A LOAD BALANCING SCHEME FOR EBXML REGISTRIES

A Thesis

Presented to the

Faculty of

San Diego State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

in

Computer Science

by

Sadhana V. Sahasrabudhe

Spring 2011
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Sadhana V. Sahasrabudhe:

A Load Balancing Scheme for ebXML Registries

Alan Riggins, Chair
Department of Computer Science

William Root
Department of Computer Science

Christopher Paolini
Department of Mechanical Engineering

22nd April 2011
Approval Date
Copyright © 2011

by

Sadhana V. Sahasrabudhe

All Rights Reserved
DEDICATION

I dedicate this work to my parents Vinaya and Vasant, and sister Swati.
ABSTRACT OF THE THESIS

A Load Balancing Scheme for ebXML Registries
by
Sadhana V. Sahasrabudhe
Master of Science in Computer Science
San Diego State University, 2011

Large Scale Service Oriented Architecture (SOA) developments are becoming increasingly reliant on registry services that manage Web Services using taxonomic attributes. A registry stores a Web Service’s interface definition and protocol bindings in WSDL format, along with one or more XML schema files that define the structure of a SOAP message exchanged between Web Service operations and client processes and other static metadata. During Web Service discovery, an ebXML registry returns the access URI associated with a service binding to allow dynamic Web Service invocation. This usually restricts a calling process to a Web Service invocation on a single host. This thesis presents an architecture that manages service bindings for Web Services that have been deployed across multiple hosts such that a URI returned by a ebXML registry can resolve to a host that satisfies different system constrains like current CPU load, physical memory, swap memory, and time of day. This architecture involves the design and development of a new scheme for ebXML registries that facilitate the periodic collection and management of dynamic system properties for registry clients and the enforcement of constraints during service discovery and operation invocation. The proposed scheme is unique as it extends the existing freebXML registry implementation by adding load balancing features along with other QoS capabilities. Furthermore, our scheme is transparent to an end user as no significant code changes are required by a user to utilize this load balancing architecture. Using our modified ebXML registry, it is possible to implement a many task computing (MTC) application using distributed Web Services in an effective way across multiple hosts where the CPU load and system memory is uniformly maintained.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>viii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>ix</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>xiii</td>
</tr>
<tr>
<td>1 INTRODUCTION TO REGISTRY/REPOSITORIES</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Function of a Web Service Registry</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Function of a Web Service Repository</td>
<td>4</td>
</tr>
<tr>
<td>1.3 Registry/Repository Standards</td>
<td>6</td>
</tr>
<tr>
<td>1.3.1 Universal Description, Discovery, and Integration</td>
<td>6</td>
</tr>
<tr>
<td>1.3.1.1 Layered Web Service Stack</td>
<td>6</td>
</tr>
<tr>
<td>1.3.1.2 How Does a UDDI Registry Work?</td>
<td>11</td>
</tr>
<tr>
<td>1.3.1.3 Technical Architecture</td>
<td>13</td>
</tr>
<tr>
<td>1.3.1.4 UDDI Data Structure</td>
<td>15</td>
</tr>
<tr>
<td>1.3.1.5 UDDI APIs</td>
<td>19</td>
</tr>
<tr>
<td>1.3.1.6 UDDI Evolution</td>
<td>22</td>
</tr>
<tr>
<td>1.3.2 Electronic Business Using eXtensible Markup Language</td>
<td>24</td>
</tr>
<tr>
<td>1.3.2.1 Introduction to ebXML</td>
<td>24</td>
</tr>
<tr>
<td>1.3.2.2 ebXML Architecture Overview</td>
<td>27</td>
</tr>
<tr>
<td>1.3.2.3 ebXML Registry</td>
<td>29</td>
</tr>
<tr>
<td>1.3.2.4 ebXML Registry Interfaces</td>
<td>35</td>
</tr>
<tr>
<td>1.3.2.5 Content Based Notification</td>
<td>37</td>
</tr>
<tr>
<td>1.4 Previous Work</td>
<td>38</td>
</tr>
<tr>
<td>2 THE FREEBXL REGISTRY/REPOSITORY</td>
<td>42</td>
</tr>
<tr>
<td>2.1 The freebXML Registry</td>
<td>42</td>
</tr>
<tr>
<td>2.2 freebXML Registry Architecture</td>
<td>42</td>
</tr>
</tbody>
</table>
2.2.1 The Registry Client Layer ................................................................. 42
2.2.2 The Registry Client API Layer ............................................................ 43
2.2.3 The freebXML Registry Server .......................................................... 44
2.3 Features and Benefits of Using freebXML ............................................. 48

3 A LOAD BALANCING SCHEME FOR THE FREEBXL REGISTRY/REPOSITORY ................................................................. 49
3.1 Motivation ............................................................................................. 49
3.2 Modifications to freebXML ................................................................. 49
3.3 Client Side Web Services ................................................................. 54
3.4 Publishing and Accessing Registry Contents ........................................ 55
  3.4.1 Software Requirements ................................................................. 56
  3.4.2 User Registration ........................................................................ 56
  3.4.3 Client Keystore Generation .......................................................... 62
  3.4.4 Publishing and Modifying Registry Contents .................................. 64
    3.4.4.1 Web UI ............................................................................... 64
    3.4.4.2 The AccessRegistry API ..................................................... 86
  3.4.5 Developing Sample Programs Using the AccessRegistry API .......... 97
  3.4.6 JUnit Test Cases .......................................................................... 105

4 RESULTS .................................................................................................. 114
  4.1 Publish Organization and Web Service .............................................. 114
  4.2 Add Web Service ............................................................................... 115
  4.3 Edit Web Service Description ........................................................... 117
  4.4 Delete a Web Service ......................................................................... 118
  4.5 Delete Organization ........................................................................... 119
  4.6 Access Web Service .......................................................................... 119

5 CONCLUSION ......................................................................................... 121
  5.1 Summary of Work ............................................................................... 121
  5.2 Future Directions ............................................................................... 121

BIBLIOGRAPHY ........................................................................................... 122
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Registry Repository Standard Comparison Matrix</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Categories Supported</td>
<td>20</td>
</tr>
<tr>
<td>1.3</td>
<td>Identifier Types</td>
<td>20</td>
</tr>
<tr>
<td>1.4</td>
<td>UDDI Registry Types</td>
<td>24</td>
</tr>
<tr>
<td>1.5</td>
<td>Pre-Defined Associations</td>
<td>34</td>
</tr>
<tr>
<td>1.6</td>
<td>Methods Supported by LifeCycleManager</td>
<td>36</td>
</tr>
<tr>
<td>1.7</td>
<td>Methods Supported by QueryManager</td>
<td>37</td>
</tr>
<tr>
<td>3.1</td>
<td>File Name and Listing</td>
<td>63</td>
</tr>
<tr>
<td>3.2</td>
<td>Options Details</td>
<td>63</td>
</tr>
<tr>
<td>3.3</td>
<td>Rules for Embedding Child Elements in a Parent Tag</td>
<td>87</td>
</tr>
<tr>
<td>3.4</td>
<td>Action Tag Type Options and Meaning</td>
<td>88</td>
</tr>
<tr>
<td>3.5</td>
<td>Symbols</td>
<td>91</td>
</tr>
<tr>
<td>3.6</td>
<td>List of Fields That Can or Cannot Be Modified</td>
<td>94</td>
</tr>
<tr>
<td>3.7</td>
<td>List of Organization and Services from PublishToRegistry.xml</td>
<td>101</td>
</tr>
<tr>
<td>3.8</td>
<td>List of Elements That Are Modified Using ModifyRegistry.xml</td>
<td>103</td>
</tr>
<tr>
<td>3.9</td>
<td>Test Case Description</td>
<td>109</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

Figure 1.1. Web Service stack showing different protocols and layers..................................10
Figure 1.2. Message flow between client and UDDI registry.................................................12
Figure 1.3. UDDI registry use case scenario. ........................................................................13
Figure 1.4. Web Service discovery using UDDI registry. .......................................................14
Figure 1.5. UDDI registry technical architecture.................................................................14
Figure 1.6. UDDI data structure. ..........................................................................................15
Figure 1.7. A businessEntity template. ................................................................................16
Figure 1.8. A publisherAssertion template. .......................................................................17
Figure 1.9. A businessService template...............................................................................17
Figure 1.10. A bindingTemplate template. .........................................................................18
Figure 1.11. A service template. .........................................................................................19
Figure 1.12. Scenario depicting registry affliction. ...............................................................23
Figure 1.13. ebXML business scenario.................................................................................26
Figure 1.14. ebXML architecture.........................................................................................28
Figure 1.15. ebXML use case. ............................................................................................30
Figure 1.16. ebXML registry architecture. .........................................................................31
Figure 1.17. Registry service and client interaction. ...........................................................31
Figure 1.18. Registry information model. ...........................................................................33
Figure 1.19. Object life cycle of a registry object. ...............................................................36
Figure 1.20. Scenario depicting event based notification. ..................................................37
Figure 2.1. The freebXML registry architecture..................................................................43
Figure 2.2. JAXR architecture. ..........................................................................................44
Figure 2.3. Inner workings of a JAXR provider. .................................................................45
Figure 2.4. Registry object life cycle. ...............................................................................47
Figure 3.1. Class diagram for TimeHits class.................................................................50
Figure 3.2. Table NodeState used to store the state of a host machine.........................51
Figure 3.3. High level data flow diagram from Web Service publishing to discovery.......52
Figure 3.4. Detail data flow diagram.

Figure 3.5. Collaboration diagram represents the relationship and interaction between ServiceDAO, ServiceBindingDAO and the LoadStatus classes.

Figure 3.6. Interaction diagram describing messaging between ServiceDAO, ServiceBindingDAO and the LoadStatus classes.

Figure 3.7. Administrator needs to deploy NodeStatus Web Service on the hosts to be load balanced and register the Web Service to freebXML.

Figure 3.8. Web UI of freebXML registry on volta.sdsu.edu.

Figure 3.9. Highlighting link to create user account.

Figure 3.10. Registration wizard for registering new users.

Figure 3.11. Registration wizard step 1, requirements.

Figure 3.12. Registration wizard step 2, user’s details.

Figure 3.13. User registration wizard, user details form.

Figure 3.14. User registration wizard: links to load key to web browser.

Figure 3.15. freebXML web UI with link to create new registry object highlighted.

Figure 3.16. Organization object type is selected to be created.

Figure 3.17. Organization details form.

Figure 3.18. Organization postal address.

Figure 3.19. Organization postal address tab.

Figure 3.20. Postal address details form.

Figure 3.21. Address displayed in the postal address tab.

Figure 3.22. Apply successful message is displayed if registry contents are modified successfully.

Figure 3.23. Click on email tab to add organization email address.

Figure 3.24. To add an organization email click add in email tab.

Figure 3.25. Enter organization email address.

Figure 3.26. Email tab with organization email listed click save to save the email.

Figure 3.27. Click on telephone number tab to add organization telephone number.

Figure 3.28. Click add on telephone numbers tab to add telephone.

Figure 3.29. Fill in the telephone details and select the telephone type.

Figure 3.30. Phone number added is listed in telephone number tab click save to save.

Figure 3.31. Organization address details.

Figure 3.32. Organization detail: telephone number and email.
Figure 3.33. Click apply to commit the information.................................................................75
Figure 3.34. Create new registry object of type service............................................................76
Figure 3.35. Web Service details page......................................................................................76
Figure 3.36. Click on ServiceBinding tab to enter access URIs associated with a Web Service..........................................................................................................................77
Figure 3.37. Click add in ServiceBinding tab to add new access URIs.....................................77
Figure 3.38. ServiceBinding details panel click save to save the changes...............................78
Figure 3.39. Service bindings tab – added service bindings are listed in the tab....................79
Figure 3.40. Lists all the service bindings added......................................................................79
Figure 3.41. *FindAllMyObjects* option lists all the objects that are published by a user.......80
Figure 3.42. Select objects to be associated and then click relate...........................................81
Figure 3.43. Association object details....................................................................................82
Figure 3.44. Association type.................................................................................................82
Figure 3.45. Commit association object by clicking apply.........................................................83
Figure 3.46. NewOrganization details....................................................................................83
Figure 3.47. NodeStatus details..............................................................................................84
Figure 3.48. List of all the objects owned by a user.................................................................84
Figure 3.49. Select object that is to be modified and then click details.....................................85
Figure 3.50. Select the object to delete and press the delete button........................................85
Figure 3.51. Execute() returns a arraylist which holds the results of different operation in different arraylist.................................................................98
Figure 3.52. A search for DemoOrganization on freebXML confirms that the organization was published successfully using AccessRegistry API..............................100
Figure 3.53. Organization search results after publishing from PublishToRegistry.xml.............102
Figure 3.54. Service search results after publishing from PublishToRegistry.xml....................102
Figure 3.55. Organization search result for organization with names starting with DemoOrg_..................................................................................................................105
Figure 3.56. Service search result for service names starting with the string DemoSrv............106
Figure 3.57. Service DemoSrv_EditDescription2....................................................................107
Figure 3.58. JUnit test cases within the AccessRegistry API...................................................108
Figure 3.59. JUnit test results screen shot showing all the test cases were executed successfully.................................................................................................................110
Figure 4.1. Search results showing organization and Web Service published.........................115
Figure 4.2. Search result after adding ServiceAdder to organization San Diego State University..................................................................................................................116
Figure 4.3. Search result after editing Web Service description...........................................117
Figure 4.4. Search result after deleting Web Service ServiceAdder......................................118
Figure 4.5. Search result after deleting organization San Diego State University (SDSU).................................................................................................................................119
ACKNOWLEDGEMENTS

I would like to thank Dr. Christopher Paolini for giving me the opportunity to work on this thesis, and for all his support and guidance, without which it would have been impossible to complete this work. I would like to thank Farrukh Najmi, from freebXML for responding to all queries promptly. A special thanks to Anupama Potluru and Chiranjeevi Manne for their help in configuring Ubuntu. I am grateful to my family and friends for all their support and encouragement throughout this period. Finally, I would like to thank the San Diego State University Graduate Division for their help on my graduate application and thesis review process.
CHAPTER 1

INTRODUCTION TO REGISTRY/REPOSITORIES

One of the primary features of SOA architecture is loose coupling among different participating components. This is achieved by using different service components called Web Services. A Web Service encompasses functionality that can be used by Internet accessible distributed applications. The entire web application can be a collection of different Web Services; however an end user is only aware of the complete application. A distributed application can be executed on multiple hosts but from the end user’s prospective it is executed on the machine to which he interfaces. A Web Service can be accessed through its end point URL or access URL. In order to use any Web Service it is essential to first locate the Web Service and then invoke the Web Service using the access URL. Hard coding end point address of a Web Service tightly couples an implementation to its location. This impairs the agility and efficiency of SOA architecture. What is needed is a way to dynamically discover Web Services that satisfies particular criteria at run time and reuse it. Location transparency of a Web Service is critical to the SOA architecture; this can be achieved through the use of SOA Registry Repositories. SOA Registry Repositories are becoming an important component in the SOA architecture. Besides making service components location independent, SOA Registry Repositories also satisfy the need for SOA policy governance and enforcement [1]. SOA policies come in two flavors: design time constraints and runtime constraints. These constraints are specified electronically and enforced programmatically. A SOA Registry Repository acts as a much needed control point that enforces design time and run time policies uniformly, thus improving overall performance. Among other things, SOA Registry Repositories facilitate storage of metadata such as links or pointers to artifacts as well as actual artifacts related to different Web Services.

