

AN ONTOLOGICAL APPROACH FOR SEARCHING MULTI DATASOURCE WEB SERVICES

by

Ahmed Sabbir Arif

A thesis submitted to the faculty of graduate studies
Lakehead University
in partial fulfillment of the requirements for the degree of
Masters of Science in Mathematical Science

Department of Computer Science
Lakehead University
April 2006

Copyright © Ahmed Sabbir Arif 2006



Library and
Archives Canada

Bibliothèque et
Archives Canada

Published Heritage
Branch

Direction du
Patrimoine de l'édition

395 Wellington Street
Ottawa ON K1A 0N4
Canada

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file *Votre référence*
ISBN: 978-0-494-24050-2
Our file *Notre référence*
ISBN: 978-0-494-24050-2

NOTICE:

The author has granted a non-exclusive license allowing Library and Archives Canada to reproduce, publish, archive, preserve, conserve, communicate to the public by telecommunication or on the Internet, loan, distribute and sell theses worldwide, for commercial or non-commercial purposes, in microform, paper, electronic and/or any other formats.

The author retains copyright ownership and moral rights in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

AVIS:

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque et Archives Canada de reproduire, publier, archiver, sauvegarder, conserver, transmettre au public par télécommunication ou par l'Internet, prêter, distribuer et vendre des thèses partout dans le monde, à des fins commerciales ou autres, sur support microforme, papier, électronique et/ou autres formats.

L'auteur conserve la propriété du droit d'auteur et des droits moraux qui protègent cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this thesis.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de cette thèse.

While these forms may be included in the document page count, their removal does not represent any loss of content from the thesis.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.


Canada

To my parents (Mr. M. A. Mannan and Mrs. Shireen Jahan) and younger brother (Ahmed Sazzid Arif).

Disclaimer

This thesis* has referred a lot of published articles. While quoting form here please acknowledge the ORIGINAL references. The author of this thesis will not be hold responsible if any portion of the thesis is copied, published or distributed without proper referencing. The following sections of the research have been collaborated with other researchers and authors, while using materials from these sections please refer accordingly: The example used in **Section 1.3** was originally given by **David Chappell** (Chappell 2002) in his blog. He is the Principal of Chappell & Associates in San Francisco, California, USA. **Section 3.4** is collaborated with **Erhard Rahm** and **Philip Bernstein's** (Rahm, Bernstein 2001) survey. Erhard Rahm is a professor at the University of Leipzig in Leipzig, Germany and Philip Bernstein is the principal researcher in the Database Group of Microsoft Research. **Section 3.5** is collaborated with **Badrul Sarwar's** (Sarwar, Karypis et al. 2001) research on item-based collaborative filtering recommendation algorithms. Badrul Sarwar works at Attosoft Corporation in Santa Clara, California, USA. Other than that some portions of the **Section 2.2** and **Section 2.3** have been collected from this book (Lemow, Newcomer 2004). I would like to thank all of them for their kind support.

* To get an **UPDATED VERSION** of this thesis please email the author [asarif@gmail.com, asarif@lakeheadu.ca]

Abstract

This thesis addresses the issue of Web Service search. This thesis focuses on solving the problem of searching for Web Services which are associated with relatively similar datasources. It attempts to overcome the search limitations of the UDDI standard Web discovery protocol which is based on simplest keyword search and have no primitives to look into datasource associated with Web Services.

This thesis developed a searching framework that take into account the Web Service with variety of ontologies and through adopting techniques like schema matching and ontology merging; the developed prototype can find relatively similar datasources. The prototype also adopted techniques based on Collaborative Filtering to infer more datasources that are relevant to the search request based on relatively similar matches from other datasources.

The prototype represents an extension to the capability of Apache Axis. The prototype has been tested on sample of locally published Web Services that have datasources on books. The developed searching techniques prove to be more flexible than other frameworks including UDDI.

Acknowledgments

I would like to express appreciation to my thesis supervisor Dr. Jinan Fiaidhi, for leading me into this interesting research field and guiding me throughout my master's studies and for providing me with the Faculty Research Scholarship from her NSERC grant. I would like to thank everyone from the Department of Computer Science, Lakehead University for their kind support. I would also like to thank Dr. Sabah Mohammed and Dr. Ruizhong Wei for providing me with the guidance and cooperation whenever I needed. I gladly acknowledge the Graduate and International Studies of Lakehead University for awarding me with NSERC Research Capacity Development and Graduate International Tuition Awards. Finally I would like to thank all of my friends and families for their love and support.

Table of Contents

Abstract	i
Acknowledgement	ii
List of Figures	v
List of Tables	vi
List of Examples	vi
1. Introduction	1
1.1. Introduction	1
1.2. How Does Web Service Work?	2
1.2.1. Simple Object Access Protocol (SOAP)	3
1.2.2. Web Service Definition Language (WSDL)	4
1.2.3. The Universal Description, Discovery and Integration	4
1.2.4. Other Web Service Registries	6
1.2.5. The Advantages of Using Web Services	7
1.3. REST: Another Way of Looking at Web Services	8
1.4. Describing the Semantics of Web Services: WS Schema	9
1.4.1. Schema & Ontology Integration	10
1.5. Proposed Web Service Searching Architecture	10
2. Service Oriented Architecture & Web Services	13
2.1. Introduction	13
2.2. What are Services?	13
2.3. Service Oriented Architecture	15
2.4. Creating Web Services	16
2.4.1. Creating Web Service in ASP.NET	17
2.4.2. Creating Web Service in Apache Axis	19
2.4.3. Apache Axis Architectural Overview	21
2.5. Apache Axis or ASP.NET?	24
3. Web Service Search via XML Schema & Ontology	26
3.1. Web Service Search	26
3.2. The Role of W3C XML Schema	27
3.2.1. The Role of XML Schema in Our System	29
3.3. The Role of Ontology	31
3.3.1. W3C Resource Description Framework (RDF)	32
3.3.2. W3C Web Ontology Language (OWL)	33
3.3.3. Role of RDF Ontology in Our System	34
3.3.4. The RDF Ontology Syntax Supported by Our System	35

3.4. Schema-level Matching	36
3.4.1. The Match Operator	36
3.4.2. Matching Strategy in Our System	38
3.4.3. An Approach to Ontology Merging	40
3.5. The Use of Collaborative Filtering (CF)	41
3.5.1. Collaborative Filtering in Our System	43
4. System Prototype Implementation	47
4.1. Web Service Implementation	47
4.1.1. The RDF Generator	50
4.2. Infoset Streaming	51
4.2.1. Pull Parsing versus Push Parsing	52
4.3. Web Service Search Prototype	53
4.3.1. Schema Matching and Ontology Merging	58
4.3.2. Searching for Web Services	59
4.3.3. Collaborative Filtering	63
4.3.4. Editing and Validating XML-type Files	65
4.3.5. About Tab	66
4.4. Source Codes	66
5. Conclusion & Future Research	67
5.1. Future Research	68
Bibliography	70

List of Figures

1.1.	Web Service Participants	1
1.2.	Web Service Runtime Environment.	7
1.3.	Conceptual Search in Our System	11
1.4.	Collaborative Filtering in Our System	12
2.1.	Breakdown of Web Service Components	14
2.2.	Requesting Different Types of Web Services	14
2.3.	(a) A Typical Three-tier Application Architecture	15
	(b) A Service-oriented Application Architecture	15
2.4.	The ASP.NET C# Web Service Status Page	18
2.5.	A Fragment of the WSDL for ASP.NET C# Service	19
2.6.	The Java Axis Web Service Status Page	21
2.7.	The Server Side Message Path of Axis	22
2.8.	The Client Side Message Path of Axis	23
3.1.	The UDDI Data Model	26
3.2.	Visual Representation of XML Schema	29
3.3.	Search for Web Services with Keys Like "author's name"	30
3.4.	Role of Ontology in Services Discovery	31
3.5.	Graph Representation of a P (R, V) Triple	32
3.6.	Triple Representation of a Resource Description	33
3.7.	RDF Generation with our Application	35
3.8.	Ontology Merging Process in Our System	41
3.9.	The Collaborative Filtering Process	42
3.10.	Isolation of Co-rated Items and Similarity Computation	44
3.11.	Item-based Collaborative Filtering Algorithm	45
3.12.	Item-based Collaborative Filtering in Our System	46
4.1.	A Fragment of the WSDL for Our Web Service	49
4.2.	The Start Page for One of Our Web Services	49
4.3.	A Portion of One of Our Web Service Datasources	50
4.4.	RDFGenerator Too	50
4.5.	(a) Screenshot of the First Tab of the WSSearch Application	53
	(b) Screenshot of the Second Tab	54
	(c) Screenshot of the Third Tab	55
4.6.	Class Diagram for WSSearch Prototype	56
4.7.	(a) Screenshot of the "endpoints.txt" File	58
	(b) Screenshot of the "webservices.txt" File	58
4.8.	The SearchHelper File	59
4.9.	(a) Shows a Screenshot of the WSSearch Application after the User Performed a Search for All Book Related Web Services	60
	(b) Shows a Screenshot of the Same Application after the User Performed Search With a Keyword	61
4.10.	Sequence Diagram for all Book Related Web Services Search	62

4.11.	Sequence Diagram for Keyword Search	62
4.12.	Similarity Computed Record File	64
4.13.	Sequence Diagram for Collaborative Filtering (Recommender)	64
4.14.	Recommender in WSearch Screenshot	65
4.15.	(a) Document Editing in WSearch Screenshot	66
	(b) Document Validating in WSearch Screenshot	66

List of Tables

2.1.	ASP.NET versus Apache Axis	25
3.1.	Match on Schemas	37
3.2.	Structure-level Match	38
3.3.	Match Cardinality	39
4.1.	XML Parsers	52

List of Examples

2.1.	A Sample C# Web Service	17
2.2.	A Sample Java Axis Web Service	20
3.1.	XML Schema	28
3.2.	Different Representation of a RDF Statement	33

CHAPTER 1

INTRODUCTION

1.1. Introduction

Web Services (WS) provide a framework for application-to-application interaction, built on top of existing Web protocols (e.g. HTTP) and based on *Extensible Markup Language (XML) Standards* (Siblini, Mansour 2005). Nowadays, Web applications are integrating Web Services from a variety of resources and can run on all kind of machines, either within an enterprise or at external sites (Siblini, Mansour 2005). This case of integration enables tighter business relationships and more efficient business processes.

In other words we can say, a Web Service is a software system identified by a *Uniform Resource Identifier (URI)*, whose public interfaces and bindings are defined and described using XML (Bray, Paoli et al. 2004). Moreover, Web Services can be written in different programming languages, usually distributed over a network or on the Internet that may not have the same runtime environment. Its definition can be discovered by other software systems. These systems may interact with a Web Service in a way described by a *Web Services Definition Language (WSDL)* document using internet protocol (Booth, Haas et al. 2004, Christensen, Curbera et al. 2001). A WSDL file is an XML document that describes a set of messages and how the messages are exchanged. The Web Service framework is divided into three areas: *exchanging messages, service description, and service discovery*. Figure 1.1 shows how Web Services connect these three participants.

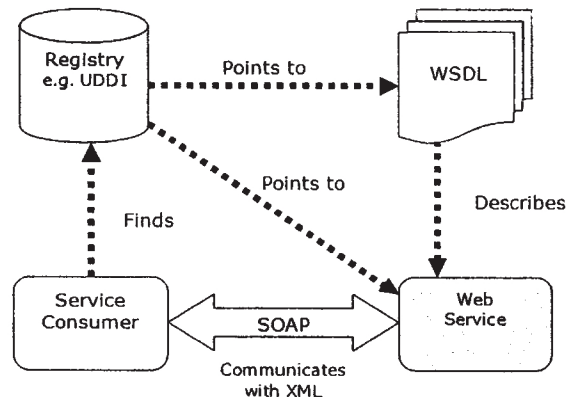


Figure 1.1: Web Service participants.

Web Services are registered so that potential users can discover them. This is done with *Universal Discovery Description and Integration (UDDI)* directory, which could be seen as the yellow pages of Web Services. Nowadays there are commercial and experimental Web Service search engines available (e.g. Woog¹ etc.). However, Web Service discovery remains a hot research area (Shen Derong, Yu Ge et al. 2005). In most of the cases clients rely on the functions or services available within a Web Service or by the Web Service datasource description created by the Web Service providers.

Web Service interoperability relies on the use of open-source data within XML with its semantics expressed in XML metadata. Schema and ontology languages (e.g. XML Schema, RDF Schema etc.) provide enhanced as well as more comprehensive and powerful features than a *Document Type Definition (DTD)*, the traditional mechanism used to describe the structure and content of XML documents (Brickley, Guha et al. 2004, Manola, Miller et al. 2005, McGuinness, Harmelen 2004).

Searching and mining Web Service datasources require intrinsically different techniques and algorithms than those been used with the traditional databases paradigm (Doan, Domingos et al. 2003, Doan, Madhavan et al. 2002, Govert, Kazai et al. 2003, Hakimpour, Geppert 2002, Rahm, Bernstein 2001, Xiao, Cruz et al.). One of the methods of datasource search is the use of schema (and also ontology) matching techniques to create a global schema and then perform search on that. This requires rewritten queries on global schema on the local XML Web Service documents validated by their representative local schemas (Sakamuri, Madria et al. 2003).

In this thesis we will demonstrate a schema & ontology oriented searching strategy to discover Web Services and to perform Collaborative Filtering on the data available in the service datasources. Our proposed architecture will increase the possibility of discovering only the services the searcher is looking for; moreover, it will add some value-added services like “product recommendations” (collaborative filtering). Our strategy also enables the use of service datasources with external services or applications; this gives the users (or searchers) tremendous flexibility and power over the service datasources.

1.2. How Does Web Service Work?

Web Services communicate via XML, which makes it “loosely coupled” architecture (Booth, Haas et al. 2004). XML is a mainstream, non-proprietary, simple but very

¹ <http://haydn.cs.washington.edu:8080/won/wonServlet>

flexible text format for exchanging data. Unlike *HyperText Markup Language (HTML)*, it provides the logical structure of data instead of a visual representation. Using XML, applications can communicate using Web Services even if they are written in different programming languages and run on different operating systems.

Web Services are the building blocks for distributed applications functioning over standard internet protocols. Typically, these are simple request or response services that can be located and invoked by the others. At the same time, each of them remains independent and self-contained (Systinet Corporation 2005). Developing a number of such Web Services into a sophisticated information system is also possible.

We can look at Web Services as a way of connecting three participants: a provider, a requester and a directory (see Figure 1.1). The first has a service to offer. The second is looking for a service to use. The third helps the other two find each other. Usually using UDDI Inquiries, the requester talks to the registry about the services it needs. The registry returns services matching the query and the service requester chooses the one it wants to access.

Web Services translate data structures and method calls into XML text. They send this text using standard transport protocols and translate it back into functions and data structures on the server end. It allows data structure of virtually any complexity be transmitted along with method calls (Booth, Haas et al. 2004). Web Services use three *standard formats or protocols* namely SOAP, WSDL and UDDI to translate into and out of XML.

1.2.1. Simple Object Access Protocol (SOAP)

SOAP is an XML based markup language. The core protocol underlying Web Services, SOAP defines a standard message format for carrying data objects (Box, Ehnebuske et al. 2000). Depending on the rules of how its contents should be serialized, the body of a SOAP document contains one or more objects to be consumed by the receiving application. The root of a SOAP message is an *envelope* in which a developer can place the XML representation of these objects. The SOAP envelope can also contain routing, state and security information, these are placed in one or more headers (Box, Ehnebuske et al. 2000).

Web Services can handle SOAP messages in two different ways. In their simplest form they wrap function arguments and return values. In this way they act as the bonding element for *Remote Procedure Call (RPC)* mechanisms. The second approach is to treat a SOAP message as a one-way document containing information to be handled by a service with the response message optional.

1.2.2. Web Services Definition Language (WSDL)

WSDL is an XML based markup language used to describe and define Web Services (Christensen, Curbera et al. 2001). A WSDL document makes public the “What, How, and Where” of a Web Service. It describes what the Web Service does, how it communicates, and where it resides. A client application developer uses the WSDL document at development-time to generate data types to be placed in SOAP messages, and service interface stubs which are then compiled into the application. Some SOAP implementations use WSDL at runtime to support dynamic communications through generated service proxies.

1.2.3. The Universal Description, Discovery and Integration (UDDI)

The UDDI project is a partnership among industry and business leaders and was founded by *IBM, Ariba, and Microsoft* and now over 300 companies participate (Bellwood 2002). UDDI provides a standards-based set of specifications for service description, discovery, as well as a set of Internet-based implementations (Clement, Hately et al. 2000). This specification has developed quickly because it is backed with rapid implementation, which proves the concepts and provides a rich experience base for further refinement of the specification.

A UDDI registry stores information about service providers and their Web Services. Service providers are typically companies, organizations, or institutions. The information stored in a registry follows a relatively straightforward schema, and many implementations use a relational database system as storage manager. The interface to a registry provides two main functionalities (Sun, Lin et al. 2004). Firstly, the information in the registry must be maintained, that is, it can be registered and updated. Secondly, users can query the registry to retrieve information about service providers and their services. The UDDI information model contains four core elements (Mahmoud 2002):

1. ***Business Information:*** *This is described using the `businessEntity` element, which represents a physical organization. It contains information such as name, description, and contacts about the organization. The `businessEntity` information includes support for yellow pages taxonomies so that searches can be performed to locate organizations who service a particular industry or product category.*

2. **Service Information:** This is described using the `businessService` element, which groups together related services offered by an organization.
3. **Binding Information:** This is described using the `bindingTemplate` element, for information that is relevant for application programs that need to connect to and then communicate with a remote Web Service. In other words, it provides instructions on how to invoke a remote Web Service. The instructions may either be in the form of an interface definition language such as WSDL, or a text-based document.
4. **Information about Specifications for Services:** This is described using the `tModel` element, which is an abstract representation of the technical specification. A `tModel` has a name, publishing organization, and URL pointers to the actual specifications themselves.

The UDDI Application Program Interfaces (API) are divided into two logical parts: the *Inquiry* API and the *Publish* API. They describe the SOAP messages that are used to publish and discover an entry in the registry.

- **The Publish API:** The Publish API provides methods for publishing and updating information contained in a UDDI registry. A business should select a UDDI registry operator site to host its information. Invoking methods in the Publish API requires authorization and is usually done through HyperText Transfer Protocol Secure (HTTPS). The API consists of methods for saving information: `save_business`, `save_service`, `save_binding`, and `save_tModel`. These methods take `authToken` (it's optional, if operators provide another mechanism of authentication such as username and password) and one or more `businessEntity` elements. This API also contains methods for deleting information: `delete_business`, `delete_service`, `delete_binding`, and `delete_tModel`. These methods take the `uuid` key (which was generated by the registry when information was first published) as an input parameter.
- **The Inquiry API:** The Inquiry API provides methods for querying the registry. Some of these methods are: `find_service`, `get_businessDetail`, `get_serviceDetail`, `get_bindingDetail`, and `get_tModelDetail`.

UDDI is not a core Web Service specification. It is clear that a service registry is required part of the Web Service platform, but it isn't clear that UDDI will ever truly become that solution (Lomow, Newcomer 2004).

1.2.4. Other Web Service Registries

*Electronic Business XML (ebXML)*² is one of the most important Web Service registry for business-to-business framework, developed by the ebXML initiative which is a joint project of the *United Nations Centre for Trade Facilitation and Electronic Business (UN/CEFACT)* and the *Organization for the Advancement of Structured Information Standards (OASIS)* (OASIS Open 2006). The ebXML membership includes representatives from over 2000 businesses, institutions, governments, standards bodies, and individuals from around the globe.

An ebXML registry allows organizations to advertise and discover information about businesses (Mahmoud 2002). It stores *Collaboration-Protocol Profile (CPP)* and *Collaboration-Protocol Agreement (CPA)* and other information relevant to business collaboration. The CPP is an XML document that contains information about a business and the way it exchanges information with other businesses. The CPA is also an XML document that describes the specific capabilities that two businesses have agreed to use in business collaboration.

Unlike the UDDI registry, which is a registry of metadata only, the ebXML registry is both a metadata registry as well as a repository that can hold arbitrary content. Only the metadata about a Web Service is published to UDDI. The actual Web Service description (that is, the WSDL document) cannot reside in UDDI and must reside in the service provider's Website. In contrast, a Web Service description may be published in an ebXML registry and repository to include all metadata as well as technical specifications and related artifacts. A common example is as follows: businesses register their profiles (CPPs) in an ebXML registry. When a business searches the registry and finds another business that it wants to collaborate with, it creates a technical agreement (CPA) using the CPP and sends the CPA to the other business. The two businesses collaborate according to the CPA. The ebXML registry offers many other unique features that are valuable for e-business collaboration.

There are other Web Service registry specifications available too, for example, *Systinet Registry*³, *METEOR-S Web Service Discovery Infrastructure (MWSDI)*⁴, etc.

² <http://www.ebxml.org>

³ <http://www.systinet.com/products/sr/overview>

⁴ <http://lsdis.cs.uga.edu/proj/meteor/mwsdi.html>

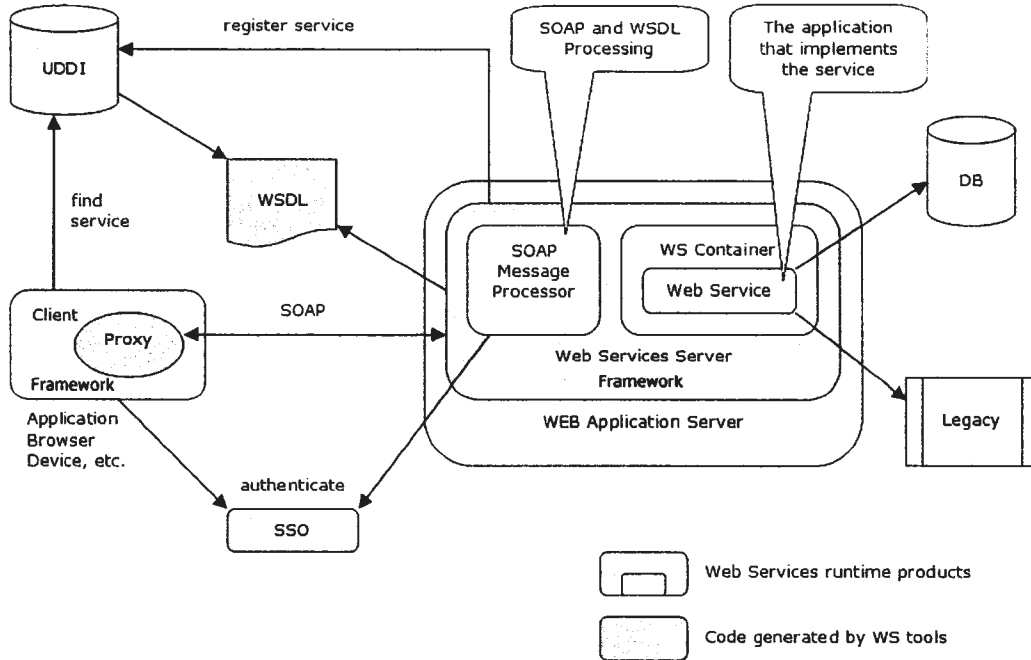


Figure 1.2: Web Service runtime environment.

