, where latencies are more stable and, therefore, produce a smaller fluctuation in the statistic calculations, it was possible to assess the execution time of the 100 ms operation with ±1% deviation and the 10ms operation with ±8-12% deviation.

One has to keep in mind, that in a real-world scenario, a service where the execution may exactly take 100ms could by all means take 150 ms to respond after all. A possible reason would be an overloaded server, where requests are scheduled for execution. From the client side, this would just result in a delayed response and, therefore, it is perfectly valid to just rate the execution time with 150 ms.

All the values presented above are results of single executions with at least one second of time difference. To get an idea how the service behaves when it receives a large amount of parallel requests, the same service and the same operation was measured during a throughput test. Each throughput test run consists of 1000 individual requests. Table 5.3 shows a comparison of the evaluated values. It can be seen that the operations per second is quite high, nevertheless, the limit of the Axis framework was reached because it produced quite a large amount of failed requests when further increasing the number of requests per throughput test-run.

Besides demonstrating the accuracy of the measurement it is also interesting to see the evaluation employed on a real-world Web service, e.g., Google in this case. In order to evaluate the `GoogleSearch` Web service, more precisely the `doGoogleSearch` operation, the template-based invocation mechanism is needed to be able to invoke the service, because

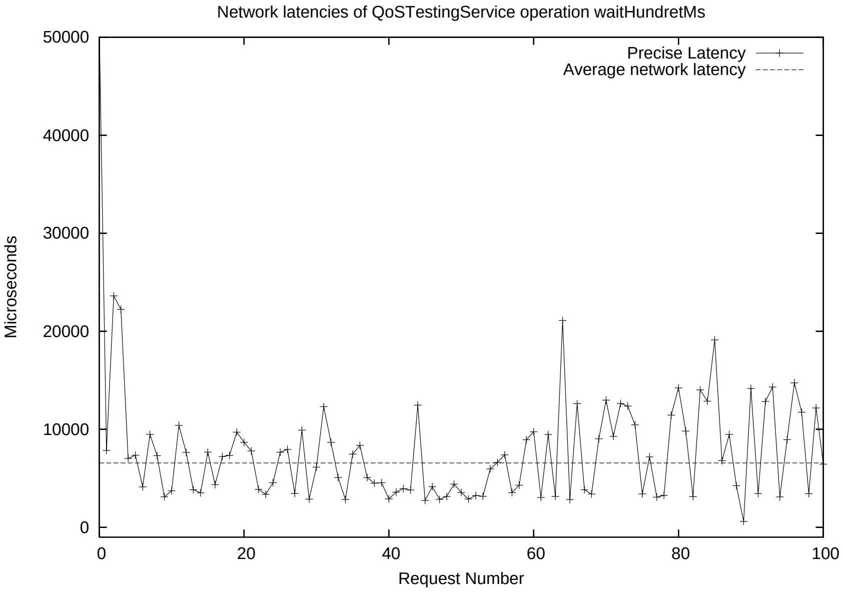


Figure 5.5: Network Latencies

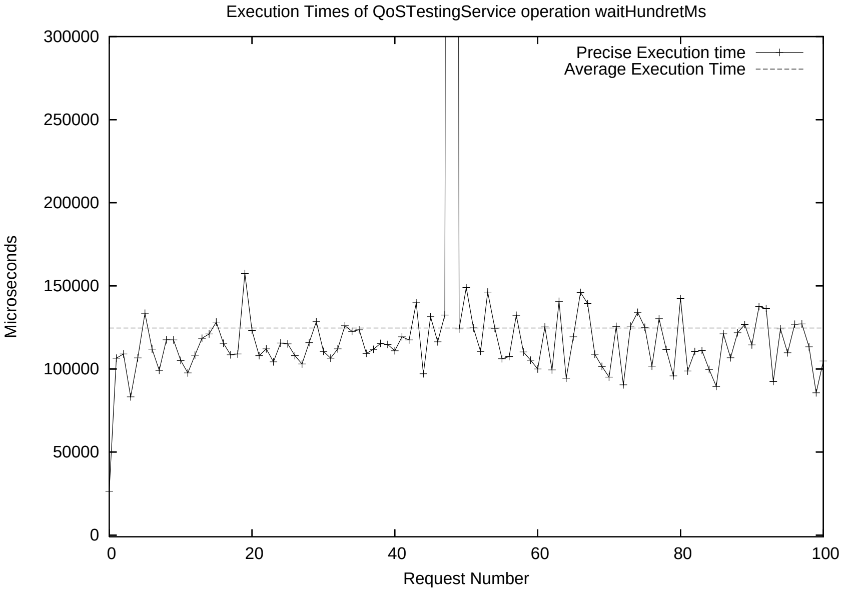


Figure 5.6: Execution times

	Throughput test-run	Standard test-run
Execution Time	127 ms	124 ms
Response Time	186 ms	172 ms
Average Latency	8,1 ms	6,5 ms
Operations per Second	76,8	———
Scalability	0,92	

Table 5.3: Test-run comparison

`GoogleSearch` requires a valid key to use the service.