Both SOA Registry and Repositories play distinct roles. SOA Registry was the earliest attempt towards a more uniform and formal way of managing and collaborating information among different organizations. SOA Registry provides digital identity to service
components like Web Services that encompasses the SOA architecture. Metadata associated
with service components such as links and pointers to service artifacts are stored in a SOA
Registry. The Universal, Description, Discovery and Integration (UDDI) registry is one such
example of a SOA Registry used to manage and share information.

As stated before, SOA Registry can only hold pointers and links to service artifacts;
the service artifacts are stored at some other location outside the registry. The SOA registry
thus cannot govern these artifacts. SOA Repository provides the solution for this deficiency.
SOA Repository acts as a store for service artifacts. SOA Repository provides extensive
search capabilities for specific service artifacts based on artifact related metadata; thus SOA
Repository is responsible for extracting metadata from service artifacts and cataloging these
artifacts. Further, SOA Repository is responsible for validating service artifacts and rejecting
artifacts that do not conform to SOA policies. SOA Repository also provides versioning and
subscription capabilities, thus making it easy to track changes to different service
components. Thus SOA Registry Repository provide governess over metadata as well as
service artifacts.

Universal, Description, Discovery and Integration (UDDI) [2], is an industry
initiative aimed at providing a platform independent method for publishing and discovering
Web Services. A UDDI registry manages information regarding service providers, service
implementations, and metadata. A UDDI registry acts as an advertising medium used by
service providers to advertise their services. Service consumers discover services they need
through UDDI registries and use services metadata to consume different Web Services. A
UDDI registry relies on distributed Web registries that implement service descriptions in a
common XML format. Similar to an SOA Registry except for Web Service metadata, all the
other service artifacts are stored outside the UDDI registry.

Electronic Business using eXtensible Markup Language (ebXML) [3] is a collection
of specifications that enables enterprises of any type located anywhere on the Internet to
conduct business over the Internet. ebXML provides a standard method to exchange business
messages, conduct trading relationships, communicate data in common terms, and define and
register business processes. ebXML provides a complete framework for businesses to
achieve ad hoc business interactions. The core of ebXML is a powerful system of registries
and distributed repositories. Along with the registry, there are specifications for the
messaging layer as well as for business process specifications and collaboration information. The set of concrete ebXML specifications maps to these concepts as follows:

- Centralized Shared Registry: Registry Information Model, Registry Services Specification (ebRIM, ebRS).
- ebRIM defines what types of objects are stored in the registry and how they are organized [4].
- ebRS provides the interface for managing a repository [5].
- ebBPSS provides a framework by which business systems may be configured to support execution of business collaborations consisting of business transactions [6].
- ebCPPA defines a standard for information used in business collaborations. The information includes details of transport, messaging, security constraints, and bindings to a Business-Process-Specification document that contains the definition of the interactions between two parties engaged in a specified electronic business collaboration [7].
- Messaging: Message Services Specification (ebMS) is a standard for business messages. The specification is build on SOAP Web Services message format, thus providing a neutral format for carrying messages between different systems [8].

An ebXML registry repository is an ideal example of a SOA Registry Repository. For this work we have used the freebXML [9] registry. freebXML is an open source implementation of the ebXML registry standard. freebXML is first to deliver a feature complete and specification compliant implementation of the latest ebXML Registry standard.

**1.1 FUNCTION OF A WEB SERVICE REGISTRY**

A Web Service can be invoked by a client that requires functionality implemented by the Web Service. Invoking a Web Service requires knowledge of its location; the endpoint address could be hard-coded in the implementation. This approach tightly couples a Web Service to client implementation and the service location. Hard coding is error prone and scales poorly as the number of Web Services grows. What is needed in this case is a component specialized in dynamically resolving service queries into endpoint addresses and invocation policies. A Web Service registry provides the most flexible and maintainable solution to this problem. A service registry, in this case, contains all the information about Web Services deployments, their locations, and the policies associated with invocations at
each location. A SOA Registry is where structured information about a SOA is stored. The concept of a service registry was introduced by the Web Services architecture group, which defined the Universal Description, Discovery, and Integration (UDDI) registry as an “intermediary” between services consumers and providers. Thus it could be said that, usage of a Web Service registry is a runtime lookup of a Web Service endpoint based on Web Service name and policies that are required by service consumer. A Web Service Registry is used to inventory and catalog Web Service assets so that they can be discovered and reused. An interesting way to describe the purpose of a SOA Registry is suggested by Keith Pijanowski in his blog “SOA Registry Repository” [10] of it as a provider of digital identities for the Web Services that make up an enterprise SOA. A digital identity has subject and attributes that further define the subject. In this case a Web Service is a subject. All other forms of metadata such as Provider Information, Configuration, Dependencies, Subscriptions (or interested parties), Policies, and Business Taxonomy are used to further define the service.

1.2 FUNCTION OF A WEB SERVICE REPOSITORY

A system can be completely described by structured data along with some form of “document” or non-structured data will be used to describe some aspect of a software system. For example, WSDL can be used to describe contracts and bindings. While WS-Policy and Service Level Agreements can be used to describe rules that must be adhered to at design time and at run time. All services-related artifacts in the enterprise-wide SOA implementation become enterprise-wide assets and require a centralized asset repository that stores all of the above information. The repository should also provide capabilities to search, modify, etc to all of the SOA stakeholders. Such a service repository integrates all of the sources of services-related information, including design artifacts, runtime topologies, information collected by service monitoring and management solutions, service code repository, and so on. A Web Service repository provides information required to support the complete service life cycle starting from its inception through design, implementation, deployment, usage, and maintenance. Thus we can say the basic SOA Repository contains information that is referenced usually via a URL from the SOA Registry. Based on common registry capabilities these documents are: WSDL files, Policy files, and any other
documentation generated during the software development lifecycle. The SOA Repository can also contain artifacts generated at run time such as logged messages, archived performance data, and archived health data.

According to [1] the essential capabilities of a service repository include:

- **Service cataloging and discovery.** The main purpose of the service repository is to provide the ability to find artifacts, based on artifact-specific metadata. This metadata is typically contained in the artifacts themselves. Consequently, a service repository should automatically extract this metadata (based on the cataloging policies) whenever new artifacts are published to the repository.

- **Validation.** As the point of access to a Web Service related information, the service repository should enforce organizational and domain-specific business rules, ensuring conformance of these artifacts to the enterprise policies and standards. This ability to enforce validation rules makes the repository a focal part of SOA governance.

- **Dependency management.** Web Service related information typically includes multiple interrelated artifacts, such as service interfaces, message schemas, implementation code, usage profiles, and so on. In addition, the Web Services themselves can be reused by other Web Services or business processes. As the number of Web Services grows, tracking all these dependencies and evaluating the impacts of changes becomes a difficult task. The service repository can simplify it by supporting the management of relationships between service artifacts. The repository should provide standard relationship types; it should also allow the organization to extend these types with additional ones based on their additional requirements.

- **Web Service evolution and versioning.** Once created, Web Services typically evolve over time. This evolution can be caused by changes in the Web Service functionality, semantic messaging, and implementation. Many of these changes will require creation and deployment of a new version of the Web Service. In order to track all of this versioning information, a Web Service repository should provide versioning capabilities. In addition a Web Service repository should provide subscription to change/versioning notification capabilities, allowing interested parties to be notified about upcoming and current changes.

- **Artifacts publishing governance.** As a Web Service repository becomes a centralized collection of all of the information about its Web Services, it requires the same governance as any other enterprise assets repository. This type of governance typically includes permissions for publishing Web Services related artifacts and artifacts publishing approval processes.

- **Support for multiple artifacts types.** One of the main challenges in creation of a service repository is a great diversity of Web Service-related artifacts, including XML documents that define services interfaces and messaging schemas, implementation code, UML diagrams, and text documents. The use of a generic representation for the different asset types can significantly simplify the repository implementation.
1.3 Registry/Repository Standards

We will be looking at UDDI and ebXML registry repository standards in this section. See Table 1.1.

1.3.1 Universal Description, Discovery, and Integration

UDDI stands for Universal Description, Discovery, and Integration. UDDI is an industry initiative aimed at providing a platform independent method for publishing and discovering Web Services. A UDDI registry manages information regarding service providers, service implementations, and metadata. A UDDI registry acts as an advertising medium used by service providers to advertise their services. Service consumers discover services that they need through UDDI registries and use services metadata to consume different services. A UDDI registry relies on distributed Web registries that implement service descriptions in a common XML format.

1.3.1.1 Layered Web Service Stack

The Web Service layered stack is shown in Figure 1.1. Each layer in the stack builds upon functionality provided by the lower layers. The Network layer acts as a foundation layer by supporting different protocols like HTTP, FTP, etc. The XML-based messaging layer, which is build on top of the networking layer, supports the Simple Object Access Protocol (SOAP) messaging protocol. All SOAP based messages support publish, bind, and find operations. The service description layer provides support for Web Service invocation. Web Service Description Language (WSDL) is an XML format for describing service implementations and service interfaces. WSDL is used to describe operations and methods supported by a Web Service, along with data format, protocol details, and access an URL. A Web Service can be located and accessed through an access URL. If a client knows in advance the Web Service it is going to use, the client can ask the service provider for the WSDL file associated with that particular service. However, this is not a ideal solution. This is where UDDI registry, the fourth layer of the Web Service stack service publication comes into the picture. By using different data structures provided by UDDI, a service provider can publish a WSDL to a public or private UDDI registry.
<table>
<thead>
<tr>
<th>Category/Feature</th>
<th>ebXML Registry 3.0</th>
<th>UDDI 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standards Leveraged</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Service Description Standards</td>
<td>WSDL 1.1</td>
<td>WSDL 1.1</td>
</tr>
<tr>
<td>Messaging Standards</td>
<td>SOAP 1.1 with Attachments</td>
<td>SOAP 1.1</td>
</tr>
<tr>
<td>Message Security Standards</td>
<td>OASIS Web Service Security</td>
<td></td>
</tr>
<tr>
<td>Access Control Policy Standards</td>
<td>XACML 1.0</td>
<td></td>
</tr>
<tr>
<td>Identity Management Standards</td>
<td>SAML 2.0</td>
<td></td>
</tr>
<tr>
<td>Architecture</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-Oriented API</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>• API offers a few task-oriented calls</td>
<td></td>
<td>• API offers type-oriented calls</td>
</tr>
<tr>
<td>that may be used by any type of metadata</td>
<td></td>
<td>that keep growing as new types</td>
</tr>
<tr>
<td>object</td>
<td></td>
<td>are added in each release of</td>
</tr>
<tr>
<td>• Consistent and uniform actions</td>
<td></td>
<td>UDDI standard</td>
</tr>
<tr>
<td>supported via few API calls across</td>
<td></td>
<td></td>
</tr>
<tr>
<td>entire information model</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-Oriented Information Model</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Extensible API</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>• New API calls may be defined using</td>
<td></td>
<td></td>
</tr>
<tr>
<td>standards-based API extensibility</td>
<td></td>
<td></td>
</tr>
<tr>
<td>features</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extensible Information Model</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>• New information model types may be</td>
<td></td>
<td></td>
</tr>
<tr>
<td>defined using standards-based type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>extensibility features</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Core Features</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Registry</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>• Rich set of ~25 standard metadata</td>
<td></td>
<td>• Inadequate set of ~6 standard</td>
</tr>
<tr>
<td>classes</td>
<td></td>
<td>metadata classes</td>
</tr>
<tr>
<td>Repository</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>• Integrated registry-repository</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Any type of electronic content</td>
<td></td>
<td></td>
</tr>
<tr>
<td>supported</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Publish</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Can publish metadata describing any</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>type of information artifact</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Can publish any type of information</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>artifact</td>
<td></td>
<td>• Actual information artifact</td>
</tr>
<tr>
<td>• Information artifacts subject to</td>
<td></td>
<td>resides external to registry</td>
</tr>
<tr>
<td>governance</td>
<td></td>
<td>and therefore not subject to</td>
</tr>
<tr>
<td>• Supports predicate combination</td>
<td></td>
<td>governance</td>
</tr>
<tr>
<td>using logical operators</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SQL query syntax</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>• Ad hoc syntax supports unlimited</td>
<td></td>
<td></td>
</tr>
<tr>
<td>number of queries</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(table continues)
Table 1.1. (continued)

<table>
<thead>
<tr>
<th>Category/Feature</th>
<th>ebXML Registry 3.0</th>
<th>UDDI 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Discovery</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>XML query syntax</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>• Ad hoc syntax supports unlimited number of queries</td>
<td>• Fixed syntax supports 4 predefined queries</td>
</tr>
<tr>
<td>Stored parameterized queries</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Life Cycle Management</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approval</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Update</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Automatic Version Control</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>• Versioning of metadata</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Versioning of information artifacts</td>
<td></td>
</tr>
<tr>
<td>Deprecation</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>• Prevents proliferation of obsolete information artifacts</td>
<td></td>
</tr>
<tr>
<td>Un-Deprecation</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Deletion</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>• Prevents accidental deletion of information artifacts that are in use</td>
<td></td>
</tr>
<tr>
<td><strong>Taxonomy/Classification Support</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Predefined taxonomies</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>User-defined taxonomies</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Taxonomy browsing and validation</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Classification of artifacts</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Classification of any metadata object</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Relationship Support</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Predefined relationship types</td>
<td>Yes - Extensive</td>
<td>Yes-Very Limited</td>
</tr>
<tr>
<td>User-defined relationship types</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Ability to relate any two objects in registry using any relationship type</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Packaging/Grouping Support</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>User-defined packages</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Group any number of objects in same package</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Group an object in multiple packages</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td><strong>Security Features</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Digital signature based authentication</td>
<td>Yes - Required</td>
<td>Yes - Optional. Most vendors do not support it.</td>
</tr>
<tr>
<td>Basic access control based on predefined roles and predefined policies</td>
<td>Yes</td>
<td>Yes - Limited</td>
</tr>
<tr>
<td>User-defined, fined-grained access control policies based on user-defined roles/groups</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>• Based on XACML 1.0</td>
<td></td>
</tr>
<tr>
<td>Federated identity management and SSO</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>• Based on SAML 2.0</td>
<td></td>
</tr>
<tr>
<td>Audit trail</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