1.2.5. The Advantages of Using Web Services

Web Services offer many benefits over other distributed computing architectures (Booth, Haas et al. 2004, Lomow, Newcomer 2004, Systinet Corporation 2005).

- ❑ **Interoperability:** *This is the most important benefit of Web Services. Web Services typically work outside of private networks, offering developers a non-proprietary route to their solutions. Services developed are likely, therefore, to have a longer life-span, offering better return on investment of the developed service. Web Services also let developers use their preferred programming languages (e.g., Java, C++, VBScript, JavaScript, etc.). In addition, because of the use of standards-based communications methods Web Services are virtually platform-independent.*
- ❑ **Usability:** *Web Services allow the business logic of many different systems to be exposed over the Web. This gives applications the freedom to choose the Web Services that they need. Instead of reinventing the wheel for each client, only including additional application-specific business logic on the client-side is required. This allows developers to develop services and/or client-side code using the languages and tools that they want.*
- ❑ **Reusability:** *Web Services provide not a component-based model of application development, but the closest thing possible to zero-coding deployment of such*

services. This makes it easy to reuse Web Service components as appropriate in other services. It also makes it easy to deploy legacy code as a Web Service.

- **Deployability:** *Web Services are deployed over standard Internet technologies. This makes it possible to deploy Web Services even over the firewall to servers running on the Internet on the other side of the globe.*

1.3. REST: Another Way of Looking at Web Services

Web Services described up to now communicates via SOAP. Although some might argue that it's possible to create Web Services without this standard protocol (Chappell 2002, Clark 2003). SOAP can be used in an asynchronous style, it evolved from an earlier protocol called XML-RPC, and the initial SOAP specification explicitly defined an RPC-style mapping of SOAP to HTTP (Chappell 2002). SOAP clearly grows out of this earlier tradition in distributed computing.

However Web Service can utilize another communication protocol called *Representational State Transfer (REST)*. Rather than growing out of the RPC world, its roots are solidly embedded in the Web itself.

Let's look at an example to understand the differences better. In the SOAP world, each endpoint has a URI, such as `http://www.lakeheadbank.com`, and each endpoint exposes various methods. Any of these can be invoked via an HTTP POST, with the specific SOAP method being called identified within a SOAP envelope that gets embedded in the POST request. Each data object that's accessed is identified using some parameter value, such as a character string. To read the balance of a savings account maintained by QwickBank, for example, a client might invoke a `GetBalance` method at `http://www.lakeheadbank.com` identifying a particular account by passing its account number as a parameter.

REST takes more strictly Web-oriented approach. But the result exposing methods using Web technologies is much the same. The way those methods are exposed, however, is quite different. For instance, rather than assigning each endpoint a URI, the REST approach argues that each data item should have a URI. Instead of a single endpoint for QwickBank, it exposes a distinct URI for each account the bank maintains. This is much more like the Web today, in which each item (for example, each page) that a client wishes to access can be directly named.

And rather than hiding arbitrary method names inside a generic HTTP POST request, it uses the HTTP methods that already exist. These methods, such as PUT, GET, POST, and DELETE can be used to *create, read, update, and delete* information. These four operations, sometimes referred to with the inelegant acronym *CRUD*, are the fundamental things we need to do to data. We can't use them directly for many reasons, like for example, then firewalls could filter based on HTTP method names rather than performing the complex (and perhaps impossible) task of deciphering each SOAP packet to filter requests on a per-method basis.

Similarly, rather than identifying parameters using character strings & other values opaquely embedded in a SOAP packet, REST uses URIs. Identifying everything with a URI is fundamental to how the Web works. Among other things, a common naming scheme allows easier composition of independently developed software, which is a core goal of Web Services. In the REST model, requesting the balance of an account maintained by QwickBank could be as simple as sending an HTTP GET to the account's URI. Rather than building a distinct infrastructure on top of the Web, REST uses what the Web provides to create a simpler and perhaps more effective means to the end of Web Services.

Though REST's ideas are attractive, it's hard to imagine SOAP being displaced (Chappell 2002). Every major vendor supports SOAP today, and its already quite well established, still the ideas embodied in REST are worth understanding (Chappell 2002).

1.4. Describing the Semantics of Web Services: WS Schema

Schemas and ontologies provide a vocabulary of terms that describes a domain of interest. They constrain the meaning of terms used in the vocabulary.

WS schema and ontology is described by XML schema and ontology languages. It resembles the *Data Definition Language (DDL)* for a relational database (Roy, Ramanujan 2001). In a relational database, we use a DDL to create a table and to specify rules and constraints for that table. Similarly, the XML schema and ontology languages provides the necessary framework for creating XML documents by specifying the valid structure, constraints, and data types for the various elements and attributes of an XML document. Ontology differs from an XML schema in that it is a knowledge representation, not a message format (Smith, Welty et al. 2004).

1.4.1. Schema and Ontology Integration

Most work on schema and ontology match has been motivated by schema and ontology integration, a problem that has been investigated since the early 1980s (Rahm, Bernstein 2001).

Since the schemas and ontologies are independently developed, they often have different structure and terminology. This can obviously occur when the schemas and ontologies are from different domains, such as a “real estate schema” and “property tax schema”. However, it also occurs even if they model the same real world domain, just because they were developed by different people in different real-world contexts (Rahm, Bernstein 2001). Thus, a first step in integrating the schemas and ontologies is to identify and characterize these inter-schema and ontology relationships. This process is called schema and ontology matching. Once they are identified, matching elements can be unified under a coherent, integrated schema and ontology. During this integration programs or queries are created that permit translation of data from the original schemas into the integrated representation (Rahm, Bernstein 2001).

1.5. Proposed Web Service Searching Architecture

General prevalent crawler based search engines are not a solution for searching highly dynamic contents of Web Services (Graupmann, Biber et al. 2003), because:

- ❑ *Information is not accessible by crawlers as sometimes it is generated as a response to a HTML form submission.*
- ❑ *Links between pages change frequently, for example the content of bidding Web Service that changes continuously.*

To solve that problem, nowadays developers wrap the portal search engines themselves into Web Services and use them in the fashion of a meta-search-engine. But discovering the services is another issue that remains a hot research area (Shen Derong, Yu Ge et al. 2005). Even though UDDI was proposed as a standard to enable universal discovery of services, the promise of dynamically finding Web Services is not fully achieved in the current specification (Colgrave, Akkiraju et al. 2004).

UDDI is limited in its search services by its inability to extend beyond the keyword-based matches (Colgrave, Akkiraju et al. 2004). For example, if we have two similar services with slight change in their description, the traditional UDDI search won't be

able to discover the one that don't have the requested keyword in it. That is why searching for Web Services containing multi-data resources need a special searching framework. Another problem with UDDI is the registry does not provide any value-added service, such as checking the quality of the registered services, collaborative filtering, etc. (Pautasso 2005).

As an attempt to resolve these problems, in this thesis we will show a new architecture for discovering Web Services. In our architecture we will use schema and ontology matching strategy. To be more specific, in our architecture we will match and merge the ontology for the service and the schema for the datasources to create an XML based search helper file; and will parse through that file during the searching process to discover services. This will extend our searching from keyword-based search to conceptual search. Figure 1.3 shows how our conceptual search works.

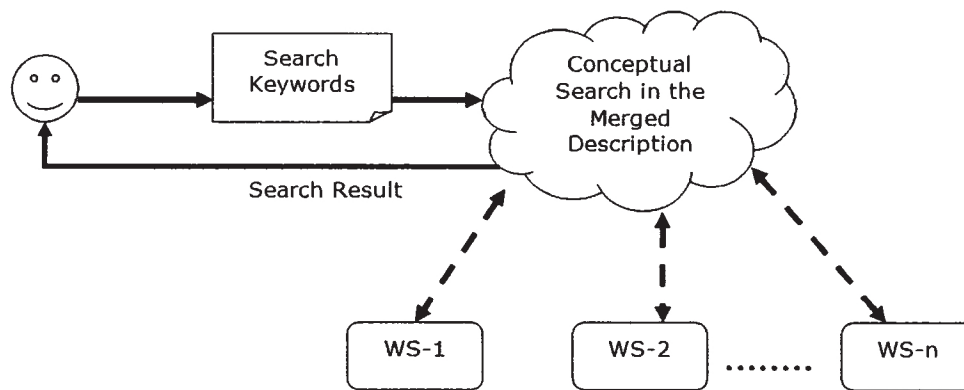


Figure 1.3: Conceptual search in our system.

At first the system generates a search helper (merged description) file that contains the similar type of Web Services' information. Later when user enters a keyword to find particular services our system parses through the search helper (merged description) to look similar services. The similarity or relation is understood from the merged description itself. If any match, relation or similarity found then the system will return the user with the Web Service URI and other information.

With our system it is also possible to search into the Web Service datasources. This kind of search is important when the user is looking for services where a specific item or information is available. In this kind of search the system looks for the requested item or information in the service datasource first (with the help of the search helper file). The system will only show the services those have similar item or information in their datasource.

Along with Web Service discovery and datasource search we added value-added service like collaborative filtering in our architecture. While the user is looking for a specific item in the service datasources, we use collaborative filtering algorithms to recommend or predict other similar items that the user might like. Figure 1.4 shows how collaborative filtering works in our system.

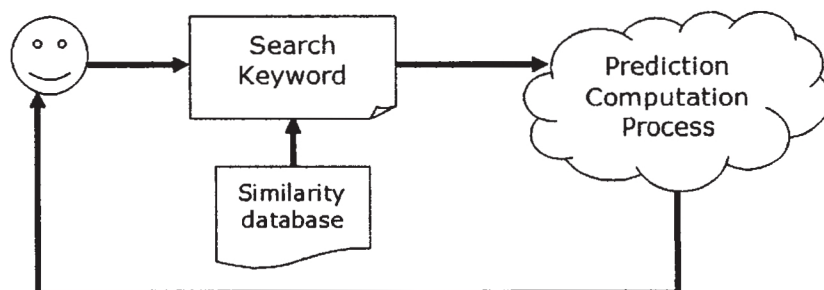


Figure 1.4: Collaborative filtering in our system.

More details on the architecture will be discussed in the next chapters. This is to be noted that, we do not attempt to replace the existing meta-data searching or Web Service discovery approaches, we just show another way of doing it.

CHAPTER 2

SERVICE ORIENTED ARCHITECTURE AND WEB SERVICES

2.1. Introduction

In this chapter we will discuss the *Service-oriented Architecture (SOA)* and differences between Web Services and other architecture for SOA. Then we will demonstrate and compare a few popular ways of creating Web Services, and finally we will show how we've created our Web Services and why we believe that our way of creating Web Service is better.

2.2. What are Services?

A service is a location on the network that has a machine-readable description of the messages it receives and optionally returns (Lomow, Newcomer 2004). In other words, a service is an abstract notion that must be implemented by a concrete *agent*. The agent is the concrete piece of software or hardware that sends and receives messages, while the service is the resource characterized by the abstract set of functionality that is provided (Booth, Haas et al. 2004). The agent may have changed but the Web Service remains same. A schema for the data contained in the message is used as the main part of the contract (i.e., description) established between a service requester and a service provider. Other items of metadata describe the *network address* for the service, the *operations* it supports, and its requirements for *reliability, security, and transactionality*.

Figure 2.1 illustrates the relationship among the parts of a service, including the description, the implementation, and the mapping layer between the two. The service implementation can be any execution environment for which Web Service support is available. The service implementation is also called the *executable agent* (Lomow, Newcomer 2004). The executable agent is responsible for implementing the Web Service processing model as defined in the various Web Service specifications. The executable agent runs within the execution environment, which is typically a software system or programming language.

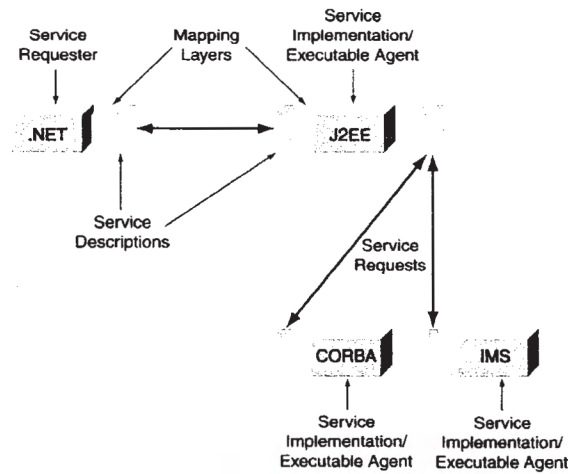


Figure 2.1: Breakdown of Web service components.

An important part of the definition of a service is that its description is separated from its executable agent. One description might have multiple different executable agents associated with it (see Figure 2.1). Similarly, one agent might support multiple descriptions. The description is separated from the execution environment using a *mapping layer* (a.k.a., *transformation layer*). The mapping layer is often implemented using proxies and stubs. The mapping layer is responsible for accepting the message, transforming the XML data to the native format, and dispatching the data to the executable agent.

The Web Service's roles include requester and provider (see Chapter 1). The service requester initiates the execution of a service by sending a message to a service provider (see Figure 2.2). The service provider executes the service upon receipt of a message and returns the results, if any are specified, to the requester. A requester can be a provider, and vice versa, meaning an execution agent can play either or both roles. The whole concept is highly abstract. One of the greatest benefits of this service abstraction is its ability to easily access a variety of service types, including newly developed services, wrapped legacy applications, and applications composed of other services (Lomow, Newcomer 2004).

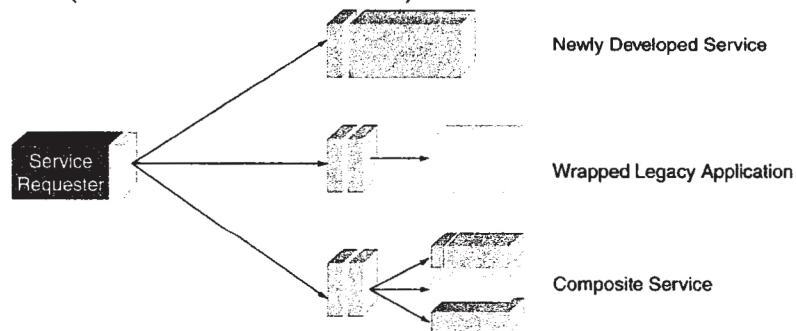


Figure 2.2: Requesting different types of Web Services.

2.3. Service-oriented Architecture (SOA)

A *Service-oriented Architecture (SOA)* is a style of design that guides all aspects of creating and using business services throughout their lifecycle. It's also a way to define and provision an *Information Technology (IT)* infrastructure to allow different applications to exchange data and participate in business processes, regardless of the operating systems or programming languages underlying those applications (Anand, Padmanabhuni et al. 2005, Channabasavaiah, Holley et al. 2003, He 2003, Lomow, Newcomer 2004). Unlike three-tier model, in a service-oriented architecture clients consume services rather than invoking discreet method calls directly. Figure 2.3 compares SOA with traditional three-tier architecture.

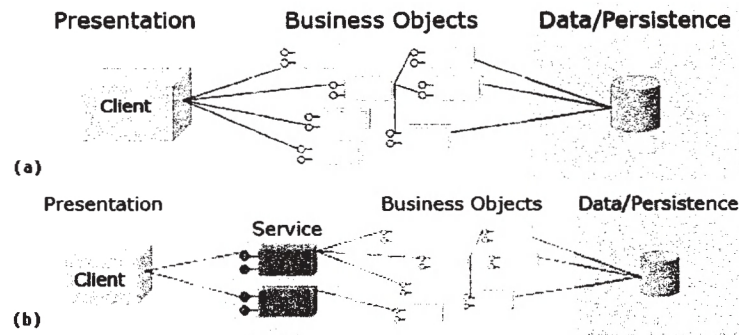


Figure 2.3: (a) A typical three-tier application architecture, (b) A service-oriented application architecture.

The concept of SOA isn't new, what is new is the ability to mix and match execution environments, clearly separating the service interface from the execution technology, allowing IT departments to choose the best execution environment for each job (whether it's a new or existing application) and tying them together using a consistent architectural approach. Previous implementations of SOA were based on a single execution environment technology (Lomow, Newcomer 2004). The prior effort that has gone into defining distributed, inter-application communication architectures are as follows (Dietzen 2004):

- **Synchronous (RPC-oriented):** CICS Distributed Program Link (DPL), Distributed Computing Environment (DCE), Distributed Component Object Model (DCOM), Common Object Request Broker Architecture (CORBA) IIOP, Java Remote Method Invocation (RMI), Relational Database Management System (RDBMS) stored procedures, and so on.

- **Asynchronous (Messaging-oriented):** *CICS Transient Data Queues (TDQs), Tuxedo ATMI, IBM MQSeries, Tibco rendezvous, Microsoft Message Queuing (MSMQ), Java Message Service (JMS), and so on.*

Although the concepts behind SOA were established long before Web Services came along; Web Services play a major role in a SOA. This is because Web Services are built on top of well-known and platform-independent protocols (e.g., HTTP, XML, UDDI, WSDL, and SOAP). These protocols help Web Service fulfill the key requirements of a SOA that a service must be dynamically discoverable and invokeable. As we have seen in Chapter 1, this requirement is fulfilled by UDDI, WSDL, and SOAP. SOA requires that a service have a platform-independent interface contract. This requirement is fulfilled by XML. SOA must have interoperability. This requirement is fulfilled by HTTP. This is why implementing a service-oriented architecture using Web Services technologies confirm a new way of building applications within a more powerful and flexible programming model (Channabasavaiah, Holley et al. 2003, Hashimi 2003, Lomow, Newcomer 2004, Vasudevan 2001).

Among prior standards CORBA is mostly compared with Web Service because, from technical perspective we can use CORBA for almost everything we can use Web Services for. But CORBA (as well other standards) didn't succeed widely because of vendor politics and for not defining a standard for interoperability (Jones 2005). The implication was that interoperability didn't matter if we had a standard interface. Web Services started with SOAP, which is an interoperability standard (Box, Ehnebuske et al. 2000). Even from human perspective Web Services are much easier to learn, and the missing features from CORBA don't matter as much as interoperability.

2.4. Creating Web Services

Various application server and programming languages can be used to create Web Services. Among application servers *Java 2 Platform Enterprise Edition (J2EE), Microsoft .NET, Apache Axis*; and among programming languages *Java, C++, C#, VB* are mostly used. Here we will demonstrate two widely used application servers and programming languages to create two test Web Services: *in programming language C# with .NET and in programming language Java with Apache AXIS.*

2.4.1. Creating Web Services in ASP.NET

In this section we will demonstrate how to create, test and deploy Web Services in ASP.NET. The examples used and information provided in this section are collected from various articles (Ferrara, MacDonald 2002, Peiris 2001, Strahl 2002).

- **Platform Requirements:**
 - Windows 2000, Windows XP Pro, Windows Server 2003.
 - Internet Information Services (IIS) 5 or later.
 - SQL Server 2000 or MSDE.
 - .NET Framework Distributable 1.0 SP2 or later installed, with ASP.NET functionality tested.
 - A recent version of Web browser (Internet Explorer 5.5 or later preferred).
 - Visual Studio .NET 2003 or 2002.
 - If running Visual Studio .NET 2002, the Visual Studio .NET Data Loss Fix installed.
 - A minimum of 512 MB RAM; 640 MB RAM or higher is strongly preferred.
- **Creating A Sample Web Service:** Though Visual Studio .NET provides a feature-rich integrated development environment for .NET development, it's possible to create Web Services using any text editor or the command-line tool provided with the .NET Framework SDK. Here for simplicity we will use Notepad. No matter what editor we use, the file extension has to be `.asmx` and must be placed in an Internet Information Service (IIS) folder on a server or workstation that has the .NET Framework installed.

After saving the code to a folder served by the IIS Web server it's immediately becomes ready to run. To get the file to our Web server, if we are running IIS locally on our workstation (we need the .NET Framework to be installed locally) we have to save the file to a suitable location on our local drive (e.g., `c:\inetpub\wwwroot\`). While using a remote server (in this case we do not need the .NET Framework to be installed locally), we might have to use FTP or a network share instead.

Example 2.1 lists the code for a C# version of a test application that delivers its message over the Web through an exposed method called `Test()`. To identify the class and method as a Web Service to the compiler, this code uses some special notation. It also includes an ASP.NET directive at the head of the file. To create a sample test Web Service in C# we have to enter the code from Example 2.1 exactly as it appears, and save the file to our web server under the web root folder for our system (in our case `c:\inetpub\wwwroot\`) with the name `firstwebservice.asmx`.

Example 2.1: A Sample C# Web Service

```
<%@ WebService Language="C#" Class="FirstWeb Service" %>
using System;
```

```

using System.Web;
using System.Web.Services;
public class FirstWebService
{
    [WebMethod]
    public string Test()
    {
        return "Test Service!";
    }
}

```

Example 2.1 begins with a `WebService` directive, an ASP.NET statement declaring that the code that follows is a Web Service:

```
<%@ WebService Language="C#" Class="FirstWebService" %>
```

To make Test Web Service work we must assign values to two `WebService` directives attributes: `Language` and `Class`. The required `Language` attribute lets .NET know which programming language the class has been written in. The acceptable values for the language attribute are currently C#, VB, and JS for JScript.NET. The `Class` attribute which is also required, tells ASP.NET the name of the class to expose as a Web Service, because a Web Service application can comprise multiple classes, some of which may not be Web Services.

It's possible to use a `using` statement to tell the compiler to alias a particular namespace to the local namespace. For example, in C#, this directive is:

```
using System.Web.Services;
```

This directive allows us to refer to objects in the `System.Web.Services` namespace without having to fully qualify the request. This statement is optional. Namespaces can contain definitions for classes, interfaces, structs, enums, and delegates, as well as other namespaces.

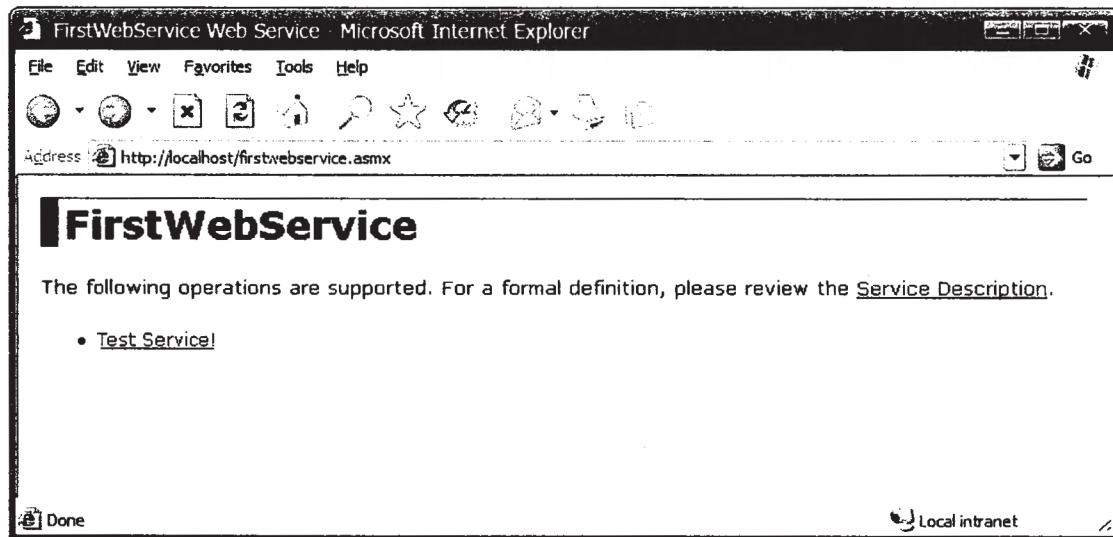


Figure 2.4: The ASP.NET C# Web Service status page.