QuATSCH automatically generates a default template during the evaluation, thus the client only has to provide some reasonable values for the service invocation. In this case the template from Listing 5.1 was used, where a key and a search query are specified. The search string is ”Monica Bellucci”, to make sure that a reasonable amount of results is found. Please note that no throughput tests were conducted on the Google search service, because this might be rated as a denial of service attack by Google on one hand, and because free keys issued by Google are usually limited to a small amount of search requests per day.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<service name="GoogleSearch">
    <operation name="doGoogleSearch">
        <param name="key" type="string">
          <value>kcXJusVQYARSFq7T+4R+Kyqq/cqQAodqm</value> <!-- key is not valid -->
        </param>
        <param name="q" type="string">
          <value>"Monica Bellucci"</value>
        </param>
        <param name="start" type="int"/>
        <param name="maxResults" type="int"/>
        <param name="filter" type="boolean"/>
        <param name="restrict" type="string"/>
        <param name="safeSearch" type="boolean"/>
        <param name="lr" type="string"/>
        <param name="ie" type="string"/>
        <param name="oe" type="string"/>
    </operation>
</service>
```

Listing 5.1: GoggleSearch Invocation Template

The results of the evaluation are quite interesting, because the accuracy is not as high as it should be expected from Google. As one can see in Figure 5.7, the response time of the first 100 test runs varies quite heavily, probably depending on the load of the service.

The most interesting values of all 222 test runs are summarized in Table 5.4. Comparing these results to searches issued by the Web-based Google search engine shows why the Web service is still considered to be in beta state and no new keys are issued for trial purposes. Both, response time and execution time are considerably slower than searches done via the Web interface. Because of the limited usage of the Web service interface it can be assumed

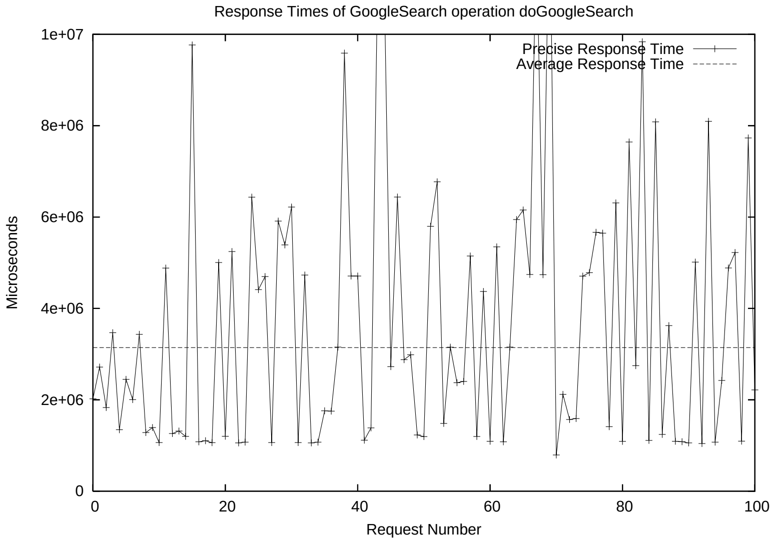


Figure 5.7: GoogleSearch Response Times

	Results
Execution Time	1497ms
Response Time	3143ms
Round Trip Time	4236ms
Average Latency	125ms
Accuracy	79 %

Table 5.4: Google Evaluation Results

that Google simply doesn't provide load-balancing and other performance-critical methods on this side. The rather low average latencies are most probably a result of the Web-server setup which is designed to handle incoming requests in a minimum of time, no matter where the request is originally aimed at. Another reason may be the time Google needs to verify the provided key and decide if the incoming request is valid.

For general purpose, the results show that the desired times and performance ratings can be assessed with a good precision. Furthermore, possible peaks in service responses or packet loss is tolerated by the statistical approach and also reflects in the service's dependability ratings.

Chapter 6

Related Work

> I don't want to achieve immortality through my work...
> I want to achieve it through not dying.
>
> Woody Allen

This chapter gives an overview on the most important publications directly related to the work presented in the previous chapters. The core concept of vector spaces and how to search them will be the main focus here. Other topics blend into the concept on different points and hence form the research background this thesis is based upon.

6.1 Vector Space basics

6.1.1 General

In the field of information retrieval, the vector space model constitutes a common method to search for a specific document or to retrieve the most relevant documents for a search query. Especially when whole articles are processed (e.g., in news repositories or for Websites), the vector space model is known to produce good results for both, document similarity rating, and query processing. In most cases, a vector space engine is used as a centralized service, where all needed data is stored in a single repository. With the original concept which was proposed by Salton [54], searching in such a repository was leveraged to a point, where performance became a secondary issue. Some additional problems arise, when there is the need to create a search engine that uses a distributed document repository with different term spaces as it was presented in Chapter 2.

This section intends to recap the related work for the Vector Space Model with special attention to the distribution capabilities of it’s different sub-categories. The principles and concepts are already well-established and are therefore mentioned as related work instead. In other words, the goal is to explain here how the developed solution evolved from the basic concept to the distributed version that forms the basis of this thesis.

6.1.2 Principle

The following example should provide a better understanding on how this term space is built in the first place. Two fictive and one real WSDL files are used as samples. The real file is the same as in Section 2.2.1 but only one of it’s elements is used for the time being.

- The start will always be a term space of zero dimensions. No vectors are available yet.
- Now three documents are added subsequently. For an easier understanding, the documents are assumed to contain less than the usual amount of keywords. The first document keeps the element ”Product” and ”Info”. This creates a two-dimensional term space:

C=

Dimension	d_1
Product	1
Info	1

- After adding a document with the elements (or methods) ”Info” alone and one with the words ”get”,”Product” and ”Info”, the term space is expanded by one term or dimension to:

C=

Dimension	d_1	d_2	d_3
Product	1	0	1
Info	1	1	1
get	0	0	1

The presented form with binary values is the simplest way of a document representation based on vectors. The result is a three dimensional, binary vector space with a population of $N = 3$. Because of its low dimensionality, it is even possible to visualize the vectors as arrows inside a cube (see Figure 6.1). This form of a vector space is suited best to visualize where the problem with distributed repositories lies. Once, all desired documents (e.g. WSDL files) are represented within the common term space, the relevance between them can be rated according to various rating procedures. A possible way to rate documents is the cosine value, where the angle between two vectors is taken as a measure for the documents’ similarity. Search queries are processed the same way. Once a query is received,

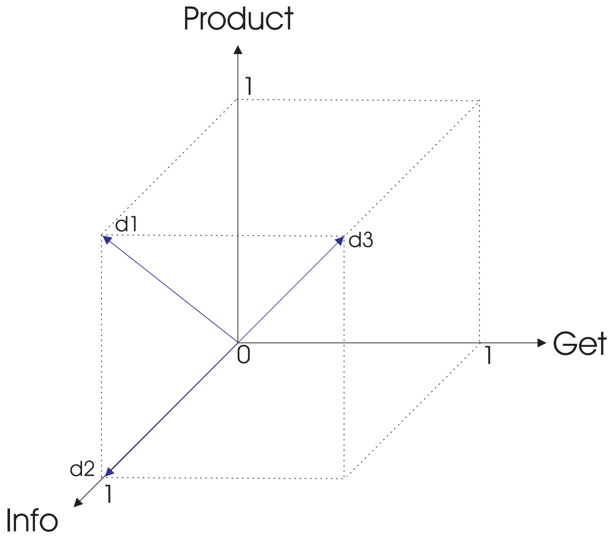


Figure 6.1: Three dimensional vector space

it is projected into the term space and treated like any other document. In the above example, a query for the word ”get” would result in a query vector $d_q = (0, 0, 1)$ which is compared to already existing documents. In the above case, document d_3 will be the most relevant result, since it is the only one with an occurrence of the dimension ”get”. Building term spaces based on binary values however, is not useful in most cases, because no weighting scheme can be applied. Therefore, raw term frequencies are assigned instead of binary values for the elements of a vector.

6.1.3 Raw term frequency:

Vectors weighted by raw term frequency have some important properties:

1. Documents with many occurrences of a specific term are considered of higher relevance to a query.
2. Long documents stand a better chance of being retrieved than short documents, because of the bigger number of overall words.
3. No matter how frequent a term is in the overall collection, it is always of the same importance within a vector.

These properties comprise some advantages when it comes to distribution. Since a single element of a vector is independent from auxiliary values, no additional considerations to allow distribution have to be taken. Instead, the weighted values are treated exactly like the boolean values in the section before. Unfortunately, this form of weighting is unsuitable for document collections with a broad spectrum of document lengths. And this must be assumed for the underlying data. Therefore, some advanced weighting had to be done, to ensure an acceptable recall rating for query operations. This applies to both, the natural language and the code-part of Web service descriptions as already explained in Section 2.5.1. For this reason, weighting was introduces.

6.1.4 Weighting and rating schemes

Where Salton uses the dual logarithm for the calculation of the *tf* x *idf*, some approaches [25] are known to use the common logarithm:

$$idf = log(\frac{N}{n_k + 1}).$$

Both variations target to achieve the same goal. Which one to actually use can be decided in the implementation.

For the rating algorithms presented in 2.5.3, certain alternatives exist. Some of these additional rating algorithms are [33] [71]:

- The Dice Coefficient which is designed to relate documents based on the number of matching bigrams. using it for this purpose is theoretically possible but limited in its significance. A vector representation of a document does not follow a predictable order in the occurrence of its keywords and, therefore, cannot be used to produce meaningful bigrams.
- The Jaccard Coefficient can be seen as the predecessor. This coefficient is defined as the size of the intersection divided by the size of the union of two sets.

  $$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

  For distance measurements, the Jaccard distance is calculated by subtracting the coefficient from 1:

  $$J_\delta(A,B) = 1 - J(A,B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|}$$

  This measurement can be used for distance measurements in binary vector spaces. For this purpose it is very well suited, because the require operations are very efficient in processing. For non-binary vectors however, the Jaccard coefficient has to be extended which essentially leads to the cosine value as it is used in this thesis.
- Overlap Coefficient. Other than the cosine value, the overlap coefficient measures the degree of similarity two documents have compared to the minimum cardinality of keywords.

  $$O(A,B) = \frac{|A \cap B|}{min(|A|, |B|)}$$

This coefficient is also suitable for binary elements.

- The Levenshtein Distance is another distance measure worth mentioning. Its purpose is to rate orthogonal relations of terms. It is used to calculate word distances. The words 'right' and 'might' for instance, would be recognized as related with just one substitution. It is thinkable to calculate the Levenshtein distance for terms within the vector space and this way relate terms to each other. For vector spaces with high dimensionality but few common terms, it could be used to produce better ratings. How good such a method would actually work is worth a discussion and is part of future investigation possibilities.

For WSDL files, the cosine value proved to be the optimal solution. When extending the search engine to other data structures however, is has to be considered if one of the mentioned algorithms may be more suitable.

6.1.5 Linguistic methods

Several methods common for information retrieval do not directly apply to the proposed concept. Stop word lists are a good example. For natural language processing, stop word lists are collections of very common words that occur in most documents without adding real value. Therefore, they can be omitted. The result is a smaller vector space with essentially the same level of information. Unfortunately, the case is different here. Most of the code entries will be method names or type labels. Those elements will not contain parts that are not useful. If a method is named getHighestUserRatio(), for example, the label consists of four important words. Applying a stop word list here would be very counterproductive. Instead, it is tried to split method names and type definitions into single keywords. Splitting method names where capital letters occur is just one of many possibilities. In cases where the splitting fails, it does not automatically mean that the result is corrupted. It just means that the chance of a match for this specific method are lower than average.

6.1.6 Vector space assembly and synchronization

The approach presented in Section 2.5.1 describes an autonomous growth for each distributed vector space. The resulting discrepancies in the weighting scheme are handled by the runtime weighting and evaluation facility. Apart from this approach, there is also the possibility to synchronize the vector spaces with each other. This method is preferable for prototypes where the repositories are of high availability [65]. In this trivial approach, once a change of any kind (adding new documents, or updating old values) occurs, every term space must update all affected values before performing any other operation on the same space. Therefore, this method is not feasible for environments, where connections between participating nodes are not guaranteed. Furthermore, when a new node joins the network, it would have to adept the already built up term space of another node first, including all vectors and keywords. The result would be a massive traffic overhead and reduced independency of the overall system.

6.2 Measuring service metadata

In comparison to the field of information retrieval, which builds the foundations of the search engine, performance related research, especially where Web services are concerned are still highly investigated on a fundamental level. Partially this is, because quality of service research encompasses different research domains such as the network, software engineering and more recently service-oriented computing community. A lot of research is dedicated to the area of QoS modeling and enriching Web services with QoS semantics. Most of the existing work leaves open the way how QoS parameters and other metadata are bootstrapped or evaluated in the first place. Even the categories and classification methods are not well established but still discussed in many research papers.

In [68] for example, Wickramage and Weerawarana define 15 distinguishable periods in time, a SOAP request goes through before completing a round trip. This value, which is also referred to as response time, can be split up into different components where it is essential to bootstrap as much of them as possible. The presented approach does not use all 15 periods identified in this work, because not all periods are interesting to consumers of a service or cannot be determined from the client.

Suzumura et al. [57] did some work that focuses on performance optimizations of Web services. Their approach is to minimize XML processing time (which is called wrapping time here) by using differential de-serialization. The idea is to de-serialize only these parts which have not been processed in the past. Here, the goal is not try to optimize the performance, but to measure the different attributes, without dealing with the wrapping time itself.

A QoS model and a UDDI extension for associated QoS to a specific Web service is proposed in [52]. The QoS model proposed in this thesis is very similar to the model at hand, however, the author does not specify how these values are actually assessed and monitored. It is assumed the QoS attributes of a service are specified by the service provider in UDDI. In [62], another approach for integrating QoS with Web services is presented. The authors implemented a tool-suite for associating, querying and monitoring QoS of a Web service. In contrast to this work, it is not specified how QoS attributes are actually measured and evaluated to associate them to certain Web services.

In [56], Song and Lee propose a simulation based Web service performance analysis tool called sPAC which allows to analyze the performance of Web process (i.e., a composition) by using simulation code. Their approach is to call the Web service once under low load conditions and then transform these testing results into a simulation model. Chapter 3 of this thesis also focuses on the performance aspects of Web services whereas it does not deal with Web processes. Furthermore, simulation code is not used. The evaluation is performed on real Web services even with the constraint that access to the Web service implementation is not available.

QoS attributes in Web service composition also raise a lot of interest due to the fact that they can be used by the compositor to dynamically choose a suitable service for the composition regarding the performance, price or other attributes. A simple, but illustrating example of such a composition is presented in [41]. In [73] and [74], the authors propose a QoS model and a middleware approach for dynamic QoS-driven service composition. They investigate a global planning approach to determine optimal service execution plans for composite service based on QoS criteria. In contrast to the presented approach, the authors do not specify how QoS attributes for atomic services are measured and assessed. It is assumed that the atomic services already have reasonable QoS attributes.

6.3 Search and Clustering

Search and search-related aspects of Web services are highly investigated fields throughout the service-oriented community. In most cases, Web service discovery is the driving force behind the research. The reason is simply that this particular area still rises some very interesting issues that need to be addressed. Service discovery in ad-hoc networks that are based on Web services [20] for instance, has to deal with some very special problems regarding information propagation and centralization. It is basically the same problem that arises with all peer-to-peer (P2P) networks. The highly fluctuating nature of such designs either depend on a single point of failure or have to encompass a feature to propagate service information suitably. Ordinary methods for service discovery, like UDDI registries most certainly fail in such environments because they are not adaptable enough. Furthermore, they lack the very important feature of joining service repositories as it is necessary for federations of Web services. This particular issue was addressed by [55], for example. Here, the authors specifically deal with this problem and developed a discovery mechanism that allows a user to find services in a federated service environment. The system is called MWSDI and relies on a decentralized structure without a central component. The general layout of the presented discovery approach allows a setup that is independent of the underlying structure. It can work in centralized environments as part of an ordinary services registry. At the same time it is possible to deploy the search engine in a federated environment or as part of an ad-hoc network.

Other work focuses not on the principle structure of the service infrastructure but on the quality of the search result itself. Almost every registry ever built encompasses some sort of search facility to pick up contained services. In most cases the search functionality is implemented by a full text search on the underlying database. In other words, searching those repositories is not always concerned as particularly important, let alone a convenient and more powerful way to relate search results and repository content. Only recently, those topics seem to have gained attention. In [11], a search engine named BASIL is introduced that tries to relate Web services by using bias-based techniques. Here, the repositories are supposed to encompass only data-intensive services like searching for DNA sequences. The similarity measure is based on the exchanged documents and is therefore a personalized type of search approach rather than a general one as proposed here. Some of the used techniques are similar to the work presented in [49] with the difference that the weighting is not performed at runtime but in advance. Furthermore, the approaches work on different repository structures. Another example for a Web service search engine is woogle [16] where services can also be queried and tested for their availability.

Some other aspects of service discovery are discussed in [52]. Here, the authors propose an extension of the current UDDI infrastructure to add QoS descriptions for a given service. Due to the relative openness of the UDDI technology, this approach can actually work with existing technologies. On the other hand, this openness is sometimes seen as one of the major reasons why UDDI has failed to dominate the public Web service domain. In [7] a rather formal approach is presented, where semantic annotations are utilized for automated service discovery. Apart from the discovery issues, the description logic could also be used to formally describe inter-service relationships. Those relationships can reach from input/output matching on syntactical level up to QoS descriptions of whole compositions. In [70] a method to select services based on their quality is proposed. The selection algorithm assumes services where QoS parameters are already described and focuses on optimizing an end-to-end composition with respect to the overall QoS.

For the statistical background, there is of course plenty of reference material available. The theoretical foundation for creating the service relationship is best described with statistical cluster analysis [18]. This methods are used in various research fields to describe similarities of metric values of all kinds. The main problem faced here is the high complexity of ordinary methods. The usually exponential growth rate limits its capabilities for high dimensional data structures enormously. To cope with this problem, [31] introduced a modified cluster algorithm that performs dimensional reduction and clustering at the same time. The method is designed to increase performance in large vector spaces with 1000 and more dimensions. Even though the application area is different in this work, some of the ideas are built on a common ground. Compared with this work, a whole data cloud with a strong cluster layout has to be processed to categorize the single elements while here is has to be dealt with the issue to find possible clusters for a particular query. Furthermore, dimensional reduction has to be applied before the original cluster algorithm to speed up the matrix generation rather than increase the performance of the algorithm itself.

Chapter 7

Conclusion and Future Work

> A conclusion is the place where you got tired of thinking.
>
> Harold Fricklestein

7.1 Conceptual implications

7.1.1 Clustering and search performance

The search and cluster capabilities presented in this thesis are created with the purpose to allow Web service consumers to easily find and relate specific services to a given query. The theoretical basis is formed by statistical methods of cluster analysis in n-dimensional vector spaces with numerical characteristics. The implementation shows that the approach is feasible and even exceeds the expectations in certain respects. The implementation of the cluster engine shows that the possibilities for the processing power does not end at $\mathcal{O}(n)$ expense. It is quite the opposite. By using the developed method, the query processing speed is the only limitation for the whole method. By further enhancing the query process for complete vectors, it is theoretically possible to enhance the cluster speed even beyond $\mathcal{O}(n)$ in the future. In the best case, the growth rate converges to the same amount as it is currently possible for the original search. This assumption is based on the observation that common queries perform better due to the dimensional reduction that can be applied when retrieving related documents. A similar method to reduce the query time for the matrix generation has to be implemented to reduce the involved processing time. Furthermore, an optimization of the generated database queries for the specific form of request issued by the cluster algorithm can further enhance performance. Without altering the query algorithm though, the gain will be proportional and is, therefore, not considered with the

utmost priority for the future.
Performance tweaks usually require some elaborate implementation work, without a conceptual gain. Furthermore, research prototypes are always bound to a limited amount of man-hours and no direct need exists to enhance performance over a certain level. Those reasons often drive researchers to advance their field in conceptual form other than the engineering perspective and explains the software quality of many research prototypes, partially including the one at hand.

7.1.2 Search engine adoptions

One of the conceptual issues worth discussing is the entry point for the cluster algorithm. In the prototype a query is first issued with the vector of one result element to fill the cluster matrix. This starting point is selected by the Web interface.
Alternatively, it is also possible to fill the matrix with the results of the original query itself. As an effect, the cluster matrix would not represent the n most related elements compared to one WSDL file but the nearest matches for the query and their relations to each other. It can be seen as a thinner populated space where the original result vectors are "highlighted". The remaining vectors are still necessary to build the right vector-relations, otherwise a reduced vector space with n elements for a n-sized matrix could be used. This would certainly speed up the processing time, but it is not possible without loosing important term information. Although formally not complete, this cluster might provide a better understanding of the document relations and, therefore, be of greater value to the user than a cluster analysis based on a complete vector. This issue is at least worth investigating and will be part of future tasks.

7.1.3 Semantic indexes

Finally, the presented indexing method opens the opportunity to switch from keyword-based indices to quality-based ones. The current method ensures to provide an unbiased representation of the contained services. The target in this thesis is to provide natural language enabled query processing which will be the preferred method for most cases. In some cases though, it could be necessary to search for certain quality elements of a service. The approach presented here works for both query methods. To extend the capabilities to quality information, the various Quality of Service (QoS) parameters have to be indexed by the search engine. By using QuATSCH, those values are available. They could be used for indexing services with quality-based vectors. Furthermore, other metadata like the location information can be used to further enrich the service vector and, therefore, leverage the discovery capability from a purely syntactic to a quality enabled level. What remains to be seen is whether it is of any use to do so. Creating a vector with QoS information is quite easy. It is possible to introduce a dimension for each criteria presented in Section 3.2.2, resulting in a static vector space with 11 dimensions. Every assessed Web service is indexed with their corresponding value or percentage rating, normalized if needed. When issuing a query though, the result would be a Web service which has the most similar QoS ratings, and not the best. When searching for response times of 650ms and execution times of 100ms for example, a service with response time 620ms and execution time 105ms would list as a very close hit. This example makes it obvious that the vector space model is not the appropriate search model for every data structure but strongly depends on the indexing procedure.

For service ranking based on QoS, a simple database query would certainly provide a much better solution. Another possibility is to use the generated metadata to enrich the results produced by the search engine. Developing an algorithm that adjusts the rating according to service quality, location or user domain would certainly be an interesting aspect.

7.2 Outlook

In this thesis a novel distributed Web service search engine based on the vector space model for information retrieval was presented. In addition to the search engine, automatic methods for metadata generation for Web services were presented along with the underlying technologies necessary to achieve this goal. The evaluation section discussed the implemented prototypes and some of the implications the technology has on used methods.

To formally evaluate and optimize the search engine's performance parameters, a test collection with real-world examples was created and tested by both prototypes. Because of the limited number of available services, the extend of entries processed by the engine was limited. With future Web service ecosystems this number can hopefully be increased to further evaluate scalability and performance issues.

Apart from the search engine, an approach was presented, that allows to automatically generate the most relevant non-functional metadata for a given Web service implementation. These parameters are required in various research fields, including automatic composition and substitution approaches. Nevertheless, the encountered problems and limitations suggest that it is and will always be very hard to automatically generate working applications out of Web services without human judgement and interaction. Creating metadata or even service ontologies may help to a limited degree to automate this processes but a final evaluation will most likely always be reserved for a human expert.

Part of the introduced metadata generation processes presented in this thesis are needed for dynamic service composition, which involves taking Quality of Service (QoS) attributes of Web services into account. Performance and availability QoS attributes enables runtime composition of composite Web services with the only limitation that service alternates are known in advance. The most critical performance and availability metrics and measurements for Web services were presented here. The evaluated measurements of both, generated and real-world services clearly show that the approach is accurate and useful for building large-scale Web service ecosystems.

Another part of future tasks will be to seamlessly integrate all components in the Web based application and even expose the most important methods as Web services. The challenge is to find the right balance between a feasible implementation that features all required functions and a possible overloaded application with possible security issues.

To summarize, the work presented in this thesis is best described as one important and currently missing element in the course of designing a usable system for service discovery and Web service relationship management.

Bibliography

[1] Marco Aiello, Christian Platzer, Florian Rosenberg, Huy Tran, Martin Vasko, and Schahram Dustdar. Web Service Indexing for Efficient Retrieval and Composition. In *Proceedings of the IEEE Joint Conference on E-Commerce Technology (CEC'06) and Enterprise Computing, E-Commerce and E-Services (EEE'06), San Francisco, USA*, June 2006.

[2] Ruj Akavipat, Le-Shin Wu, and Filippo Menczer. Small world peer networks in distributed web search. *WWW 2004*, May 17-22 2004.

[3] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services – Concepts, Architectures and Applications*. Springer Verlag, 2004.

[4] Vo Ngoc Anh, Owen de Kretser, and Alistair Moffat. Vector-space ranking with effective early termination. *SIGIR 01*, 2001.

[5] Apache Software Foundation – Apache Axis. http://ws.apache.org/axis (Last accessed: Dec. 05, 2007), 2005.

[6] Robert Baumgartner, Sergio Flesca, and Georg Gottlob. Visual web information extraction with lixto. In *The VLDB Journal*, pages 119–128, 2001.

[7] Boualem Benatallah, Mohand-Said Hacid, Alain Leger, Christophe Rey, and Farouk Toumani. On automating web services discovery. *The International Journal on Very Large Data Bases*, 14(1):84–96, 3 2005.

[8] David Booth, Hugo Haas, Francis McCabe, Eric Newcomer, Michael Champion, Chris Ferris, and David Orchard. Web Services Architecture – W3C Working Group Note. http://www.w3.org/TR/ws-arch (Last accessed: Dec. 05, 2007), 2004.

[9] Christoph Bussler, Dieter Fensel, and Alexander Maedche. A conceptual architecture for semantic web enabled web services. *SIGMOD Record, 2002*, 2002.

[10] Fabio Casati and Ming-Chien Shan. Dynamic and adaptive composition of e-services. *Information Systems*, 26(3):143–163, 2001.

[11] James Caverlee, Ling Liu, and Daniel Rocco. Discovering and ranking web services with BASIL: a personalized approach with biased focus. In *Proceedings of the 2nd International Conference on Service Oriented Computing (ICSOC'04)*, pages 153–162. ACM Press, 2004.

[12] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) 1.1.* W3C, 2001. URL: http://www.w3.org/TR/wsdl (Last accessed: Dec. 05, 2007).

[13] Christian Platzer. V.U.S.E. - The Vector Space Web Service Search Engine. http://vuse.de.vu/ (Last accessed: Dec. 05, 2007), 2007.

[14] Owen Conlan, David Lewis, Steffen Higel, Declan O'Sullivan, and Vincent Wade. Applying adaptive hypermedia techniques to semantic web service composition. *International Workshop on Adaptive Hypermedia and Adaptive Web-based Systems (AH 2003)*, 2003.

[15] Francisco Curbera, Matthew Duftler, Rania Khalaf, William Nagy, Nirmal Mukhi, and Sanjiva Weerawarana. Unraveling the web services web: an introduction to soap, wsdl, and uddi. *IEEE Internet Computing*, 2002.

[16] Xing Dong, Alon Halevy, Jayant Madhavan, Ema Nemes, and Jun Zhang. Similarity Search for Web Services. In *Proceedings of the 30th VLDB Conference, Toronto, Canada*, 2004.

[17] Schahram Dustdar and Wolfgang Schreiner. A Survey on Web services Composition. *International Journal of Web and Grid Services*, 1, 2005.

[18] Hans-Friedrich Eckey, Reinhold Kosfeld, and Martina Rengers. *Multivariate Statistics*. Gabler, 9 2002.

[19] Eclipse Foundation, Inc. Eclipse AspectJ. http://www.eclipse.org/aspectj/ (Last accessed: Dec. 05, 2007), 2005.

[20] Roy Friedman. Caching web services in mobile ad-hoc networks: opportunities and challenges. In *Proceedings of the second ACM International Workshop on Principles of Mobile Computing (POMC'02)*, pages 90–96. ACM Press, 2002.

[21] Keita Fujii. Jpcap – Java package for packet capture. http://netresearch.ics.uci.edu/kfujii/jpcap/doc/index.html (Last accessed: Dec. 05, 2007), 2005.

[22] Keita Fujii and Tatsuya Suda. Dynamic service composition using semantic information. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 39–48, New York, NY, USA, 2004. ACM Press.

[23] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[24] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, and Henrik Frystyk Nielsen. SOAP Version 1.2. http://www.w3.org/TR/soap12-part1/ (Last accessed: Dec. 05, 2007), 2003.

[25] Monika Henzinger, Brian Milch, BayWei Chang, and Sergey Bin. Query-free news search. *ACM/WWW*, 2003.

[26] David Holmes and M. Catherine McCabe. Improving precision and recall for soundex retrieval. In *Proc. of ITTC (IEEE)*, 2002.

[27] IBM. Ibm business registry. https://uddi.ibm.com/ubr/registry.html (Last accessed: Dec. 05, 2007), 2005.

[28] IBM, BEA Systems, Microsoft, SAP AG, Computer Associates, Sun Microsystems, webMethods. *Web Service Meta Data Exchange Specification*, August 2006. http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange.pdf (Last accessed: Jan. 05, 2008).

[29] Lukasz Juszczyk, Anton Michlmayr, Christian Platzer, Florian Rosenberg, Alexander Urbanec, and Schahram Dustdar. Large Scale Web Service Discovery and Composition using High Performance In-Memory Indexing. In *Proceedings of the IEEE Joint Conference on E-Commerce Technology (CEC'07) and Enterprise Computing, E-Commerce and E-Services (EEE'07), Tokio*, June 2007.

[30] Nick Koudas, Beng chon Ooi, Heng Tao Shen, and Anthony K.H. Tung. Lcd: Enabling search by partial distance in a hyper-dimensional space. *ICDE*, 2004.

[31] Fernando De la Torre and Takeo Kanade. Discriminative cluster analysis. In *Proceedings of the 23rd International Conference on Machine learning (ICML'06)*, pages 241–248. ACM Press, 2006.

[32] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications, 2003.

[33] Michael D. Lee, Brandon Pincombe, and Matthew Welsh. A comparison of machine measures of text document similarity with human judgments. *Submitted Manuscript*, 2004.

[34] Yutu Liu, Anne H. Ngu, and Liang Z. Zeng. QoS computation and policing in dynamic web service selection. In *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters (WWW'04)*, pages 66–73, New York, NY, USA, 2004. ACM Press.

[35] James B. MacQueen. Some Methods for classification and Analysis of Multivariate Observations. In *Proceedings of 5-th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297, Berkeley, 1967. University of California Press.

[36] Ivan Magdalenic, Boris Vrdoljakand, and Zoran Skocir. Towards dynamic web service generation on demand. *International Conference on Software in Telecommunications and Computer Networks, (SoftCOM 2006)*, 2006.

[37] Anbazhagan Mani and Arun Nagarajan. Understanding quality of service for web services. http://www-128.ibm.com/developerworks/library/ws-quality.html (Last accessed: Dec. 05, 2007).

[38] E. Michael Maximilien and Munindar P. Singh. Toward autonomic web services trust and selection. In *Proceedings of the 2nd international conference on Service oriented computing (ICSOC'04)*, pages 212–221, New York, NY, USA, 2004. ACM Press.

[39] Sheila McIlraith, Tran Cao Son, and Honglei Zeng. Semantic web services. *IEEE Intelligent Systems (Special Issue on the Semantic Web)*, 2001.

[40] Daniel A. Menasce. QoS issues in Web services. *IEEE Internet Computing*, 6(6):72–75, November/December 2002.

[41] Daniel A. Menasce. Composing Web Services: A QoS View. *IEEE Internet Computing*, 8(6):88–90, November/December 2004.

[42] Anton Michlmayr, Florian Rosenberg, Christian Platzer, Martin Treiber, and Schahram Dustdar. Towards Recovering the Broken SOA Triangle - A Software Engineering Perspective. In *Proceedings of the 2nd International Workshop on Service-oriented Software Engineering (IW-SOSWE'07), Dubrovnik, Croatia*, September 2007.

[43] Microsoft. Microsoft public uddi registry. http://uddi.microsoft.com/inquire (Last accessed: Mar. 14, 2005), 2005.

[44] OASIS. *Universal Description, Discovery and Integration v3.0 (UDDI) Specification*, February 2005. http://www.oasis-open.org/committees/uddi-spec (Last accessed: Dec. 05, 2007).

[45] Dimitris Papadias, Qiongmao Shen, Yufei Tao, and Kyriakos Mouratidis. Group nearest neighbor queries. *Proceedings of ICDE*, 2004.

[46] Mike P. Papazoglou. Service-oriented computing: concepts, characteristics and directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering*, pages 3–12, Dezember 2003.

[47] Mike P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-Oriented Computing Research Roadmap. http://infolab.uvt.nl/pub/papazogloump-2006-96.pdf (Last accessed: Dec. 05, 2007), 2006. Technical Report/Vision Paper on Service oriented computing European Union Information Society Technologies (IST), Directorate D - Software Technologies (ST).

[48] Mike P. Papazoglou and Willem-Jan van den Heuvel. Service Oriented Architectures: Approaches, Technologies and Research Issues. *VLDB Journal*, 2006. forthcoming.

[49] Christian Platzer and Schahram Dustdar. A Vector Space Search Engine for Web Services. In *Proceedings of the 3rd European IEEE Conference on Web Services (ECOWS'05)*, 2005.

[50] Christian Platzer, Florian Rosenberg, and Schahram Dustdar. *Securing Web Services: Practical Usage of Standards and Specifications*, chapter Enhancing Web Service Discovery and Monitoring with Quality of Service Information. Idea Publishing Inc., 2007.

[51] Martin Porter. Porter stemming algorithm, 10 2004. http://www.tartarus.org/~martin/PorterStemmer/ (Last accessed: Dec. 05, 2007).

[52] Shuping Ran. A model for web services discovery with QoS. *SIGecom Exch.*, 4(1):1–10, 2003.

[53] Florian Rosenberg, Christian Platzer, and Schahram Dustdar. Bootstrapping performance and dependability attributes of web services. pages 205–212. IEEE Computer Society, 2006.

[54] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*, volume 1. McGraw-Hill, Inc., 1983.

[55] Kaarthik Sivashanmugam, Kunal Verma, and Amit Sheth. Discovery of Web Services in a Federated Registry Environment. *ICWS*, 0:270, 2004.

[56] Hyung Gi Song and Kangsun Lee. sPAC (Web Services Performance Analysis Center): Performance Analysis and Estimation Tool of Web Services. In *Proceedings of the 3rd International Conference on Business Process Management (BPM'05)*, pages 109–119, 2005.

[57] Toyotaro Suzumura, Toshiro Takase, and Michiaki Tatsubori. Optimizing Web services performance by differential deserialization. In *Proceedings of the IEEE International Conference on Web Services (ICWS'05)*, pages 185–192, 2005.

[58] Tanveer Syeda-Mahmood, Gauri Shah, Rama Akkiraju, Anca-Andrea Ivan, and Richard Goodwin. Searching service repositories by combining semantic and ontological matching. *ICWS*, 0:13–20, 2005.

[59] Stefan Tai, Nirmit Desai, and Pietro Mazzoleni. Service Communities: Applications and Middleware. In *Proceedings of the 6th International Workshop on Software Engineering and Middleware (SEM'06)*, pages 17–22. ACM Press, 2006.

[60] Stefan Tai, Rania Khalaf, and Thomas Mikalsen. Composition of coordinated web services. In *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, pages 294–310, New York, NY, USA, 2004. Springer-Verlag New York, Inc.

[61] Xiaoying Tai, Minoru Sasaki, and Yasuhito Tanaka. Improvement of vector space information retrieval model based on supervised learning. *ACM*, 2000.

[62] Min Tian, A. Gramm, Hartmut Ritter, and Jochen Schiller. Efficient Selection and Monitoring of QoS-aware Web services with the WS-QoS Framework. In *Proceedings of the International Conference on Web Intelligence (WI'04), Beijing, China*, 2004.

[63] W3C. Resource Description Framework (RDF). http://www.w3.org/RDF (Last accessed: Dec. 05, 2007), 2000.

[64] W3C. OWL Web Ontology Language Overview. http://www.w3.org/TR/owl-features/ (Last accessed: Dec. 05, 2007), 2004. W3C Recommendation 10 February 2004.

[65] Zhiwei Wang, Michael Wong, and Yiyu Yao. An analysis of vector space models based on computational geometry. *ACM/SIGIR*, 1992.

[66] Web Services Policy Framework. http://www-128.ibm.com/developerworks/library/specification/ws-polfram/ (Last accessed: Dec. 05, 2007), 2004.

[67] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More.* Prentice Hall PTR, 2005.

[68] Narada Wickramage and Sanjiva Weerawarana. A benchmark for web service frameworks. In *Proceedings of the IEEE International Conference on Service Computing (SCC'05)*, 2005.

[69] SKM Wong, Wojciech Ziarko, and Patrick Wong. Generlized vector space model in information retrieval. *ACM*, 1985.

[70] Tao Yu, Yue Zhang, and Kwei-Jay Lin. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web*, 1(1):6, 2007.

[71] Zhi-Wen Yu, Xing-She Zhou, Jian-Hua Gu, and Xiao-Jun Wu. Adaptive proram filtering under vector space model and relevance feedback. *Proceedings of ICMLC*, 2003.

[72] Budi Yuwono and Dik L. Lee. Search and ranking algorithms for locating resources on the world wide web. *IEEE*, 1996.

[73] Liangzhao Zeng, Boualem Benatallah, Marlon Dumas, Jayant Kalagnanam, and Quan Z. Sheng. Quality driven web services composition. In *Proceedings of the 12th International Conference on World Wide Web (WWW'03)*, pages 411–421, New York, NY, USA, 2003. ACM Press.

[74] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Transactions on Software Engineering*, 30(5):311–327, May 2004.

Appendix A

Code listings

A.1 UDDI cross-reference downloads