(table continues)
Table 1.1. (continued)

<table>
<thead>
<tr>
<th>Category/Feature</th>
<th>ebXML Registry 3.0</th>
<th>UDDI 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Protocol Bindings Support</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTP binding (REST)</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>• Allows any metadata or</td>
<td></td>
<td></td>
</tr>
<tr>
<td>artifact to be addressable via</td>
<td></td>
<td></td>
</tr>
<tr>
<td>an HTTP URL module access</td>
<td></td>
<td></td>
</tr>
<tr>
<td>control</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SOAP API binding</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Advanced Features</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Information Management</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Metadata Validation</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>• Unrestricted, may be used to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>validate any metadata type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• XSLT-based, content-specific</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cataloging of XML artifacts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Extensible via custom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>validation services (requires</td>
<td></td>
<td></td>
</tr>
<tr>
<td>programming)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Artifact Validation</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>• Unrestricted, may be used to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>validate any artifact type</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Extensible via custom</td>
<td></td>
<td></td>
</tr>
<tr>
<td>validation services (requires</td>
<td></td>
<td></td>
</tr>
<tr>
<td>programming)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Event Subscription and Notification</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ability to select events using</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>custom query</td>
<td>• Can use any user-defined query (see Discovery features)</td>
<td></td>
</tr>
<tr>
<td>Content-based event notification</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>• Can specify interest in</td>
<td></td>
<td></td>
</tr>
<tr>
<td>specific types of content</td>
<td></td>
<td></td>
</tr>
<tr>
<td>within specific types of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>artifacts</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delivery of notifications to</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>registered Web service</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delivery of notifications to</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>registered e-mail address</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Federation Support</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Federated Queries</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Object references between</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>any object in one registry to</td>
<td></td>
<td></td>
</tr>
<tr>
<td>any object in any other registry</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object replication from any</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>registry to any other registry</td>
<td>• Supports selective replication</td>
<td></td>
</tr>
<tr>
<td>• All data replicated across all</td>
<td>• All data replicated across all registries all the time</td>
<td></td>
</tr>
<tr>
<td>registries all the time</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Client SDK Support</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JAXR API</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>• JAXR level 0 and level 1 support</td>
<td></td>
<td></td>
</tr>
<tr>
<td>• JAXR level 0 support only</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(table continues)
Table 1.1. (continued)

<table>
<thead>
<tr>
<th>Category/Feature</th>
<th>ebXML Registry 3.0</th>
<th>UDDI 3.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Other Features</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Web Services Support</td>
<td>Yes • Several predefined, WSDL discovery parameterized stored queries (such as “Find all WSDLs that use a specified namespace or name space pattern”) • Automatic validation of WSDL upon publish to ensure compliance with WS-I Basic profile • Automatic cataloging of WSDL upon publish to support WSDL discovery</td>
<td>Yes • Limited to ~5 predefined discovery queries • No automatic validation of WSDL upon publish • No automatic cataloging, requires publisher to manually catalog the WSDL</td>
</tr>
<tr>
<td>Domain-Specific Profile Support</td>
<td>Yes • Extensibility features allow standard extensions to be defined for domain-specific use cases • Allows interoperability within and across domains • Examples of standard profile include: Web Services Profile, WSRP Profile • Open Geographic Information System (GIS) Profile • Institute of Health Education (IHE) – XML Data Representation (XDR) Profile • Health Level 7 (HL7) Conformance Profile</td>
<td>No</td>
</tr>
</tbody>
</table>

Figure 1.1. Web Service stack showing different protocols and layers.
Service discovery relies on service publication. In order for a Web Service to be discovered dynamically it is essential that it is published in a registry. A Web Service can then be discovered at design or run time using a public or private UDDI registry.

Web Service Flow Language (WSFL) is at the top of the stack. WSFL is used to describe how Web Services interact in work-flows and how they perform in case of service-to-service communication or collaborations.

1.3.1.2 HOW DOES A UDDI REGISTRY WORK?

Before we get into details of how does a UDDI registry actually works, it is essential to know what kind of information is stored in a UDDI registry. A UDDI registry is divided into two core components: the UDDI business registration component and the UDDI business registry component. Information provided by the business registration component can be categorized into three distinct categories: White Pages, Yellow Pages, and Green Pages. White Pages hold basic company information such as: business name, general business description, contact information, and a unique identifier such as a D-U-N-S numbers or tax IDS. D-U-N-S stands for “data universal numbering system”. These numbers are used to identify and track business records worldwide. This information helps others to discover Web Services based on business identifiers. Yellow Pages hold information that describes Web Services based on different categorizations and standard taxonomies. Three basic categories listed below are provided as a part of UDDI version 1:

1. North American Industry Classification System (NAICS) taxonomy - an industry classification
2. The Universal Standard Products and Services Code System (UNSPSC) taxonomy - a classification of products and services
3. The International Organization for Standardization Geographical taxonomy (ISO 3166)

Green pages provide actual technical information necessary for invoking Web Services. Green pages are binding information associated with Web Services and provide references to technical specifications implemented by Web Services. Green pages also provide pointers to various files and URL-based discovery mechanisms.
Information flow between a UDDI registry client and a UDDI service registry is shown in Figure 1.2. As mentioned in Section 1.3.1.1 messaging layer of Web Service stack uses SOAP, an XML-based messaging protocol, which is a messaging layer is built upon the network layer. All UDDI specifications APIs are defined in XML, wrapped in a SOAP message envelope, and sent over HTTP. A client SOAP request is transported over HTTP to a UDDI registry. The registry takes care of the client request, processes it, and an appropriate SOAP response is send back to the client. A client request for a registry information modification is required to be a secured and authenticated transaction. Using a cross-language, cross-platform approach such as XML and SOAP simplifies the problem of making systems at different companies compatible with one other.

Figure 1.2. Message flow between client and UDDI registry.

A high level view of how a UDDI registry works is shown in Figure 1.3. The figure illustrates how a UDDI registry is populated with data from a service provider and how it is discovered by a service consumer. The entire process is divided into four steps. The first step is publishing the specifications to a UDDI registry. These specifications are defined by software companies and standards bodies and are relevant to an industry or business and are registered in UDDI. These specifications are known as technical models or tModels. In step 2, companies populate a registry with information regarding business and services they have to offer. A UDDI registry assigns each a unique identifier called a Unique Universal Identifier or UUID. This is shown is step 3. AUDDI registry keeps track of all the
Figure 1.3. UDDI registry use case scenario.

information using UUID. Each UUID is guaranteed to be unique and never changes within a registry. In step 4, clients like marketplaces and search engines use a UDDI registry to discover services provided by other companies. At this point it is important to note that UDDI does not have a full-featured discovery service. Search for Web Services is restricted to Web Service names and classifications. However, a UDDI registry cannot search for a Web Services based on their cost or geographic location.

Business with such needs can locate potential partners through on-line marketplaces and search engines that use UDDI registries as a data source. Figure 1.4 depicts a relationship between the discovery layer defined by UDDI registries and the role of aggregations and specialized search capabilities that address business level searching. In the final step, businesses use data that is made available through step 4. Thus, a software service can invoke other Web Services using simple or dynamic integration.

1.3.1.3 TECHNICAL ARCHITECTURE

Figure 1.5 shows technical architecture of a UDDI registry. The architecture consists of a UDDI data model, UDDI API specifications, and UDDI cloud services. A UDDI data model is a XML scheme for describing businesses and Web Services. The data model consists of data structures such as businessEntity, businessService, bindingTemplate, tModel,
and publisherAssertion. UDDI API specification describes specification APIs for searching and publishing data to UDDI registries. A UDDI Business Registry (UBR), also known as a public cloud, is conceptually a single system build of multiple nodes. Data in nodes are synchronized through replications. The global grouping of operator nodes is jointly known as UBR. Contents are inserted in a UBR through one node and that node becomes the master owner of that content. Any subsequent update or deletion of data occurs at the operator node where data was inserted. A company can have private nodes that are not part of a UBR. Data in a private node is not synchronized with data in a UBR, thus data contained in a private node is distinct. While a UBR has widely accessible inquiry services, Web Services may be
published only by authenticated entities. Nodes can be made available by any business over the Internet.

### 1.3.1.4 UDDI Data Structure

As mentioned in Section 1.3.1.3 UDDI data model consists of different data structures: businessEntity, businessService, bindingTemplate, tModel, and publisherAssertion. Figure 1.6 shows the UDDI information model. A single businessEntity can hold multiple businessService objects. A businessService is physically contained in a businessEntity, whereas a businessService holds a businessKey as a reference to the businessEntity. Similarly, a single businessService can hold multiple businessTemplate objects as we will see later. A tModel is referenced by all the other data types.

![UDDI data structure diagram]

Figure 1.6. UDDI data structure.

A businessEntity object provides business information such as contact information, industry categorizations, business descriptions, and business identifiers. All of this information form white-pages information. A businessEntity object holds one or more businessService elements that represent services provided by a business. Yellow-pages
information is also stored in a businessEntity in the form of a categoryBag, as it categorizes businesses. A sample template for businessEntity is shows in Figure 1.7.

```xml
<businessEntity businessKey="uuid:...."
    operator="http://...">
    <authorizedName="..."/>
    <name/>
    <description/>
    <contacts>
        <contact useType="general info">
            <description/>
            <personName/>
            <phone/>
            <email/>
        </contact>
    </contacts>
    <businessServices>
    ...
</businessServices>
<identifierBag>
    <keyedReference
        tModelKey="UUID:...."
        name="D-U-N-S"
        value="123456789"/>
</identifierBag>
<categoryBag>
    <keyedReference
        tModelKey="UUID:...."
        name="NAICS"
        value="111330"/>
</categoryBag>
</businessEntity>
```

Figure 1.7. A businessEntity template.

A publisherAssertion object is used to establish a public relationship between two businessEntity objects. This relationship is visible to the public when both companies create the same relationship using two different publisherAssertion documents independently. The publisherAssertion structure consists of three elements: fromKey (the first business key), toKey (the second business key), and keyedReference. A KeyedReference designates asserted relationship type in terms of a keyName, keyValue pair within a tModel, and id uniquely referenced by a tModelKey. Figure 1.8 shows a template for publisherAssertion.
A businessService object represents a single logical web service provided by a businessEntity. It mostly contains white-pages information such as a name and a description along with unique service identifier. A businessService object holds yellow-pages information such as a categoryBag to categorize a service. It contains a list of bindingTemplates which in turn contains tModelInstance details encoding the technical service information, green-pages information on how to bind the Web Service. Figure 1.9 shows a sample template for a businessService object.

A bindingTemplate, as mentioned previously, holds green-pages information and stores pointers to technical descriptions and access point URLs. A businessService contains one or more bindingTemplate structures. A bindingTemplate also contains an optional description of a Web Service, and one or more tModel structures. It is important to understand here that bindingTemplates hold operational parameters for only one Web Service. In an organization, at any given time, there might be different versions of a Web Service in operation. In this case a businessService may have multiple bindingTemplates, a service may specify different implementations of the same service, and each bound to a different set of protocols or a different network address. Figure 1.10 shows a sample template for a bindingTemplate object.
tModel stands for technical model. tModels are used to describe compliance with a specification, a concept, or a shared design. A tModel has various uses in a UDDI registry. In case of WSDL mapping to UDDI, a tModel acts as an UDDI construct to refer an interface describing a WSDL document. In this case, a tModel is used for two purposes. First a tModel is used to represent technical specifications such as service types, bindings, and wire protocols. Second, a tModel is used to categorize technical specifications and services. Apart from this, a tModel is also used for defining find qualifiers for a UDDI Web Service. Find qualifiers are values used to modify the result of find APIs part of the UDDI Web Service APIs. For example, find_business results can be arranged according to the order in which they were published. A unique key, called a tModelKey, is assigned when a particular specification is registered in a UDDI registry as a tModel. This key is used by other UDDI entities to reference the tModel and to indicate compliance with the specification. Each specification tModel contains an overviewURL, Figure 1.11, which provides the address of
Figure 1.11. A tModel template.

the specification itself. Additional metadata can be associated with a specification tModel using any number of identifier and category systems. This is done with the help of a construct called an identifierBag, and categories are grouped in a construct called a categoryBag.

A categoryBag object helps to categories data. categoryBags can appear in a businessEntity, businessService or tModel. The different categorization systems used are listed in Table 1.2. Each category is represented by a tModel within a UDDI registry and has a unique UUID. The tModel name is the same across all UDDI registries.

An identifierBag acts like a keyword used to uniquely identify a business or a specification. identifier Bags can be attached to businessEntities and tModels. Table 1.3 lists two identifier schemas that can be used. As in case of a categoryBag, an identifierBag is represented as a tModel; the tModel name is unique throughout UDDI.

These bags contain a set of keyedReference elements. Each keyedReference specifies the tModelKey of the category system tModel and a name/value pair that specifies the metadata. For example, a keyedReference referencing the namespace category system can be used to specify a WSDL namespace. The metadata values specified in keyedReference elements can be used as selection criteria when searching a UDDI registry.