- **Testing the Web Service:** To check out the Web Service all we can access the URL for it in a Web browser like this:
`http://localhost/firstwebservice.asmx`
 this will open the Web Service status page which will look something like Figure 2.4. It lets us see and test the methods that the Web Service exposes. We can also review and optionally capture the WSDL description for the service.

The runtime automatically creates a service description in WSDL which we can see by clicking the service description link. This page can also be viewed in a Web browser by appending `?WSDL` (e.g., `firstwebservice.asmx?WSDL`) to the page URL. The service description for our service is shown in Figure 2.5.

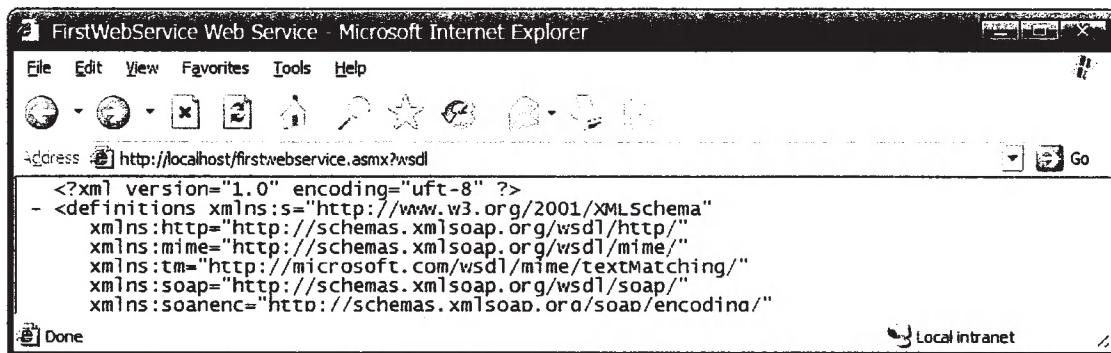


Figure 2.5: A fragment of the WSDL for ASP.NET C# service.

- **Visual Studio .NET and IIS:** It's preferred to use Microsoft's Visual Studio .NET (VS.NET) environment to create ASP.NET Web Services (Ferrara, MacDonald 2002). It provides many features to support creating complex Web Services. With Visual Studio .NET we can get a Web Service up and running real fast. The deployment process is also easy as long as we properly configure Visual Studio .NET to be able to deploy to our instance of IIS. It's also possible to deploy a Web Service directly to IIS (Ferrara, MacDonald 2002).

2.4.2. Creating Web Services in Apache Axis

In this section we will demonstrate how to *create*, *test* and *deploy* Web Services in Apache Axis. The examples used and information provided in this section are collected from various articles (Almaer 2002, Apache Axis 2005, Gibbs, Goodman et al. 2003, Hansen).

- **Platform Requirements:**
 - An application server up (full distribution of Jakarta Tomcat⁵ version 4.1.x is recommended) running on the localhost at port 8080.

⁵ <http://tomcat.apache.org>

- Full installation of Apache AXIS (including supporting .jar files in the CLASS-PATH).
 - A recent version of Web browser (Internet Explorer or Firefox preferred).
- **Creating A Sample Web Service:** We can create Axis Web Services using any text editor. Here for simplicity we will use Notepad. No matter what editor we use, the file extension has to be .jws and must be placed in an Axis\WEB-INF subdirectory on a server or workstation that has the Apache Tomcat and Axis installed (in our case, C:\Program Files\Apache Software Foundation\Tomcat 5.5\webapps\axis). After saving the code to that directory it's immediately becomes ready to run.

Axis needs to be able to find an XML parser. If the application server or Java runtime does not make one visible to web applications, we have to download and add it. Java 1.4 includes the Crimson⁶ parser, so we can omit this stage, though the Axis team prefers Xerces⁷.

To add an XML parser, we have to acquire any JAXP 1.1 XML compliant parser. Apache recommend Xerces jars from the xml-xerces distribution. In case our JRE or app server doesn't have its own specific requirements, we have to add the parser's libraries to ...\axis\WEB-INF\lib. If using Xerces, we have to add xml-apis.jar and xercesImpl.jar to the AXIS-CLASS-PATH so that Axis can find the parser.

Example 2.2 lists the code for a Java version of a test application that delivers its message over the Web through an exposed method called Test(). Unlike .NET we don't need any special notation to identify the class and method as a Web Service to the compiler, that's what the code will appear as an ordinary Java source.

Example 2.2: A Sample Java Axis Web Service

```

public class FirstWebService
{
    public String Test()
    {
        return "Test Service!";
    }
}

```

To create a Java version of the test Web Service we have to enter the code from Example 2.2 exactly as it appears, and save the file to our Web server under the web-info folder for our system with the name firstwebservice.jws.

⁶ <http://xml.apache.org/crimson>

⁷ <http://xerces.apache.org/xerces-j>

- ❑ **Testing the Web Service:** To check out the Web Service we can access the URL for it in a Web browser like this:

`http://localhost:8080/axis/firstwebservice.jws`

this will open the Web Service status page that will look like Figure 2.6. It will let us see and test the methods that the Web Service exposes. We can also review and optionally capture the WSDL description for the service.

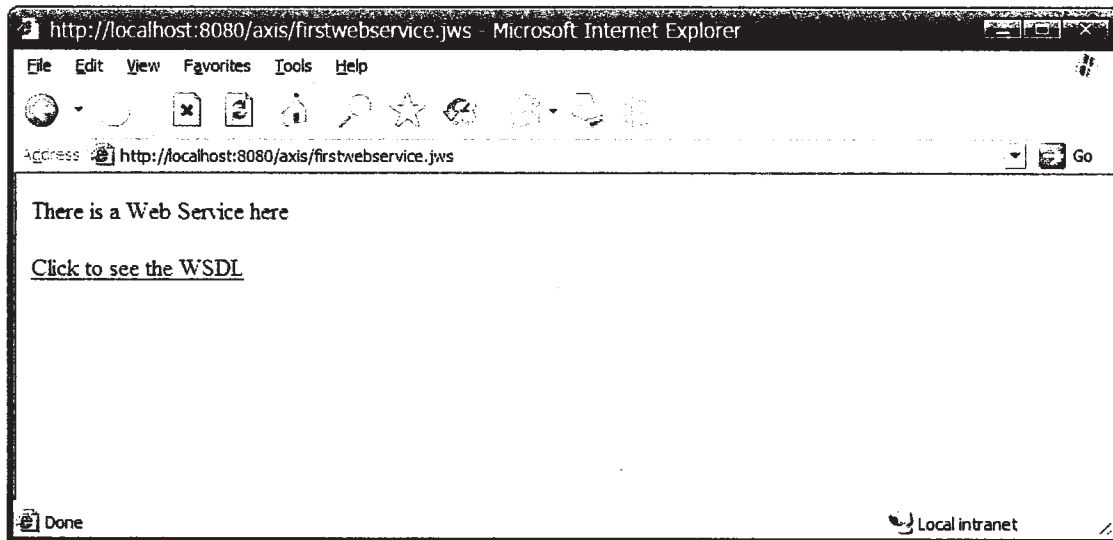


Figure 2.6: The Java Axis Web Service status page.

Like ASP.NET, Axis creates a service description in WSDL which on runtime that we can see by clicking the service description link. This page can also be viewed in a browser by appending `?WSDL` to the page URL, as in `firstwebservice.jws?WSDL`. The service description for java service will look the same as .NET Web Service's service description as shown in Figure 2.5.

2.4.3. Apache Axis Architectural Overview

In this section we'll give an overview of how the core of Axis works. The core of Axis can be divided into three sections (the information provided here is collected from the online Axis Architecture Guide⁸):

1. **Handlers and the Message Path in Axis:** When the central Axis processing logic runs, a series of Handlers are each invoked in order. The particular order is determined by two factors, deployment configuration and whether the engine is a client or a server. The object which is passed to each Handler invocation is a `MessageContext`. A `MessageContext` is a structure which contains several

⁸ <http://ws.apache.org/axis/java/architecture-guide.html>

important parts: 1) a request message, 2) a response message, and 3) a bag of properties.

There are two basic ways in which Axis is invoked:

- a) As a server, a Transport Listener will create a MessageContext and invoke the Axis processing framework.
- b) As a client, application code (usually aided by the client programming model of Axis) will generate a MessageContext and invoke the Axis processing framework.

In either case, the Axis framework's job is simply to pass the resulting MessageContext through the configured set of Handlers, each of which has an opportunity to do whatever it is designed to do with the MessageContext.

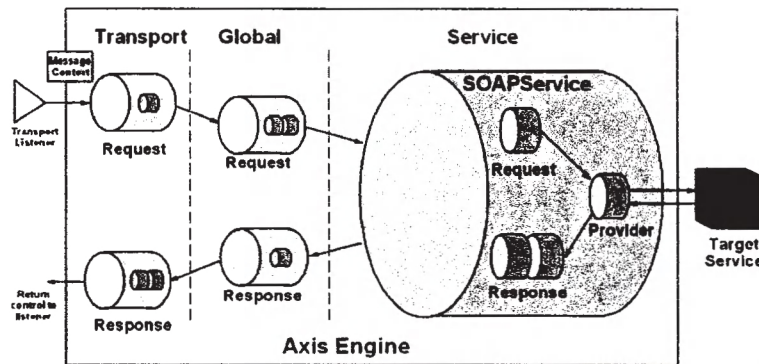


Figure 2.7: The server side message path of Axis (small cylinders represent Handlers and the larger, enclosing cylinders represent Chains).

2. **Message Path on the Server:** Figure 2.7 illustrates the server side message path. A message arrives at a Transport Listener. Here we are assuming that the Listener is a HTTP servlet. It's the Listener's job to package the protocol-specific data into a Message object (`org.apache.axis.Message`), and put the Message into a MessageContext. The MessageContext is also loaded with various properties by the Listener. In this example, the property `http.SOAPAction` would be set to the value of the SOAPAction HTTP header. The Transport Listener also sets the `transportName` String on the MessageContext, in this case to `http`. Once the MessageContext is ready to go, the Listener hands it to the AxisEngine.

The AxisEngine's first job is to look up the transport by name. The transport is an object which contains a request Chain, a Response Chain, or both. A Chain is a Handler consisting of a sequence of Handlers which are invoked in turn. If a transport request Chain exists, it will be invoked, passing the MessageContext into the `invoke()` method. This will result in calling all the Handlers specified in the request Chain configuration.

After the transport request Handler, the engine locates a global request Chain, if configured, and then invokes any Handlers specified therein.

At some point during the processing up until now, some Handler must have set the `serviceHandler` field of the `MessageContext` (this is usually done in the HTTP transport by the `URLMapper` Handler, which maps a URL like `http://localhost/axis/services/AdminService` to the `AdminService` service). This field determines the Handler going to be invoked to execute service-specific functionality, such as making an RPC call on a back-end object. Services in Axis are typically instances of the `SOAPService` class (`org.apache.axis.handlers.soap.SOAPService`), which may contain request and response Chains (similar to what we saw at the transport and global levels), and must contain a provider, which is simply a Handler responsible for implementing the actual back end logic of the service.

The provider is the `org.apache.axis.providers.java.RPCProvider` class for RPC-style requests. This is just another Handler that, when invoked, attempts to call a backend Java object whose class is determined by the `className` parameter specified at deployment time. It uses the SOAP RPC convention for determining the method to call, and makes sure the types of the incoming XML-encoded arguments match the types of the required parameters of the resulting method.

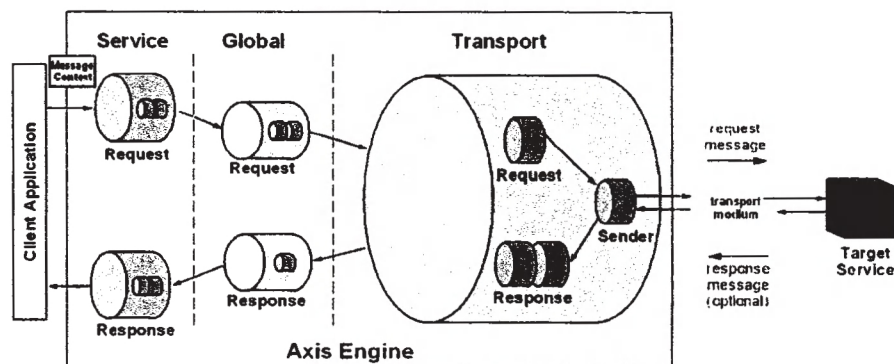


Figure 2.8: The client side message path of Axis (small cylinders represent Handlers and the larger, enclosing cylinders represent Chains).

3. **Message Path on the Client:** The Message Path on the client side is similar to that on the server side, except the order of scoping is reversed, as shown in Figure 2.8. The service Handler, if any, is called first. On the client side, there is no “provider” since the service is being provided by a remote node, but there is still the possibility of request and response Chains. The service request and response Chains perform any service-specific processing of the request message on its way out of the system, and also of the response message on its way back to the caller.

After the service request Chain, the global request Chain, if any, is invoked, followed by the transport. The Transport Sender, a special Handler whose job it is to actually perform whatever protocol-specific operations are necessary to get the message to and from the target SOAP server, is invoked to send the message. The response (if any) is placed into the responseMessage field of the MessageContext, and the MessageContext then propagates through the response Chains - first the transport, then the global, and finally the service.

2.5. Apache Axis or ASP.NET?

We have demonstrated two popular approaches to Web Services. But among those why we might want to use one method over others? From the issues we've discussed in the above sections and from the Chapter 1 we understood no matter what approach we use a Web Services must fulfill 4 basic requirements as follows:

1. **Service Description:** *Web Services must be described as collections of message-enabled endpoints or ports in WSDL. The abstract definition of endpoints and messages must be separated from their concrete deployment or bindings. The concrete protocol and data format specifications for a particular endpoint type must constitute a binding. An endpoint must be defined by associating a Web address with a binding, and a collection of endpoints defines a service.*
2. **Service Implementation:** *Implementing Web Services means structuring data and operations inside of an XML document that complies with the SOAP specification. Once a Web Service component is implemented, a client sends a message to the component as an XML document and the component sends an XML document back to the client as the response.*
3. **Service Publishing, Discovery and Binding:** *Once a Web Service has been implemented, it must be published somewhere that allows interested parties to find it. Information about how a client would connect to a Web Service and interact with it must also be exposed somewhere accessible to them. This connection and interaction information is referred to as binding information. Registries are currently the primary means to publish, discover, and bind Web Services. Registries contain the data structures and taxonomies used to describe Web Services and Web Service providers. A registry can either be hosted by private organizations or by neutral third parties.*
4. **Service Invocation and Execution:** *Web Service recipients must operate as SOAP listeners and notify interested parties when a Web Service request is received. The SOAP listener validates a SOAP message against corresponding XML schemas as defined in a WSDL file. The SOAP listener then un-marshals the SOAP message.*

Within the SOAP listener, message dispatchers can invoke the corresponding Web Service code implementation. Finally, business logic is invoked to get the reply. The result of the business logic is transformed into a SOAP response and returned to the Web Service caller

Both Axis and ASP.NET (along with many others) handle these challenges with great sophistication. The key advantage of using the ASP.NET approach to Web Services is that it has been designed for that purpose. On the other hand Axis is being retrofitted by the addition of further APIs. Another advantage of using Axis as a base for our system is that we have a much wider choice of vendor for our pre-built software, including numerous open source projects.

Table 2.1: ASP.NET versus Apache Axis

	Microsoft ASP.NET	Apache Axis
Basic challenges can be achieved	√	√
Has no preferred editor	X	√
Easy to install/Low system requirements	X	√
Free of cost	X	√
Programming language support	C#, VB, JS	Java, C++

If we consider these two platforms from a developer's point of view, Axis clearly beats ASP.NET. First of all, it takes a long time to make the system compatible with ASP.NET. In case we are working on an operating system other than Windows, things get even more complicated. Usually ASP.NET is used with Visual Studio.NET, which is a very heavyweight editor that takes huge amount of space and time to install. On the other hand, Axis is an open-source project, using and distributing it is open to all. Moreover it takes lesser time to understand and adopt Axis than other approaches. This is why in this thesis we have used Apache Axis to create our sample Web Services. Table 2.1 compares a few aspects of ASP.NET and Apache Axis from a developer's perspective.

CHAPTER 3

WEB SERVICE SEARCH VIA XML SCHEMA AND ONTOLOGY

3.1. Web Service Search

Discovery and composition of services are key steps to build Web applications. Web Services are usually published and searched in central registry of UDDI servers. To understand how UDDI is being used to discover services we have to understand how UDDI Registries contain information about businesses and the services they offer, that we have already discussed in Chapter 1, Section 1.2.3. Figure 3.1 shows the information or data model of UDDI.

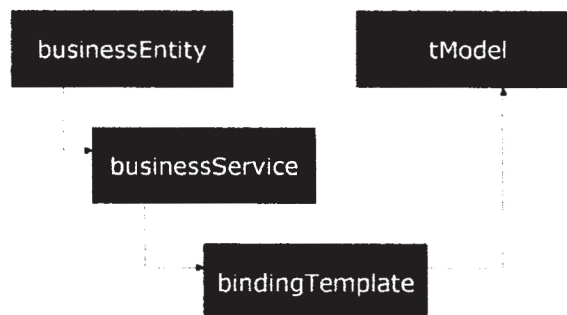


Figure 3.1: The UDDI Data Model.

When looking for a Web Service, a developer queries the UDDI registry, searching for a service offered by a business. From the `businessService` entry for the specific service, the developer can obtain the service access point and a pointer to the `tModel` that describes the service type. From the `tModel`, the developer can obtain the WSDL description describing the service interface. Using the access point and the WSDL description, the developer can construct a SOAP client interface that can communicate with the Web Service. Because of UDDI's dependency on taxonomy and `tModel` it is also through for the businesses to find each other and the services meet their needs (Xu Bin, Wang Yan et al. 2005).

This taxonomy and `tModel` based search also stand as a barrier in Web Services composition (Xu Bin, Wang Yan et al. 2005). Nowadays, composition of Web Services has received much interest to support B2B or enterprise application integration. Applications are to be assembled from a set of appropriate Web Services

and no longer be written manually. Seamless composition of Web Services has enormous potential in development distributed application. To composite Web Services into new application, it is often inside a domain to search related Web Services. For example, to develop an application about “books”, the developer should search Web Services like “ordering used books”, “preordering unreleased books” etc. Because of most of the Web Services’ searching are based on tModel of UDDI, it is not convenient to search Web Services in a *domain*. By domain we mean a specific area (for example, the “book” domain is about concepts of publishing). Instead the developer has to search through keywords like “book”, “comics”, so forth separately. Furthermore, Web Services which are not registered in UDDI servers can’t be discovered through tModel.

We try to overcome these issues with our approach where we use Web Service datasources’ schema matching and Web Service ontology merging to search for services as well as in related datasources. In this chapter we will discuss briefly the methods used for schema matching and ontology merging. We will also discuss how *Collaborative Filtering (CF)* algorithms are being used in our architecture to recommend or predict items (or next search keywords).

3.2. The Role of W3C XML Schema

One of the most closely watched developments within the XML community is the XML schema language for describing the legal structure, content, and constraints of XML documents (Roy, Ramanujan 2001). Schema language provides enhanced as well as more comprehensive and powerful features than a DTD, the traditional mechanism used to describe the structure and content of XML documents. The *W3C Schema Working Group*, which supervises the development of XML schemas, issued the language as a candidate recommendation on 24 October 2000 (Thompson, Beech et al. 2004).

The main features that W3C recommended XML Schema are as follows (Li, Miller 2005, Roy, Ramanujan 2001, Thompson, Beech et al. 2004):

- *Features for Reuse: XML schema supports inheritance, so we can create new schemas by deriving features from existing schemas. We can also override derived features when new ones are required. The XML schema language also provides for breaking a schema into separate components. We can then refer to appropriate predefined components in writing schemas.*

Inheritance enables efficient software reuse and help developers avoid building everything from scratch again and again. It significantly improves XML software development process, code maintainability, and programmer productivity.

- ❑ **Tight Integration with Namespaces:** Every XML Schema uses at least two namespaces - the target namespace and the XMLSchema namespace, with the exception of no-namespace schemas (out of scope of our discussion). The namespace plays an important role in the identification process. However, namespaces are also the source of much confusion in XML. Most of the problems during developing the XML Schema documents are related to namespaces in one way or another. The confusion, however, is related to namespace semantics as opposed to the syntax outlined by the specification.
- ❑ **User-defined Types:** One of the most powerful aspects of W3C XML Schema is that the language support for user-defined types, and more specifically for custom value/lexical spaces. There are two custom types: `simpleType` and `complexType`, which W3C XML Schema makes possible for users to define in the schema documents. In addition, W3C XML Schema defines a set of type characteristics. Instead of treating all XML data as just plain text, users can enforce formal syntax and semantics in XML documents.

In schemas, models are described in terms of *constraints*. A constraint defines what can appear in any given context. There are basically two kinds of constraints that we can give, *content model* constraints describe the order and sequence of elements and *datatype* constraints describe valid units of data.

For example, one of our Web Service schema describes a valid `<book>` with the content model constraint that it consist of `<title>`, `<author>`, `<coverType>`, `<genre>`, `<year>` and `<new>` elements. The contents of these elements can have further datatype constraint. The schema can also define how many times an element can appear. Example 3.1 shows fragment of one of our XML data source and the XML schema that has been used with that. Figure 3.2 shows the visual representation of the same schema.