```
public class UDDIregistry
{
    private WsDescription description = new WsDescription();
    public ArrayList keys = new ArrayList();
    public ArrayList wsdls = new ArrayList();
    private string URL = ”http://uddi.microsoft.com/inquire”;
    public bool running = false;
    // The constructor just takes a UDDI inquiry URL
    public UDDIregistry(string URL)
    {
        if (URL != ””) this.URL = URL;
        keys.Clear();
        wsdls.Clear();
   }
    // starts data extraction
    public void startExtraction()
    {
        Thread t;
        t =new Thread(new ThreadStart(retrieveTmodelKeysThread));
        t.Start();
        Thread.Sleep(20);
    }
    public void retrieveTmodelKeysThread()
    {
        char query = ’a’;
        this.running = true;
        // Read all UDDI entries
        Inquire.AuthenticationMode =
        AuthenticationMode.WindowsAuthentication;
        Inquire.Url = URL;
        query--;
        while (query != 'z')
        {
            query ++;
            FindTModel ftm = new FindTModel();
            ftm.Name = query.ToString();
            try
            {
                TModelList tml = ftm.Send();
                foreach( TModelInfo tmi in tml.TModelInfos )
                {
                    keys.Add(tmi.TModelKey);
                }
            }
            catch (Exception e)
            {
                // no error handling required
            }
        }
        retrieveURLS();
    }
    private void retrieveURLS()
    {
        for (int i=0; i<keys.Count; i++)
        {
            // Processing each TModel Key
            GetTModelDetail gtmd = new GetTModelDetail();
            gtmd.TModelKeys.Add((string) keys[i]);
            TModelDetail tmd = gtmd.Send();
            foreach( TModel tm in tmd.TModels )
            {
                // URLS are in the overview documents
                wsdls.Add(tm.OverviewDoc.OverviewURL);
                downloadURL(tm.OverviewDoc.OverviewURL);
            }
        }
    }
    private void downloadURL(URL)
    {
        string filename;
        string destDir=”C:\\WSDLS”;
        try
        {
            HttpWebRequest myReq =
                (HttpWebRequest)WebRequest.Create(URL);
            HttpWebResponse resp =
                (HttpWebResponse)myReq.GetResponse();
            Stream receiveStream = resp.GetResponseStream();
            filename = destDir+URL;