1.3.1.5 UDDI APIs

This section discusses how data is accessed in a UDDI registry. The UDDI specification describes all UDDI data accesses in terms of Web Services. The term “API Set”
Table 1.2. Categories Supported

<table>
<thead>
<tr>
<th>Taxonomy Name</th>
<th>&lt;tModel&gt; Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNSPSC</td>
<td>unspsc-org:unspsc:3-1</td>
<td>The Universal Standard Products and Services Classification. It is the first system to classify products and services for worldwide use. More information can be found at <a href="http://www.unspsc.org">http://www.unspsc.org</a>.</td>
</tr>
<tr>
<td>ISO 3166</td>
<td>iso-ch:3166:1999</td>
<td>International standard geographical regions. This taxonomy includes codes for countries and their administrative support staffs. More information can be found at <a href="http://www.din.de/gremien/nas/nabd/iso3166ma">http://www.din.de/gremien/nas/nabd/iso3166ma</a>.</td>
</tr>
<tr>
<td>Other</td>
<td>uddi-org:general_keywords</td>
<td>General-purpose associations that a business might want to make. This taxonomy allows operator nodes to promote invalid entries or entries that would otherwise be rejected by another classification system. There is no specification on how this works; it is operator-node specific.</td>
</tr>
</tbody>
</table>

Table 1.3. Identifier Types

<table>
<thead>
<tr>
<th>Identifier name</th>
<th>tModel name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>D-U-N-S</td>
<td>dnb-com:D-U-N-S</td>
<td>The Dun &amp; Bradstreet D-U-N-S number is a unique nine-digit identification sequence. This sequence provides unique identifiers for single business entities, while linking corporate family structures. More information can be found at <a href="http://www.d-u-n-s.com">http://www.d-u-n-s.com</a>.</td>
</tr>
<tr>
<td>ThomasNet</td>
<td>thomasnet-com:supplierID</td>
<td>This scheme provides identifiers for over 150,000 manufacturing and e-commerce companies worldwide. More information can be found at <a href="http://www.thomasnet.com">http://www.thomasnet.com</a>.</td>
</tr>
</tbody>
</table>

refers to these Web Services. An API Set is essentially a single Web Service containing related operations. UDDI specifies nine API Sets to be used for all forms of data access and data manipulation. These sets are:

- Inquiry
- Publication
The Inquiry API set allows entries to be located in a UDDI registry and information to be retrieved. It supports get operations and five find operations:

- find_binding
- find_business
- find_relatedBusinesses
- find_service
- find_tModel
- get_bindingDetails
- get_businessDetails
- get_operationalInfo
- get_serviceDetail
- get_tModelDetail

The Publication API set is used to add, update, and delete information in a UDDI registry. The API set is also used to manage a publisherAssertion. This API set supports:

- add_publisherAssertions
- delete_binding
- delete_business
- delete_publisherAssertions
- delete_service
- delete_tModel
- get_assertionStatusReport
- get_publisherAssertions
- get_registeredInfo
- save_service
- save_binding
- save_business
- save_tModel
- set_publisherAssertion

The Security API set is used to request an authentication token. Registry policy decides if authentication is required or not. An authentication token can be acquired using a call to get_authToken. This token is then passed to all the operations that require authentication. discard_authToken is used to inform a UDDI registry that the passed token is to be discarded, effectively ending the session.

The Replication API set is used to replicate information within different UDDI registry nodes. A logical UDDI registry may be composed of one or more physical nodes. Operations supported by this set are: get_changeRecord, notify_changeRecordsAvailable, do_ping, get_highWaterMarks, transfer_custody.

The Custody Transfer API set enables any node in a UDDI registry to transfer custody of one or more businessEntity or tModel from one node to another, as well as transferring the ownership of these structures from one publisher to another. Operations found in this set are discard_transferToken, get_transferToken, and transfer_entity.
The Subscription API set allows registry clients to register requests to track new and changed registry contents. save_subscription, get_subscription, and delete_subscription is used to create, modify, save, retrieve, and delete subscription. Operation get_subscriptionResults returns the subscription result. When get_subscriptionResults is executed it returns registry data that has changed for a particular subscription within a specified time period.

The Subscription Listener API Set can be implemented by a UDDI Registry client that is interested in being sent subscription results by the UDDI Registry. The UDDI Registry will invoke operation notify_subscriptionListener passing in data that has changed since notify_subscriptionListener was last invoked for the subscription.

The Validation API set is used to validate the core data type values before they are stored in a UDDI registry. It supports operation validate_values.

Finally the Validation Set Caching API set is used if all the values are cached by a UDDI registry, it supports the get_allValidValues operation.

### 1.3.1.6 UDDI EVOLUTION

UDDI version 1.0 was released in the year 2000. This created foundation for a registry of Internet based business services. The second Version 2.0 was released in the year 2001. This version aligned the UDDI specification with emerging Web Services standards. In this version taxonomy capabilities were substantially improved, this was done by adding three industry standard classification systems to UDDI. The classification systems added to UDDI are:

1. North American Industry Classification System (NAICS)
2. Universal Standard Products and Services Classification (UNSPSC)
3. The international standard for geographical regions, including codes for countries and first-level administrative subdivision of countries (ISO 3166)

A fourth classification system named “Other Taxonomy” was also added. This is a general-purpose classification system that allows key-value pairs to be used to represent company specific taxonomies. This UDDI version provides capability for two business entities to document their relationship with each other. Version 3.0 was released in year 2004, key features added to this version are listed below.
**Support for multi-registry environments** and movement of data between registries. The UDDI Version 3.0 introduces the concept of *Registry Affliction*. *Registry Affliction* means the ability of a UDDI registry to support different network topologies. The topologies such as a single stand alone registry approach to hierarchical, peer-based, delegated, and others are supported. Table 1.4 lists different flavors of a UDDI registry. Figure 1.12 represents conceptual illustration of Registry Affliction. The figure illustrates several modes of registry interaction enabled by Version 3. Through mechanisms such as publish/subscribe and replication among peer nodes, information in UDDI servers can be fully publish as in case of UBR, or kept semi-private as in case of affiliated registries show in the figure or kept fully private and isolated from the public domain as show by *Private Domain* in the Figure 1.12.

**Figure 1.12. Scenario depicting registry affliction.**

- **Subscription capabilities** provide a means for parties interested in (or dependent upon) any of the metadata contained within a UDDI registry to subscribe for change notifications.

- **Support for digital signatures** this feature was added to a UDDI registry to make data integrity an inherent feature of UDDI registries. This is important when UDDI registries from different organizations share SOA data.
Table 1.4. UDDI Registry Types

<table>
<thead>
<tr>
<th>Registry Type</th>
<th>Description</th>
<th>Example Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Corporate/Private</td>
<td>An internal registry, behind a firewall, that is isolated from the public network. Access to both administrative features and registry data is restricted. Data is not shared with other registries.</td>
<td>Enterprise Web Service registry</td>
</tr>
<tr>
<td>Affiliated</td>
<td>A registry deployed within a controlled environment, but with limited access by authorized clients. Administrative features may be delegated to trusted parties. Data Network may be shared with other registries in a controlled manner.</td>
<td>Trading Partner Network</td>
</tr>
<tr>
<td>Public</td>
<td>From an end-user’s perspective, a public registry appears to be a service in a cloud. Although administrative functions may be secured, access to the registry data itself is essentially open and public. Data may be shared or transferred among other registries, and content may or may not be moderated.</td>
<td>UDDI Business Registry (UBR)</td>
</tr>
</tbody>
</table>

1.3.2 Electronic Business Using eXtensible Markup Language

ebXML stands for Electronic Business using eXtensible Markup Language, is a standard that aims at providing XML-based infrastructure that would enable businesses to conduct business on internet in an interoperable, secure, and consistent manner. Details about ebXML specifications are explored here in this section.

1.3.2.1 INTRODUCTION TO EBXML

ebXML is a collection of specifications that enables enterprises of any type located anywhere to conduct business over the Internet. ebXML provides a standard method to exchange business messages, conduct trading relationships, communicate data in common terms and define and register business processes. One of the most powerful features of ebXML is its ability to achieve ad hoc business interactions. For example, if two businesses arrange to trade electronically, they must pay the price for the infrastructure and the software, and to formalize the business interactions and policies. Over time, if new businesses emerge that offer lower costs the mode of interaction and the technologies required become an economic barrier. ebXML provides an ideal solution to such problems as it is built around XML, SOAP, HTTP, and SMTP, all open standards. ebXML provides a complete framework to achieve ad hoc business interaction. At the heart of ebXML is a powerful system of
registries and distributed repositories. Along with the registry, there are specifications for the messaging layer as well as for business process specifications and collaboration information. The set of concrete ebXML specifications maps to these concepts as follows:

- Centralized Shared Registry: Registry Information Model, Registry Services Specification (ebRIM, ebRS)
- Messaging: Message Services Specification (ebMS)

Figure 1.13 depicts a high-level use case scenario of two trading partners: Company A and Company B, engaged in a simple business transaction. To start with Company A reviews contents of an ebXML registry, mainly the Core Library, as shown in step 1, this allows Company A to determine what is required for their own local implementations to be compatible with ebXML standards, as shown in step 2. In step 3, Company A submits its own business profile to the ebXML registry, the business profile states company capabilities, constraints and business scenarios it supports. These business scenarios are XML versions of the business processes and associated information in which the company is able to engage.

After receiving verification, that the format and usage of a business scenario is correct, the ebXML registry sends an acknowledgment to Company A, step 3. Company B discovers the business scenarios supported by Company A through ebXML registry as shown in step 4, it then sends a request to Company A stating that it would like to engage in a business scenario using ebXML, step 5. As a part of step 5, before engaging in the business scenario Company B submits a proposed business arrangement directly to Company A. The proposed business arrangement outlines mutually agreed upon business scenarios and specific agreements. The business arrangement also contains information pertaining to messaging requirements for business transactions, contingency plans, and security-related requirements. If Company A accepts the business agreement, then both Company A and B are now ready to engage in eBusiness using ebXML, step 6.

The business scenario described above highlights following concepts:

- A standard mechanism for describing business processes and associated information model.
Figure 1.13. ebXML business scenario.

- Discovery of information about each participant including:
  - Business Processes supported.
  - Business Service Interfaces offered in support of the business process.
  - Business Messages that are exchanged between their respective Business Service Interfaces.
  - Technical configuration of supported transport, security and encoding protocols.
- A mechanism for registering and storing business processes and information models so they can be shared and reused.
- A mechanism for registering information so that it may be discovered and retrieved.
- A mechanism for describing the execution of a mutually agreed upon business arrangement which can be obtained from information provided by each participant. (Collaboration Protocol Agreement – CPA).
- A standardized business Messaging Service framework that enables interoperable, secure and reliable exchange of messages between trading partners.
1.3.2.2 ebXML Architecture Overview

EbXML is designed to meet three basic concepts: provide an infrastructure that ensures data communication interoperability; provide a semantics framework that ensures commercial interoperability; and provide a mechanism that allows enterprises to find each other, agree to become trading partners and conduct business with each other.

Infrastructure to ensure data communication interoperability is provided through, a standard message transport mechanism with a well defined interface, packaging rules, and a predictable delivery and security model, a ‘business service interface’ that handles incoming and outgoing messages at either end of the transport.

Semantic Framework to ensure commercial interoperability is provided through: metamodel for defining business process and information models, set of re-useable business logic based on core components that reflect common business processes, and process for defining actual message structures and definitions as they relate to the activities in the Business Process model.

Mechanism to allow enterprises to find each other, agree to establish business relationships, and conduct business, is provided through: shared repository where enterprises can register and discover each other’s business services via partner profile information. EbXML provides a process for defining and agreeing to a formal Collaboration Protocol Agreement (CPA).

The technical architecture is composed of five main area of emphasis (Figure 1.14):

- Business Process and Information Model
- Company Profiles
- Messaging Services
- Registry & Repository
- Collaborative Partner Agreements

The Business Process models define how business processes are described and can be represented using modeling tools. The specification for business process definition enables an organization to express its business processes so that they are understandable by other organizations. This enables the integration of business processes within a company, or between companies.
The Information models define reusable components that can be applied in a standard way within a business context. They are defined using identity items that are common across all businesses. This enables users to define data that is meaningful to their business while also maintaining interoperability with other business applications.

The ebXML Messaging Service specification defines the set of services and protocols that enables electronic business applications to exchange data. The specification allows any application-level protocol to be used. These can include common protocols such as SMTP, HTTP, and FTP. Different cryptographic techniques can be used to implement strong security. For example, secure protocols such as HTTPS can be used to guarantee confidentiality.

In addition, digital signatures can be applied to individual messages or a group of related messages to guarantee authenticity.
The Registry and Repository provides a number of key functions. For the user it stores company profiles and Trading Partner specifications. These give access to specific business processes and information models to allow updates and additions over time. For the application developer it will store not only the final business process definitions, but also a library of core components.

The Collaborative Partner Agreement defines the technical parameters of the Collaborative Partner Profiles (CPP) and Collaborative Partner Agreements (CPA). This captures critical information for communications between applications and business processes and also records specific technical parameters for conducting electronic business.

We now explore a business use case to see how different parties interact to perform eBusiness using ebXML. See Figure 1.15.

- Step 1 involves designing and registering business processes and information models. In this the implementer browses the repository for appropriate business processes, or for the process the intended partner is registered to support.

- In Step 2 an implementer buys, builds, or configures application(s) capable of participating in the selected business process. Then the implementer registers his (software’s) capability to participate, in the form of a Collaborative Partner Profile.

- Step 3 is concerned with negotiation and defining a Collaborative Partner Agreement (CPA). Two parties negotiate technical details and/or functional overrides, and draw up the result in the form of a CPA.

- Step 4 involves message exchanging between the parties. The parties (software) send and receive ebXML messages containing ebXML business documents, over the secure and reliable ebXML Messaging Service.

### 1.3.2.3 EBXML REGISTRY

The ebXML Registry architecture shown in Figure 1.16 consists of: ebXML Registry Clients, Client API, ebXML Registry Service Interfaces, Service Interface protocol bindings, authentication, authorization, metadata registry and content repository.

A Registry Client is a software program that interacts with the registry using registry protocols. The Registry Client may be a Graphical User Interface (GUI), software service or agent. The Registry Client typically accesses a registry using SOAP 1.1 with Attachments [SwA] protocol. A Registry client may access a registry interface directly or may use a registry client API such as the Java API for XML Registries [JAXR] to access a registry.
Figure 1.15. ebXML use case.

Client APIs such as [JAXR] provide programming convenience and are typically specific to a programming language.

The ebXML Registry Service consists of a robust set of interfaces designed to fundamentally manage objects and inquiries associated with a ebXML Registry. The two primary interfaces provided by Registry Service are: LifeCycleManager and QueryManager interface. See Figure 1.17.
Figure 1.16. ebXML registry architecture.

Figure 1.17. Registry service and client interaction1.3.2.4 ebXML registry information model.
• A LifeCycleManager interface provides a collection of operations for end-to-end lifecycle management of metadata and content within a registry. This includes publishing, update, approval and deletion of metadata and content. Referred in Figure 1.17 as LM.

• A QueryManager interface provides a collection of operations for discovery and retrieval of metadata and content within a registry. Referred in Figure 1.17 as QM.

Service Interface protocol binding specifications defines following concrete protocol bindings for abstract service interfaces for a ebXML Registry:

• SOAP Binding that allows a Registry Client to access a registry using SOAP 1.1 with Attachments (SwA).