Example 3.1: XML Schema

(a) Fragment of an XML data source

```
<?xml-stylesheet type="text/xsl"
  href="bookstore.xsl" version="1.0" encoding="UTF-8"?>
<bookstore xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="bookstore.xsd">
  <book no="1">
    <title>Deception Point</title>
    <author>Dan Brown</author>
    <coverType>Paperback</coverType>
    <genre>Thriller</genre>
    <year>2002</year>
    <new>7.19</new>
  </book>
  ...
```

</bookstore>

(b) XML Schema for the data source

```
<?xml version="1.0" encoding="utf-8"?>
<!-- edited with XMLSpy v2006 sp2 U (http://www.altova.com) by Ahmed Arif
(Lakehead University) -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="bookstore">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string"/>
              <xs:element name="author" type="xs:string"/>
              <xs:element name="coverType" type="xs:string"/>
              <xs:element name="genre" type="xs:string"/>
              <xs:element name="year" type="xs:integer"/>
              <xs:element name="new" type="xs:decimal"/>
            </xs:sequence>
            <xs:attribute name="no" type="xs:integer"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Every specific, individual document which doesn't violate any of the constraints of the model is, by definition, valid according to that schema.

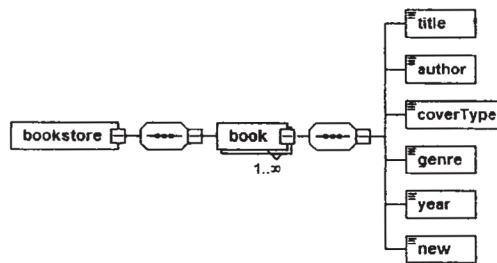


Figure 3.2: Visual representation of XML schema.

3.2.1. The Role of XML Schema in Our System

XML Schemas can be used for service discovery. As XML schema contains information about datasources (e.g., datatypes, etc.), matching schemas to understand the Web Services or Web Service datasources might prove useful, especially considering specific domains.

For example, let's consider some services like "OrderBook", "GetBooks", "OrderItem", all of these three services let's the user search and order fiction books. Using UDDI based searing strategy we might be able to discover all of these three services. But, what if the user wants to look only for the services those have "Harry Potter" or "Robert Langdon" series books available in their stock? Our schema matching strategy can solve this problem. Instead of "Book", "Fiction" like keywords the user can search with keywords like "Harry Potter (central character's name)", "Humayun Azad (author's name)" so forth.

Let's say the user is looking for book related Web Services with the "Humayun Azad" keyword. Before starting the search process the user needs to do the schema matching or ontology merging (any one of these strategies could be used; we will discuss ontologies in later sections). Both schema matching and ontology merging process creates a simple XML file, which we are calling the SearchHelper file. During the schema matching the system goes directly to the service datasource schema. By parsing the schema it can understand that in that particular Web Service datasource "author's name" is tagged as, let's say <my:writer> (this matching is done using a Dictionary, more on this will be discussed in next sections). The system stores these kinds of useful information in the SearchHelper file.

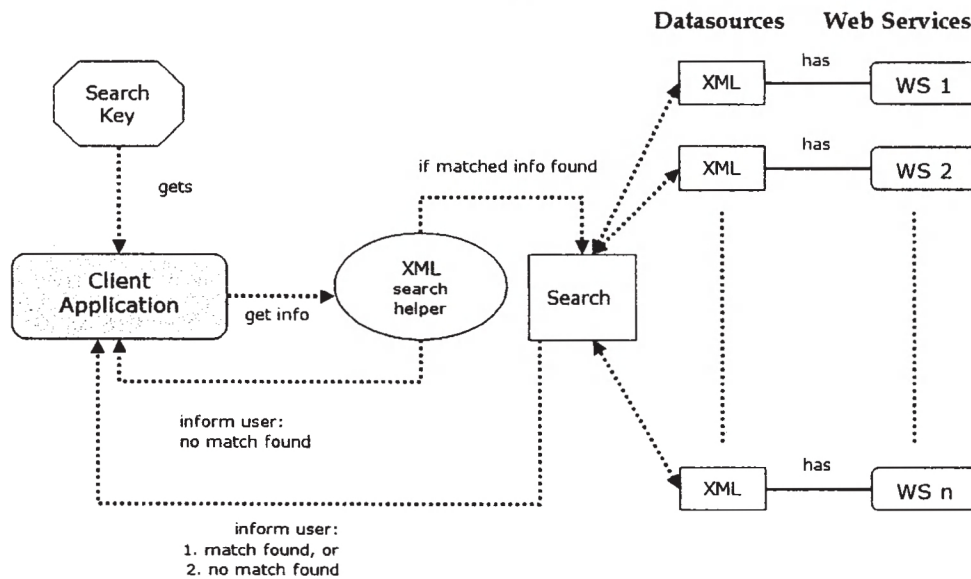


Figure 3.3: Search for Web Services with keys like "author's name".

This SearchHelper file is stored locally, so that the user doesn't have to go through the schema matching and ontology merging process before every search operation. He/she has to recreate the SearchHelper file, only when new services are register with the system. After the completing the SearchHelper creating process, the system will go directly to the SearchHelper file. By parsing it the system will remember - in that particular Web Service datasource "author's name" is tagged as <my:writer>.

Then it will go to the datasource to look if there is any author called “Humayun Azad” available; if yes, it will return the user that Web Service’s URI.

So we can say using schema matching and ontology merging strategy for Web Service discovery increases the possibility of the discovering only the services the searcher really interested in. Figure 3.3 illustrates this process.

The schema matching strategies will be discussed in depth in later sections.

3.3. The Role of Ontology

Ontologies specify a conceptualization of a domain in terms of *concepts*, *attributes*, and *relations* (Fensel 2003, Fiaidhi, Passi et al. 2004). They are a way of specifying the structure of domain knowledge in a formal logic designed for machine processing. The effect on IT is to shift the burden of capturing the meaning of data content from the procedural operations of algorithms and rules to the representation of the data itself (Denny 2004).

For example, let’s say we have a Web Service called “findFlat”. In some regions of the world (like Eastern Asia) “flat” means “apartment”. Now if that service doesn’t have ontology to describe that the task of it is to look for accommodation, the search application will have to rely on its own logic to understand that. If the application fails, the service will be undiscovered. Figure 3.4 illustrates this issue.

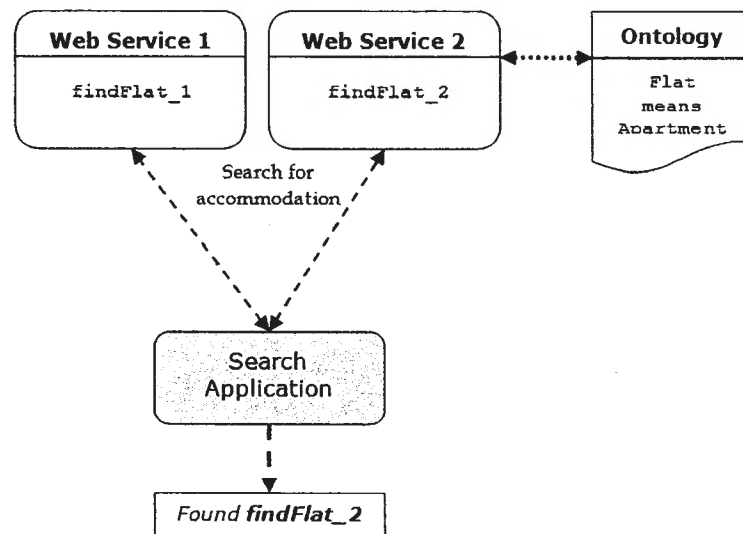


Figure 3.4: Role of ontology in service discovery. Service description with ontology can increase the possibility of service discovery. Here the search application understood from the service ontology that flat means apartment.

Infusing even a little semantic quality into our data (like, residing in Web pages, Web Service, database tables, electronic documents etc.) can mean that data is more immediately, broadly, and profoundly usable by all applications (including ours) aware of the knowledge-representation scheme, the ontology (Denny 2004, Fiaidhi, Passi et al. 2004).

Practical ontology languages are being adopted; the *Resource Description Framework (RDF)* and the *Web Ontology Language (OWL)* are recently recommended by the W3C for building Web ontologies (Denny 2004, Manola, Miller et al. 2005, Smith, Welty et al. 2004). These language specifications were developed over several years both within and outside of the organization.

3.3.1. W3C Resource Description Framework (RDF)

The *Resource Description Framework (RDF)* is the first W3C standard for enriching information resources of the Web with detailed descriptions (Manola, Miller et al. 2005). RDF provides a model for data, and syntax so that independent parties can exchange and use it. RDF was designed to provide a common way to describe information so it can be read and understood by computer applications. RDF descriptions are not designed to be displayed on the Web.

The RDF data model defines the structure of the RDF language. The data model consists of three data types (Bonstrom, Hinze et al. 2003, Manola, Miller et al. 2005):

- 1) **Resources:** All data objects described by a RDF statement are called resource. It can be anything that can have an URI, like Web Service.
- 2) **Properties:** A specific aspect, characteristic or relation of a resource is described by a property, like the creator of the service.
- 3) **Statements:** A statement combines a resource with its describing property and the value of the property, like the name of the creator. RDF statements are the structural building blocks of the language.

A RDF statement is typically expressed as “resource-property-value” triple, commonly written as $\mathcal{P}(\mathcal{R}, \mathcal{V})$ where a resource \mathcal{R} has a property \mathcal{P} with value \mathcal{V} . These triples can also be seen as object-attribute-value triple and as graphs with nodes for resources and values where directed edges represent the properties. Figure 3.4 shows the graph of the resource \mathcal{R} with an edge for the property \mathcal{P} directed to the property value \mathcal{V} .

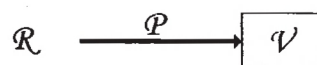


Figure 3.5: Graph representation of a $\mathcal{P}(\mathcal{R}, \mathcal{V})$ triple.

Resources are represented in the graph as circles. Properties are represented by directed arcs. Property-values are represented by a box. These values are called graph *endnodes*. Values can also become resources if they are described by further properties, i.e., if a value forms a resource in another triple. They are then represented by a circle. The Example 3.2 shows one of our Web Service descriptions and Figure 3.5 shows the triple representation for the metadata of the same description.

Example 3.2: Different Representation of a RDF Statement

This is the description created for one of our Web Services, URL, <http://localhost:8080/axis/MYWS1>.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:my="http://lakehead.fake/">
  <rdf:Description rdf:about="http://localhost:8080/axis/MYWS1">
    <my:xml my:is="bookstore.xml"/>
    <my:author my:is="author"/>
    <my:book my:is="title"/>
    <my:character my:is="character"/>
    <my:genre my:is="genre"/>
    <my:year my:is="year"/>
  </rdf:Description>
</rdf:RDF>
```

Triple: See Figure 3.6

Number	Subject	Predicate	Object
1	genid:ARP1192	http://lakehead.fake/is	"bookstore.xml"
2	http://localhost:8080/axis/MYWS1	http://lakehead.fake/xml	genid:ARP1192
3	genid:ARP1193	http://lakehead.fake/is	"author"
4	http://localhost:8080/axis/MYWS1	http://lakehead.fake/author	genid:ARP1193
5	genid:ARP1194	http://lakehead.fake/is	"title"
6	http://localhost:8080/axis/MYWS1	http://lakehead.fake/book	genid:ARP1194
7	genid:ARP1195	http://lakehead.fake/is	"character"
8	http://localhost:8080/axis/MYWS1	http://lakehead.fake/character	genid:ARP1195
9	genid:ARP1196	http://lakehead.fake/is	"genre"
10	http://localhost:8080/axis/MYWS1	http://lakehead.fake/genre	genid:ARP1196
11	genid:ARP1197	http://lakehead.fake/is	"year"
12	http://localhost:8080/axis/MYWS1	http://lakehead.fake/year	genid:ARP1197

Figure 3.6: Triple representation of a resource description.

3.3.2. W3C Web Ontology Language (OWL)

W3C *Web Ontology Language (OWL)* is a semantic markup language for publishing and sharing ontologies on the semantic Web, OWL is designed as an extension of RDF/S and is derived from the *DARPA Markup Language (DAML+OIL)* Web ontology language (zhihong, Mingtian 2003). OWL was designed to provide a

common way to process the content of web information. Like RDF, OWL is also not meant for to be read by humans.

RDF and OWL are not similar but eventually they are much of the same thing, only OWL is a stronger language with greater machine interpretability and it comes with a larger vocabulary and stronger syntax than RDF (McGuinness, Harmelen 2004). We can call OWL as an extension of RDF Schema, in the sense that OWL would use the RDF meaning of classes and properties and would add language primitives to support the richer expressiveness.

Unfortunately, the desire to simply extend RDF Schema clashes with the trade-off between expressive power and efficient reasoning. RDF Schema has some very powerful modelling primitives, such as the `rdfs:Class` (the class of all classes) and `rdf:Property` (the class of all properties); these primitives are very expressive, and will lead to uncontrollable computational properties if the logic is extended with the expressive primitives (Antoniou, Harmelen 2003).

3.3.3. Role of RDF Ontology in Our System

It is possible to replace XML schema with ontologies in our system to understand the datasource elements (please see Section 3.2.1). Ontologies have one advantage over schemas that is - it can describe the services too, though, not necessarily all services will or want to have ontologies. That's why in our system we kept both options; the searching can be done either with the help of schema matching or ontology merging.

Let's consider the same example we used in section 3.2.1, a user searches for services with keyword like author's name, "Humayun Azad". If he/she wants to do the search through ontologies, then exactly like the schema matching, before starting the search he/she needs to go through the ontology merging process. Like schema matching, ontology merging process also creates the SearchHelper file. During the ontology merging the system goes directly to the service ontology and by parsing the ontology it realizes - in that particular Web Service datasource "author's name" is tagged as `<my:writer>`. The system stores these kinds of useful information in the SearchHelper file.

We already mentioned in Section 3.2.1 that, the SearchHelper file is stored locally, so that the user doesn't have to go through the schema matching and ontology merging process before every search operation. But, he/she has to recreate the SearchHelper file, when new services are registered with the system. After the user is done with creating the SearchHelper file, he/she can proceed with the search. During search the system will go directly to the SearchHelper file and by parsing it will remember in that particular Web Service datasource "author's name" is tagged as

<my:writer>. Then it will go to the datasource to look if there is any author called “Humayun Azad” available; if yes, it will return the user that Web Service’s URI.

For describing our Web Services and service datasources we used RDF. We have mentioned, RDF and OWL shares same syntax only that OWL is stronger language with greater machine interpretability and it comes with a larger vocabulary and stronger syntax. While describing a simple Web Service or Website it is better to use RDF as developing and handling RDF is easier than OWL. But no matter in what format our description is, converting RDF to OWL or OWL to RDF for simple ontologies can easily be done. Especially in our case, the ontologies are written in very simple logics that if we change a few lines in the source and change the extension in OWL it will become OWL ontology. If we change our ontologies from RDF to OWL our system will still work fine with a very minor change in coding.

For the convenience of service providers, we took the opportunity to create an application that will generate RDF descriptions for a service page by taking minimal user input. Figure 3.7 shows the RDF generation process with that tool.

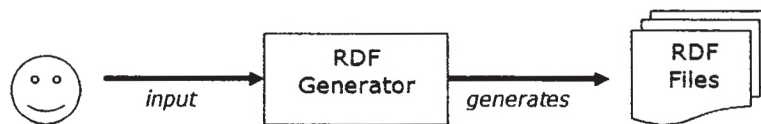


Figure 3.7: RDF generation with our application.

3.3.4. The RDF Ontology Syntax Supported by Our System

The RDF descriptions can be created without the use of this tool, as long as the providers follow the syntax, semantics our system supports. Expressed in shorthand form, the Description element must have the following structure (where “?” denotes zero or one occurrence; “+” denotes one or more occurrences; “*” denotes zero or more occurrences; and the empty element tag means the element must be empty):

```

<rdf:Description rdf:about+>+
  <my:xml my:is?/>
  <my:author my:is?/>
  <my:book my:is?/>
  <my:character my:is?/>
  <my:genre my:is?/>
  <my:year my:is?/>
</rdf:Description>
  
```

Here the Description element and the about attribute are from RDF vocabulary. xml, author, book, character, genre and year are mandatory elements from

our namespace `http://lakehead.fake/`. These elements provide the following data:

- ⇒ `xml`: Web Service datasource name (e.g., `name.xml`)
- ⇒ `author`: The database tag where author names are stored.
- ⇒ `book`: The database tag where book titles are stored.
- ⇒ `character`: The database tag where character names are stored.
- ⇒ `genre`: The database tag where book genres (e.g., `fiction`) are stored.
- ⇒ `year`: The database tag where publishing years are stored.

The `is` attribute is a mandatory attribute to be used with the above elements to describes the corresponded information (e.g., `author is writer`).

3.4. Schema-level Matching

A fundamental operation in the manipulation of schema information is *Match*, which takes two schemas as input and produces a mapping between elements of the two schemas that correspond semantically to each other (Doan, Domingos et al. 2003, Miller, Ioannidis et al. 1994, Milo, Zohar 1998, Sakamuri, Madria et al. 2003). Match plays a central role in numerous applications, in our case we will be using it for schema integration and ontology merging. In this section we will cover some existing approaches for schema-level matching, and will demonstrate how we have done our schema matching and ontology merging.

Most work on schema match has been motivated by schema integration that is, given a set of independently developed schemas; construct a global view (Batini, Lenzerini et al. 1986, Sheth, Larson 1990). In an artificial intelligence setting, this is the problem of integrating independently developed ontologies into a single ontology.

Since schemas are independently developed, they often have different structure and terminology. Thus, a first step in integrating the schemas is to *identify* and *characterize* these *inter-schema relationships*. This is a process of schema matching. Once they are identified, matching elements can be unified under a coherent, integrated schema or view. Sometimes during this integration, or as a separate step, programs or queries are created that permit translation of data from the original schemas into the integrated representation (Rahm, Bernstein 2001, Sakamuri, Madria et al. 2003).

3.4.1. The Match Operator

To define the match operator `Match` we need to choose a representation for its input schemas and output mapping. We define a mapping to be a set of “mapping elements”, each of which indicates that certain elements of schema *S1* are mapped to

certain elements in $S2$. Furthermore, each mapping element can have a “mapping expression” which specifies how the $S1$ and $S2$ elements are related. The mapping expression may be directional, for example, a certain function from the $S1$ elements referenced by the mapping element to the $S2$ elements referenced by the mapping element. Or it may be non-directional, that is, a relation between a combination of elements of $S1$ and $S2$. It may use simple relations over scalars (e.g., $=$, \leq), functions (e.g., addition or concatenation), ER-style relationships (e.g., is-a, part-of) or any other terms that are defined in the expression language being used.

We define the `Match` operation to be a function that takes two schema elements $S1.element$ and $S2.element$ as input and returns a mapping between those two elements as output, called the “match result”. Each mapping element of the match result specifies that certain elements of schema $S1$ logically correspond to certain elements of $S2$.

Unfortunately, the criteria used to match elements of $S1$ and $S2$ are based on heuristics that are not easily captured in a precise mathematical way that can guide us in the implementation of `Match` (Rahm, Bernstein 2001).

Table 3.1: Match on Schemas

$S1$ elements	$S2$ elements
<i>Book</i>	<i>Item1</i>
<i>BookName</i>	<i>Title</i>
<i>Author</i>	<i>Writer</i>
<i>Binding</i>	<i>Cover</i>
	<i>Page</i>

In this thesis we represent a mapping as a similarity relation \cong , where each pair in \cong represents one mapping element of the mapping. For example, the result of calling `Match` on the schemas of Table 3.1 could be $Book.BookName \cong Item1.Title$, $Book.Author \cong Item1.Writer$ and $\{Item1.Cover, Item1.Page\} \cong Book.Binding$. A complete specification of the result of the invocation of `Match` would also include the mapping expression of each element that is $Book.BookName = Item1.Title$, $Book.Author = Item1.Writer$ and $Concatenate(Item1.Cover, Item1.Page) = Book.Binding$ describes a mapping between two $S2$ elements and one $S1$ element. When mapping expressions are involved, we will explicitly mention them. Otherwise, we will simply use \cong .

Schema-level matchers only consider schema information, not instance data. The available information includes the usual properties of schema elements, such as name, description, data type, relationship types (part-of, is-a, etc.), constraints, and schema structure. In general, a matcher will find multiple match candidates. For each candidate, it is customary (but not mandatory) to estimate the degree of similarity by a normalized numeric value in the range 0-1, in order to identify the

best match candidates (Bergamaschi, Castano et al. 1999, Castano, De Antonellis 2001, Doan, Domingos et al. 2000).

3.4.2. Matching Strategy in Our System

There are two main alternatives for the “Granularity of Match”, *element-level* and *structure-level*. In element-level matching for each element of *S1*, system determines the matching elements in *S2*. In the simplest case, only elements at the finest level of granularity are considered, which we call the atomic level, such as attributes in an XML schema. In our system we considered this kind of granularity. But it’s not restricted to the atomic level, but may also be applied to coarser grained, higher level elements (Rahm, Bernstein 2001). For the schema fragments shown in Table 3.2, a sample atomic-level match is $MyBook.Author \cong MyItem1.Writer$.

Table 3.2: Structure-level Match

S1 elements	S2 elements	
<i>MyBook</i>	<i>MyItem1</i>	<i>Full structural match.</i>
<i>BookName</i>	<i>Title</i>	
<i>Author</i>	<i>Writer</i>	
<i>Binding</i>	<i>CoverType</i>	
<i>Book</i>	<i>Item1</i>	<i>Partial structural match.</i>
<i>BookName</i>	<i>Title</i>	
<i>Author</i>	<i>Writer</i>	
<i>Binding</i>	<i>Cover</i>	
	<i>Page</i>	

On the other hand, structure-level matching refers to matching combinations of elements that appear together in a structure. Arrange of cases is possible depending on how complete and precise a match of the structure is required. In the ideal case, all components of the structures in the two schemas fully match. Alternatively, only some of the components may be required to match (i.e., a partial structural match). Examples of the two cases are shown in Table 3.2. For more complex cases, the effectiveness of structure matching can be enhanced by considering known equivalence patterns, which may be kept in a library (Rahm, Bernstein 2001).

There could be 4 kinds of relations or “Match Cardinality” between elements, namely *1:1*, *1:n*, *n:1*, and *n:m*. Element-level matching is typically restricted to local cardinalities of *1:1*, *n:1*, and *1:n*. Obtaining *n:m* mapping elements usually requires considering the structural embedding of the schema elements and thus requires structure-level matching (Rahm, Bernstein 2001).

Table 3.3: Match Cardinality

	Local match cardinalities	S1 elements	S2 elements	Matching expression
1	1:1, element-level	BookName	Title	BookName = Title
2	n:1, element-level	Cover, Page	Binding	Binding = Cover, Page
3	1:n, element-level	Binding	Cover, Page	Cover, Page = Extract (Binding, ...)
4	n:1 structure-level n:m element-level	B.Title, B.PuNo, P.PuNo, P.Name	A.Book, A.Publisher	A.Book, A.Publisher = Select B.Title, P.Name From B, P Where B.PuNo=P.PuNo

Table 3.3 shows examples of the four local cardinality cases for individual mapping elements. In row 1 and 2, the match is *1:1* and *n:1*. Row 3 explains how *Cover* and *Page* are extracted from *Binding*, where row 4 uses a SQL expression combining attributes from two tables. It corresponds to an *n:m* relationship at the attribute level and an *n:1* relationship at the structure level. But this is out of our scope here, because in our system only the elements at the finest level of granularity are considered, we identified only *1:1* relationship cardinalities because successfully matching these means matching other relations can also be adopted (Doan, Madhavan et al. 2002, Sakamuri, Madria et al. 2003).

We identified the relations and did matching using a “Linguistic Approach”. Linguistic matchers use names and text (i.e., words or sentences) to find semantically similar schema elements. There are usually two types of linguistic approaches, *name matching* and *description matching*. We used name-matching in our system. Name-based matching matches schema elements with equal or similar names. Similarity of names can be defined and measured in various ways, including:

- **The Equality of Names:** An important subcase is the equality of names from the same XML namespace, since this ensures that the same names indeed bear the same semantics.
- **The Equality of Canonical Name:** Representations after stemming and other preprocessing. This is important to deal with special prefix/suffix symbols. One example of this kind of equality is *Title* → *BookName*.
- **The Equality of Synonyms:** As name implies, the equality of synonyms considers synonyms as equality, like *price* \cong *cost*.
- **The Equality Hypernyms:** Considers hypernyms as equalities. For example, *book is-a publication* and *article is-a publication* imply *book* \cong *publication*, *article* \cong *publication*, and *book* \cong *article*.
- **The Similarity of Names:** This is based on common substrings, edit distance, pronunciation and soundex (an encoding of names based on how they sound rather than how they are spelled), etc. For example *writtenBy* \cong *writer*.