            StreamWriter sw = File.CreateText(filename);
            StreamReader readStream =
                new StreamReader( receiveStream, true );
            char[] read = new Char[bufferlength];
            int count = readStream.Read(read, 0, bufferlength);
            while (count > 0)
            {
                for (int k = 0 ; k < count; k++)
                sw.Write(read[k]);
                count = readStream.Read(read, 0, bufferlength);
            }
            resp.Close();
            sw.Flush();
            sw.Close();
            readStream.Close();
        }
        catch (Exception s)
        {
            Console.WriteLine(”Error downloading URL”+s.Message);
        }
    }
}
```

Listing A.1: UDDI data extraction code

A.2 V-USE table generation

```
private void checkTables() throws VSException
{
  boolean termflag = false;
  boolean idflag   = false;
  boolean joinflag = false;

  try
  {
    Statement stm = con.createStatement();
    ResultSet rs = stm.executeQuery(”SHOW TABLES;”);

    while (rs.next()){
      // For Windows Systems use .tolowerCase() on the comparison
      if (rs.getString(1).equals(_termTable)) termflag = true;
      if (rs.getString(1).equals(_relTable))  joinflag = true;
      if (rs.getString(1).equals(_idTable))   idflag   = true;
    }

    // If one of the needed Tables does not exist, create them.
  if (!(termflag&&idflag&&joinflag))
   {
     // Drop the tables
    stm.executeUpdate(”DROP TABLE IF EXISTS ‘”+_termTable+”‘;”);
    stm.executeUpdate(”DROP TABLE IF EXISTS ‘”+_idTable+”‘;”);
    stm.executeUpdate(”DROP TABLE IF EXISTS ‘”+_relTable+”‘;”);

    // Create the new Tables

/*--------------------------------------------------------------------*/
    stm.executeUpdate(”CREATE TABLE ‘”+_relTable+”‘(term_id bigint(20)
    NOT NULL default '0', document_id bigint(20) NOT NULL default '0',
    frequency DOUBLE NOT NULL default '0', PRIMARY KEY
    (term_id,document_id)) ENGINE=MyISAM DEFAULT CHARSET=utf8
    COMMENT='The m_n relation';”);
/*--------------------------------------------------------------------*/


    stm.executeUpdate(”CREATE TABLE ‘”+_idTable +”‘(id bigint(20)
    NOT NULL auto_increment, ‘document‘ VARCHAR(255) NOT NULL,
    PRIMARY KEY  (id)) ENGINE=MyISAM DEFAULT CHARSET=utf8
    COMMENT='The documents for the VSE';”);

/*-------------------------------------------------------------*/
    stm.executeUpdate(”CREATE TABLE ‘”+_termTable+”‘(id bigint(20)
    NOT NULL auto_increment, term VARCHAR(255) NOT NULL
    COMMENT ’The actual term name’,PRIMARY KEY (id))
    ENGINE=MyISAM DEFAULT CHARSET=utf8 COMMENT=’Terms for Engine’;”);
/*-------------------------------------------------------------*/



    // Add indizes for faster query execution
    stm.executeUpdate(”ALTER TABLE ‘”+_termTable+”‘ ADD INDEX ‘”+
        _termTable+”index‘(‘term‘);”);
    stm.executeUpdate(”ALTER TABLE ‘”+_idTable+”‘ ADD INDEX ‘”+
        _idTable+”index‘(‘document‘);”);
   }
  }
  catch (SQLException e)
  {
    throw new VSException(”Unable to create required tables for VM
        − please check TABLE permissions.”+e.getMessage());
  }
}
```

Listing A.2: VUSE table creation

A.3 V-USE query execution

```
private boolean processQuery(String[] QueryWords, double[] frequencies,
                             boolean useWeighting,long maximumTimeoutMS)throws VSException
{
    StringBuffer sb = new StringBuffer();
        for (String st : QueryWords)
        sb.append(st+”;”);

    // Processing each Query word
    Hashtable<String, Double> qhash = new Hashtable<String, Double>();
    if (QueryWords.length != frequencies.length)
        // If the call signature was not correct, throw an exception
        throw new VSException(”Supplied arrays are not of the same length”);
            // Otherwise, put all terms into the Hashtable
    else
        {
        for (int i = 0; i < QueryWords.length; i++)
        {
            qhash.put(QueryWords[i],frequencies[i]);
                        //Putting word QueryWords[i] into the queryword hashtable”);
                }
        }

    // Now try to process the query
    try
    {
        // Take begin time (for early termination)
        long begintime = System.currentTimeMillis();

        // _result is an instance variable that keeps the final result
        // initialized by its maximum size
        _result = new Result(QueryWords.length);

        // First, get all term id's represented by this Vector
        // All database access are processed as prepared statements
        StringBuffer statement = new StringBuffer(”SELECT id, term FROM ”);
        statement.append(_termTable);
        statement.append(” WHERE term IN (”);
        for (int i = 0; i < QueryWords.length; i++)
        statement.append(”'”+QueryWords[i]+”',”);
        statement.append(”'');”);
        PreparedStatement pstm=con.prepareStatement(”SELECT count(*) from ”+_idTable+”;”);
        ResultSet number = pstm.executeQuery(); number.next();
        int N  = number.getInt(1);
        int nk = 0;

        // Now N is known
        pstm = con.prepareStatement(statement.toString());
        ResultSet query = pstm.executeQuery();
        ResultSet                fqResult;
        PreparedStatement frequencystatement = con.prepareStatement
            (”SELECT frequency, document_id from ”+_relTable+” where term_id = ?;”);

        // query is the iterator with the id's of all contained terms of the Query
        while (query.next() && (System.currentTimeMillis()-maximumTimeoutMS < begintime))
        {
            long currTermID = query.getLong(1);

            // Get all frequencies of all documents containing that term
            frequencystatement.setLong(1,currTermID);
            fqResult = frequencystatement.executeQuery();
            // Determine nk
            fqResult.last();
            nk = fqResult.getRow();
            fqResult.beforeFirst();

            // now commence the document iteration
            while (fqResult.next())
            try
            {
                if (useWeighting)
                    _result.addElement(fqResult.getLong(”document_id”),
                    getWeight(fqResult.getDouble(”frequency”),N+1,nk+1),
                    getWeight(qhash.get(query.getString(”term”)),N+1,nk+1));
                else _result.addElement(fqResult.getLong(”document_id”),
                    fqResult.getDouble(”frequency”),qhash.get(query.getString(”term”)));
            }
            catch (Exception e)
            {
                // One element skipped of this query, resume with the others
                Log.ERROR(”Skipped an Element of the Query.”+query.getString(”term”));
                Log.ERROR(”docID was:”+fqResult.getLong(”document_id”));
                Log.ERROR(”frequency was:”+fqResult.getDouble(”frequency”));
                Log.ERROR(”Term was:”+query.getString(”term”));
            }
        }

        // Query processing finished, finalize the result
        _result.setFinished(System.currentTimeMillis()−maximumTimeoutMS < begintime);
        _result.setProcessingTime(System.currentTimeMillis()−begintime);

        if (_result.isFinished())
            Log.INFO(”Finished query in ”+_result.getProcessingTime()+” milliseconds”);
        else Log.ERROR(”Hit query time limit. Returned partial result.”);
        return _result.isFinished();
    }
    catch (Exception e)
    {
        Log.ERROR(”Exception while processing: ”+e.getMessage());
        _result = null;
        return false;
    }
}

private double getWeight (double frequency, double N, double nk)
{
    return (frequency * (Math.log((N/nk)+1)/(Math.log(2))));
}


/*--------------------------------------------------------------------------------
* The following two methods are code from the Result class which _result is an instance of
* zHash, n1Hash and n2Hash are hashtables with the signature Hashtable<Long,Double>.
*/