• HTTP Binding that allows a Web Browser client to access a registry using HTTP 1.1 protocol.

Registry clients are authenticated by a registry to determine the identity associated with them. Typically, this is a identity of a user this is associated with a registry client. Once a registry determines the identity it performs authorization and access control checks before permitting a client’s request to be processed.

An ebXML Registry is both a registry of metadata and a repository of content. A typical ebXML Registry implementation uses some form of persistent data store such as a database to store its metadata and content. Architecturally, registry is distinct from a repository. However, all access to a registry as well as a repository is through the operations defined by the Registry Service interfaces.

The registry information model defined in ebRIM defines: what types of objects are stored in an ebXML registry, and how stored objects are organized in a ebXML registry. The core information model used by an ebXML registry is hierarchical, Figure 1.18 taken from [4] shows a high level public view of objects in a registry and their relationships as a UML Class Diagram.

The RegistryObject class is an abstract base class used by most classes in the model. It provides minimal metadata for registry objects. It also provides methods for accessing related objects that provide additional dynamic metadata for the registry object.

Slot instances provide a dynamic way to add arbitrary attributes to RegistryObject instances. This ability to add attributes dynamically to RegistryObject instances enables extensibility within the Registry Information Model. For example, if a company wants to add
a "copyright" attribute to each RegistryObject instance that it submits, it can do so by adding a slot with name "copyright" and value containing the copyrights statement.

Association instances are RegistryObject instances that are used to define many-to-many associations between objects in the information model. Table 1.5 types in an ebXML Registries.

ExternalIdentifier instances provide additional identifier information to RegistryObject instance, such as DUNS number, Social Security Number, or an alias name of the organization.

ExternalLink instances are RegistryObject instances that model a named URI to content that is not managed by the Registry. Unlike managed content, such external content may change or be deleted at any time without the knowledge of the Registry.
Table 1.5. Pre-Defined Associations

<table>
<thead>
<tr>
<th>Related To</th>
<th>Relates Registry Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>HasMember</td>
<td>Defines the members of the Registry Package</td>
</tr>
<tr>
<td>EquivalentTo</td>
<td>Defines that Source Registry Object is equivalent to the Target Registry Object</td>
</tr>
<tr>
<td>Extends</td>
<td>Defines that Source Registry Object inherits from the Target Registry Object</td>
</tr>
<tr>
<td>Implements</td>
<td>Defines that Source Registry Object implements the functionality defined by the Target Registry Object</td>
</tr>
<tr>
<td>InstanceOf</td>
<td>Defines that Source Registry Object is an instance of the Target Registry Object</td>
</tr>
</tbody>
</table>

ClassificationScheme instances are RegistryEntry instances that describe a structured way to classify or categorize RegistryObject instances. The structure of the classification scheme may be defined internal or external to the registry, resulting in a distinction between internal and external classification schemes.

ClassificationNode instances are RegistryObject instances that are used to define tree structures under a ClassificationScheme, where each node in the tree is a ClassificationNode and the root is the ClassificationScheme. Classification trees constructed with ClassificationNodes are used to define the structure of Classification schemes or ontology’s.

RegistryPackage instances are RegistryEntry instances that group logically related RegistryObject instances together.

AuditableEvent instances are RegistryObject instances that are used to provide an audit trail for RegistryObject instances.

User instances are RegistryObject instances that are used to provide information about registered users within the Registry. User objects are used in audit trail for RegistryObject instances.

PostalAddress is a simple reusable Entity Class that defines attributes of a postal address.

EmailAddress is a simple reusable Entity Class that defines attributes of an email address.

Organization instances are RegistryObject instances that provide information on organizations such as a Submitting Organization. Each Organization instance may have a reference to a parent Organization.
Service instances are RegistryEntry instances that provide information on services (e.g., web services).

ServiceBinding instances are RegistryObject instances that represent technical information on a specific way to access a specific interface offered by a Service instance. A Service has a collection of ServiceBindings.

A SpecificationLink provides the linkage between a ServiceBinding and one of its technical specifications that describes how to use the service with that ServiceBinding. For example, a ServiceBinding may have a SpecificationLink instance that describes how to access the service using a technical specification in the form of a WSDL document or a CORBA IDL document.

### 1.3.2.4 ebXML Registry Interfaces

The ebXML Registry Architecture is defined in terms of registry service and registry client. Registry service has two main interfaces for managing objects in the information model:

- LifeCycleManager Interface (LCM)
- QueryManager Interface (QM)

LifeCycleManager interface is exposed by Registry Service it implements the object life cycle management functionality of a Registry. Its methods are invoked by a registry client, to modify registry objects. For example, a client may use this interface to submit objects, to classify and associate objects, and to deprecate or remove objects. Access to LCM is restricted, and requires authentication and authorization. All LCM actions are logged in an audit trail; it supports automatic versioning of objects. A typical object life cycle is shown in Figure 1.19. Table 1.6 lists methods supported by LifeCycleManager.

QueryManager interface is exposed by a Registry implements the Query management service of a Registry. Its methods are invoked by a registry client. For example, a client may use this interface to browse registry content or to perform drill down queries or ad hoc queries on registry content. The QueryManager interface provides access to browsing and discovery capabilities using the standard AdhocQuery protocol defined by ebRS[]. Query syntax such as: SQL 92, and XML Filter Query syntax are supported. Different query format are supported by QueryManager such as:
Table 1.6. Methods Supported by LifeCycleManager

<table>
<thead>
<tr>
<th>RegistryResponse</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegistryResponse</td>
<td>approveObjects</td>
<td>Approves one or more previously submitted objects.</td>
</tr>
<tr>
<td>RegistryResponse</td>
<td>deprecateObjects</td>
<td>Deprecates one or more previously submitted objects.</td>
</tr>
<tr>
<td>RegistryResponse</td>
<td>removeObjects</td>
<td>Removes one or more previously submitted objects from the Registry.</td>
</tr>
<tr>
<td>RegistryResponse</td>
<td>submitObjects</td>
<td>Submits one or more objects and possibly related metadata such as Associations and Classifications.</td>
</tr>
<tr>
<td>RegistryResponse</td>
<td>updateObjects</td>
<td>Updates one or more previously submitted objects.</td>
</tr>
<tr>
<td>RegistryResponse</td>
<td>addSlots</td>
<td>Add slots to one or more registry entries.</td>
</tr>
<tr>
<td>RegistryResponse</td>
<td>removeSlots</td>
<td>Remove specified slots from one or more registry entries.</td>
</tr>
</tbody>
</table>

- Pre-defined or application specific
- Ad hoc queries using predicate conjunction, disjunction, negations etc.
- Content based queries based on metadata that was cataloged automatically from content. Examples: Find me all images that are color images or Find me all images that are greater than 10cm x 15cm.
- Parameterized and stored queries within the registry server as metadata of type AdhocQuery. Web UI and Java UI allow mapping stored parameterized queries to Search Forms where each query parameter maps to a form field.
- AdhocQueryRequest contains: Standard SQL-92 query – Normative SQL Schema, Parameterized query invocation, XML Filter Query, and Iterative query parameters: startIndex, maxResults
- AdhocQueryResponse contains: objects matched by query, and Iterative query parameters: startIndex, totalResultsCount

Unauthenticated clients can access the QueryManager interface. However, such clients are only able to see public content and metadata within the registry.

Table 1.7 lists the methods supported by QueryManager.

<table>
<thead>
<tr>
<th>Table 1.7. Methods Supported by QueryManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegistryResponse</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

1.3.2.5 CONTENT BASED NOTIFICATION

One of the features of ebXML registry is content based notification. A client can create subscription that can notify the client of any changes in the registry contents. A subscription created by the used contents: a selector query that selects events of interest and actions that deliver notification to events to interested parties. See Figure 1.20.

![Figure 1.20. Scenario depicting event based notification.](image)

In return a response send by a registry contains: Registry objects that were changed and matched client notification requirements.
1.4 Previous Work

Previous work by Keidl et al. investigates techniques for dynamic service selection in Flexible and Reliable Web Service Execution [11]. It describes technique for dynamic service selection at run time by specifying different constraints and using semantic classification. They suggest a load balancing technique using automatic service replication. The service constraints are of two types: hard constraints or conditions that must be fulfilled and soft constraints or preferences that should be fulfilled. One such constraint is metadata constraint used to filter tModel [2] instances returned by UDDI registry. A dispatcher service that acts as a proxy for an arbitrary service is used for load balancing. The dispatcher monitors a set of service hosts and it dispatches a service request to a service instance running on a service host with least load. In case if all service hosts are experiencing heavy load, the dispatcher generates a new service instance on a service host with low load.

Three models presented by Kee-Leong et al. in [12] enable customization of UDDI registry query results using static or dynamic parameters. The parameters could be predefined or real-time changing parameters. All models consist of a user interface, a UDDI proxy and a UDDI server. Load balancing is achieved by keeping user interface and UDDI proxy on different servers. UDDI server fully supports UDDI registry like jUDDI, Microsoft Enterprise UDDI Services, etc. UDDI proxy acts as an intermediary between User interface and UDDI server. UDDI proxy is responsible for rearranging UDDI query results using predefined criteria. UDDI proxy provides a mechanism to automate the criteria used to order the results. The first proposed model comprises of a UDDI server and a UDDI proxy. The ranking parameters are stored in the server using tModel [2] definition to describe parameter information. A parameter can be referenced in businessEntity or its service through identifierBag [2] structure. UDDI proxy then uses these parameters to rank query result if ranking is on. The model proposed next uses server logs to create dynamic parameter values by using log files of SOAP server, application server and UDDI server. The required functionality to search, match and count is carried out by UDDI proxy. The final model makes use of external files to specify ranking criteria. One file for each parameter type is used. The files are kept directly accessible to UDDI proxy.

Lages et al. presents SATYA: A reputation-based approach for service discovery and selection in service oriented architecture [13]. SATYA computes a reputation value for
service providers, based on values gathered from monitoring entities and service consumers. By assigning Web Services a trustable reputation value, SATYA provides a way to rank Web registry query results, thus providing additional information that can be used during service discovery. Using SATYA, a consumer is able to search a service registry for providers that not only have the best QoS (Quality of Service) value for a given matrices, and highest reputation value for a given metric, but also for a provider that has good balance between reputation and QoS values.

Customizable descriptions and dynamic discovery for Web Services [14] presented by Kim and Karp, describes a framework based on notations used for dynamic environments like Web Services for resources discovery. The system incorporates features such as attribute based matching rules and corresponding constraint based searching capability, attributes with dynamic values, and active queries.

A lookup system for peer-to-peer networks is proposed by Sioutas et al. in [15]. The system works with a peer that hosts Web Service information. This information can be efficiently located using a Balanced Distributed Tree (BDT). A BDT provides support for different queries like exact match queries and range match queries. A BDT is flexible and updates queries as nodes join and leave the system.

A load balancing technique using self adapting Web Services is suggested by Porter et al. in [16]. They propose a design for a self adaptive Web Service using static monitoring. This involves a statistical technique for monitoring requests and their effects on multi tire systems, component monitoring, a visualization tool that summarizes statistical results that facilitate human decision, and an efficient technique for operators to invoke admission control on different requests based on statistical findings.

A service discovery mechanism with load balancing in decentralized peer-to-peer networks is suggested by Chen et al. in [17]. They propose a discovery mechanism that involves creating an agent called a P2P registry (“P2PReg”) as a middleware within DNS and peers. Using this schema, a peer is able to publish its services and discover other services automatically through current DNS using P2PReg. Searches are performed in least time, maintaining balanced load in P2P system through replication.

On similar lines as of [14], Surgihalli and Vidyasankar in [18] suggests a lazy replication schema for UDDI registries. According to the schema, the update transactions are
executed at a primary site, committed, and the updates are transmitted to other sites asynchronously. This facilitates a quick response and scalability. They also discuss, a session guarantee mechanism needed to ensure consistent access. A replication and synchronization scheme used to keep a UDDI registry up-to-date in a distributed and an ad-hoc network is proposed by [16], which also leads to a better response time.

Li et al. proposes a Rough Sets based Search Engine for Web Service discovery (ROSSE) [19]. One of the highlights of ROSSE is its capability to take into account uncertainty of service properties while matching services.

An extension to existing UDDI registry search is proposed by Zou et al. [20] to improve category-based Web Service retrieval processes. The refinement is based on operations provided by a service. A service is marked as positive only if at least one operation provided by the service can be used by the user.

A framework for “Architecture driven Web Service discovery” (ASD) is proposed by Li et al. [21] is concerned with discovery of Web Services that can provide required functionality at the same time satisfy different constraints and properties that were specified during design phase. The framework consists of a query extractor which derives queries from UML design models of a service centric system and other documents. It also consists of a query execution engine that executes these queries against services registries.

Kozlenkov et al. proposes a self configuration approach of service based systems for service level agreement (SLA) and resource optimization [22]. The issues that are being addressed here are rapid and continue changes in user requirements, runtime system resources, and other environmental factors. The scheme involves automatic system reconfiguration through continuous monitoring, analyzing, planning, and execution.

On similar lines as [12], dynamic web service discovery architecture for peer-to-peer network is proposed by Mehlorn et al. called NIPPERS (Network for InterPolated PeERS) [23]. The proposed solution provides support for different queries such as exact match queries and range queries. A “NIPPERS” is an extension of Interpolation Search Data Structure [21].

UDDIe is an extension to UDDI registry proposed by Ali et al. [24]. As a part of the extension, a new notion of “blue pages” is introduce. This enables recording of user defined properties associated with a Web Service. UDDIe adds to the existing search capabilities of a
UDDI registry by enabling searching on user recorded properties. The properties could be such as CPU load, network bandwidth, etc.

A UDDI registry extension that takes into account quality of service QoS is suggested by Ran [25]. The model works with both functional as well as non-functional properties of a Web Service during service discovery. Using this model, a consumer search of a UDDI registry for a Web Service is complemented with additional constrains like response time or other quality of service constraints. A new architecture that supports the mapping of requirements regarding network performance from higher layer to network layer is proposed by Tian et al. [26]. They introduce QoSBroker, which acts as a proxy between a UDDI registry and a consumer. The broker searches for appropriate Web Services that meet not only the functional but also QoS requirements for the consumer. The architecture also provides tools using which the consumer can monitor the server runtime performance giving instant QoS feedback to the consumer.

A Web Service discovery schema that takes into account a Web Service’s context is proposed by Lee et al. [27] and Mostefaoui et al. [28]. Different context parameters are grouped under the name of Context Attribute. For example, server load can be one such context attribute that can influence Web Service discovery.