- **User-provided Equality:** The user defined name matches, for example the user defined *Item1* is actually a book, $Item1 \cong book$.

Exploiting synonyms and hypernyms requires the use of thesauri or dictionaries (Rahm, Bernstein 2001). General natural language dictionaries, perhaps even multi-language dictionaries (e.g., English-Bengali) are being used to deal with input schemas of different languages. In addition, name matching can use domain or enterprise specific dictionaries and *is-a* taxonomies containing common names, synonyms and descriptions of schema elements, abbreviations, etc.

Name-based matching is possible for elements at different levels of granularity. Furthermore, it can be applied across levels, e.g., for a lower-level schema element to also consider the names of the schema elements it belongs to (e.g., to find that *author.name* \cong *AuthorName*). This is similar to context-based disambiguation of homonyms.

Though we used name-based matching for finding only *1:1* matches; it's not limited to *1:1* matches. It can identify multiple relevant matches for a given schema element. For example, it can match *Type* with both *cover type* and *binding type*. Name matching can also be driven by element-level matching.

Schema matching is typically semi-automatic, sometimes supported by a graphical user interface which is sometimes tedious, time consuming, error-prone, and therefore expensive process (Rahm, Bernstein 2001). In our system the schema matching is fully automatic.

3.4.3. An Approach to Ontology Merging

As we have discussed before, our system deals with ontologies in a very straightforward manner. It requires the service providers to create their ontologies according to the syntax our system supports (the syntax is provided in Section 3.3.4). The "RDF Ontology Generator" tool has also been developed to help this process. As all the ontologies are in same syntax, we simply "name (id)" each ontology with the Web Service they collaborate with. For example, suppose we have 3 Web Services MYWS1, MYWS2 and MYWS3. When we ask our system to do the ontology merging, it goes to each endpoint, parse the ontology files and merge them all together depending on the WS name (id):

```
...
<Web service id = 1 ontology1>
<Web service id = 2 ontology2>
...
```

Figure 3.8 illustrates this process.

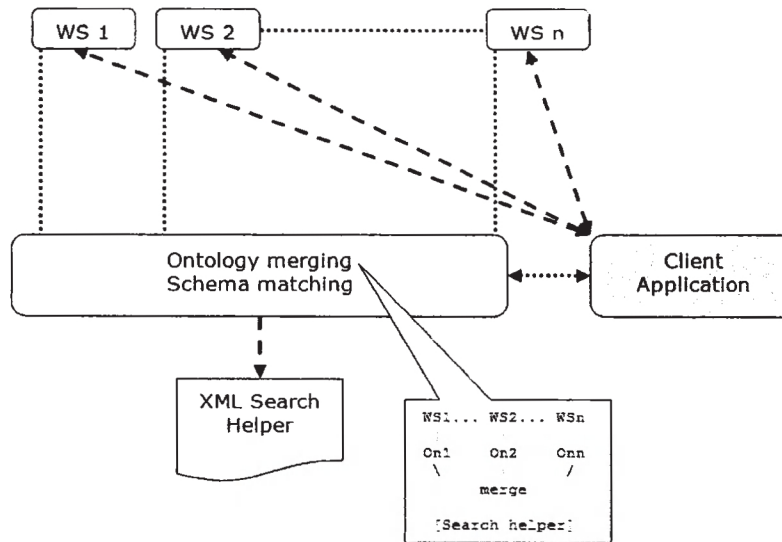


Figure 3.8: Ontology merging process in our system.

Later when a user looks for a Web Service or Web Service datasource with a key like “Robert Langdon (central character’s name)” the system goes to the merged description to look for that key. As the description contains possible linguistic 1:1 matches (e.g. $WS1.item1 \cong WS2.book$), the system recognizes similar items other than that key (e.g., *item1* in Web Service 1) and shows the result (see Figure 3.3).

3.5. The Use of Collaborative Filtering (CF)

The main idea of *collaborative filtering* is to automate the process of “word-of-mouth” by which people recommend products or services to one another (Breese, Heckerman et al. 1998, Heylighen 2001, Resnick, Iacovou et al. 1994, Shardanand, Maes 1995). When we need to choose between varieties of options with which we do not have any experience, we will often rely on the opinions of others who do have such experience. However, when there are thousands or millions of options, like in the Web, it becomes practically impossible for an individual to locate reliable experts that can give advice about each of the options.

By shifting from an individual to a collective method of recommendation, the problem becomes more manageable. Instead of asking opinions to each individual, we can determine an *average opinion* for the group. This, however, ignores user’s particular interests, which may be different from those of the *average person* (Heylighen 2001). Another way is to hear the opinions of those people who have similar interests, that is to say, a *division-of-labor* type of organization, where people only contribute to the domain they are specialized in.

There are many approaches to collaborative filtering. Usually collaborative filtering algorithms (CF-algorithms) use collection of user profiles to identify interesting information for these users. A particular user gets a recommendation based on the user profiles of other, similar users (Breese, Heckerman et al. 1998, Heylighen 2001, Wang, Vries, Arjen P. de et al. 2006). User profiles are commonly obtained by explicitly asking users to rate the items. Collaborative filtering has often been formulated as a self-contained problem, apart from the classic information retrieval problem (Wang, Vries, Arjen P. de et al. 2006).

In a typical CF scenario, there is a list of m users $U = \{u_1, u_2, \dots, u_m\}$ and a list of n items $I = \{i_1, i_2, \dots, i_n\}$. Each user u_i has a list of items I_{u_i} , which the user has expressed his/her opinion about. Opinion can be explicitly given by the user as a rating score, generally within a certain numerical scale, or can be implicitly derived from purchase records, by analyzing timing logs, by mining web hyperlinks and so on (Konstan, Miller et al. 1997, Terveen, Hill et al. 1997). Notable that $I_{u_i} \subseteq I$ and it is possible for I_{u_i} to be null-set. There exists a distinguished user $u_a \in U$ called the active user for whom the task of a collaborative filtering is to find an item likeness that can be of two forms.

- **Prediction:** That is a numerical value, $P_{a,j}$ expressing the predated likeness of item $i_j \notin I_{u_a}$ for the active user u_a . This predicted value is within the same scale (e.g., from 1 to 5, 1 to 10) as the opinion values provided by u_a .
- **Recommendation:** That is a list of N items, $I_r \subset I$, that the active user will like the most. Notable that, the recommended list must be on items already purchased by the active user, (i.e., $I_r \cap I_{u_a} = \Phi$) this interface of CF algorithms is also known as Top-N recommendation.

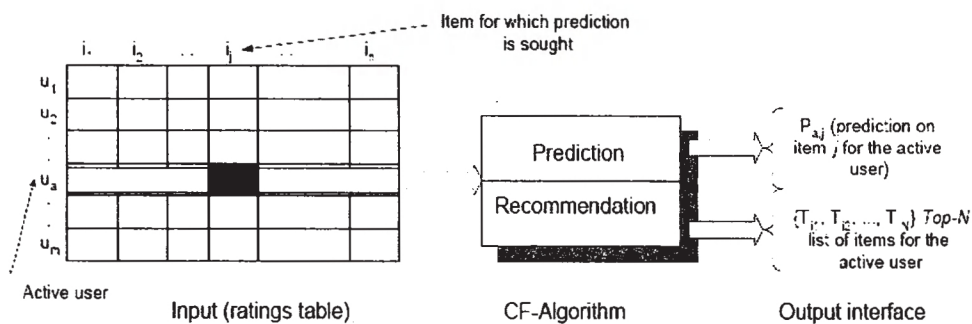


Figure 3.9: The collaborative filtering process.

Figure 3.9 shows the schematic diagram of the collaborative filtering process. CF algorithms represent the entire $m \times n$ user-item data as a ratings matrix, \mathcal{A} . Each entry $a_{i,j}$ in \mathcal{A} represent the preference score (ratings) of the i th user on the j th

item. Each individual rating is within a numerical scale and it can as well be 0 indicating that the user has not yet rated that item.

Researchers have devised a number of collaborative filtering algorithms that can be divided into two main categories: *User-based* (a.k.a., *memory-based*) and *Item-based* (a.k.a., *model-based*) algorithms. In this section we provide a brief idea about existing CF-based recommender system algorithms.

- ***User-based Collaborative Filtering Algorithms:*** *User-based or Memory-based algorithms utilize the entire user-item database to generate a prediction (Breese, Heckerman et al. 1998). These systems employ statistical techniques to find a set of users, known as neighbors that have a history of agreeing with the target user (i.e., they either rate different items similarly or they tend to buy similar set of items). Once a neighbor of users is formed, these systems use different algorithms to combine the preferences of neighbors to produce a prediction or top-N recommendation for the active user. The techniques, also known as nearest-neighbor or user-based collaborative filtering are more popular and widely used in practice.*
- ***Item-based Collaborative Filtering Algorithms:*** *Item-based or Model-based collaborative filtering algorithms provide item recommendation by fast developing a model of user ratings (Breese, Heckerman et al. 1998). Algorithms in this category take a probabilistic approach and envision the collaborative filtering process as computing the expected value of a user prediction, given his/her ratings on other items. The model building process is performed by different machine learning algorithms such as Bayesian network, clustering, and rule-based approaches. The Bayesian network model formulates a probabilistic model for collaborative filtering problem (Breese, Heckerman et al. 1998). Clustering model treats collaborative filtering as a classification problem and works by clustering similar users in same class and estimating the probability that a particular user is in a particular class C , and from there computes the conditional probability of rating (Basu, Hirsh et al. 1998, Breese, Heckerman et al. 1998, Ungar, Foster 1998). The rule-based approach applies association rule discovery algorithms to find associations between co-purchased items and then generates item recommendation based on the strength of the association between items (Sarwar, Karypis et al. 2000).*

3.5.1. Collaborative Filtering in Our System

Our system uses item-based algorithm. Unlike the user-based collaborative filtering algorithm the item-based approach looks into the set of items the target user has rated and computes how similar they are to the target item i and then selects k most similar items $\{i_1, i_2, \dots, i_k\}$; at the same time their corresponding similarities $\{s_{i1}, s_{i2}, \dots, s_{ik}\}$ are also computed (Sarwar, Karypis et al. 2001). Once the most similar

items are found, the prediction is then computed by taking a weighted average of the target user's ratings on these similar items.

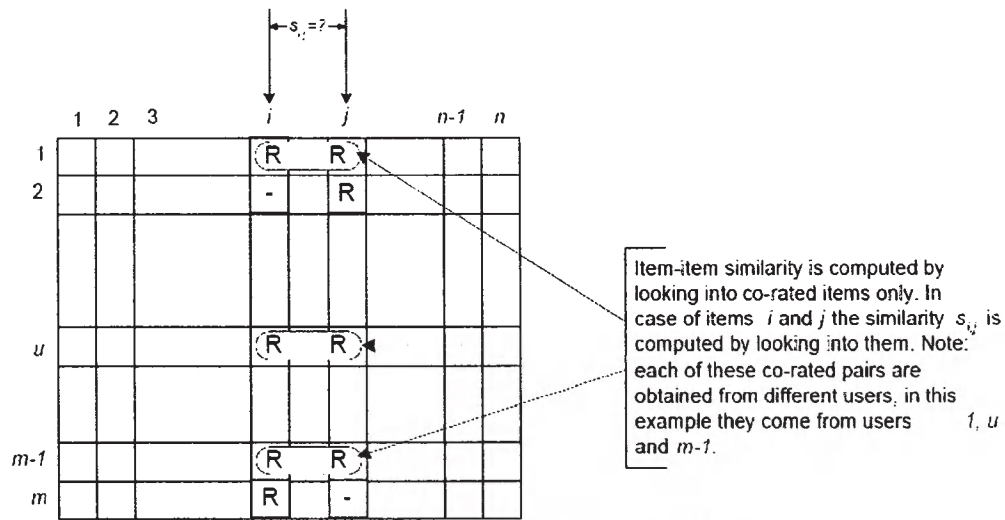


Figure 3.10: Isolation of co-rated items and similarity computation.

One critical step in the item-based collaborative filtering algorithm is to compute the similarity between items and then to select the most similar items; this process is known as “Similarity Computation”. The basic idea in similarity computation between two items i and j is to first isolate the users who have both of these items and then to apply a similarity computation technique to determine the similarity $s_{i,j}$ (Deshpande, Karypis 2004, Sarwar, Karypis et al. 2001). Figure 3.10 illustrates this process, here the matrix rows represent users and the columns represent items. There are many similarity computation algorithms. In our system we used the *correlation-based* similarity algorithm. In this algorithm, similarities between two items i and j is measured by computing the Pearson-r correlation $corr_{i,j}$. To make the correlation computation accurate we must first isolate co-rated cases (i.e., cases where the users rated both i and j) as shows in Figure 3.10. The set of users who both rated i and j are denoted by U then the correlation similarity is given by:

$$sim(i, j) = corr_{i,j} = \frac{\sum_{u \in U} (R_{u,i} - \bar{R}_i)(R_{u,j} - \bar{R}_j)}{\sqrt{\sum_{u \in U} (R_{u,i} - \bar{R}_i)^2} \sqrt{\sum_{u \in U} (R_{u,j} - \bar{R}_j)^2}}$$

Here $R_{u,i}$ denotes the rating of user u on item i , \bar{R}_i is the average rating of the i -th item.

Another important step in a collaborative filtering system is to generate the output interface in terms of prediction; this process is known as the “Prediction Computation”. Once we isolate the set of most similar items based on the similarity measures, the next step is to look into the target user's ratings and use a technique to obtain predictions (Sarwar, Karypis et al. 2001). There are many algorithms available for prediction computation; in our system we used the weighted sum algorithm. As

the name implies, this method computes the prediction on an item i for a user u by computing the sum of the rating given by the user on the items similar to i . Each ratings is weighted by the corresponding similarity $s_{i,j}$ between items i and j . We can denote the prediction $P_{u,i}$ as:

$$P_{u,i} = \frac{\sum_{\text{all similar items } N} (s_{i,N} * R_{u,N})}{\sum_{\text{all similar items } N} (|s_{i,N}|)}$$

Figure 3.11 illustrates the prediction generation process for five neighbors.

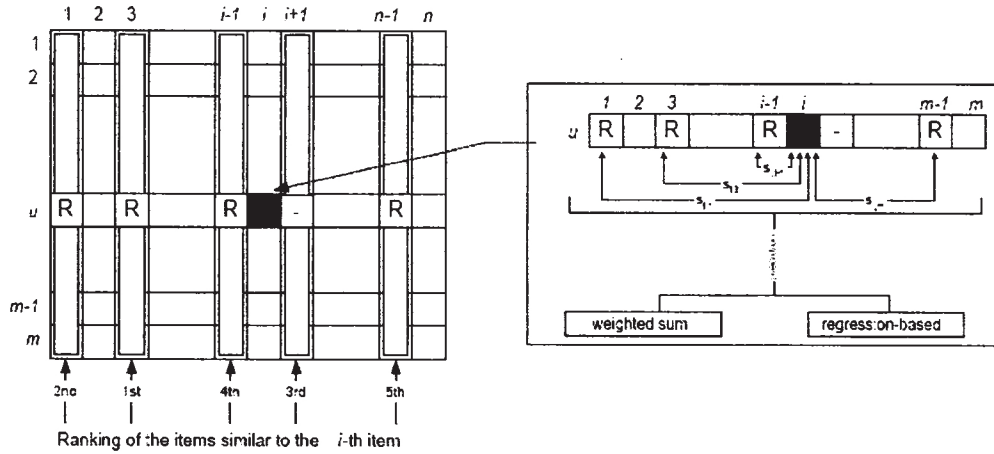


Figure 3.11: Item-based collaborative filtering algorithm. The prediction generation process is illustrated for 5 neighbors.

In our system when a user looks for a product (e.g., book title) it will try to find similar items (that has been generated manually using correlation-based). If there are similar items exists in the similarity database then it will start computing the prediction using weighted sum algorithm. We used weighted sum algorithm because it can be easily converted to regression model (Deshpande, Karypis 2004, Sarwar, Karypis et al. 2001). If the prediction computation indicates some items that could interest the user, our system will recommend those items along with the search result. Figure 3.12 illustrates this process.

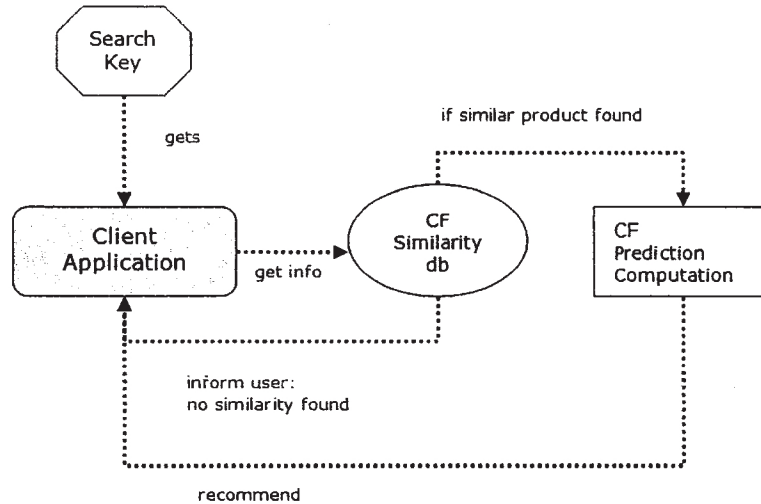


Figure 3.12: Item-based collaborative filtering in our system.

CHAPTER 4

SYSTEM PROTOTYPE IMPLEMENTATION

4.1. Web Service Implementation

In this section, the details of implementing multiple Web Services will be discussed where they share the same ontological similarities. All of our service location contains the following files:

1. **A Web Service (*.jws):** A Web Service, which has been created in Apache Axis⁹ (we've already discussed in Chapter 2, Section 2.4.2 how to create Web Services using Axis). To test our architecture we used three simple services (these services will be denoted as MYWS1, MYWS2 and MYWS3 from now on). All of these services perform the same functionality; they take the endpoint as input from the system and return the system either the ontology location or schema location, whatever the system asked for. The prototype is designed to search for book related data from family of Web Services (MYWS1, MYWS2, ..., MYWSn). The services contain data on books with varying structures and ontology. In real world, this service could be a "book bidding service", "book ordering service" or something more complex. Our system assumes all service locations have a service that returns the schema or ontology location when asked for.

Creating the service is straightforward. We create a call that typically is associated with WSDL. Then we set the target endpoint (provided by the system), operation name (e.g., `getSchemaURI()`, `getDBName()`), and request intent (either schema or ontology location). Then we use an overloaded `invoke` provided by Axis that deliver the return value.

2. **A Datasource File (*.xml):** This file contains item information, as we picked up "book" domain, in our case it would be book information (e.g., book title, author, etc.). The file could be in any semantic. The code fragment below show a portion of our bookstore datasource for MYWS1:

```
<?xml-stylesheet type="text/xsl" href="bookstore.xsl"
  version="1.0" encoding="UTF-8"?>
<bookstore xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="bookstore.xsd">
...
  <book no="3">
    <title>Digital Fortress</title>
    <author>Dan Brown</author>
```

⁹ <http://ws.apache.org/axis>

```

    <type>Paperback</type>
    <genre>Thriller</genre>
    <character>Susan Fletcher</character>
    <year>2003</year>
    <price>
      <new>6.59</new>
      <used>4.47</used>
    </price>
  </book>
  ...
</bookstore>

```

The first tag in the code fragment denotes the datasource uses a stylesheet named `bookstore.xml`; the second tag denotes it uses a schema named `bookstore.xsd`.

3. **An XSL Stylesheet (*.xsl):** The Extensible Stylesheet Language Family (XSL)¹⁰ is a family of recommendations for defining XML document transformation and presentation. We used it with our datasource to give it a presentable and user readable look. As this file is not a mandatory part of our system.
4. **An XML Schema File (*.xsd):** We've already discussed about what schema does, and how it plays an important role in Web Services and in our system (Chapter 3, Section 3.2.1). The schema can be in any semantics, the system don't require it to be in a specific format.
5. **A RDF Ontology File (*.rdf):** We've also discussed RDF ontologies and their standing in our architecture in Chapter 3, Section 3.3.3. There we mentioned that the service ontologies must follow the syntax our system supports (the syntax is showed in Chapter 3, Section 3.3.4), otherwise the system will be unable to recognize the file. In below we provide a fragment of the ontology we are using:

```

<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:my="http://lakehead.fake/">
  <rdf:Description rdf:about="http://localhost:8080/axis/MYWS1">
    <my:xml my:is="bookstore.xml"/>
    ...
    <my:genre my:is="genre"/>
    <my:year my:is="year"/>
  </rdf:Description>
</rdf:RDF>

```

The forth line of the ontology indicates the description is for the location `http://localhost:8080/axis/MYWS1`. The 5th line says the datasource this ontology represents is "bookstore.xml". Then it starts providing synonyms to help the Match process:

```

<RDF>
  <Description about="service-location">
    <database is="database name">
      <tag-in-database-for-genre is="synonyms">
        ...
      </Description>
    </RDF>

```

¹⁰ <http://www.w3.org/Style/XSL>

6. *An Index File (*.html):* We kept a simple HTML file for to show welcome message for the service location (e.g., `http://localhost:8080/axis/MYWS1`, will bring up the index file in any Web browser).
7. *Some Image Files (*.jpg, *.gif):* We also have two image files for decorating the HTML index file.

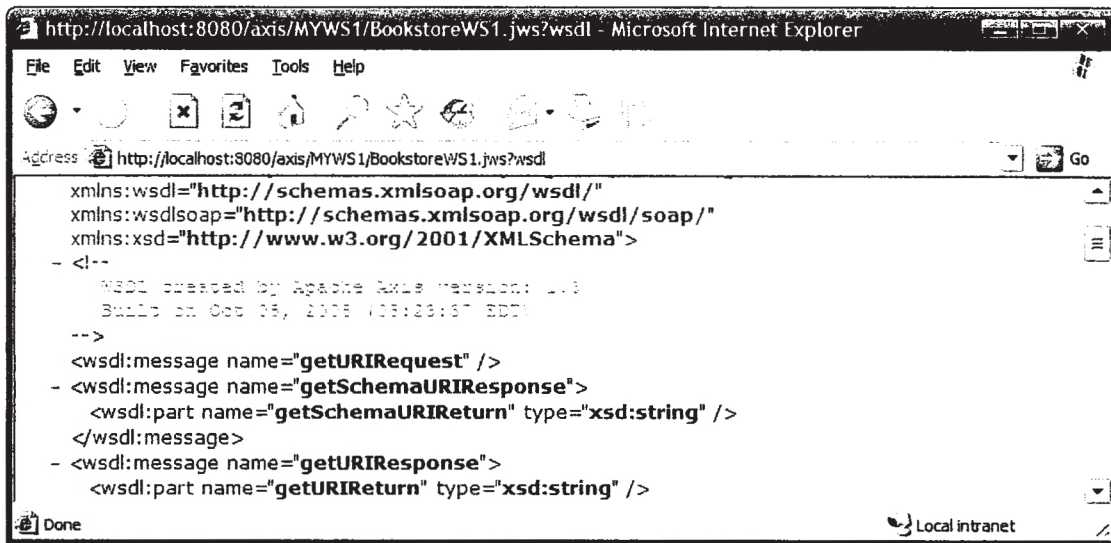


Figure 4.1: A fragment of the WSDL for our Web Service.