//gets the relevance of one element in the resultset
public double getRelevance(long documentID)
{
    if (zHash.containsKey(documentID))
    {
        double Z  = zHash.get(documentID);
        double N1 = n1Hash.get(documentID);
        double N2 = n2hash.get(documentID);
        return (((Z/Math.pow((N1*N2),0.5))*qHash.get(documentID))/(_queryLength));
    }
    // If this document is NOT in the resultset, its not relevant
    else return −1D;
```

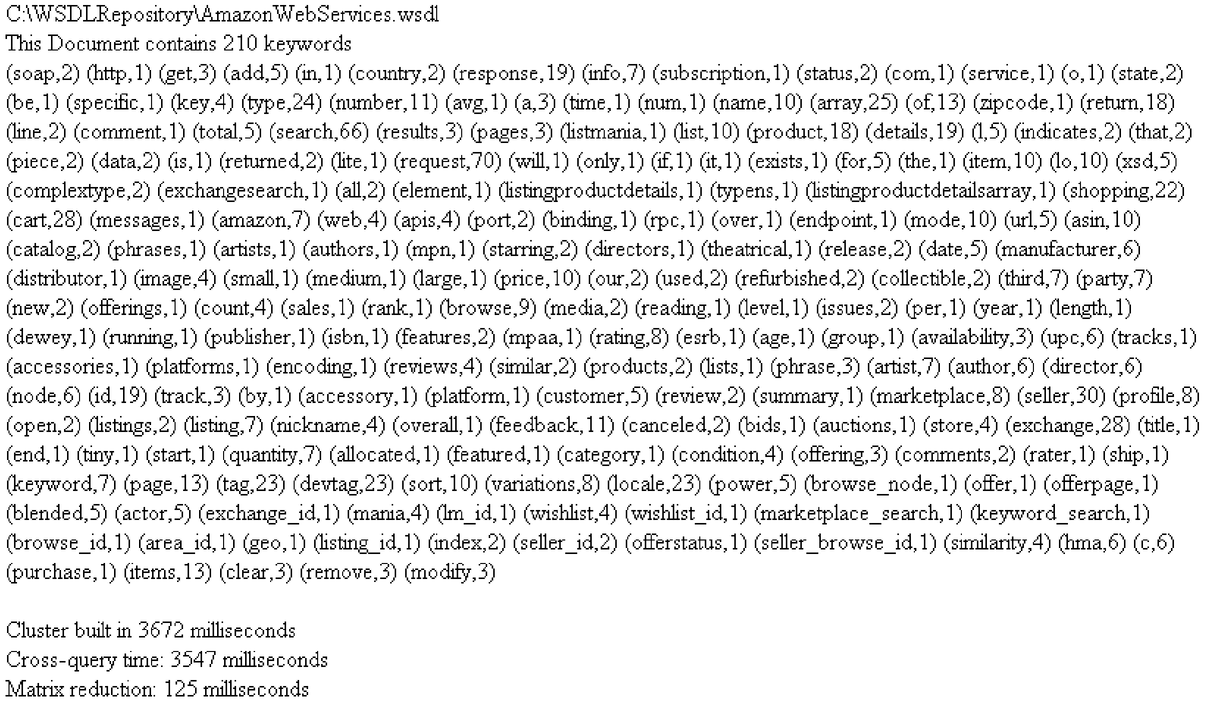
```
}

// Adds one element to the resultset
public void addElement(long documentID, double value, double queryFrequ)
{
        // Adds one document with a specific document ID and the corresponding value
        double Z  = value*queryFrequ;
        double N1 = Math.pow(value,2);
        double N2 = Math.pow(queryFrequ,2);
        int    N = 0;

        if (zHash.containsKey(documentID))
        {
                Z  += zHash.get(documentID);
                N1 += n1Hash.get(documentID);
                N2 += n2hash.get(documentID);
                N   = qHash.get(documentID);
        }

        zHash.put(documentID,Z);
        n1Hash.put(documentID,N1);
        n2hash.put(documentID,N2);
        qHash.put(documentID,++N);
}
```

Listing A.3: VUSE query execution

Appendix B

Screenshots

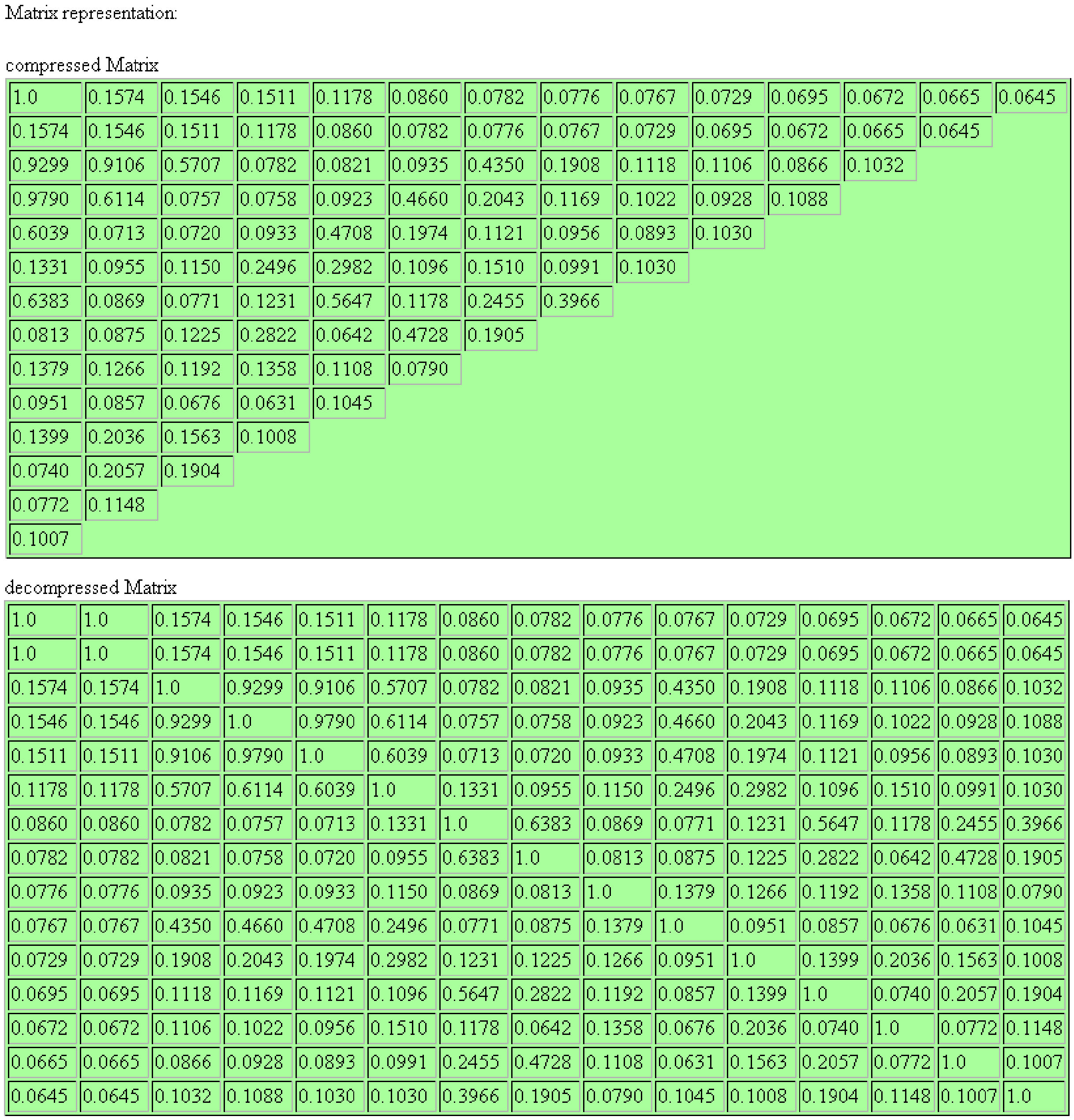
B.1 Amazon Cluster - Initial Vector



```
C:\WSDLRepository\AmazonWebServices.wsdl
This Document contains 210 keywords
(soap,2) (http,1) (get,3) (add,5) (in,1) (country,2) (response,19) (info,7) (subscription,1) (status,2) (com,1) (service,1) (o,1) (state,2)
(be,1) (specific,1) (key,4) (type,24) (number,11) (avg,1) (a,3) (time,1) (num,1) (name,10) (array,25) (of,13) (zipcode,1) (return,18)
(line,2) (comment,1) (total,5) (search,66) (results,3) (pages,3) (listmania,1) (list,10) (product,18) (details,19) (l,5) (indicates,2) (that,2)
(piece,2) (data,2) (is,1) (returned,2) (lite,1) (request,70) (will,1) (only,1) (if,1) (it,1) (exists,1) (for,5) (the,1) (item,10) (lo,10) (xsd,5)
(complextype,2) (exchangesearch,1) (all,2) (element,1) (listingproductdetails,1) (typens,1) (listingproductdetailsarray,1) (shopping,22)
(cart,28) (messages,1) (amazon,7) (web,4) (apis,4) (port,2) (binding,1) (rpc,1) (over,1) (endpoint,1) (mode,10) (url,5) (asin,10)
(catalog,2) (phrases,1) (artists,1) (authors,1) (mpn,1) (starring,2) (directors,1) (theatrical,1) (release,2) (date,5) (manufacturer,6)
(distributor,1) (image,4) (small,1) (medium,1) (large,1) (price,10) (our,2) (used,2) (refurbished,2) (collectible,2) (third,7) (party,7)
(new,2) (offerings,1) (count,4) (sales,1) (rank,1) (browse,9) (media,2) (reading,1) (level,1) (issues,2) (per,1) (year,1) (length,1)
(dewey,1) (running,1) (publisher,1) (isbn,1) (features,2) (mpaa,1) (rating,8) (esrb,1) (age,1) (group,1) (availability,3) (upc,6) (tracks,1)
(accessories,1) (platforms,1) (encoding,1) (reviews,4) (similar,2) (products,2) (lists,1) (phrase,3) (artist,7) (author,6) (director,6)
(node,6) (id,19) (track,3) (by,1) (accessory,1) (platform,1) (customer,5) (review,2) (summary,1) (marketplace,8) (seller,30) (profile,8)
(open,2) (listings,2) (listing,7) (nickname,4) (overall,1) (feedback,11) (canceled,2) (bids,1) (auctions,1) (store,4) (exchange,28) (title,1)
(end,1) (tiny,1) (start,1) (quantity,7) (allocated,1) (featured,1) (category,1) (condition,4) (offering,3) (comments,2) (rater,1) (ship,1)
(keyword,7) (page,13) (tag,23) (devtag,23) (sort,10) (variations,8) (locale,23) (power,5) (browse_node,1) (offer,1) (offerpage,1)
(blended,5) (actor,5) (exchange_id,1) (mania,4) (lm_id,1) (wishlist,4) (wishlist_id,1) (marketplace_search,1) (keyword_search,1)
(browse_id,1) (area_id,1) (geo,1) (listing_id,1) (index,2) (seller_id,2) (offerstatus,1) (seller_browse_id,1) (similarity,4) (hma,6) (c,6)
(purchase,1) (items,13) (clear,3) (remove,3) (modify,3)