Though major work has been done in the field of Web Service discovery that takes into account not only functional aspects of a Web Service but also QoS aspects, all this work is centered on UDDI or peer-to-peer networks. In this thesis we work with freebXML [9] which is an open source implementation of the ebXML specification and standards. We propose a schema for the ebXML registry that manages the service bindings of a Web Service deployed across multiple hosts such that the URI returned by the registry can resolve to a host that satisfies different system constraints.
CHAPTER 2

THE FREEBXML REGISTRY/REPOSITORY

2.1 THE FREEBXML REGISTRY

freebXML [9] registry is an open source implementation of the ebXML registry standard. freebXML is first to deliver a feature complete and specification compliant implementation of latest ebXML Registry standard.

2.2 FREEBXML REGISTRY ARCHITECTURE

The layered architecture of a freebXML registry is shown in Figure 2.1 [9]. The architecture is divided into three layers: the registry client layer, the registry client API layer, and the freebXML registry server layer. The functionality and purpose of each of these layers is explained next.

2.2.1 The Registry Client Layer

The registry client layer is used to access registry contents and handle administrative functions. A registry client can access registry contents through Web browser, Web UI, Java UI, and a SOAP Client. Administrative tools are also part of this layer.

Using the HTTP protocol, a client connects with a registry through a web browser. The HTTP GET method is used to access and modify registry contents. The Get method works just as defined by the HTTP Protocol Binding [3] in case of ebXML registries. The registry can also be accessed using a Web UI which is hosted by a Web container and can be accessed through a Web browser. A Web UI uses JAXR APIs provided by JAXR providers to connect and modify registry contents. The freebXML registry supports a special feature called the \textit{localCall} mode. By default it is set to \textit{false}; in this case all communication between JAXR provider and registry server is through the ebXML Registry protocol over SOAP 1.1 with attachments [5]. When localCall mode is set to \textit{true} then an optimization occurs which bypasses SOAP communication and instead communicates via local method calls directly from the JAXR Provider to the QueryManager and LifeCycleManager.
interfaces [4] defined by the registry server. localCall is set to true in case of Web UI. A Java UI is a fat registry client that also uses JAXR APIs to communicate with the registry. This UI is provided as part of freebXML source code. A SOAP client is another way to access registry contents, not shown in figure. Command line utilities are provided as part of administrative tools. These tools communicate with the registry using JAXR.

2.2.2 The Registry Client API Layer

JAXR APIs are part of the registry client APIs. JAXR [29] is Java APIs for XML registries. These APIs provide a uniform way to access business registries based on different open standards like ebXML and industry initiatives like UDDI. A JAXR architecture [29] consists of a JAXR client and JAXR provider (Figure 2.2). A JAXR client is any program that uses JAXR APIs to access a Web registry.
A JAXR provider on the other hand provides implementation of the JAXR AP, which in turn provides access to a specific registry provider or to a class of registry providers. The freebXML Registry provides its own JAXR Provider which is an advanced implementation of JAXR API.

The JAXR Provider uses JAXB to generate ebXML Registry Protocol request and response messages used to communicate with the registry server, shown in Figure 2.3 [9].

### 2.2.3 The freebXML Registry Server

In Figure 2.1, the freebXML Registry server is located below the JAXR Provider. The server is typically deployed and runs within a Web container such as Tomcat or a commercial J2EE server such as Sun Java Enterprise System Application Server, Glassfish [30]. The freebXML Registry server is divided into nine parts: registry and repository, HTTP interface, SOAP interface, QueryManager interface, LifeCycleManager interface, Authentication, Authorization, Persistence and the Relational Database Management System (RDBMS).

The Repository holds contents such as XML Schemas, XML Instances, WSDL specifications, GIF images, etc. A content instance in a repository has an associated metadata instance in a registry. A registry contains metadata instances. A metadata instance describes a content instance or another metadata instance. Metadata instances enable content discovery.
Classified as mandatory by the ebXML Registry standard [5], the freebXML Registry Server exposes an HTTP interface that enables accessing registry content and metadata using HTTP GET via an HTTP URL. The HTTP GET access only supports search queries and does not support functionality to publish or modify registry contents because the freebXML Registry Server HTTP interface provides bindings to only the QueryManager [4] interface of the registry. The HTTP interface implementation within the registry server uses JAXB to process registry protocol request messages and to generate HTTP response message containing the object instances matching the query.

The freebXML Registry Server exposes a SOAP 1.1 with Attachments interface to both the QueryManager and LifeCycleManager interfaces of the registry. The SOAP Interface uses SAAJ (SOAP with Attachment API for Java) for processing SOAP messages and JAXB to process registry request and response messages.

The QueryManager interface provides access to browsing and discovery capabilities using the standard AdhocQuery protocol defined by ebRS [5]. Query syntax such as SQL 92 (preferred query syntax, used pervasively in freebXML Registry), and XML Filter Query syntax (discouraged, used rarely in freebXML Registry) are supported. Different query formats are supported by QueryManager such as:
• Pre-defined or application specific.
• Ad hoc queries using predicate conjunction, disjunction, negations, etc.
• Content based queries based on metadata that was cataloged automatically from content.
• Parameterized and stored queries within the registry server as metadata of type AdhocQuery [5]. Web UI and Java UI allow mapping stored parameterized queries to Search Forms where each query parameter maps to a form field.

Unauthenticated clients can access the QueryManager interface. However, such clients are only able to see public content and metadata within the registry.

The LifeCycleManager interface helps publish contents to a registry using various standard ebXML Registry protocols defined by ebRS [5]. SubmitObjectsRequest Protocol is used to publish content and metadata to a registry. Registry contents are updated using the UpdateObjects Request Protocol. The submitted object and metadata are approved using the ApproveObjectsRequestProtocol. An approved registry object is deprecated using the DeprecateObjectsRequestProtocol; the action can be reversed using the DeprecateObjectsRequestProtocol. It is possible to relocate contents from one registry to another using the RelocateObjectsRequestProtocol. Finally an object or metadata that is no longer required can be removed from the registry using the RemoveObjectsRequestProtocol, Figure 2.4 shows an objects life cycle. Note that unauthenticated clients cannot access the LifeCycleManager interface as it requires authenticated access.

freebXML registry server first authenticate an incoming request, next it performs authorization checks for the request using the authentication credentials. Authorization is done based on XACML (eXtensible Access Control Markup Language) [27] access control policies. XACML is an XML-based standard that provides a policy language and an access control decision/response language for managing access to resources. Authorization policies are defined with a collection of rules. Both rules and requests use Subjects, Resources and Actions attributes. A subject attribute is based on user authentication and includes: Subject id (user identity), SubjectRoles (roles assigned to the user), and SubjectGroups (groups to which a user has been assigned). Resource attributes include attributes defined by ebRIM. Action attributes include attributes defined by ebRIM as an action. This is an id that identified an action such as create, update, deprecate, delete, etc.
The abstract Persistence layer is responsible for storing content and metadata to a persistent store and providing access to it. It is made up of the SQLPersistenceManager interface and SQLPersistenceManagerImpl, DAO classes, RepositoryManager interface and RepositoryManagerImpl. The SQLPersistenceManager interface provides an abstract interface for using an RDBMS and SQL-92 to provide persistence for the registry server. It has an implementation class SQLPersistenceManagerImpl which provides a concrete implementation over RDBMS using JDBC 3.0. Not shown in Figure 2.1 are classes named XxxDAO where Xxx maps to a class defined by ebRIM. For example ServiceDAO maps to Service class. These classes provide support for the corresponding RIM class using an RDBMS and JDBC 3.0. The RepositoryManager interface provides an abstract interface for using an RDBMS and SQL-92 to provide persistence for repository items within the registry server. It has an implementation class RepositoryManagerImpl which provides a concrete implementation over RDBMS using JDBC 3.0.

The relational database stores content and metadata for a freebXML Registry while providing ACID (atomicity, consistency, isolation, durability) transactional characteristics. The registry server’s Persistence layer uses JDBC 3.0 to access the RDBMS. The default database for freebXML Registry is Apache Derby. By default, Derby runs in embedded mode within the same JVM as the registry server.
2.3 Features and Benefits of Using freebXML

freebXML provides a standard variety of features such as:

- Web Service publish and discovery
- Web content management (SOAP and HTTP)
- Web content discovery standard taxonomies
- SQL, XML filter query syntax
- Content validation
- Cataloging
- Classification
- Federated content management and content-based event notification
- Authorization control (XML-DSIG)
- Role based access control (XACML)
- Audit trail/event log
- User defined taxonomies
- Extensible information model
- Object oriented architecture
- Internationalization features
CHAPTER 3

A LOAD BALANCING SCHEME FOR THE FREEBXL REGISTRY/REPOSITORY

3.1 Motivation

Many-Task Computing (MTC) [31] emphasizes using large numbers of computing resources over short periods of time to accomplish computational tasks, where primary metrics are measured in seconds. Tasks executed by MTC applications can be individually scheduled on many different computing resources to achieve larger application goals. There are many challenges that can impair efficiency and utilization in running applications on large-scale systems, such as local resource scalability, efficient utilization of resources, data management, I/O management, and application scalability [32]. Lack of proper scheduling of jobs on different resources can impair the overall performance of a system. Some resources may become overwhelmed while others may starve for lack of jobs. Proper load balancing among distributed computing resources is thus paramount for effective job execution.

A MTC application can be deployed in a distributed environment using Web Services deployed on multiple hosts. An ebXML registry with a built-in load balancing capability will go a long way to ensure proper load distribution on various hosts resulting in improved system performance.

3.2 Modifications to FreebXML

In this work we propose a load balancing schema for ebXML registries. We have used and extended the freebXML registry; freebXML is an open source implementation of the ebXML registry standard. During Web Service discovery, a freebXML registry returns the access URIs associated with a service binding to allow dynamic discovery and invocation. This usually restricts a calling process to a Web Service invocation on one host. This behavior results in increased load on one particular host, while the other hosts on which the Web Service has been deployed may starve for computation. A desired scenario under such conditions would be uniform distribution of load on all hosts. The idea is to order the
access URIs that are returned by a freebXML registry in accordance to different system constraints like CPU load, available swap memory, physical memory, and time of day. This calls for periodic performance monitoring on different hosts and collection of data for different performance parameters like current CPU load, available memory, and available swap memory, etc. However, this solves only one part of the problem. What is also needed is a way to communicate periodically collected information to a freebXML registry. Both of these problems are solved using a Web Service named NodeStatus. The NodeStatus Web Service is deployed on multiple hosts. At the same time it is published to a freebXML registry along with end point URLs of different hosts on which the Web Service has been deployed. The registry then periodically invokes the NodeStatus Web Service on all the hosts that deploys the Web Service and collects performance data. The data is collected every 25 seconds; however this period can be reconfigured by the freebXML administrator. The duration at which the data is being collected at present was decided upon after observing the frequency of load change on our system. The data obtained is stored in a database table called NodeStatus. Figure 3.1 shows class TimeHits which is responsible for two things: to invoke the NodeStatus Web Service periodically and to collect and store current host performance data into the database.

<table>
<thead>
<tr>
<th>TimeHits</th>
</tr>
</thead>
<tbody>
<tr>
<td>-contextObj : ServerRequestContext</td>
</tr>
<tr>
<td>-java.util.Timer : fTimer</td>
</tr>
<tr>
<td>-timerInstance : static TimeHits</td>
</tr>
<tr>
<td>+getInstance() : TimeHits</td>
</tr>
<tr>
<td>-TimeHits()</td>
</tr>
<tr>
<td>+makeHist() : void</td>
</tr>
<tr>
<td>+Task() : Class</td>
</tr>
</tbody>
</table>

**Figure 3.1. Class diagram for TimeHits class.**

Figure 3.2 shows the NodeState table schema. NodeState is used to store constraint information that is retrieved periodically. At present the system supports four performance constraints: CPU load, swap-memory, physical memory, and time of day. CPU load is the number of processes waiting in the ready to execute queue. Field HOST acts as a primary key and holds the host name from the access URI. LOAD is used to store current CPU load. Fields MEMORY and SWAP MEMORY are used to store current physical memory and swap memory quantities, respectively, available on the host. Note that there is no field in
Figure 3.2. Table NodeState

Table used to store the state of a host machine.

table NodeState for the time of day constraint, which specifies a time frame during which a Web Service is available for operation. This constraint is handled at run time by the registry by comparing the current time and the constraint time window.

All constraints or combination of constraints can be applied to different Web Services. The constraints are published to a registry whenever a Web Service is published to a freebXML registry. These constraints are specified in XML format as shown below in the Web Service description field.

```
<constraint>
  <cpuLoad>load ls 1.0 </cpuLoad>
  <memory>memory gr 3GB</memory>
  <swapmemory>swapmemory gr 5MB </swapmemory>
  <starttime>1000</starttime>
  <endtime>1200</endtime>
</constraint>
```

The <constraint> tag defines individual constraints. As the name suggests, tags <cpuLoad>, <memory>, <swapmemory>, <starttime> and <endtime> are used to define CPU load, physical memory, swap memory and time of the day respectively. Words load, memory, swapmemory, starttime, and endtime are keywords. Starttime and endtime is specified in military time format. Keyword gr identifies the greater than operator, geq the greater than or equal operator, ls the less than operator, leq the less than or equal operator and eq identifies the equal to operator. Standard units are used to specify memory size: KB, MB, and GB.

As shown in Figure 3.3, the entire process of obtaining a constraint satisfied access URI is transparent to the registry user. A producer publishes Web Services that they support along with their organization details to a freebXML registry. An end user can search the
Figure 3.3. High level data flow diagram from Web Service publishing to discovery.

registry for Web Services that they want to use. The proposed load balancing scheme is hidden from the end user perspective as the end user is not required to make any changes to their code. End users can still continue to access registry contents in the same way.

As shown in Figure 3.4, the NodeStatus Web Service is first published to the freebXML registry. A list of all the hosts that deploy this service is maintained in the NodeState table in the database. The registry periodically invokes the NodeStatus Web Service on all the hosts that deploy it in order to collect different performance parameters. Finally, when an end user queries a freebXML registry for a Web Service, the registry first checks if there are any performance constraints specified for that specific Web Service. If the constraints are satisfied, then the access URIs of only those hosts that satisfy these performance constraints are returned in the form of service bindings to the end user.

Figure 3.4. Detail data flow diagram.
It is important to note that it is not enough to just specify Web Service constraints. The host on which the Web Service is deployed should also deploy the NodeStatus Web Service and the freebXML registry should be made aware of this by publishing the end point URL of the host to the registry.