Figure 4.1 shows a fragment of the WSDL generated for our Web Service (how to get see WSDL of a service has been demonstrated in Chapter 2).

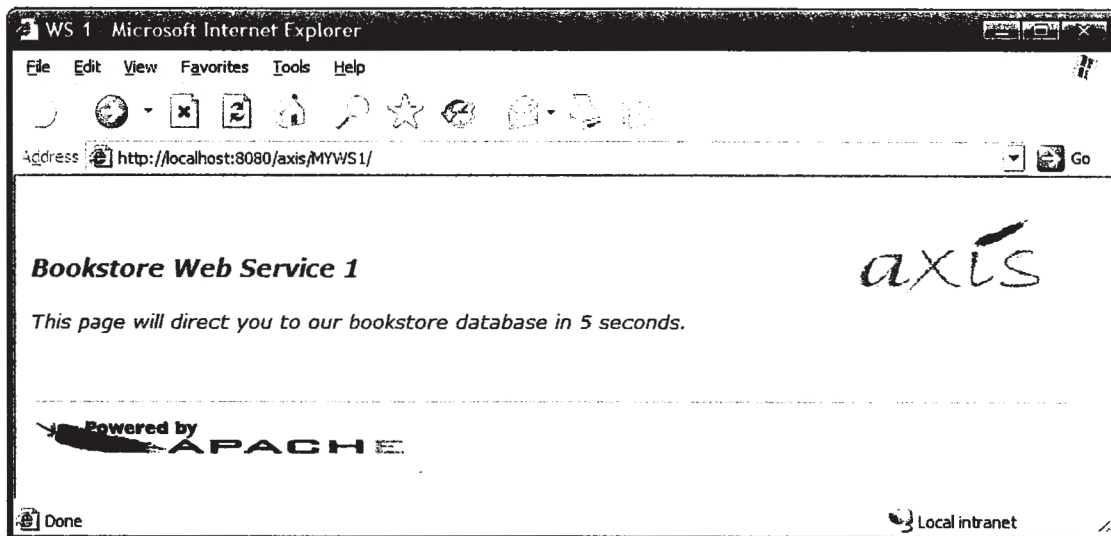
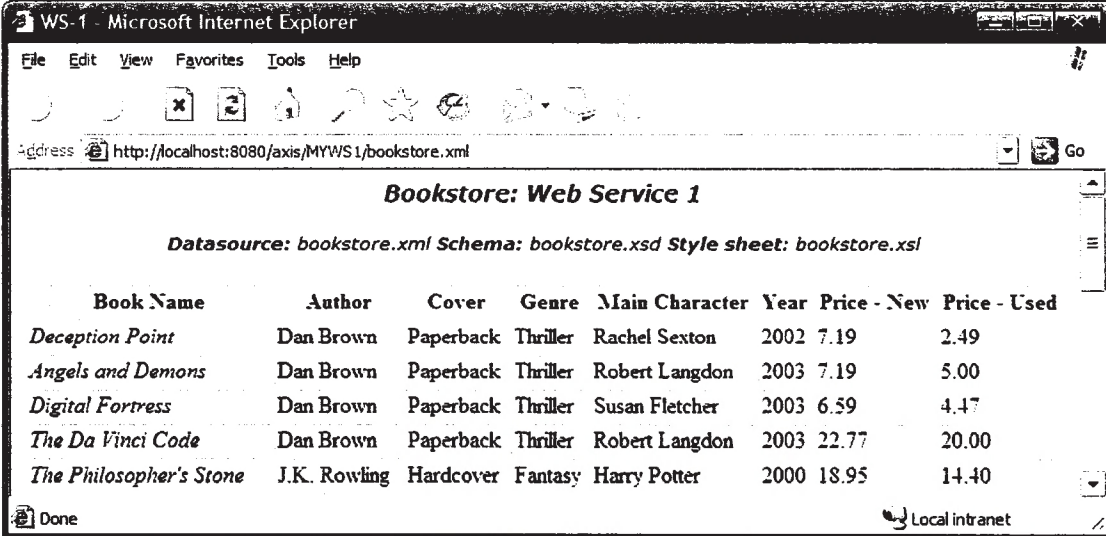


Figure 4.2: The start page for one of our Web Services. This page is written in HTML.

Figure 4.2 shows the welcome page of our Web Service (combination of HTML and image files).



Bookstore: Web Service 1
 Datasource: bookstore.xml Schema: bookstore.xsd Style sheet: bookstore.xsl

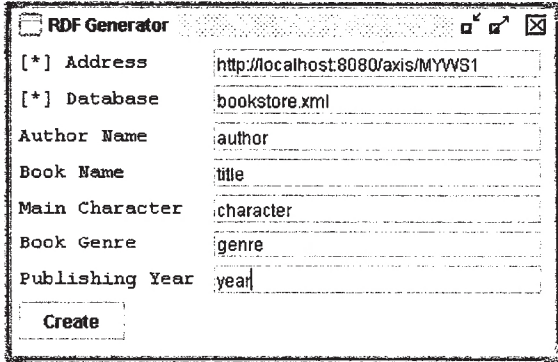
Book Name	Author	Cover	Genre	Main Character	Year	Price - New	Price - Used
<i>Deception Point</i>	Dan Brown	Paperback	Thriller	Rachel Sexton	2002	7.19	2.49
<i>Angels and Demons</i>	Dan Brown	Paperback	Thriller	Robert Langdon	2003	7.19	5.00
<i>Digital Fortress</i>	Dan Brown	Paperback	Thriller	Susan Fletcher	2003	6.59	4.47
<i>The Da Vinci Code</i>	Dan Brown	Paperback	Thriller	Robert Langdon	2003	22.77	20.00
<i>The Philosopher's Stone</i>	J.K. Rowling	Hardcover	Fantasy	Harry Potter	2000	18.95	14.40

Figure 4.3: A portion of one of our Web Service datasources. This pleasant look is created with a stylesheet.

Figure 4.3 shows a portion of MYWS1 datasource that has been made presentable using a stylesheet.

4.1.1 The RDF Generator

As we've seen in Chapter 3, Section 3.3.4 our system requires the RDF ontology to follow a specific syntax. Generating a file following a specific semantic sometime is irritating and error prone. That's why we've provided a simple RDF generation tool (this tool is discussed in Chapter 3, Section 3.3.3) that will take inputs from user and will create the ontology by itself. We generated this tool considering our test case, to show that providing a simple tool like this is really helpful and effective. Figure 4.4 shows a screenshot of this tool.



RDF Generator

[*] Address:

[*] Database:

Author Name:

Book Name:

Main Character:

Book Genre:

Publishing Year:

Figure 4.4: RDFGenerator tool.

In this tool the provider will have to insert the Web Service URI in the “Address” field and the database name in the “Database” field. These two fields are mandatory. Other than that, in the “Author Name” field he/she has to insert the tag name that represents author’s name in the service database. Similarly in the “Book Name” field the tag name that represents book title, in the “Main Character” field the tag name that represents the central character of a fiction book, in the “Book Genre” field tag name that represents the book type and finally in the “Publishing Year” field the tag name that represents the publishing year of a book in the service database. These fields can be empty if any of these information is/are not available in the service database.

4.2. Infoset Streaming

There are mainly two programming models for working with XML infosets, *document streaming* and the *Document Object Model (DOM)* (the java web services tutorial 2005).

The DOM model involves creating in-memory objects representing an entire document tree and the complete infoset state for an XML document. Once in memory, DOM trees can be navigated freely and parsed arbitrarily, and as such provide maximum flexibility for developers. However the cost of this flexibility is a potentially large memory footprint and significant processor requirements, as the entire representation of the document must be held in memory as objects for the duration of the document processing (the java web services tutorial 2005). This may not be an issue when working with small documents, but memory and processor requirements can escalate quickly with document size.

Streaming refers to a programming model in which XML infosets are transmitted and parsed serially at application runtime, often in real time, and often from dynamic sources whose contents are not precisely known beforehand. Moreover, stream-based parsers can start generating output immediately, and infoset elements can be discarded and garbage collected immediately after they are used (the java web services tutorial 2005).

While providing a smaller memory footprint, reduced processor requirements, and higher performance in certain situations, the primary trade-off with stream processing is that we can only see the infoset state at one location at a time in the document. The implication being that we need to know what processing we want to do before reading the XML document. Table 4.1 compares the common XML parser API features.

Table 4.1: XML Parsers

Feature	SAX	StAX	DOM	TrAX
API Type	<i>Push, streaming</i>	<i>Pull, streaming</i>	<i>In memory tree</i>	<i>XSLT rule</i>
Ease of Use	<i>Medium</i>	<i>High</i>	<i>High</i>	<i>Medium</i>
XPath Capability	<i>No</i>	<i>No</i>	<i>Yes</i>	<i>Yes</i>
CPU & Memory Efficiency	<i>Good</i>	<i>Good</i>	<i>Varies</i>	<i>Varies</i>
Forward Only	<i>Yes</i>	<i>Yes</i>	<i>No</i>	<i>No</i>
Read XML	<i>Yes</i>	<i>Yes</i>	<i>Yes</i>	<i>Yes</i>
Write XML	<i>No</i>	<i>Yes</i>	<i>Yes</i>	<i>Yes</i>
Create, Read, Update, Delete	<i>No</i>	<i>No</i>	<i>Yes</i>	<i>No</i>

Streaming models for XML processing are particularly useful when an application has strict memory limitations, as with a cell phone running J2ME, or when an application needs to simultaneously process several requests, as with an application server. In fact, it can be argued that the majority of XML business logic can benefit from stream processing, and does not require the in-memory maintenance of entire DOM trees (the java web services tutorial 2005). That's why we proffered streaming APIs over DOM to implement our prototype.

The Transformation API for XML (TRaX) API is a standard interface for Extensible Stylesheet Language Transformation (XSLT) engines. TRaX is not a good choice for us because it is designed to be used as a general-purpose transformation interface for XML documents. TRaX bridges various XML transformation methods (e.g., JDBC, JNDI, etc.) including SAX Events and XSLT Templates. TRaX relies upon a SAX2 and DOM-level-2-compliant XML parser and XSLT engine.

Most of our system is implemented in the *Simple API for XML (SAX)*¹¹. This is because, other than all the advantages discussed above, the performance of push parser like SAX is better considering anonymous XML files than pull parser like *The Streaming API for XML (StAX)*¹². We used StAX only to merge multiple files, because StAX can read multiple documents at one time with a single thread. More on this will be discussed in Section 4.3.1. We discuss the differences between push and pull parsing in the next section.

4.2.1. Pull Parsing Versus Push Parsing

Streaming pull parsing (e.g., StAX) refers to a programming model in which a client application calls methods on an XML parsing library when it needs to interact with

¹¹ <http://www.saxproject.org>

¹² <http://stax.codehaus.org>

an XML infoSet; that is, the client only gets (pulls) XML data when it explicitly asks for it (the java web services tutorial 2005).

Streaming push parsing (e.g., *Simple API for XML-SAX*) refers to a programming model in which an XML parser sends (pushes) XML data to the client as the parser encounters elements in an XML infoSet; that is, the parser sends the data whether or not the client is ready to use it at that time (the java web services tutorial 2005).

In our system we used both models, depending on which one solves our problem in a better way.

4.3. Web Service Search Prototype

Our Web Service searching and collaborative filtering prototype (a.k.a., *WSSearch*) is written in Java 2 Standard Edition with the help of SAX, StAX and Axis APIs. Figure 4.5 (a, b, c) shows three screenshots of three tabs of the WSSearch.

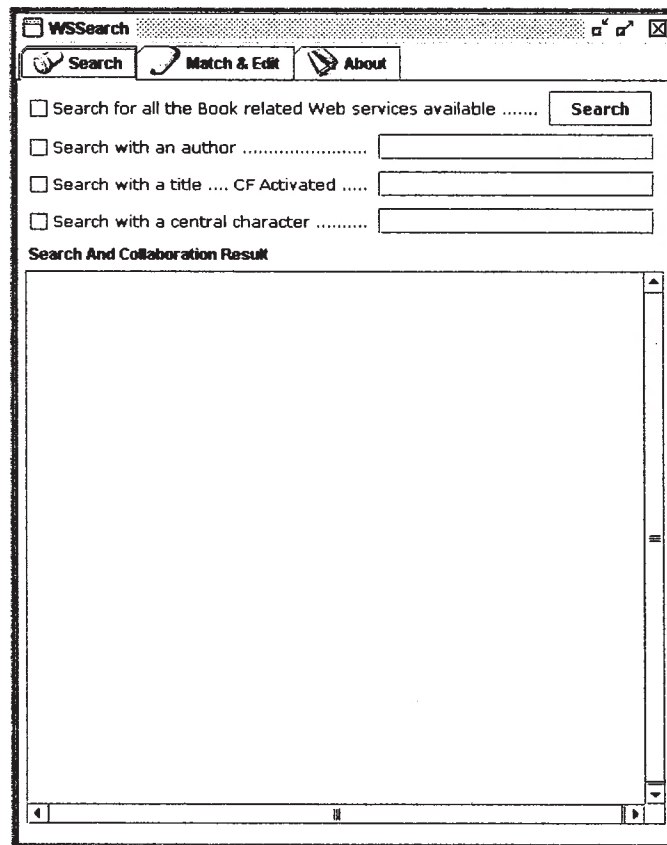


Figure 4.5 (a): Screenshot of the first tab of the WSSearch application. Users can search for Web Services in this tab.

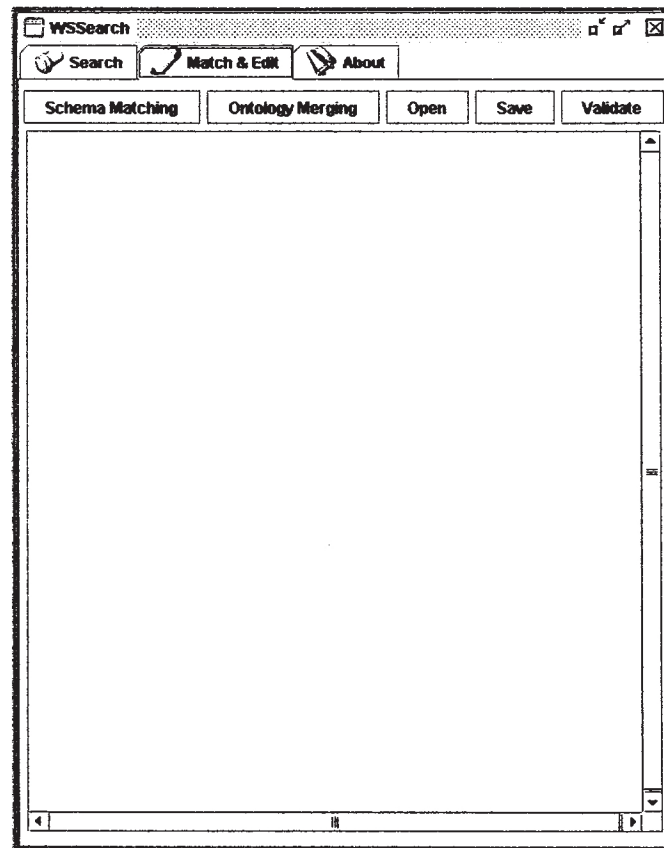


Figure 4.5 (b): Screenshot of the second tab. Users can generate SearchHelper by schema matching or ontology merging strategy from this tab. They can also edit and save XML and text type documents, and validate XML type documents in this tab.

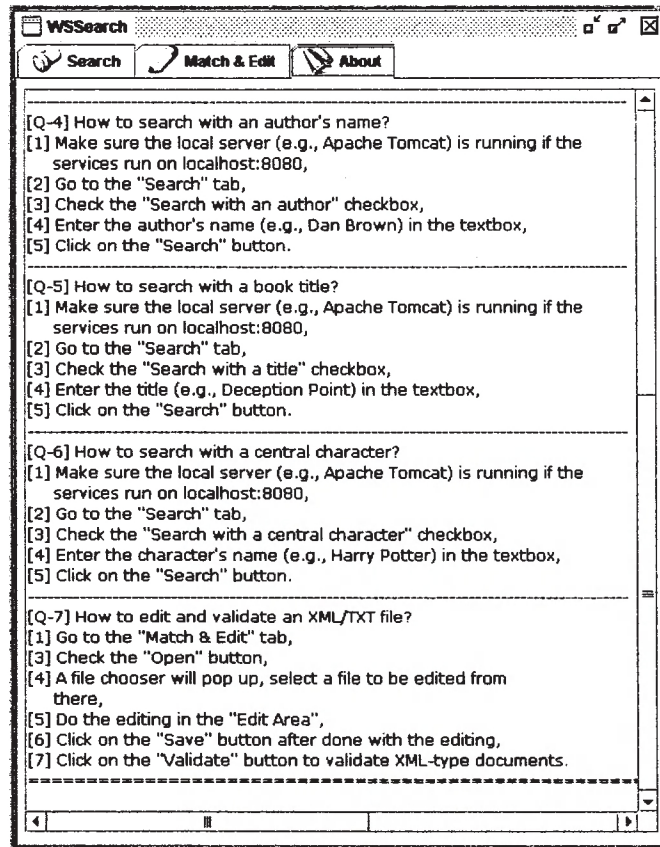


Figure 4.5 (c): Screenshot of the third tab, which shows the ReadMe file to help the user.

WSSearch is the combination of ten classes: WSSearch, BookstoreClient, RDFInformation, SchemaInformation, Merger, Search, DataCollector, KeySearch, CFRecommender and EditorClass. Figure 4.6 shows the *Unified Modeling Language* (UML) class diagram of these classes.

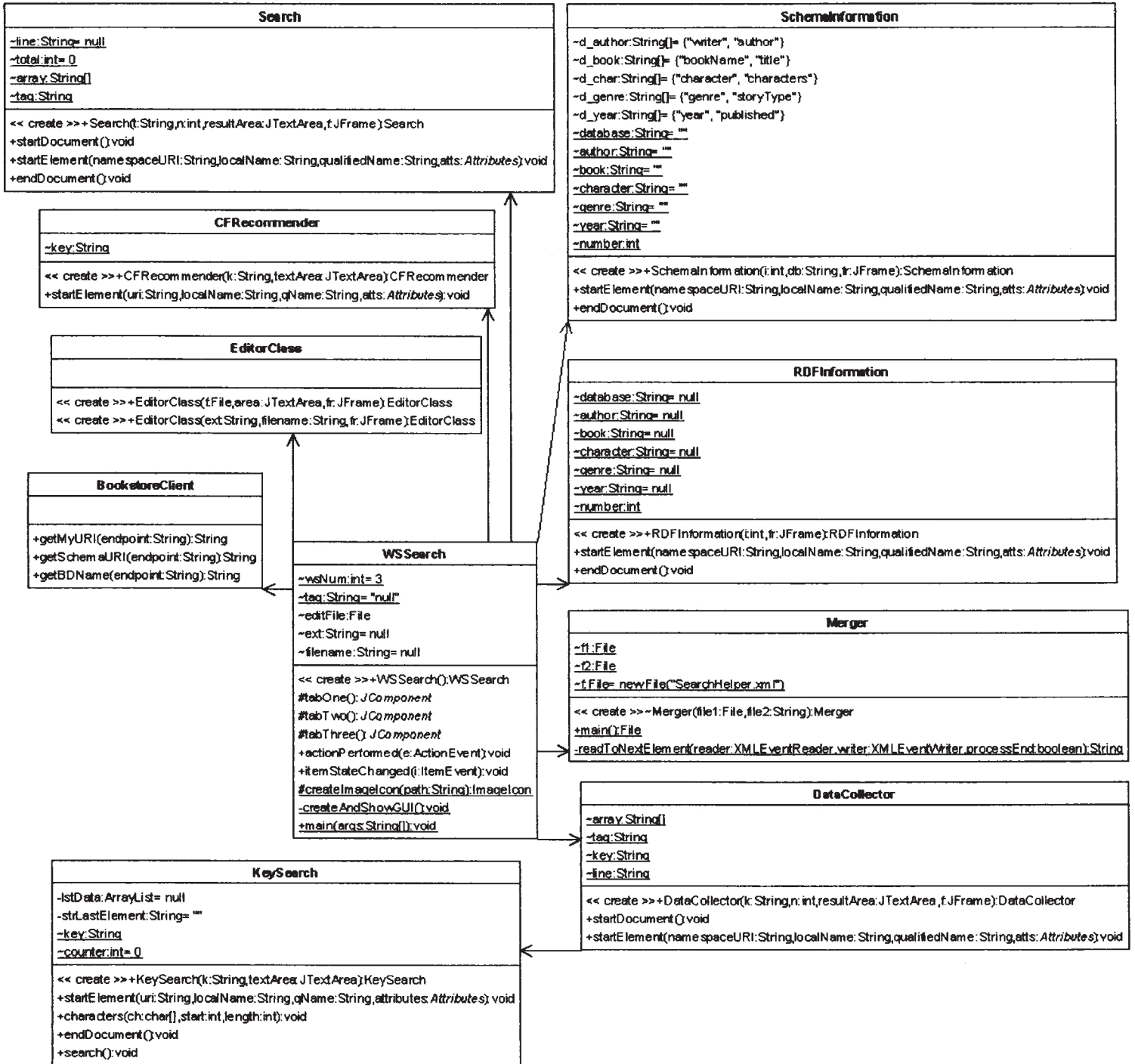


Figure 4.6: The class diagram for the WSearch prototype.

The responsibilities of these classes are as stated below:

- **WSSearch:** This is the top-level JFrame. It creates several objects and builds the graphical user interface.
- **BookstoreClient:** This class works as a Web Service client. It invokes Web Services and gets the Web Service datasource's file name, schema or ontology location as requested.
- **RDFInformation:** This class parses through RDF ontology files and creates an XML file for each in SearchHelper's syntax. These XML files will be referred as "Chunk/Chunks" in this thesis from now on.
- **SchemaInformation:** This class is responsible for schema matching. It parses through the XML Schema files and does the matching operation with the help of a build in dictionary (arrays of synonyms). The outcome of this class is the same as *RDFInformation*, it creates Chunks for each schema in SearchHelper's syntax.
- **Merger:** Takes all the Chunks and merge those together to create the SearchHelper.
- **Search:** This class is responsible for searching all book related Web Services.
- **DataCollector:** This class collects data from SearchHelper file to proceed with keyword searching and collaborative filtering.
- **KeySearch:** Responsible for keyword searching (search for all services those have author Humayun Azad's books in their database).
- **CFRecommender:** This class does the collaborative filtering prediction computation and recommends users related items.
- **EditorClass:** This class is responsible for editing, saving and validating XML type documents (e.g., XML, XML schema, RDF, etc.). This class also shows the ReadMe file to the users when requested.