Cluster built in 3672 milliseconds
Cross-query time: 3547 milliseconds
Matrix reduction: 125 milliseconds
```

Figure B.1: Query vector for Matrix creation

B.2 Amazon Cluster - Matrix reduction



Matrix representation:

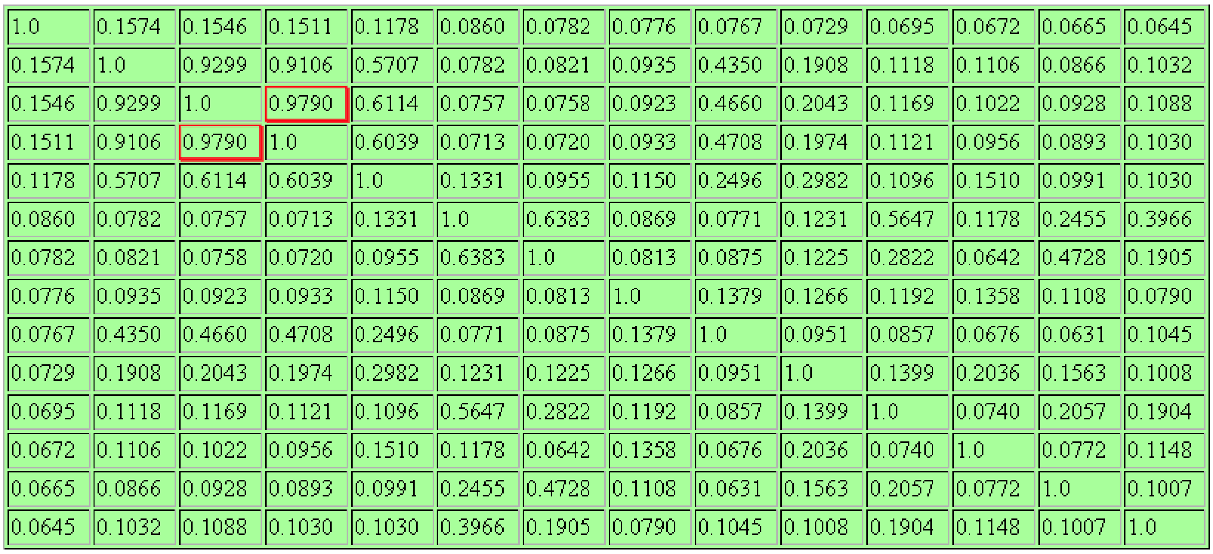
compressed Matrix

1.0	0.1574	0.1546	0.1511	0.1178	0.0860	0.0782	0.0776	0.0767	0.0729	0.0695	0.0672	0.0665	0.0645
0.1574	0.1546	0.1511	0.1178	0.0860	0.0782	0.0776	0.0767	0.0729	0.0695	0.0672	0.0665	0.0645	
0.9299	0.9106	0.5707	0.0782	0.0821	0.0935	0.4350	0.1908	0.1118	0.1106	0.0866	0.1032		
0.9790	0.6114	0.0757	0.0758	0.0923	0.4660	0.2043	0.1169	0.1022	0.0928	0.1088			
0.6039	0.0713	0.0720	0.0933	0.4708	0.1974	0.1121	0.0956	0.0893	0.1030				
0.1331	0.0955	0.1150	0.2496	0.2982	0.1096	0.1510	0.0991	0.1030					
0.6383	0.0869	0.0771	0.1231	0.5647	0.1178	0.2455	0.3966						
0.0813	0.0875	0.1225	0.2822	0.0642	0.4728	0.1905							
0.1379	0.1266	0.1192	0.1358	0.1108	0.0790								
0.0951	0.0857	0.0676	0.0631	0.1045									
0.1399	0.2036	0.1563	0.1008										
0.0740	0.2057	0.1904											
0.0772	0.1148												
0.1007													

decompressed Matrix

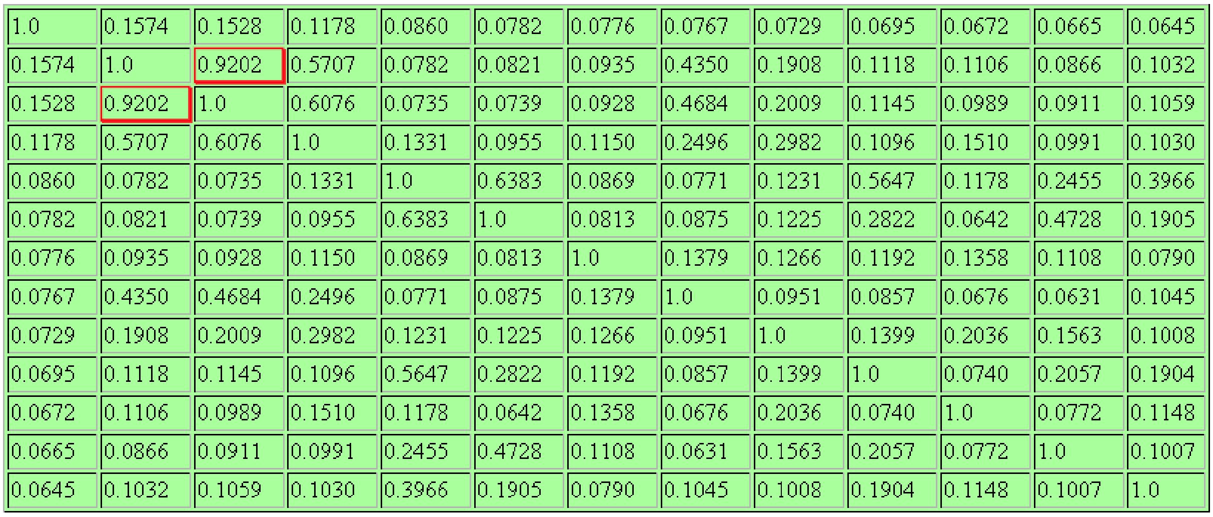
1.0	1.0	0.1574	0.1546	0.1511	0.1178	0.0860	0.0782	0.0776	0.0767	0.0729	0.0695	0.0672	0.0665	0.0645
1.0	1.0	0.1574	0.1546	0.1511	0.1178	0.0860	0.0782	0.0776	0.0767	0.0729	0.0695	0.0672	0.0665	0.0645
0.1574	0.1574	1.0	0.9299	0.9106	0.5707	0.0782	0.0821	0.0935	0.4350	0.1908	0.1118	0.1106	0.0866	0.1032
0.1546	0.1546	0.9299	1.0	0.9790	0.6114	0.0757	0.0758	0.0923	0.4660	0.2043	0.1169	0.1022	0.0928	0.1088
0.1511	0.1511	0.9106	0.9790	1.0	0.6039	0.0713	0.0720	0.0933	0.4708	0.1974	0.1121	0.0956	0.0893	0.1030
0.1178	0.1178	0.5707	0.6114	0.6039	1.0	0.1331	0.0955	0.1150	0.2496	0.2982	0.1096	0.1510	0.0991	0.1030
0.0860	0.0860	0.0782	0.0757	0.0713	0.1331	1.0	0.6383	0.0869	0.0771	0.1231	0.5647	0.1178	0.2455	0.3966
0.0782	0.0782	0.0821	0.0758	0.0720	0.0955	0.6383	1.0	0.0813	0.0875	0.1225	0.2822	0.0642	0.4728	0.1905
0.0776	0.0776	0.0935	0.0923	0.0933	0.1150	0.0869	0.0813	1.0	0.1379	0.1266	0.1192	0.1358	0.1108	0.0790
0.0767	0.0767	0.4350	0.4660	0.4708	0.2496	0.0771	0.0875	0.1379	1.0	0.0951	0.0857	0.0676	0.0631	0.1045
0.0729	0.0729	0.1908	0.2043	0.1974	0.2982	0.1231	0.1225	0.1266	0.0951	1.0	0.1399	0.2036	0.1563	0.1008
0.0695	0.0695	0.1118	0.1169	0.1121	0.1096	0.5647	0.2822	0.1192	0.0857	0.1399	1.0	0.0740	0.2057	0.1904
0.0672	0.0672	0.1106	0.1022	0.0956	0.1510	0.1178	0.0642	0.1358	0.0676	0.2036	0.0740	1.0	0.0772	0.1148
0.0665	0.0665	0.0866	0.0928	0.0893	0.0991	0.2455	0.4728	0.1108	0.0631	0.1563	0.2057	0.0772	1.0	0.1007
0.0645	0.0645	0.1032	0.1088	0.1030	0.1030	0.3966	0.1905	0.0790	0.1045	0.1008	0.1904	0.1148	0.1007	1.0

reduced Matrix; Size 14



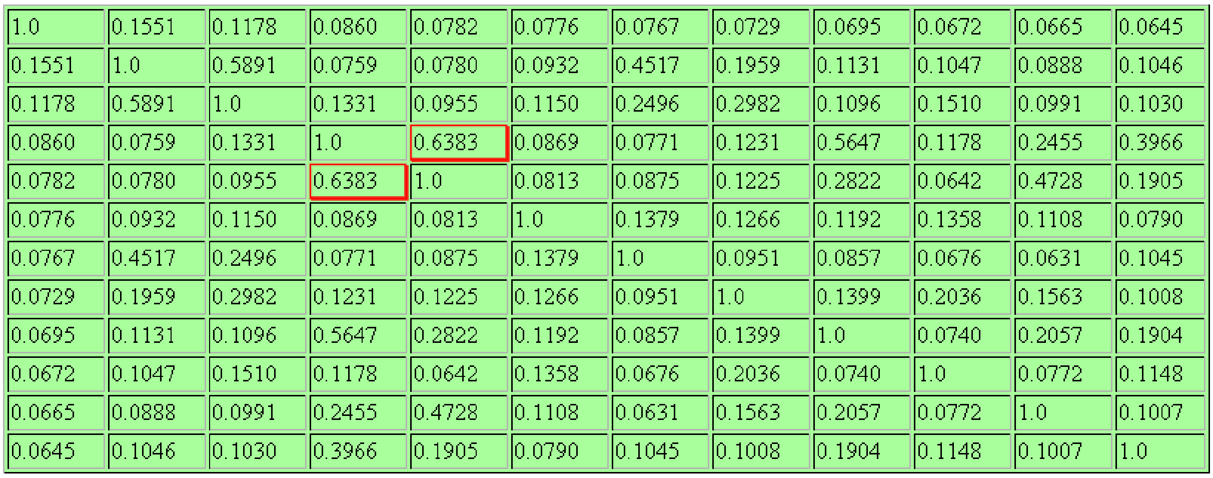
1.0	0.1574	0.1546	0.1511	0.1178	0.0860	0.0782	0.0776	0.0767	0.0729	0.0695	0.0672	0.0665	0.0645
0.1574	1.0	0.9299	0.9106	0.5707	0.0782	0.0821	0.0935	0.4350	0.1908	0.1118	0.1106	0.0866	0.1032
0.1546	0.9299	1.0	0.9790	0.6114	0.0757	0.0758	0.0923	0.4660	0.2043	0.1169	0.1022	0.0928	0.1088
0.1511	0.9106	0.9790	1.0	0.6039	0.0713	0.0720	0.0933	0.4708	0.1974	0.1121	0.0956	0.0893	0.1030
0.1178	0.5707	0.6114	0.6039	1.0	0.1331	0.0955	0.1150	0.2496	0.2982	0.1096	0.1510	0.0991	0.1030
0.0860	0.0782	0.0757	0.0713	0.1331	1.0	0.6383	0.0869	0.0771	0.1231	0.5647	0.1178	0.2455	0.3966
0.0782	0.0821	0.0758	0.0720	0.0955	0.6383	1.0	0.0813	0.0875	0.1225	0.2822	0.0642	0.4728	0.1905
0.0776	0.0935	0.0923	0.0933	0.1150	0.0869	0.0813	1.0	0.1379	0.1266	0.1192	0.1358	0.1108	0.0790
0.0767	0.4350	0.4660	0.4708	0.2496	0.0771	0.0875	0.1379	1.0	0.0951	0.0857	0.0676	0.0631	0.1045
0.0729	0.1908	0.2043	0.1974	0.2982	0.1231	0.1225	0.1266	0.0951	1.0	0.1399	0.2036	0.1563	0.1008
0.0695	0.1118	0.1169	0.1121	0.1096	0.5647	0.2822	0.1192	0.0857	0.1399	1.0	0.0740	0.2057	0.1904
0.0672	0.1106	0.1022	0.0956	0.1510	0.1178	0.0642	0.1358	0.0676	0.2036	0.0740	1.0	0.0772	0.1148
0.0665	0.0866	0.0928	0.0893	0.0991	0.2455	0.4728	0.1108	0.0631	0.1563	0.2057	0.0772	1.0	0.1007
0.0645	0.1032	0.1088	0.1030	0.1030	0.3966	0.1905	0.0790	0.1045	0.1008	0.1904	0.1148	0.1007	1.0

reduced Matrix; Size 13

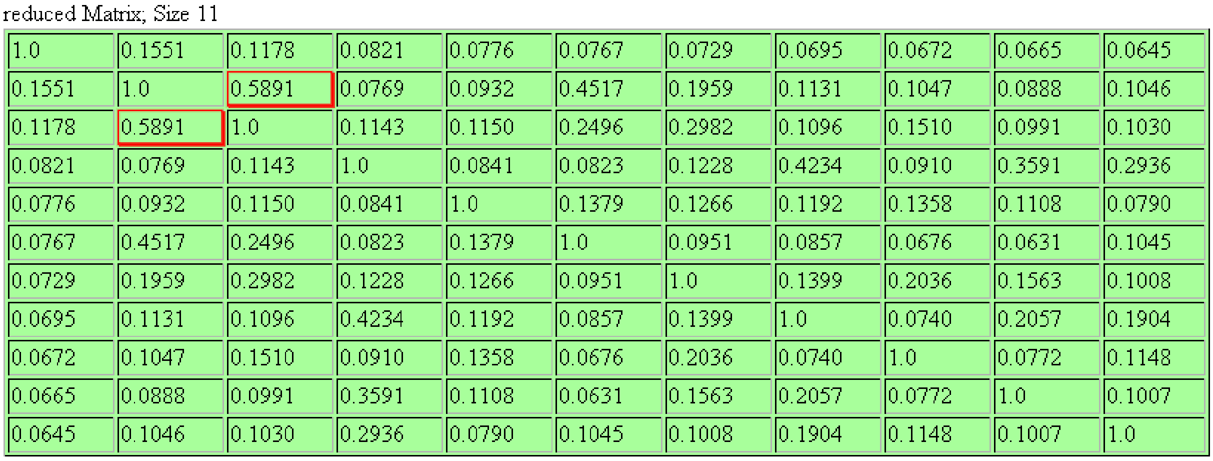


1.0	0.1574	0.1528	0.1178	0.0860	0.0782	0.0776	0.0767	0.0729	0.0695	0.0672	0.0665	0.0645
0.1574	1.0	0.9202	0.5707	0.0782	0.0821	0.0935	0.4350	0.1908	0.1118	0.1106	0.0866	0.1032
0.1528	0.9202	1.0	0.6076	0.0735	0.0739	0.0928	0.4684	0.2009	0.1145	0.0989	0.0911	0.1059
0.1178	0.5707	0.6076	1.0	0.1331	0.0955	0.1150	0.2496	0.2982	0.1096	0.1510	0.0991	0.1030
0.0860	0.0782	0.0735	0.1331	1.0	0.6383	0.0869	0.0771	0.1231	0.5647	0.1178	0.2455	0.3966
0.0782	0.0821	0.0739	0.0955	0.6383	1.0	0.0813	0.0875	0.1225	0.2822	0.0642	0.4728	0.1905
0.0776	0.0935	0.0928	0.1150	0.0869	0.0813	1.0	0.1379	0.1266	0.1192	0.1358	0.1108	0.0790
0.0767	0.4350	0.4684	0.2496	0.0771	0.0875	0.1379	1.0	0.0951	0.0857	0.0676	0.0631	0.1045
0.0729	0.1908	0.2009	0.2982	0.1231	0.1225	0.1266	0.0951	1.0	0.1399	0.2036	0.1563	0.1008
0.0695	0.1118	0.1145	0.1096	0.5647	0.2822	0.1192	0.0857	0.1399	1.0	0.0740	0.2057	0.1904
0.0672	0.1106	0.0989	0.1510	0.1178	0.0642	0.1358	0.0676	0.2036	0.0740	1.0	0.0772	0.1148
0.0665	0.0866	0.0911	0.0991	0.2455	0.4728	0.1108	0.0631	0.1563	0.2057	0.0772	1.0	0.1007
0.0645	0.1032	0.1059	0.1030	0.3966	0.1905	0.0790	0.1045	0.1008	0.1904	0.1148	0.1007	1.0

reduced Matrix; Size 12

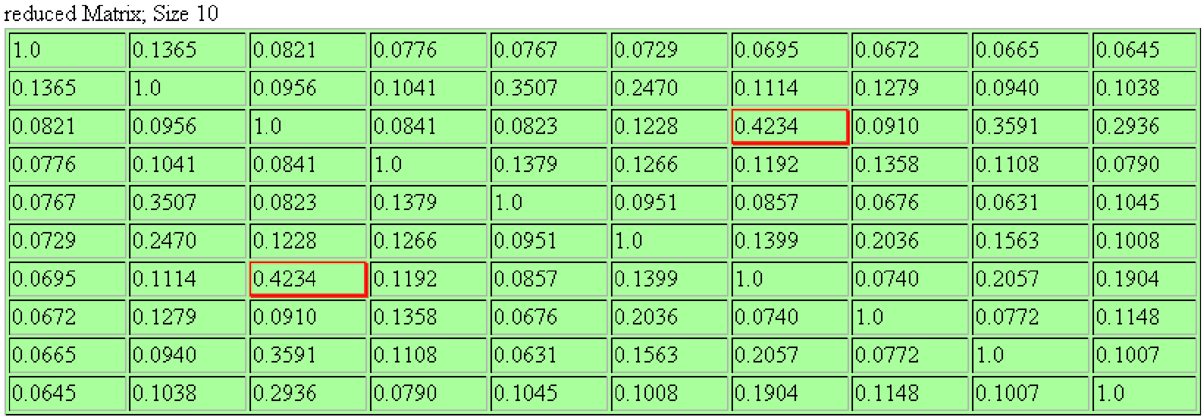


1.0	0.1551	0.1178	0.0860	0.0782	0.0776	0.0767	0.0729	0.0695	0.0672	0.0665	0.0645
0.1551	1.0	0.5891	0.0759	0.0780	0.0932	0.4517	0.1959	0.1131	0.1047	0.0888	0.1046
0.1178	0.5891	1.0	0.1331	0.0955	0.1150	0.2496	0.2982	0.1096	0.1510	0.0991	0.1030
0.0860	0.0759	0.1331	1.0	0.6383	0.0869	0.0771	0.1231	0.5647	0.1178	0.2455	0.3966
0.0782	0.0780	0.0955	0.6383	1.0	0.0813	0.0875	0.1225	0.2822	0.0642	0.4728	0.1905
0.0776	0.0932	0.1150	0.0869	0.0813	1.0	0.1379	0.1266	0.1192	0.1358	0.1108	0.0790
0.0767	0.4517	0.2496	0.0771	0.0875	0.1379	1.0	0.0951	0.0857	0.0676	0.0631	0.1045
0.0729	0.1959	0.2982	0.1231	0.1225	0.1266	0.0951	1.0	0.1399	0.2036	0.1563	0.1008
0.0695	0.1131	0.1096	0.5647	0.2822	0.1192	0.0857	0.1399	1.0	0.0740	0.2057	0.1904
0.0672	0.1047	0.1510	0.1178	0.0642	0.1358	0.0676	0.2036	0.0740	1.0	0.0772	0.1148
0.0665	0.0888	0.0991	0.2455	0.4728	0.1108	0.0631	0.1563	0.2057	0.0772	1.0	0.1007
0.0645	0.1046	0.1030	0.3966	0.1905	0.0790	0.1045	0.1008	0.1904	0.1148	0.1007	1.0



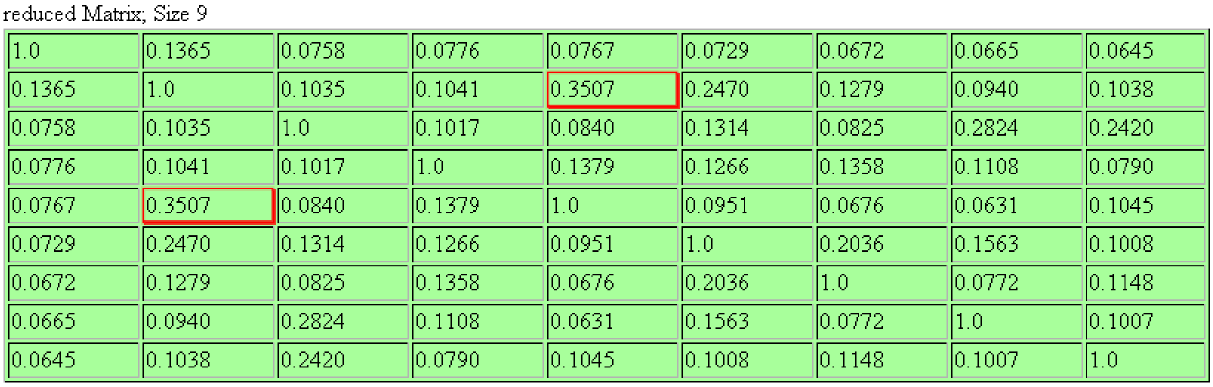
reduced Matrix; Size 11

1.0	0.1551	0.1178	0.0821	0.0776	0.0767	0.0729	0.0695	0.0672	0.0665	0.0645
0.1551	1.0	0.5891	0.0769	0.0932	0.4517	0.1959	0.1131	0.1047	0.0888	0.1046
0.1178	0.5891	1.0	0.1143	0.1150	0.2496	0.2982	0.1096	0.1510	0.0991	0.1030
0.0821	0.0769	0.1143	1.0	0.0841	0.0823	0.1228	0.4234	0.0910	0.3591	0.2936
0.0776	0.0932	0.1150	0.0841	1.0	0.1379	0.1266	0.1192	0.1358	0.1108	0.0790
0.0767	0.4517	0.2496	0.0823	0.1379	1.0	0.0951	0.0857	0.0676	0.0631	0.1045
0.0729	0.1959	0.2982	0.1228	0.1266	0.0951	1.0	0.1399	0.2036	0.1563	0.1008
0.0695	0.1131	0.1096	0.4234	0.1192	0.0857	0.1399	1.0	0.0740	0.2057	0.1904
0.0672	0.1047	0.1510	0.0910	0.1358	0.0676	0.2036	0.0740	1.0	0.0772	0.1148
0.0665	0.0888	0.0991	0.3591	0.1108	0.0631	0.1563	0.2057	0.0772	1.0	0.1007
0.0645	0.1046	0.1030	0.2936	0.0790	0.1045	0.1008	0.1904	0.1148	0.1007	1.0



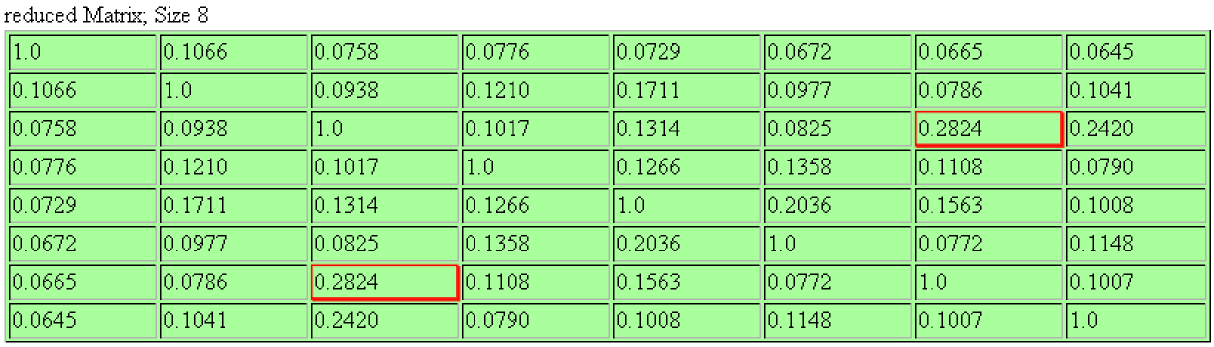
reduced Matrix; Size 10

1.0	0.1365	0.0821	0.0776	0.0767	0.0729	0.0695	0.0672	0.0665	0.0645
0.1365	1.0	0.0956	0.1041	0.3507	0.2470	0.1114	0.1279	0.0940	0.1038
0.0821	0.0956	1.0	0.0841	0.0823	0.1228	0.4234	0.0910	0.3591	0.2936
0.0776	0.1041	0.0841	1.0	0.1379	0.1266	0.1192	0.1358	0.1108	0.0790
0.0767	0.3507	0.0823	0.1379	1.0	0.0951	0.0857	0.0676	0.0631	0.1045
0.0729	0.2470	0.1228	0.1266	0.0951	1.0	0.1399	0.2036	0.1563	0.1008
0.0695	0.1114	0.4234	0.1192	0.0857	0.1399	1.0	0.0740	0.2057	0.1904
0.0672	0.1279	0.0910	0.1358	0.0676	0.2036	0.0740	1.0	0.0772	0.1148
0.0665	0.0940	0.3591	0.1108	0.0631	0.1563	0.2057	0.0772	1.0	0.1007
0.0645	0.1038	0.2936	0.0790	0.1045	0.1008	0.1904	0.1148	0.1007	1.0



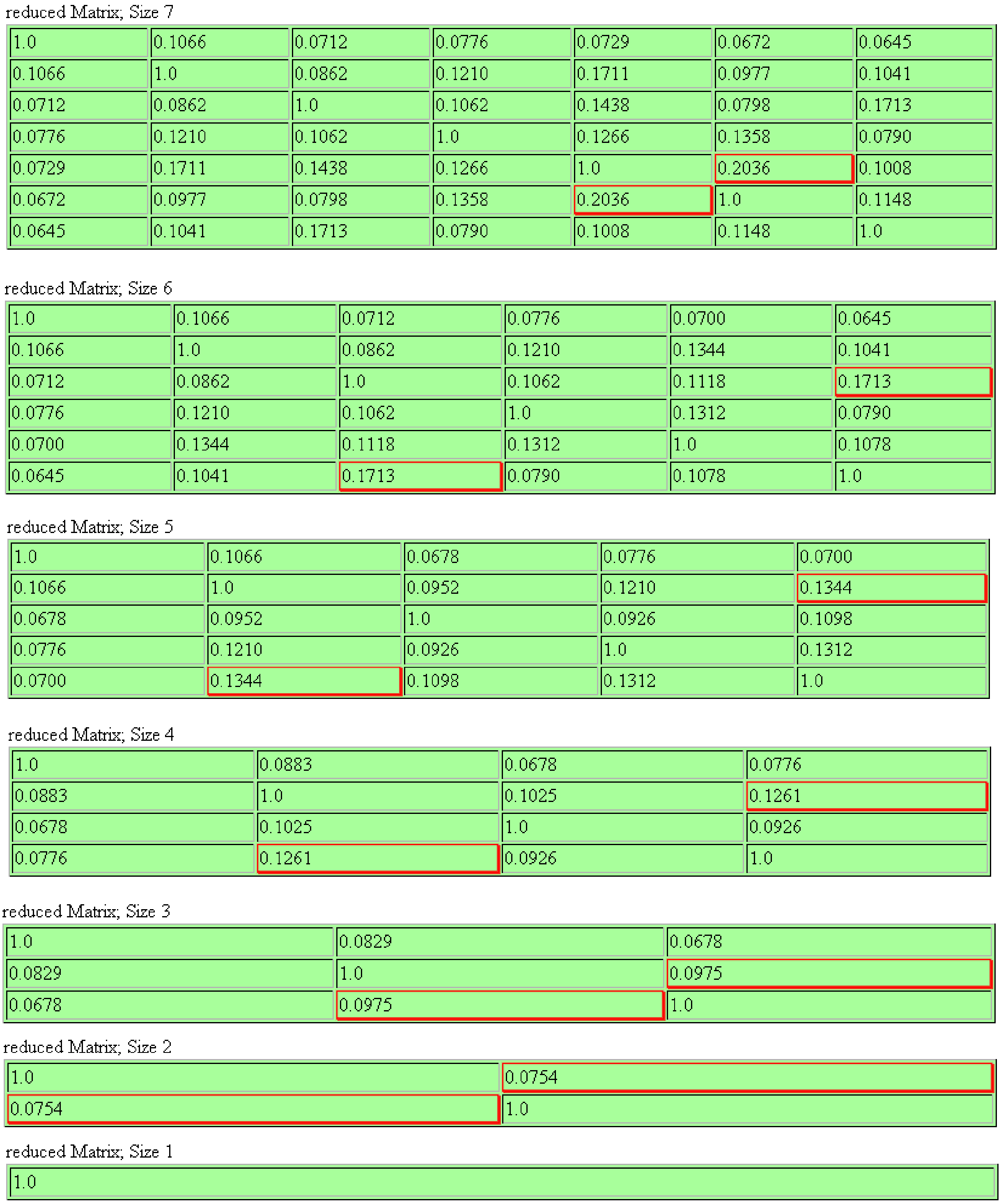
reduced Matrix; Size 9

1.0	0.1365	0.0758	0.0776	0.0767	0.0729	0.0672	0.0665	0.0645
0.1365	1.0	0.1035	0.1041	0.3507	0.2470	0.1279	0.0940	0.1038
0.0758	0.1035	1.0	0.1017	0.0840	0.1314	0.0825	0.2824	0.2420
0.0776	0.1041	0.1017	1.0	0.1379	0.1266	0.1358	0.1108	0.0790
0.0767	0.3507	0.0840	0.1379	1.0	0.0951	0.0676	0.0631	0.1045
0.0729	0.2470	0.1314	0.1266	0.0951	1.0	0.2036	0.1563	0.1008
0.0672	0.1279	0.0825	0.1358	0.0676	0.2036	1.0	0.0772	0.1148
0.0665	0.0940	0.2824	0.1108	0.0631	0.1563	0.0772	1.0	0.1007
0.0645	0.1038	0.2420	0.0790	0.1045	0.1008	0.1148	0.1007	1.0



reduced Matrix; Size 8

1.0	0.1066	0.0758	0.0776	0.0729	0.0672	0.0665	0.0645
0.1066	1.0	0.0938	0.1210	0.1711	0.0977	0.0786	0.1041
0.0758	0.0938	1.0	0.1017	0.1314	0.0825	0.2824	0.2420
0.0776	0.1210	0.1017	1.0	0.1266	0.1358	0.1108	0.0790
0.0729	0.1711	0.1314	0.1266	1.0	0.2036	0.1563	0.1008
0.0672	0.0977	0.0825	0.1358	0.2036	1.0	0.0772	0.1148
0.0665	0.0786	0.2824	0.1108	0.1563	0.0772	1.0	0.1007
0.0645	0.1041	0.2420	0.0790	0.1008	0.1148	0.1007	1.0



reduced Matrix; Size 7

1.0	0.1066	0.0712	0.0776	0.0729	0.0672	0.0645
0.1066	1.0	0.0862	0.1210	0.1711	0.0977	0.1041
0.0712	0.0862	1.0	0.1062	0.1438	0.0798	0.1713
0.0776	0.1210	0.1062	1.0	0.1266	0.1358	0.0790
0.0729	0.1711	0.1438	0.1266	1.0	0.2036	0.1008
0.0672	0.0977	0.0798	0.1358	0.2036	1.0	0.1148
0.0645	0.1041	0.1713	0.0790	0.1008	0.1148	1.0

reduced Matrix; Size 6

1.0	0.1066	0.0712	0.0776	0.0700	0.0645
0.1066	1.0	0.0862	0.1210	0.1344	0.1041
0.0712	0.0862	1.0	0.1062	0.1118	0.1713
0.0776	0.1210	0.1062	1.0	0.1312	0.0790
0.0700	0.1344	0.1118	0.1312	1.0	0.1078
0.0645	0.1041	0.1713	0.0790	0.1078	1.0

reduced Matrix; Size 5

1.0	0.1066	0.0678	0.0776	0.0700
0.1066	1.0	0.0952	0.1210	0.1344
0.0678	0.0952	1.0	0.0926	0.1098
0.0776	0.1210	0.0926	1.0	0.1312
0.0700	0.1344	0.1098	0.1312	1.0

reduced Matrix; Size 4

1.0	0.0883	0.0678	0.0776
0.0883	1.0	0.1025	0.1261
0.0678	0.1025	1.0	0.0926
0.0776	0.1261	0.0926	1.0

reduced Matrix; Size 3

1.0	0.0829	0.0678
0.0829	1.0	0.0975
0.0678	0.0975	1.0

reduced Matrix; Size 2

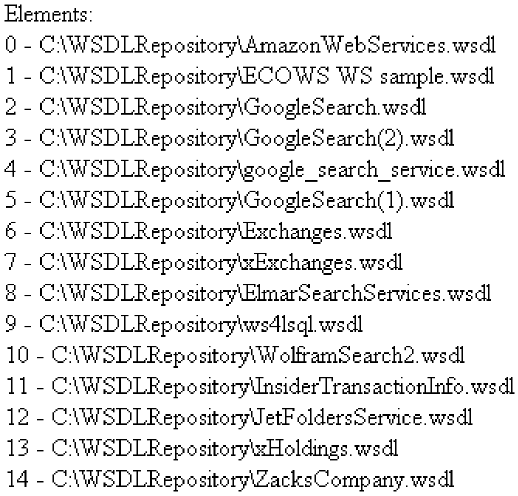
1.0	0.0754
0.0754	1.0

reduced Matrix; Size 1

1.0

Figure B.2: Matrix reduction steps

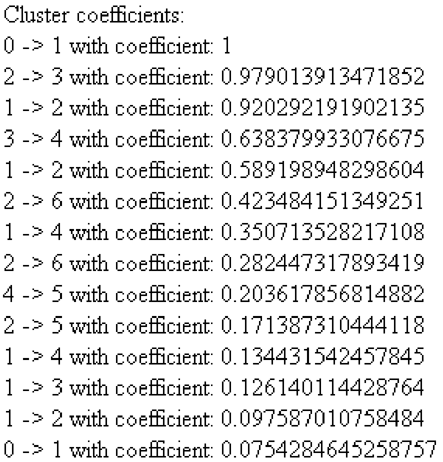
B.3 Amazon Cluster - Cluster elements