A Web Service that is published to a freebXML registry is represented by an object instance of the ServiceDAO class. The ServiceDAO class is associated with the ServiceBindingDAO class that holds the access URIs to the host that deploys that particular service. In order to impose different performance constraints on the Web Service, we modified these original freebXML classes.

Figure 3.5 and Figure 3.6 illustrate how we modified the freebXML registry. The ServiceDAO object comes into picture whenever a Web Service is published, modified, or accessed. ServiceDAO is responsible for populating a ServiceBindingDAO instance with appropriate service bindings. Under normal circumstances all the service bindings associated with a Web Service are returned with a ServiceDAO object. The same is true if no performance constraints are specified for a Web Service. However, in the event performance constraints are specified, the ServiceDAO is responsible for populating ServiceBindingDAO with access URIs for only those hosts that satisfy the constraints.

As shown in Figure 3.5, a ServiceDAO instance first checks if there are any valid constraints associated with a Web Service. Class ServiceConstraint is responsible for verifying if valid service constraints are defined for a given Web Service. A ServiceConstraint instance validates Web Service constraints that are part of service description field. It is important to note that ServiceConstraint validates time constraints. As stated before, it is possible to impose a time frame during which a Web Service is available for service. The time constraint is specified using <starttime> and <endtime> tags. ServiceConstraint returns false if no valid service constraints are specified or if the time constraint is not satisfied. If valid service constraints are specified and time constraints are also satisfied, ServiceDAO then checks if the remaining constraints are satisfied. This is where class LoadStatus comes in to play. Class LoadStatus is responsible for identifying hosts that deploy the Web Service and satisfy the performance constraints. This is done by querying the NodeState table in the database for hosts that satisfy the constraints. LoadStatus returns the access URI for only those hosts that satisfy the performance constraints. Thus we
rearrange the access URI so that hosts that currently provide optimal service conditions are given preference over the ones that don’t currently satisfy Web Service performance constraints.

### 3.3 Client Side Web Services

The important aspect of this proposed scheme is its ability to monitor different hosts. This is achieved with the help of a Web Service name NodeStatus. NodeStatus is dormant software that is invoked periodically. The NodeStatus Web Service, when invoked, returns the CPU load along with the physical and swap memory available on the host.

As shown in Figure 3.7, the administrator needs to deploy the NodeStatus Web Service on all the hosts that need to be load balanced. The NodeStatus Web Service needs to be published to freebXML with the access URIs for all the hosts’ machines on which NodeStatus is deployed. NodeStatus needs to be deployed and published to freebXML only once and all the Web Services that are deployed on these hosts will be load balanced.
3.4 PUBLISHING AND ACCESSING REGISTRY CONTENTS

This section covers different ways in which the contents of a freebXML registry can be accessed or modified. A registry can be modified either by using a Web-based user interface (UI) or the AccessRegistry API. The Access Registry API acts as an interface between a user and the JAXR APIs. With the Access Registry API, a user need not bother about learning the details of how to use the JAXR API. The Access Registry API parses user input and makes appropriate calls to JAXR API calls. Each of these methods is covered step by step in this chapter.
3.4.1 Software Requirements

Listed below are the software’s that a client needs to install in order to collect to freebXML registry:

- JDK 1.4.2 or above.
- JWSDP 1.6 recommended.
- freebXML registry source code. This is an optional requirement just in case if user would prefer to use Java UI instead of Web UI.
- AccessRegistry source code or jar.

The registry used to demonstrate how these APIs work is accessible at http://volta.sdsu.edu:8080/omar/registry/thin/browser/login.jsp

3.4.2 User Registration

It is not necessary for a user to register with a freebXML registry if he or she intends to browse registry contents. However, only a registered authorized user is able to modify or publish to a freebXML registry. A user can register with a registry using Web UI.

As mentioned above a user can register with a freebXML registry using a Web UI. The Web UI referred to in this document can be accessed through this URL
http://volta.sdsu.edu:8080/omar/registry/thin/browser.jsp, as shown in Figure 3.8. The link to new user account is highlighted in Figure 3.9.

The “Create User Account” link leads to the “User Registration Wizard” shown in Figure 3.10. The registration wizard takes a user through four steps: Requirements, User Details, User Authentication Wizard, and loading a key to web browser. The key is part of X.509 certificate that identifies the client uniquely. Each of these steps is explained next.

Registration Wizard Step 1 Requirement: In order to successfully create new users account an X.509 certificate that holds user credentials is required. X.509 is a most widely accepted digital certificate format. This certificate is used by freebXML to authenticate the user. Every time a user logs in to freebXML the certificate is presented to freebXML by client side browser. A user can either provide an X.509 certificate or the freebXML registry can generate one for the user. Having made the decision the user can then proceed to the next step by clicking Next as shown in Figure 3.11.

As part of step 2, a user is required to fill out certain user details as shown in Figure 3.12.
During step 3, as part of the *User Registration Wizard*, a user enters a desired username and password, also called as *alias* and *password*, which he will use to login to the registry. In response, the *User Registration Wizard* generates a private key pair and a self signed X.509 certificate. See Figure 3.13.

At the end of step 4, the freebXML registry will generate a Private Key (.p12 file) / client certificate.p12 is a file extension for files that contains certificates and private keys, these files are password protected. In order to successfully complete the authentication process, a user must first download this file. The file can be downloaded by clicking on the
download button. Next, these credentials should be imported by the user into his or her Web-browser to enable SSL authentication, Figure 3.14. The Registry authenticates Web UI clients using client certificates. To import the certificate, the following steps need to be followed in accordance with the user’s particular web browser.

Mozilla/ Firefox:

- Click menu options Edit->Preference.
- Click Advance category in sidebar to expand options.
- Click Security tab.
- Click View Certificate button.
- The Your Certificate tab should be in focus. Click Import button.
- Select the certificate that needs to be imported.
- When prompted, in the Password Dialog, enter account password for the ‘master password for the Security Device’. This password is specific to the browser and assigned by the browser user.
- In the Password Entry Dialog enter the certificate password.
- If everything goes well, “Successfully restored your security certificate(s) and private key(s)” message will be displayed. Click the OK button.
- Close the Preference dialog box.

Internet Explorer:

- In Windows Explorer, double click your .p12 file. The Certificate Import Wizard should pop up. Click next twice.
Figure 3.12. Registration wizard step 2, user’s details.

- When prompted enter the certificate password.
- In the next window, let the wizard automatically select the certificate store. Click next and finish the wizard.
  Safari:
  - Start the application “Keychain Access” found under Application->Utilities.
Go to File, select option Import.

Import the certificate that was downloaded or a previously existing certificate. Enter the certificate password when prompted.

Opera:

Go to menu option Tools, in it select Preference.

Go to Advance tab, choose Security.

Click on Manage Certificate, Certificate Manager Window will pop up.

Select Personal tab, click Import button and select the certificate to be imported.
When prompted enter the certificate password.

When prompted enter Opera Certificate store password.

If successful the certificate can be seen listed in the certificate list. Close the Certificate manager.

### 3.4.3 Client Keystore Generation

A user can make his newly generated certificate (and private key) available to a JAXR provider from the Java command line. The Java keystore is used as a default client keystore. The client details are presented to freebXML every time a client starts a new session. This is the only way for the freebXML registry to authenticate a client. The command below exports the client details from the X.509 certificate to Java default keystore.

```bash
java -classpath <path_to>/omar-common.jar(*) \ 
org.freebxml.omar.common.security.KeystoreMover \ 
-sorceKeystorePath <.p12 file> \ 
-sorceKeystoreType PKCS12 \ 
-sorceKeystorePassword <keypasswd> \ 
-sorceAlias <alias> \ 
-sorceKeyPassword <keypasswd> \ 
-destinationKeystorePath ~/omar/<version>/jaxr-ebxml/security/keystore.jks \ 
-destinationAlias <alias>
```
Some jar files that are part of the freebXML source code are required; these files are also available with the AccessRegistry API source code. See Table 3.1.

Table 3.1. File Name and Listing

<table>
<thead>
<tr>
<th>File Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>build/lib/omar-common.jar</td>
</tr>
<tr>
<td>build/lib/commons-logging.jar</td>
</tr>
<tr>
<td>build/lib/jaxr-api.jar</td>
</tr>
</tbody>
</table>

The meaning of different options is explained in Table 3.2.

Table 3.2. Options Details

<table>
<thead>
<tr>
<th>Options</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>-sourceKeystorePath</td>
<td>This is the path to the certificate that was created as part of user registration.</td>
</tr>
<tr>
<td>-sourceKeystorePassword</td>
<td>Password that was specified during user registration along with the alias.</td>
</tr>
<tr>
<td>-sourceAlias</td>
<td>Alias that was specified during user registration.</td>
</tr>
<tr>
<td>-sourceKeyPassword</td>
<td>This is same as sourceKeystorePassword.</td>
</tr>
<tr>
<td>-destinationKeystorePath</td>
<td>This could be the path to java default keystore or to client keystore in omar source package if the user has downloaded it.</td>
</tr>
<tr>
<td>-destinationAlias</td>
<td>This is same as sourceAlias.</td>
</tr>
<tr>
<td>-destinationKeyPassword</td>
<td>This is as sourceKeyPassword.</td>
</tr>
<tr>
<td>-destinationKeystorePassword</td>
<td>This is the password of Java default keystore or in case of omar it is ebxmlrr.</td>
</tr>
</tbody>
</table>

Given below is a sample of how a client can export the client details from X.509 certificate to client keystore.

```
   -sourceKeystorePath generated-key_gold123.p12
   -sourceKeystoreType PKCS12
   -sourceKeystorePassword gold123
   -sourceAlias gold
   -sourceKeyPassword gold123
   -destinationKeystorePath /home/sadhana/omar/3.1/jaxr-ebxml/security/keystore.jks
```
One of the steps to complete user authentication process is to add a registryOperator alias to client keystore. The certificate that is generated as a part of user registration is issued by the registryOperator. When a client tries to connect to a freebXML registry using JAXR APIs, client details from client keystore are sent to the freebXML registry. The registry verifies it with client records it has with itself. In addition to this it also tries to verify the authority that issued the certificate to see if it is valid. For this it also verifies the identity registryOperator. The registry does this by verifying the registryOperator records that it has with itself with one that is part of the client credentials. This is taken care of by importing the registryOperator details into a client keystore from a Servier.cer file. The file is located in AccessRegistry/ServerCert/. If this step is completed successfully it means required authentication is established and the user can proceed further.

The registryOperator can be imported to client keystore using the following command line.

```
```

-trustcacerts is specified indicates that the certificates that are part of the -file option should be added as a trusted certificate.
-keystore option specifies the path to client keystore by default it is same as path to java keystore.

### 3.4.4 Publishing and Modifying Registry Contents

A freebXML’s registry contents can be modified and accessed using a Web user interface or the AccessRegistry API developed for this thesis. Both of these methods are covered in this section.

#### 3.4.4.1 WEB UI

As mentioned earlier, only authorized users can publish new registry objects or modify existing registry objects. A user can publish a registry object by clicking on the Create a New Registry Object link under the Task tab shown in Figure 3.15.

The type of object that a user wants to create can be selected from the Object type drop down list on right hand side, as shown in Figure 3.16. For example to create an
As stated before, in order to publish an organization, object type *Organization* should be selected as shown in Figure 3.16. A web page with organization details is displayed as shown in Figure 3.17. In order to publish an organization to the registry, details displayed in the form on the right should be filled out. The information is divided into different tabs; to fill in particular information the appropriate tab should be clicked, as shown in Figure 3.17.

The *Organization Details* tab has a field called *Name* which holds the name of the organization that you are publishing. The organization name makes it easy to search/locate
Figure 3.16. Organization object type is selected to be created.

the organization. All the information keyed in can be saved by clicking the Save button. Remember that this will save information in memory; in order to store information permanently in the database the user needs to click the Apply button. If a user fails to click Apply, and logs out, any information entered will be lost.

An organization address can be published to the registry by clicking on the Postal Address tab and by filling out the required fields, as shown in Figure 3.18.

Once inside the Postal Address tab, click the Add button to add a new address for this organization, as shown in Figure 3.19.
Next, fill in the address in the form displayed, as shown in Figure 3.20, and click Save to store the information.

After clicking Save in the step above, the Postal Address tab is displayed again, with the postal address, as shown in Figure 3.21, click Save to save information to memory. Click Apply to commit information to the database. A web page displaying message “Apply”
Figure 3.18. Organization postal address.

*Successful* will be displayed if information is committed successfully, as shown in Figure 3.22.

Organization email can be added by clicking on the *Email* tab, as shown in Figure 3.23. Once inside email table click *Add* to enter email address, as show in Figure 3.24.
Enter the email address for the organization in the web page displayed, as shown in Figure 3.25, and click Save to save the address.

Email tab is displayed next and the organization email address is listed in it, as shown in Figure 3.26. Click Apply to commit the email address to the freebXML database. A web page displaying message “Apply Successful” will be displayed if information is committed successfully, as shown in Figure 3.22.
In order to enter an organization’s telephone number, click on the *Telephone Number* tab, as shown in Figure 3.27.

Once inside the telephone number tab click *Add* to add new telephone number, as show in Figure 3.28.
Figure 3.23. Click on email tab to add organization email address.

Figure 3.24. To add an organization email click add in email tab.

Figure 3.25. Enter organization email address.
Figure 3.26. Email tab with organization email listed click save to save the email.

Figure 3.27. Click on telephone number tab to add organization telephone number.

Select the type of telephone number from the drop down list, enter the number and click Save to save the number, as shown in Figure 3.29.

The phone number entered is listed in the Telephone Number tab, if saved, as shown in Figure 3.30; click Apply to commit the entry to database. A web page displaying the message “Apply Successful” will be displayed if information is committed successfully, as shown in Figure 3.22.
Figure 3.28. Click add on telephone numbers tab to add telephone.

Figure 3.29. Fill in the telephone details and select the telephone type.

Figure 3.30. Phone number added is listed in telephone number tab click save to save.
The organization information entered so far can be displayed by click on *Organization Detail* tab, as shown in Figure 3.17. The information listed can be seen in Figure 3.31 and Figure 3.32.

![Figure 3.31. Organization address details.](image)

After reviewing the information, the user should commit the *Organization* object to the database by clicking *Apply*, as shown in Figure 3.33. A web page displaying the message "*Apply Successful*" will be displayed if information is committed successfully, as shown in Figure 3.22. A failure to do so would lead to loss of information.

In order to represent a Web Service provided by an organization the first step is to create a *Service* object corresponding to a Web Service and then create *Association* objects that represent the relationship between the Web Service and the organization.