Some other supporting files are also required to make this program fully functional. Like, we will need three *Portable Network Graphics (PNG)* images in the "images" folder. These three images are used as icons in the tabs (see Figure 4.5 (a, b, c)). We also need three *endpoints.txt*, *webservices.txt* and *readme.txt* text files in the "locations" folder. The *endpoints.txt* file works like a Web Service register; it stores one Web Service URIs in each line, on the other hand *webservices.txt* stores one service locations in line. Figure 4.7 shows the screenshots of these files. The *readme.txt* file contains instructions on how to use the prototype. This file will be loaded in the "About" tab (see Figure 4.5). Finally we need some XML files in the "similarity" folder. These XML files are supposed to be generated using the similarity computational algorithm. As we didn't put user login and product rating system in our program, these files are being created manually. These files are mandatory to produce recommendations.

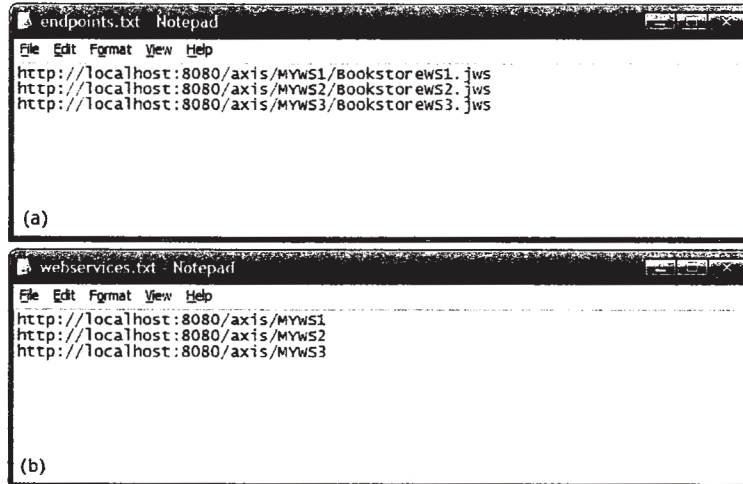


Figure 4.7: (a) Screenshot of the "endpoints.txt" file. Each line stores a Web Service URI. (b) Screenshot of the "webservices.txt" file. Each line stores a Web Service location.

4.3.1. Schema Matching and Ontology Merging

The schema matching can be done independently by clicking on the "Schema Matching" button located in the "Match & Edit" tab. When we click on the button the program starts the following loop:

- ⇒ *Takes a endpoint (Web Service location) from the "endpoints.txt";*
- ⇒ *Invoke the service by calling the `BookstoreClient` class;*
 - ⇒ *Get the schema location;*
- ⇒ *Parse the schema with the `SchemaInformation` class*
 - ⇒ *Recognizes the elements by using a dictionary;*
 - ⇒ *Create a Chunk file.*

This loop creates one Chunk file (see the class responsibility list) for each service using schema matching strategies. We've already discussed the matching strategies in detail in Chapter 3, Section 3.4.

After all Chunks have been created, the program calls the `Merger` class to merge those into one file (a.k.a., `SearchHelper`). The `Merger` class is written in StAX. This is because:

- ❑ With pull parsing, the client controls the application thread, and can call methods on the parser when needed. By contrast, with push processing, the parser controls the application thread, and the client can only accept invocations from the parser. As we know the exact format of the Chunk files, this function of the pull parsing will be beneficial.

- ❑ Pull parsing libraries can be much smaller and the client code to interact with those libraries much simpler than with pushes libraries.
- ❑ Pull clients can read multiple documents at one time with a single thread. This is the main reason of using StAX in the `Merger` class. In this class we will be parsing two Chunks at the same time.

The `Merger` class is being called in a loop that runs for each service location. In this class we used classical merging algorithm to merge the lists from Chunks. Depending on the comparison between the merge criteria from the Chunks, we either copy events from Chunk 1 to the `SearchHelper` or from Chunk 2 to the `SearchHelper`. We used some extra logic for detecting the end of the book list.

The ontology merging process (that could be started independently by clicking on the “Ontology Merging” button, located right beside the “Schema Matching” button) works almost the same as schema matching process. The only difference is from the `WSSearch` class we call `RDFInformation` class instead of `SchemaInformation`, that parses all the Web Service RDF file in loop and creates a Chunks all of them. All other processes works exactly same as before.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <Search>
  <book id="1" db="bookstore.xml" au="author" bk="title" cr="character" gn="genre" yr="year" />
  <book id="2" db="bookdatabase.xml" au="writer" bk="bookName" cr="" gn="" yr="published" />
  <book id="3" db="bookstore.xml" au="author" bk="title" cr="" gn="genre" yr="year" />
</Search>
```

Figure 4.8: The `SearchHelper` file.

A screenshot of a `SearchHelper` file is provided in Figure 4.8. In that file each `book` tag represents a book related Web Service. Attribute `id` tells us the Web Service’s id number and `db` tells us the Web Service datasource’s name. On the other hand `au` tells us under which tag author’s names are stored in that Web Service’s data source; like that `au` tells us about author’s name, `bk` book title, `cr` central character’s name, `gn` book type and `yr` publishing year. If any attribute value is null, that means that information is not provided in that Web Service datasource.

4.3.2. Searching for Web Services

After the `SearchHelper` file has been created by using schema matching and ontology merging method, we can start searching for services. The service search can be done in two ways:

1. **All Book Related Web Services:** Where the `WSSearch` program returns the user all the book related services available by parsing the `SearchHelper`. No user input is required for this search. Figure 4.9 (a) shows this kind of searching process.

2. *Filtered Web Services by Keywords (Search in Web Service Datasources):* In this search the user enters book related keywords like the author's name (e.g., Dan Brown), a book title (e.g., Angels and Demons) or the central character's name (e.g., Robert Langdon) and the program returns the services those have the keyword related items available in the service datasource. Collaborative filtering is enabled in this kind of search (to be more specific, in the search with a book title). Figure 4.9 (b) shows filtered searching process.

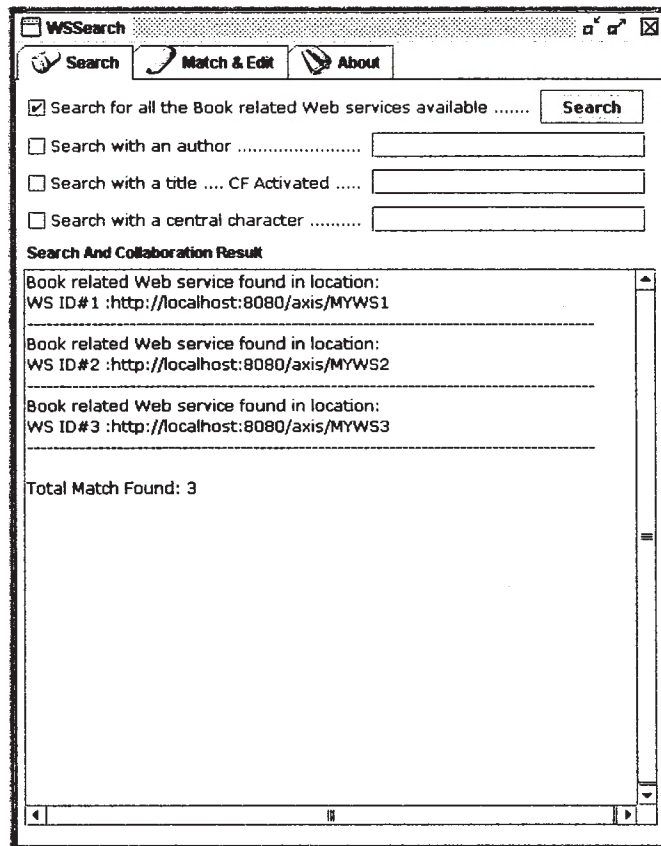


Figure 4.9 (a): Shows a screenshot of the WSSearch application after the user performed a search for all book related Web Services. The application found 3 such services.

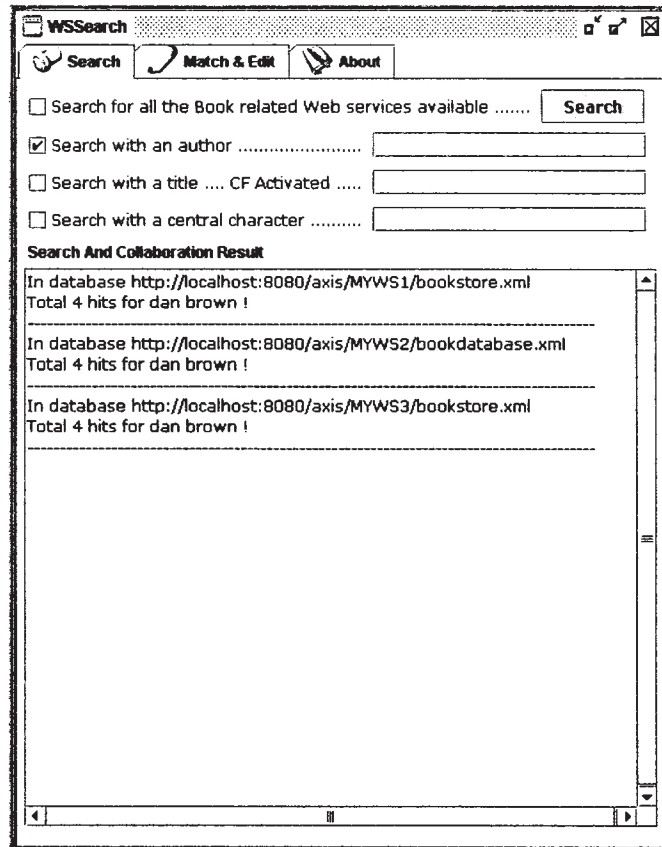


Figure 4.9 (b): Shows a screenshot of the same application after the user performed search with a keyword, author's name, "dan brown." The application found 4 matches match for this keyword in three Web Service datasources (in total 12 hits).

When the user checks the "Search for all Book related Web Services available" checkbox and clicks on the "Search" button (see Figure 4.9 (a)) the `WSSearch` class calls the `Search` class to parser the `SearchHelper` file.

The `Search` class works like an XML parser. It reads Web Service locations from the `webservices.txt` file and store all the Web Service locations in an array. We do that so that we can inform the user the exact line number of the `webservices.txt` where the service is recorded (or registered). Then we see if there is any tag named `book` is/are available, if yes, that means we have a book related Web Service. The `id` attribute in `SearchHelper` is same as the line number of the `webservices.txt` where the service is recorded (or registered). The class then prints all book related service location, id number (same as line number) and total number of matches found to the user. Figure 4.10 shows the UML sequence diagram for all book related Web Service search.

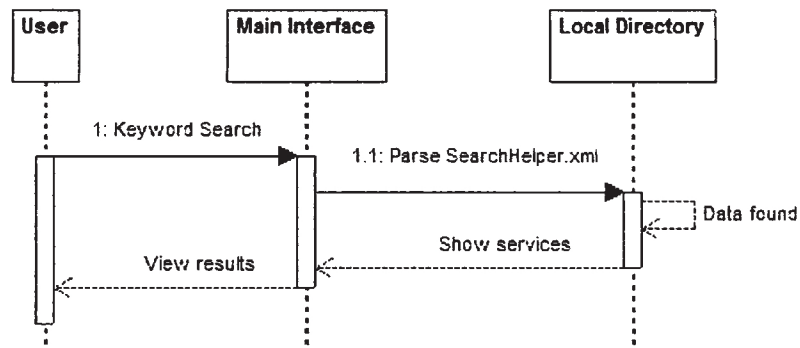


Figure 4.10: Sequence diagram for all book related Web Services search.

If the user is willing to search for Web Services with keywords, he/she has three options (see Figure 4.5). He/she can search services with an “author”, a “book title” or with a “central character”. When the user searches with a “book title” the collaborative filtering will start working. We will discuss the collaborative filtering in the next section. In this section we will just see how the keyword search is done in the prototype.

Let’s assume the user is searching for a Web Service with an author’s name, Dan Brown (see Figure 4.9 (b)). In that case the program will do the following:

- ⇒ Store the search keyword in a String variable;
- ⇒ Pass the variable (along with other parameters) to the DataCollector class;
- ⇒ Start parsing SearchHelper file with the DataCollector class.

Figure 4.11 shows the UML sequence diagram for keyword search.

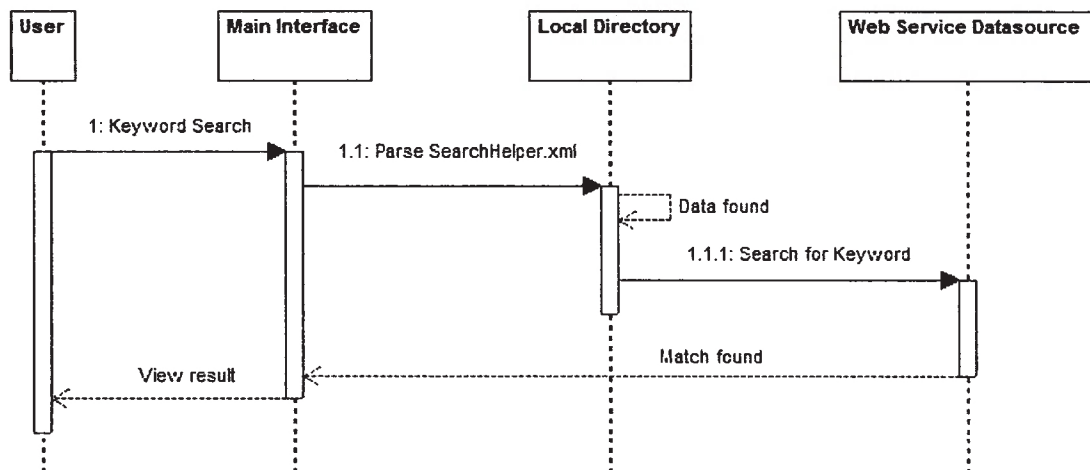


Figure 4.11: Sequence diagram for keyword search.

We have already seen the format of the SearchHelper file (Figure 4.8) and we know in this XML file each book tag represents a Web Service. Each book tag keeps the

Web Service id, database name and the tag names (those represents author name, book title so forth in the Web Service datasource) as its attributes. While the `DataCollector` class parsing the `SearchHelper`, it does the following:

- ⇒ *When the document starts, read the service locations and store those in an array;*
- ⇒ *For each start-tag, check if the tag name is `book` and if it has the tag value the user looking for (in this case author's name);*
 - ⇒ *If yes, then do the following:*
 - ⇒ *Get the id number;*
 - ⇒ *Get the database name;*
 - ⇒ *Start parsing the database with the `KeySearch` class for the keyword;*
 - ⇒ *If keyword found then print the matches for the user.*
 - ⇒ *If not, then inform user.*
 - ⇒ *If no, then do the following:*
 - ⇒ *Skip to the next tag.*

In the `KeySearch` class we used a new method to get the `pdata` from the XML datasource. For each start-tag the `KeySearch` class gets a tag and its corresponded `pdata` and stored these two as a pair (`tag:pdata`) in an `ArrayList`, unless there is no corresponded `pdata` available. In that case the class will skip to the next tag. After the parsing is done, the class searches for the keyword in the `ArrayList` and prints the result (*match found* or *no match found*).

Searching with a “book title” or a “central character” works exactly the same way. Only while searching with a “book title” the collaborative filtering gets enabled. We didn't use collaborative filtering with search with “author's name” or “central character” because the sample datasources we are using don't have many items. All of the three datasources contains 10 books from common authors and central characters. So in total we have just 3 unique authors and 3 central characters, which is not enough to proceed with collaborative filtering based recommendation. On the other hand we have 10 different book titles; that's somewhat okay to proceed with testing the prediction computation. The collaborative filtering implementation is discussed in detail in the next section.

4.3.3. Collaborative Filtering

When the user searches with a book title, `WSSearch` does everything as mentioned in section 4.3.2 and then it does the followings:

- ⇒ *See if there is a similarity computed record file exists in the “similarity” folder for the searched book title (Figure 4.12 shows a sample similarity computed record file for book “Angel and Demon” and Figure 4.14 shows the collaborative filtering algorithm based recommendations while the user searched with the same book title);*

- ⇒ If yes, then do the following for each *item* tag:
 - ⇒ Get the “title”, “author’s name” and the “similarity” (that is a number between 1 to 5);
 - ⇒ Start the prediction computation (the result will be a number between 1 to 5);
 - ⇒ If the prediction is greater than or equal to 3:
 - ⇒ Recommend the item.
 - ⇒ If the prediction is less than 3:
 - ⇒ Skip to the next tag.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <similar>
  <item title="Deception Point" author="Dan Brown" s="4" />
  <item title="Digital Fortress" author="Dan Brown" s="4" />
  <item title="The Da Vinci Code" author="Dan Brown" s="5" />
  <item title="The Broker" author="John Grisham" s="3" />
</similar>
```

Figure 4.12: Similarity computed record file. Here each item tag represents a similar item. Attribute title denotes book title; author the author’s name; and s the similarity.

Figure 4.13 shows the UML sequence diagram for the collaborative filtering process.

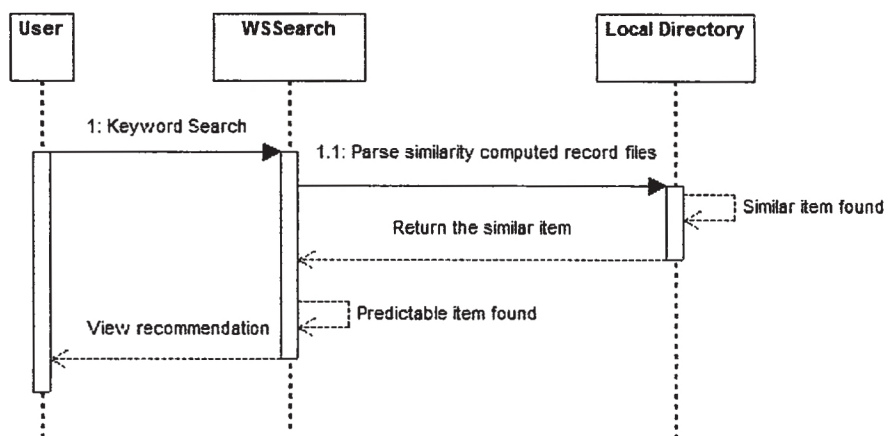


Figure 4.13: Sequence diagram for collaborative filtering (recommender).

In the `CFRecommender` class we assume that the item an active user is looking for is his/her favorite. So we calculate the prediction assuming the user rating for that specific item is 5/5 (5 out of 5).

Figure 4.11 shows the screenshot of the `WSSearch` application after the user searches with the book name keyword “Angels and Demons”.

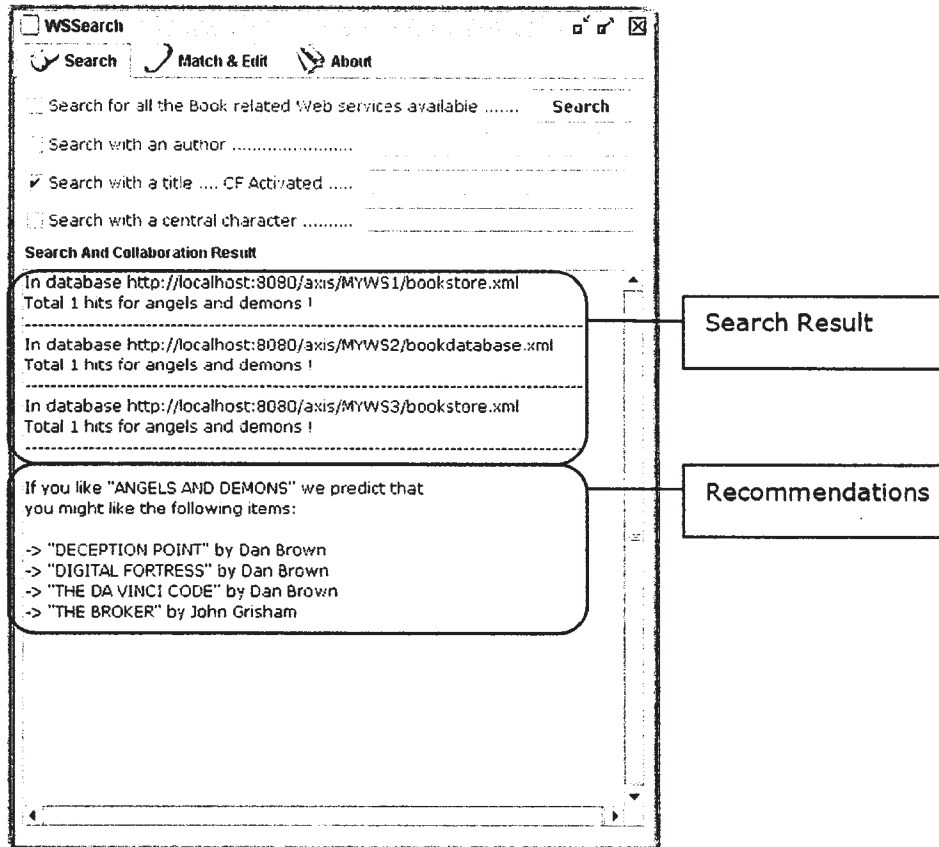


Figure 4.14: Recommender in WSSearch screenshot. This screenshot is taken after the user searched with the keyword, book title, "Angel's and Demons."

4.3.4. Editing and Validating XML-type Files

In the second tab, along with the "Schema Matching" and "Ontology Merging" buttons we have three more buttons called "Open", "Save" and "Validate". These three buttons are for editing and validating XML type documents which are done in a simple class named `EditorClass`. When the user presses the "Open" button a `JFileChooser` pops up. From there he/she can select an XML type file (like, XML Schema, RDF etc.) and can edit in the edit area (see Figure 4.15 (a)). After editing is done, the user can save and validate the document by clicking on the "Save" and "Validate" button (see Figure 4.15 (b)).

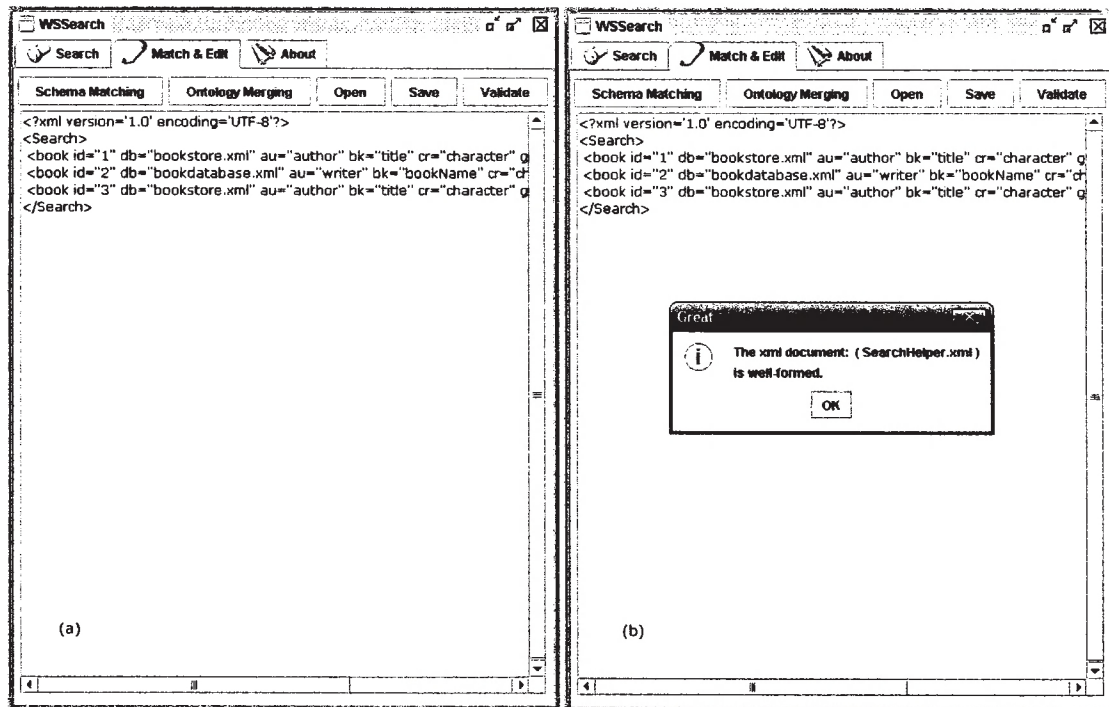


Figure 4.15: (a). Document editing in WSSearch screenshot. This screenshot is taken after a user opened an XML document by clicking on the “Open” button, and made some changes in the code. (b) Document validating in WSSearch screenshot. This screenshot is taken after the user saved the edited document and tried to validate it by clicking on the “Validate” button. A JOptionPane information message window pops up and tells the user, the document is well formed.