```
Elements:
0 - C:\WSDLRepository\AmazonWebServices.wsdl
1 - C:\WSDLRepository\ECOWS WS sample.wsdl
2 - C:\WSDLRepository\GoogleSearch.wsdl
3 - C:\WSDLRepository\GoogleSearch(2).wsdl
4 - C:\WSDLRepository\google_search_service.wsdl
5 - C:\WSDLRepository\GoogleSearch(1).wsdl
6 - C:\WSDLRepository\Exchanges.wsdl
7 - C:\WSDLRepository\xExchanges.wsdl
8 - C:\WSDLRepository\ElmarSearchServices.wsdl
9 - C:\WSDLRepository\ws4lsql.wsdl
10 - C:\WSDLRepository\WolframSearch2.wsdl
11 - C:\WSDLRepository\InsiderTransactionInfo.wsdl
12 - C:\WSDLRepository\JetFoldersService.wsdl
13 - C:\WSDLRepository\xHoldings.wsdl
14 - C:\WSDLRepository\ZacksCompany.wsdl
```

Figure B.3: Matrix element listing

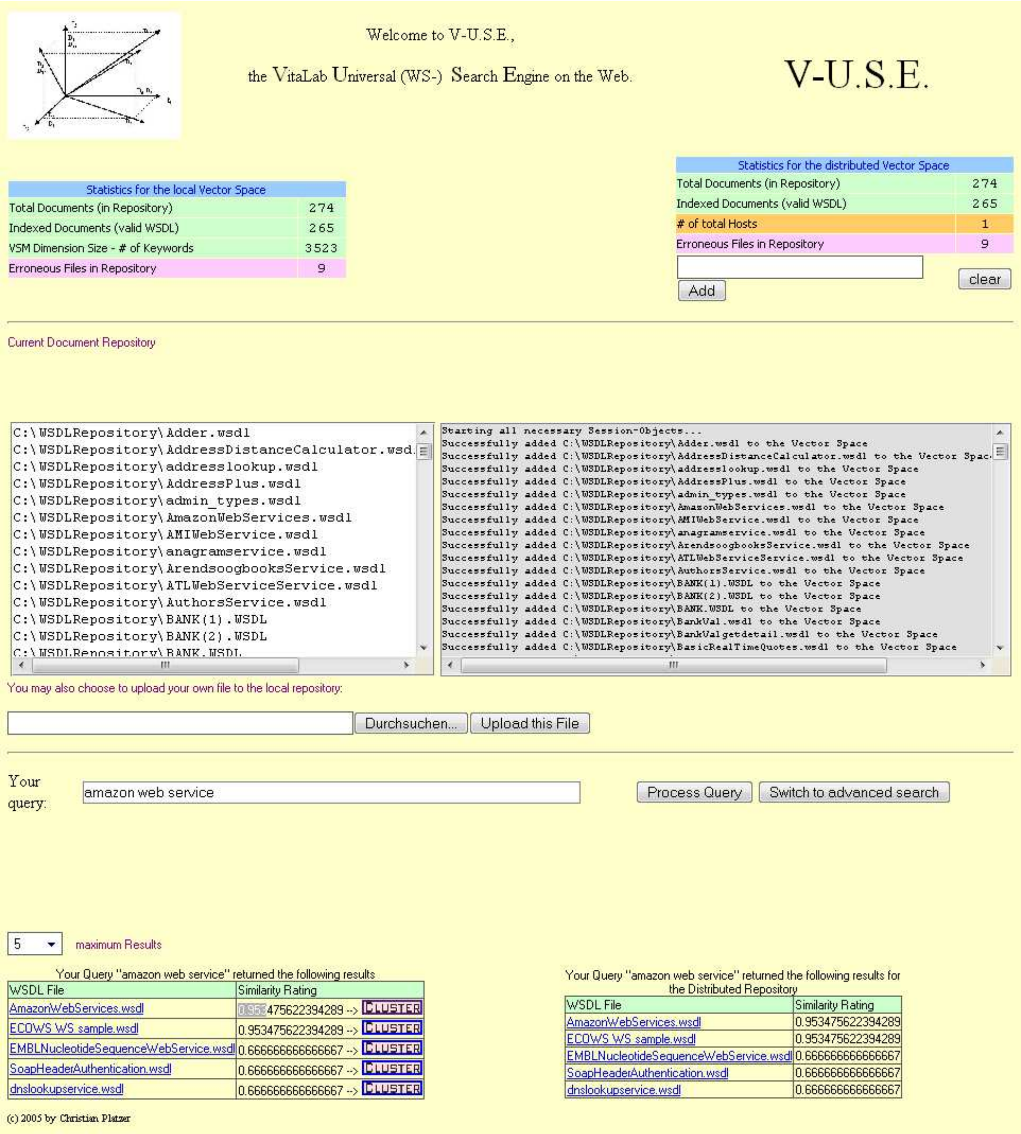
B.4 Amazon Cluster - Cluster coefficients