Editing text documents like `endpoints.txt`, `webservices.txt` or `readme.txt` are also possible using the same editing tool. The validating is done with SAX API.

4.3.5. About Tab

The about tab contains readme information for the users. This is done with a very simple strategy. The system reads the `readme.txt` file from the “locations” folder and shows the file content in the about JTextArea (see Figure 4.5 (c)).

4.4. Source Codes

The source codes for the Web Services, WSSearch and RDFGenerator applications can be made available upon request. Please contact Ahmed Sabbir Arif (email: asarif@lakeheadu.ca) or Dr. Jinan Fiaidhi (email: jfiaidhi@lakeheadu.ca).

CHAPTER 5

CONCLUSION AND FUTURE RESEARCH

In this thesis we focused on issues of Web Service searching with multiple datasources within a particular domain of knowledge.

A main problem of the current strategy of Web Service searching (UDDI discovery) is that it is based on tModel. The search depends on UDDI service description and tries to discover services on the search keywords. If the service description doesn't have that particular keyword in it then the searcher will fail to discover it. Another problem is that the current searching strategy doesn't provide any value-added services like a recommender system. Moreover searching in service datasources is not possible. Our strategy, which is based on Web Service datasource schema matching and service description ontology merging, tries to solve these problems. In our model a searcher can search for items in the service datasources and can filter out the services that don't have the items the searchers is looking for. This increases the possibility of discovering only the services the users really interested in. As well, our system adds value added services like collaborative filtering recommender system. Depending on the user-rating and the search keywords, this recommender system recommends similar items.

In summery we have discussed in the previous four chapters:

In Chapter 1 we described the architecture of the Web Service and why we might want to use it. We showed that the main attraction of Web Service is the interoperability and how standards like WSDL, XML, SOAP helps Web Service to maintain it. We showed SOAP is not the only way to create Web Service; we can also do it with REST. We also explained why UDDI is not considered as a mandatory standard for service implementation. Finally we pointed out some drawbacks of current UDDI dependent service discovery and demonstrated how we intend to solve those with our proposed architecture.

In the second chapter we explained the Service Oriented Architecture (SOA) in depth and explained why Web Service and SOA are meant to be together. We demonstrated and compared two popular ways (ASP.NET, Apache Axis) and languages (Java, C#) of creating Web Services and made our points of why creating Web Services in Apache Axis is a better choice. We also showed the architectural view of Apache Axis.

Chapter 3 was focused on Web Service search and discovery. We showed how UDDI search is dependent on `tModel` and how that makes the service discovery strictly keyword-bases (taxonomy-based). We also pointed out that UDDI based search doesn't provide any value-added services and searching in service datasources with UDDI is not possible. We explained how our model can solves these problems with the help of XML Schema and RDF Ontology with examples. We provided the RDF syntax supported by our architecture and also showed how our tool, `RDFGenerator`, can help the providers with creating service descriptions in the supported syntax. We explained the role of global `SearchHelper` file in our architecture and after that we provided an elaborated demonstration of the Matching strategy used to match service datasource's schemas (with the help of dictionary) to create that file. We also provided elaborated demonstration of how we can use RDF ontologies merging strategy to do the same task. At last we discussed in depth how we adopted collaborative filtering based recommender system with our architecture.

In Chapter 4 we discussed how we have implemented the prototype for Schema-ontology based searching architecture. This chapter starts with the demonstration of the creation process of Web Services. Then we compare the XML infoset APIs currently available and showed why stream-based APIs fits well with our architecture. We compare push and pull parsers for stream-based APIs. We showed which API is good for which part of our system. Then we discuss the implementation of our prototype in detail with the help of pseudocodes and UML structures like class diagram and sequence diagram. We also talked about the implementation of the `RDFGenerator` tool in this chapter.

5.1. Future Research

The taxonomy and `tModel` of UDDI based search is limited in its search services by its inability to extend beyond the keyword-based matches, which stands as a barrier in Web Services composition where applications are to be assembled from a set of appropriate Web Services. To composite Web Services into new application, it is often inside a domain (a specific area) to search related Web Services (Shen Derong, Yu Ge et al. 2005). With our searching strategy we can search for services inside a domain; in our future research we would like to make use of this advantage to provide facilities for Web Service compositions.

We would like explore the options of making the collaborative filtering more personalized. Use of a user login, item-rating service and maintaining separate similarity database for each user might help us in that direction. We would also like to find if sharing similarity databases of commercial services is possible (e.g., Amazon.com) (Linden, Smith et al. 2003).

We would like to extend our research to make the architecture compatible with current service registries (e.g., UDDI, ebXML, etc.). This would be an important add on to our research as this will make sure the providers don't have to register their services in various registries (Colgrave, Akkiraju et al. 2004, Mahmoud 2002, Pautasso 2005, Xu Bin, Wang Yan et al. 2005). The combination of UDDI `tModel` search and our schema-ontology based search also might prove useful while searching outside the domain.

We wish to examine more advanced matching strategies to do the Match and Merge operation. Our name-matching strategy works great but it depends on dictionary and service descriptions. To eliminate this dependency we would like to go through the artificial intelligence approach to natural language understanding, etc. (Breese, Heckerman et al. 1998, Melville, Mooney et al. 2002).

How to add more value-added services with our architecture is another field we wish to continue our research with. One of the services we are currently thinking of is a user notifier. For example, the user is interested in Harry Potter books. This user notifier will look for updates in Harry Potter books, and if there is any Harry Potter related new service has created or entry in service database has made, the notifier will notify the user about this change. XML-diff algorithms might help us in this direction.

Bibliography

The java web services tutorial, 06/14/2005, 2005-last update [Homepage of Sun Microsystems], [Online]. Available: <http://java.sun.com/webservices/tutorial.html> [03/12, 2006].

ALMAER, D., 05/22/2002, 2002-last update, creating web services with apache axis [Homepage of O'Reilly Media, Inc.], [Online]. Available: <http://www.onjava.com/pub/a/onjava/2002/06/05/axis.html> [02/22, 2006].

ANAND, S., PADMANABHUNI, S. and GANESH, J., 2005. Perspectives On Service Oriented Architecture, 2005, ppxvii vol.2.

ANTONIOU, G. and HARMELEN, F.V., 2003. Web Ontology Language: OWL, S. STAAB and R. STUDER, eds. In: *Handbook on Ontologies in Information Systems*, 2003.

APACHE AXIS, 06/15/2005, 2005-last update, apache axis documentation [Homepage of The Apache Software Foundation], [Online]. Available: <http://ws.apache.org/axis/java/index.html> [02/22, 2006].

BASU, C., HIRSH, H. and COHEN, W.W., 1998. Recommendation as Classification: Using Social and Content-Based Information in Recommendation, *AAAI/IAAI*, 1998, pp714-720.

BATINI, C., LENZERINI, M. and NAVATHE, S.B., 1986. A Comparative Analysis of Methodologies for Database Schema Integration. *Computing Surveys*, **18**(4), pp. 323-64.

BELLWOOD, T., 07/01/2002, 2002-last update, understanding UDDI: tracking the evolving specification [Homepage of IBM], [Online]. Available: <http://www-128.ibm.com/developerworks/webservices/library/ws-featuddi/> [02/13, 2006].

BERGAMASCHI, S., CASTANO, S. and VINCINI, M., 1999. Semantic Integration of Semistructured and Structured Data Sources. *SIGMOD Record*, **28**(1), pp. 54-9.

BONSTROM, V., HINZE, A. and SCHWEPPE, H., 2003. Storing RDF as a Graph, 2003, pp27-36.

BOOTH, D., HAAS, H., MCCABE, F., NEWCOMER, E., CHAMPION, M., FERRIS, C. and ORCHARD, D., 02/11/2004, 2004-last update, web services architecture [Homepage of The World Wide Web Consortium], [Online]. Available: <http://www.w3.org/TR/ws-arch/> [06/02, 2006].

BOX, D., EHNEBUSKE, D., KAKIVAYA, G., LAYMAN, A., MENDELSON, N., NIELSEN, H.F., THATTE, S. and WINER, D., 05/08/2000, 2000-last update, simple object access protocol (SOAP) 1.1 [Homepage of World Wide Web Consortium], [Online]. Available: <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/2006>.

BIBLIOGRAPHY

- BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C.M., MALER, E. and YERGEAU, F., 01/04/2004, 2004-last update, extensible markup language (XML) 1.0 (third edition) [Homepage of The World Wide Web Consortium], [Online]. Available: <http://www.w3.org/TR/REC-xml/> [02/06, 2005].
- BREESE, J.S., HECKERMAN, D. and KADIE, C., 1998. Empirical Analysis of Predictive Algorithms for Collaborative Filtering, *Proceedings 14th Conference on Uncertainty in Artificial Intelligence*, 1998, pp43-52.
- BRICKLEY, D., GUHA, R.V. and MCBRIDE, B., 2/10/2004, 2004-last update, RDF vocabulary description language 1.0: RDF schema [Homepage of The World Wide Web Consortium], [Online]. Available: <http://www.w3.org/TR/rdf-schema/2006>].
- CASTANO, S. and DE ANTONELLIS, V., 2001. Global Viewing of Heterogeneous Data Sources. *IEEE Transactions on Knowledge and Data Engineering*, **13**(2), pp. 277-97.
- CHANNABASAVAIHAH, K., HOLLEY, K. and TUGGLE JR., E., 12/16/2003, 2003-last update, migrating to a service-oriented architecture, part 1 *introduction and overview* [Homepage of IBM], [Online]. Available: <http://www-128.ibm.com/developerworks/webservices/library/ws-migratesoa/> [02/22, 2006].
- CHAPPELL, D., 07/12/2002, 2002-last update, REST: another way of looking at web services [Homepage of Addison Wesley], [Online]. Available: <http://www.awprofessional.com/articles/article.asp?p=27645&seqNum=1> [02/14, 2006].
- CHRISTENSEN, E., CURBERA, F., MEREDITH, G. and WEERAWARANA, S., 03/15/2001, 2001-last update, web services description language (WSDL) 1.1 [Homepage of The World Wide Web Consortium], [Online]. Available: <http://www.w3.org/TR/wsdl> [02/06, 2006].
- CLARK, K.G., 06/18/2003, 2003-last update, A tour of the web services architecture [Homepage of O'Reilly Media, Inc.], [Online]. Available: <http://www.xml.com/pub/a/2003/06/18/ws-arch.html> [02/14, 2006].
- CLEMENT, L., HATELY, A., RIEGEN, C.V. and ROGERS, T., 10/19/2000, 2000-last update, UDDI version 3.0.2 specification [Homepage of OASIS Open], [Online]. Available: <http://uddi.org/pubs/uddi-v3.0.2-20041019.htm> [02/13, 2006].
- COLGRAVE, J., AKKIRAJU, R. and GOODWIN, R., 2004. External Matching in UDDI, 2004, pp226-233.
- DENNY, M., 07/14/2004, 2004-last update, ontology tools survey, revisited [Homepage of O'Reilly Media, Inc.], [Online]. Available: <http://www.xml.com/pub/a/2004/07/14/onto.html> [02/28, 2006].
- DESHPANDE, M. and KARYPIS, G., 2004. Item-based top-N recommendation algorithms. *ACM Trans.Inf.Syst.*, **22**(1), pp. 143-177.
- DIETZEN, S., 05/24/2004, 2004-last update, standards for service-oriented architecture [Homepage of dev2dev], [Online]. Available: http://dev2dev.bea.com/pub/a/2004/05/soa_dietzen.html [03/07, 2006].

BIBLIOGRAPHY

- DOAN, A., DOMINGOS, P. and HALEVY, A., 2003. Learning to Match the Schemas of Data Sources: A Multistrategy Approach. *Mach.Learn.*, **50**(3), pp. 279-301.
- DOAN, A., DOMINGOS, P. and LEVY, A.Y., 2000. Learning Source Description for Data Integration, *WebDB (Informal Proceedings)*, 2000, pp86.
- DOAN, A., MADHAVAN, J., DOMINGOS, P. and HALEVY, A., 2002. Learning to Map Between Ontologies on the Semantic Web, *WWW '02: Proceedings of the 11th international conference on World Wide Web*, 2002, ACM Press pp662-673.
- FENSEL, D., 2003. Ontologies: A Silver Bullet for Knowledge Management and Electronic Commerce. Secaucus, NJ, USA: Springer-Verlag New York, Inc.
- FERRARA, A. and MACDONALD, M., 2002. Programming .NET Web Services. USA: O'Reilly Media, Inc.
- FIAIDHI, J., PASSI, K. and MOHAMMED, S., 2004. Developing a Framework for Learning Objects Search Engine, 2004.
- GIBBS, K., GOODMAN, B. and TORRES, E., 09/30/2003, 2003-last update, create web services using apache axis and castor
how to integrate axis and castor in a document-style web service client and server [Homepage of IBM], [Online]. Available: <http://www-128.ibm.com/developerworks/webservices/library/ws-castor> [02/22, 2006].
- GOVERT, N., KAZAI, G., FUHR, N. and LALMAS, M., 2003. *Evaluating the Effectiveness of Content-oriented XML Retrieval*. University of Dortmund, Computer Science 6.
- GRAUPMANN, J., BIWER, M. and ZIMMER, P., 2003. Towards Federated Search Based on Web Services, 02/26/2003 2003, pp384-384-393.
- HAKIMPOUR, F. and GEPPERT, A., 2002. Global Schema Generation Using Formal Ontologies, *ER '02: Proceedings of the 21st International Conference on Conceptual Modeling*, 2002, Springer-Verlag pp307-321.
- HANSEN, K.H., , web services with axis [Homepage of Java Boutique], [Online]. Available: <http://javaboutique.internet.com/tutorials/Axis> [02/22, 2006].
- HASHIMI, S., 08/18/2003, 2003-last update, service-oriented architecture explained [Homepage of O'Reilly Media, Inc.], [Online]. Available: http://www.ondotnet.com/pub/a/dotnet/2003/08/18/soa_explained.html [03/07, 2006].
- HE, H., 09/30/2003, 2003-last update, what is service-oriented architecture [Homepage of O'Reilly Media, Inc.], [Online]. Available: <http://webservices.xml.com/lpt/a/ws/2003/09/30/soa.html> [02/22, 2006].
- HEYLIGHEN, F., 01/31/2001, 2001-last update, collaborative filtering [Homepage of Principia Cybernetica], [Online]. Available: <http://pespmc1.vub.ac.be/COLLFILT.html> [03/09, 2006].

BIBLIOGRAPHY

JONES, S., 2005. Toward An Acceptable Definition of Service [Service-oriented Architecture]. *Software, IEEE*, **22**(3), pp. 87-93.

KONSTAN, J.A., MILLER, B.N., MALTZ, D., HERLOCKER, J.L., GORDON, L.R. and RIEDL, J., 1997. GroupLens: Applying Collaborative Filtering to Usenet News. *Communications of the ACM*, **40**(3), pp. 77-87.

LI, J.B. and MILLER, J., 2005. Testing the Semantics of W3C XML Schema, 2005, pp443-448 Vol. 2.

LINDEN, G., SMITH, B. and YORK, J., 2003. Amazon.com Recommendations: Item-to-Item Collaborative Filtering, *IEEE Internet Computing*, vol. 07, 2003, pp76-80.

LOMOW, G. and NEWCOMER, E., 2004. Understanding SOA with Web Services. 1st edn. UK: Addison Wesley Professional.

MAHMOUD, Q.H., 06/01/2002, 2002-last update, registration and discovery of web services using JAXR with XML registries such as UDDI and ebXML [Homepage of Sun Microsystems], [Online]. Available: <http://java.sun.com/developer/technicalArticles/WebServices/jaxrws/02/13, 2006>].

MANOLA, F., MILLER, E. and MCBRIDE, B., 2004, 2005-last update, RDF primer [Homepage of The World Wide Web Consortium], [Online]. Available: <http://www.w3.org/TR/rdf-primer/02/03, 2006>].

MCGUINNESS, D.L. and HARMELEN, F.V., 02/10/2004, 2004-last update, OWL web ontology language overview [Homepage of The World Wide Web Consortium], [Online]. Available: <http://www.w3.org/TR/owl-features/02/05, 2006>].

MELVILLE, P., MOONEY, R.J. and NAGARAJAN, R., 2002. Content-boosted Collaborative Filtering for Improved Recommendations, *Eighteenth national conference on Artificial intelligence*, 2002, American Association for Artificial Intelligence pp187-192.

MILLER, R.J., IOANNIDIS, Y.E. and RAMAKRISHNAN, R., 1994. Schema Equivalence in Heterogeneous Systems: Bridging Theory and Practice, *4th International Conference on Extending Database Technology*, 28-31 March 1994, | 1994, Springer-Verlag pp73-80.

MILO, T. and ZOHAR, S., 1998. Using Schema Matching to Simplify Heterogeneous Data Translation, *Proceedings of 24th Annual International Conference on Very Large Data Bases (VLDB'98)*, 24-27 Aug. 1998, | 1998, Morgan Kaufmann Publishers Inc pp122-33.

OASIS OPEN, 02/13/2006, 2006-last update, ebXML home [Homepage of OASIS Open], [Online]. Available: <http://www.ebxml.org/02/13, 2006>].

PAUTASSO, C., 2005. *Distributed Systems UDDI and Beyond*. 1. Information and Communication Systems Research Group: Swiss Federal Institute of Technology.

PEIRIS, C., 30/04/2001, 2001-last update, creating a .NET web service [Homepage of Jupitermedia Corp.], [Online]. Available: <http://www.15seconds.com/Issue/010430.htm02/22, 2006>].

BIBLIOGRAPHY

- RAHM, E. and BERNSTEIN, P.A., 2001. A Survey of Approaches to Automatic Schema Matching. *The Very Large Data Bases Journal* 10, 10, pp. 334-334–350.
- RESNICK, P., IACOVOU, N., SUCHAK, M., BERGSTORM, P. and RIEDL, J., 1994. GroupLens: An Open Architecture for Collaborative Filtering of Netnews, *Proceedings of ACM 1994 Conference on Computer Supported Cooperative Work*, 1994, pp186.
- ROY, J. and RAMANUJAN, A., 2001. XML Schema Language: Taking XML to the Next Level. *IT Professional*, 3(2), pp. 37-40.
- SAKAMURI, B.C., MADRIA, S.K., PASSI, K., CHAUDHRY, E., MOHANIA, M.K. and S., S., 2003. AXIS: A XML Schema Integration System, *Proceedings of the 22nd International Conference on Conceptual Modeling*, 2003, pp576-578.
- SARWAR, B.M., KARYPIS, G., KONSTAN, J.A. and REIDL, J., 2001. Item-based Collaborative Filtering Recommendation Algorithms, *World Wide Web*, 2001, pp285-295.
- SARWAR, B.M., KARYPIS, G., KONSTAN, J.A. and RIEDL, J., 2000. Analysis of Recommendation Algorithms for E-commerce, *ACM Conference on Electronic Commerce*, 2000, pp158-167.
- SHARDANAND, U. and MAES, P., 1995. Social Information Filtering: Algorithms for Automating "Word of Mouth", *Proceedings of ACM CHI'95 Conference on Human Factors in Computing Systems*, 1995, pp210-217.
- SHEN DERONG, YU GE, CAO YU, KOU YUE and NIE TIEZHENG, 2005. An Effective Web Services Discovery Strategy for Web Services Composition, 2005, pp257-263.
- SHETH, A.P. and LARSON, J.A., 1990. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *Computing Surveys*, 22(3), pp. 183-236.
- SIBLINI, R. and MANSOUR, N., 2005. Testing Web Services, 2005, pp135.
- SMITH, M.K., WELTY, C. and MCGUINNESS, D.L., 02/10, 2004-last update, OWL web ontology language guide [Homepage of The World Wide Web Consortium], [Online]. Available: <http://www.w3.org/TR/owl-guide/> [02/14, 2006].
- STRAHL, R., 03/07/2002, 2002-last update, creating and using web services with the .NET framework and visual studio.NET [Homepage of West Wind Technologies], [Online]. Available: <http://www.westwind.com/presentations/dotnetwebservices/DotNetWebServices.asp> [02/22, 2006].
- SUN, C., LIN, Y. and KEMME, B., 2004. Comparison of UDDI Registry Replication Strategies, 2004, pp218-225.
- SYSTINET CORPORATION, 2005. SOA Simplified, Systinet Server for Java 6.5 Systinet Primer Tutorials and White Pages. Tutorials and White Pages.
- TERVEEN, L., HILL, W., AMENTO, B., MCDONALD, D. and CRETER, J., 1997. PHOAKS: A System for Sharing Recommendations.

BIBLIOGRAPHY

THOMPSON, H., BEECH, D., MALONEY, M. and MENDELSON, N., 28 October 2004, 2004-last update, XML schema part 1: structures second edition [Homepage of The World Wide Web Consortium], [Online]. Available: <http://www.w3.org/TR/2004/REC-xmlschema-1-20041028> [01/26, 2006].

UNGAR, L. and FOSTER, D., 1998. Clustering Methods for Collaborative Filtering, *Proceedings of the Workshop on Recommendation Systems*, 1998.

VASUDEVAN, V., 04/04/2001, 2001-last update, A web services primer [Homepage of O'Reilly Media, Inc.], [Online]. Available: <http://webservices.xml.com/pub/a/ws/2001/04/04/webservices/index.html> [02/22, 2006].

WANG, J., VRIES, ARJEN P. DE and REINDERS, M.J.T., 2006. A User-Item Relevance Model for Log-based Collaborative Filtering, *European Conference on Information Retrieval (ECIR 2006)*, 2006.

XIAO, H., CRUZ, I.F. and HSU, F., Semantic Mappings for the Integration of XML and RDF Sources, "*Workshop on Information Integration on the Web (IIWeb 2004)*".

XU BIN, WANG YAN, ZHANG PO and LI JUANZI, 2005. Web Services Searching based on Domain Ontology, 2005, pp51-58.

ZHIHONG, Z. and MINGTIAN, Z., 2003. Web Ontology Language OWL and Its Description Logic Foundation, 2003, pp157-160.