```
Cluster coefficients:
0 -> 1 with coefficient: 1
2 -> 3 with coefficient: 0.979013913471852
1 -> 2 with coefficient: 0.920292191902135
3 -> 4 with coefficient: 0.638379933076675
1 -> 2 with coefficient: 0.589198948298604
2 -> 6 with coefficient: 0.423484151349251
1 -> 4 with coefficient: 0.350713528217108
2 -> 6 with coefficient: 0.282447317893419
4 -> 5 with coefficient: 0.203617856814882
2 -> 5 with coefficient: 0.171387310444118
1 -> 4 with coefficient: 0.134431542457845
1 -> 3 with coefficient: 0.126140114428764
1 -> 2 with coefficient: 0.097587010758484
0 -> 1 with coefficient: 0.0754284645258757
```

Figure B.4: Matrix reduction coefficients

B.5 Search result example



Welcome to V-U.S.E.,

the VitaLab Universal (WS-) Search Engine on the Web.

V-U.S.E.

Statistics for the local Vector Space	
Total Documents (in Repository)	274
Indexed Documents (valid WSDL)	265
VSM Dimension Size - # of Keywords	3523
Erroneous Files in Repository	9

Statistics for the distributed Vector Space	
Total Documents (in Repository)	274
Indexed Documents (valid WSDL)	265
# of total Hosts	1
Erroneous Files in Repository	9

Current Document Repository

```
C:\WSDLRepository\Adder.wsdl
C:\WSDLRepository\AddressDistanceCalculator.wsd
C:\WSDLRepository\addresslookup.wsdl
C:\WSDLRepository\AddressPlus.wsdl
C:\WSDLRepository\admin_types.wsdl
C:\WSDLRepository\AmazonWebServices.wsdl
C:\WSDLRepository\AMIWebService.wsdl
C:\WSDLRepository\anagramservice.wsdl
C:\WSDLRepository\ArendsoogbooksService.wsdl
C:\WSDLRepository\ATLWebServiceService.wsdl
C:\WSDLRepository\AuthorsService.wsdl
C:\WSDLRepository\BANK(1).WSDL
C:\WSDLRepository\BANK(2).WSDL
C:\WSDLRepository\BANK.WSDL
```

```
Starting all necessary Session-Objects...
Successfully added C:\WSDLRepository\Adder.wsdl to the Vector Space
Successfully added C:\WSDLRepository\AddressDistanceCalculator.wsdl to the Vector Spac
Successfully added C:\WSDLRepository\addresslookup.wsdl to the Vector Space
Successfully added C:\WSDLRepository\AddressPlus.wsdl to the Vector Space
Successfully added C:\WSDLRepository\admin_types.wsdl to the Vector Space
Successfully added C:\WSDLRepository\AmazonWebServices.wsdl to the Vector Space
Successfully added C:\WSDLRepository\AMIWebService.wsdl to the Vector Space
Successfully added C:\WSDLRepository\anagramservice.wsdl to the Vector Space
Successfully added C:\WSDLRepository\ArendsoogbooksService.wsdl to the Vector Space
Successfully added C:\WSDLRepository\ATLWebServiceService.wsdl to the Vector Space
Successfully added C:\WSDLRepository\AuthorsService.wsdl to the Vector Space
Successfully added C:\WSDLRepository\BANK(1).WSDL to the Vector Space
Successfully added C:\WSDLRepository\BANK(2).WSDL to the Vector Space
Successfully added C:\WSDLRepository\BANK.WSDL to the Vector Space
Successfully added C:\WSDLRepository\BankVal.wsdl to the Vector Space
Successfully added C:\WSDLRepository\BankValgetdetail.wsdl to the Vector Space
Successfully added C:\WSDLRepository\BasicRealTimeQuotes.wsdl to the Vector Space
```

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

Your query: amazon web service

5 maximum Results

Your Query "amazon web service" returned the following results

WSDL File	Similarity Rating
AmazonWebServices.wsdl	0.953475622394289 --> CLUSTER
ECOWS WS sample.wsdl	0.953475622394289 --> CLUSTER
EMBLNucleotideSequenceWebService.wsdl	0.666666666666667 --> CLUSTER
SoapHeaderAuthentication.wsdl	0.666666666666667 --> CLUSTER
dnslookupservice.wsdl	0.666666666666667 --> CLUSTER

Your Query "amazon web service" returned the following results for the Distributed Repository

WSDL File	Similarity Rating
AmazonWebServices.wsdl	0.953475622394289
ECOWS WS sample.wsdl	0.953475622394289
EMBLNucleotideSequenceWebService.wsdl	0.666666666666667
SoapHeaderAuthentication.wsdl	0.666666666666667
dnslookupservice.wsdl	0.666666666666667

(c) 2005 by Christian Platzer

Figure B.5: Search result example