In order to create a service object first click on the *Create a New Registry Object* link again in the *Search* tab, as shown in Figure 3.15. Next, from the drop down list on right, select *Service* object and click *Add*, as shown in Figure 3.34.

A web page with a Web Service’s *Service* details is displayed, as shown in Figure 3.35. After entering the Web Service name and description click *Save*. As in case of the *Organization* object, it is a good idea to name a Web Service that is being published. Web Service name should be intuitive of the actions that it is suppose to carry out. A good
Web Service name helps in searching and locating a Web Service. Similarly a user can add Web Service description, to inform other users about the functionality that a Web Service is capable of providing. The Version filed comes into use if Versioning is on, and a user can create different versions of a Web Service if any new functionality is added. However, for this work Versioning is off.
The Access URL of the Web Service is entered through the Service Binding tab, as shown in Figure 3.36. Access URL is same as the end point URL of a Web Service.

Once inside the Service Binding tab click Add to proceed further, as shown in Figure 3.37.

Enter the name and description for the service binding; enter the end point reference URL for the Web Service. In the Target Binding field, a unique identifier of another ServiceBinding to which this service binding refers, can be added. Some times
Figure 3.36. Click on ServiceBinding tab to enter access URIs associated with a Web Service.

Figure 3.37. Click add in ServiceBinding tab to add new access URIs.
ServiceBinding does not refer to service implementation directly but it specifies a reference to another ServiceBinding, this reference can be place in the Target Binding field. Note that even though Target Binding is an optional field both Access URL and Target Binding can be specified in a Service Binding Object entry. Click Save to save the information, as shown in Figure 3.38.

![Figure 3.38. ServiceBinding details panel click save to save the changes.](image)

On clicking save, control returns to the Service Bindings tab again, service bindings added are listed in this tab, as shown in Figure 3.39. More access URLs can be added similarly. After adding all the access URLs, click the Apply button to commit all service objects and dependent service binding objects to the freebXML database, as shown in Figure 3.40. A web page displaying message “Apply Successful” will be displayed if information is committed successfully, as shown in Figure 3.21.

The Service object that is created above is associated with its parent organization using an Association object. The first step is to list all the objects that a user owns. This is done by selecting FindAllMyObjects from the drop down list in the Search Tab, as shown in Figure 3.41.

Select the Organization and the Service object by clicking on the check box then press Relate, as shown in Figure 3.42.
Figure 3.39. Service bindings tab – added service bindings are listed in the tab.

Figure 3.40. Lists all the service bindings added. To add or delete a service binding, click the respective button.
Figure 3.41. *FindAllMyObjects* option lists all the objects that are published by a user.

Details of the *Association* object are displayed next, as shown in Figure 3.43. The relationship box indicates the type of relationship the object represents; in this case the *Association* radio button is enabled. A user can name the *Association* object.

The *Association details* page lists the *Organization* object as a source object while the *Service* object is listed as a target object. The association type between an organization and its services is called an “*OfferService*”. To configure this type of relationship, select the association type *OfferService* from the association type drop down list, as shown in Figure 3.44.

An *Associations* object needs to be committed to the freebXML database by clicking *Apply*, as shown in Figure 3.45. A web page displaying message “*Apply Successful*” will be displayed if information is committed successfully, as shown in Figure 3.21.
Figure 3.42. Select objects to be associated and then click relate.

The Association object is now listed as part of the Organization and Service object, whose relationship it represents. In this case it is listed as part of NewOrganization and NodeStatus objects, as shown in Figure 3.46 and Figure 3.47.

A user can modify or delete the objects that he owns. The objects that a user owns can be listed using “FindAllMyObjects” from the Search Tab, as shown in Figure 3.48.
To modify any object, pick the object by clicking across the checkbox and then click details. Refer Figure 3.49 for assistance. Object details are displayed on the web page after modifying the contents click Apply to commit the changes.

To delete an object pick the object by clicking the check box, as shown in Figure 3.50, and then click Delete.
Figure 3.45. Commit association object by clicking apply.

Figure 3.46. NewOrganization details.
Figure 3.47. NodeStatus details.

Figure 3.48. List of all the objects owned by a user.
Figure 3.49. Select object that is to be modified and then click details.

Figure 3.50. Select the object to delete and press the delete button.
3.4.4.2 The AccessRegistry API

In order to make communication with a freebXML registry easier without getting much involved with using the JAXR API, a new of API has been developed as part of this thesis effort. These APIs allow a user to access, modify, and publish metadata to a registry. The source code for these APIs is available in a Java project called AccessRegistry. AccessRegistry has dependency on following jars: jaxr-ebxml.jar, jaxb-api.jar, jaxb-impl.jar, jaxb-libs.jar, and jaxr-api.jar; these jars are part of freebXML source code.

The AccessRegistry API is responsible for two things: connecting to a freebXML registry and accessing registry contents. All the information needed to carry out both of these activates is made available through two different XML files: connection.xml and action.xml. The connection.xml file contains information required to connect to freebXML registry, such as client alias/password, path to client keystore, and registry URL. Action.xml is used to specify registry objects details that are supposed to be published, accessed or modified. RegistryObject is a class, whose instances are used to represents anything that is published to a registry. For instance an organization that is published to a registry is represented by an instance of RegistryObject called OrganizationDAO.

Next, we explore in detail how both connection.xml and action.xml files are structured. We start with connection.xml first. connect.dtd, available as part of AccessRegistry source code, specifies the XML file format for Connection.XML, this file can be used to verify if the XML file generated is in proper format. Few sample files are included with the AccessRegistry source code in directory sampleFiles. ConnectLocal.XML and ConnectVolta.XML demonstrate how to connect to local registry and registry on Volta receptively. Note that adding

```
<! DOCTYPE connection SYSTEM 'Connect.dtd'>
```

to the XML file will help validate the XML using NetBeans. Example:

```
<? xml version="1.0" encoding="UTF-8"?>
<! DOCTYPE connection SYSTEM 'Connect.dtd'>
<connection>
  <user>
    <alias>gold</alias>
    <password>gold123</password>
  </user>

  <url>https://volta.sdsu.edu:8443/omar/registry/soap</url>
```
Once a connection has been established with a registry, its contents can either be modified or accessed through the AccessRegistry API. Following XML files are used to publish, access, and modify registry contents respectively: PublishToRegistry.xml, AccessRegistry.xml, and ModifyRegistry.xml. RegistryAccess.dtd specifies precisely the format in which data appears in all these files and thus can be used to validate the XML files. As stated before all these files are part of AccessRegistry source code and are located in directory SampleFiles.

In order to be able to carry out any activity using AccessRegistry it is essential to get familiar with RegistryAccess.dtd. Different data format syntax that is defined in RegistryAccess.dtd are explained next. The first rule in RegistryAccess.dtd conveys two things: the name of the parent element is root, and root element embeds in it a child element called action.

```
<!ELEMENT root (action+)>
```

This is a general DTD rule, the ELEMENT tag specifies the parent element and the child elements it can embed.

```
<!ELEMENT element-name (element-content)>
```

For example:

```
<!ELEMENT action (organization+)>
```

The above rule conveys that parent element action can embed one or more child elements named organization. The number of child elements that a parent can embed is specified as follows in Table 3.3.

<table>
<thead>
<tr>
<th>Rules</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum one occurrence</td>
<td>&lt;!ELEMENT element-name (child-name+)&gt;</td>
</tr>
<tr>
<td>Zero or more occurrence</td>
<td>&lt;!ELEMENT element-name (child-name*)&gt;</td>
</tr>
<tr>
<td>One or Zero occurrence</td>
<td>&lt;!ELEMENT element-name (child-name?)&gt;</td>
</tr>
<tr>
<td>Only one occurrence</td>
<td>&lt;!ELEMENT element-name (child-name)&gt;</td>
</tr>
</tbody>
</table>

The ATTLIST tag specifies the attributes an element can have.

```
<!ATTLIST action type (publish |access |modify) “access“>
```
The rule above states: element action has an attribute named type, and type can take either of these values: publish, access, and modify; the default value is access. Thus the preliminary structure of any xml file for accessing the registry would be as follows:

```
<root>
  <action type ="publish">
    <organization>
      </organization>
  </action>
</root>
```

It should be noted here that an action element and its type attribute specifies the activity to be carried out, for example shown in Table 3.4.

**Table 3.4. Action Tag Type Options and Meaning**

<table>
<thead>
<tr>
<th>Type option</th>
<th>Option details</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;action type =&quot;publish&quot;&gt;</td>
<td>Publish to registry.</td>
</tr>
<tr>
<td>&lt;action type =&quot;access&quot;&gt;</td>
<td>Access registry contents.</td>
</tr>
<tr>
<td>&lt;action type =&quot;modify&quot;&gt;</td>
<td>Modify registry contents.</td>
</tr>
</tbody>
</table>

One other thing to be noted here is that at least one organization should to be specified in order to carry out any of the above mentioned activities.

In order to publish contents to a registry, the object details should be specified in an XML file such as `PublishToRegistry.xml`. As mentioned previously, rules governing the format of this file are specified in RegistryAccess.dtd. The basic file structure for any such file is:

```
<root>
  <action type ="publish">
    <organization>
      </organization>
  </action>
</root>
```

Element organization is defined as follows:

```
<!ELEMENT organization (name, description?, postaladdress?, telephone?, service*)>
```

The *Organization* tag embeds a non optional element called *name*. An organization can have only one name. It also embeds optional elements such as: *description*, *postaladdress*, *telephone*, and *service*. Example:

```
<root>
```
The description element is used for adding description to both organization and service element, and is defined as follows:

```xml
<!ELEMENT description (#PCDATA | constrain)*>
<!ATTLIST description type CDATA #IMPLIED>
<!-- type = “add” or type = “delete” or “edit”-->
```

Description can embed either #PCDATA or an element named constrain. #PCDATA stands for parsed character data. Any text that is specified will be parsed by the parser. This field is used to specify description text.

```xml
<description> Used to describe a organization. </description>
```

Element constrain and attribute type are covered later.

Postaladdress is used to add organization postal address.

```xml
<!ELEMENT postaladdress (streetnumber | street | city | state | country | postalcode | type)*>
<!ELEMENT streetnumber (#PCDATA)>
<!ELEMENT street (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT state (#PCDATA)>
<!ELEMENT country (#PCDATA)>
<!ELEMENT postalcode (#PCDATA)>
<!ELEMENT type (#PCDATA)>
```

Postaladdress can embed streetnumber, street, city, state, country, postalcode, and type. Each of these embed elements are of type #PCDATA. Example:

```xml
<postaladdress>
  <streetnumber>5500</streetnumber>
  <street>Campanile Drive</street>
  <city>San Diego</city>
  <state>CA</state>
  <country>US</country>
  <postalcode>92182</postalcode>
  <type>TYPE-US</type>
</postaladdress>
```

Element telephone is used to add telephone number for a organization.

```xml
<!ELEMENT telephone (type | number | areacode | countrycode)*)
<!ELEMENT countrycode (#PCDATA)>
<!ELEMENT areacode (#PCDATA)>
```
The telephone element embeds type, number, areacode, and countrycode elements; each of these elements is of type #PCDATA. Example:

```xml
<telephone>
  <countrycode>1</countrycode>
  <areacode>619</areacode>
  <number>594-5200</number>
  <type>OfficePhone</type>
</telephone>
```

Integrating all elements above an organization can be described as follows:

```xml
<root>
  <action type="publish">
    <organization>
      <name>DemoOrganization</name>
      <description>A university in southern California</description>
      <postaladdress>
        <streetnumber>5500</streetnumber>
        <street>Campanile Drive</street>
        <city>San Diego</city>
        <state>CA</state>
        <country>US</country>
        <postalcode>92182</postalcode>
      </postaladdress>
      <telephone>
        <countrycode>1</countrycode>
        <areacode>619</areacode>
        <number>594-5200</number>
        <type>OfficePhone</type>
      </telephone>
    </organization>
  </action>
</root>
```

A service element is used to define the Web Service provided by an organization.

```xml
<!ELEMENT service (name, description?, accessuri*)>
<!ATTLIST service type CDATA #IMPLIED> <!-- type = add | delete -->
```

A service element embeds name, description and access URIs elements. Description and access URI elements are optional; however service name is a mandatory element. Service attribute type is covered later. Unlike organization description, service description is more complex. As seen before, description can embed within it a #PCDATA, element named constrain. Constrain is used to describe service elements only and it cannot be used to
describe an organization. Constraint can embed within it: `endtime`, `starttime`, `swapmemory`, `memory`, `cpuLoad`, and `#PCDATA`.

```xml
<!ELEMENT constraint (#PCDATA | endtime | starttime |
                         swapmemory | memory | cpuLoad)>  
<!ELEMENT cpuLoad (#PCDATA)>  
<!ELEMENT memory (#PCDATA)>  
<!ELEMENT swapmemory (#PCDATA)>  
<!ELEMENT starttime (#PCDATA)>  
<!ELEMENT endtime (#PCDATA)>  
```

As the name suggest, `constraint` element is used to specify different constraints. An access URI of a server that hosts the Web Service and that satisfies all the constraint is fetched from the registry. Constraints specify resource and time requirements that are essential for a Web Service to function properly. Special symbols are used to specify different constraint values and these details are documented in Table 3.5. Standard units are used to specify memory size in KB, MB, and GB. Words such as `load`, `memory`, `swapmemory`, `starttime`, and `endtime` are keywords. Starttime and endtime is specified in military time format. Example:

```xml
<description>
  <constraint>
    <cpuLoad>load gt 0.01</cpuLoad>
    <memory>memory geq 5MB</memory>
    <swapmemory>swapmemory leq 3KB</swapmemory>
    <starttime>0700</starttime>
    <endtime>2200</endtime>
  </constraint>
</description>
```

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Arithmetic Symbol</th>
<th>Stands for</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>gt</td>
<td>&gt;</td>
<td>Greater than</td>
<td>load gt 0.01</td>
</tr>
<tr>
<td>geq</td>
<td>&gt;=</td>
<td>Greater than or equals</td>
<td>memory geq 5MB</td>
</tr>
<tr>
<td>ls</td>
<td>&lt;</td>
<td>Less than</td>
<td>load ls 0.05</td>
</tr>
<tr>
<td>leq</td>
<td>&lt;=</td>
<td>Less than or equals</td>
<td>swapmemory leq 3KB</td>
</tr>
<tr>
<td>eq</td>
<td>=</td>
<td>Equals</td>
<td>memory eq 5MB</td>
</tr>
</tbody>
</table>

Table 3.5. Symbols

Element `accessuri` is used to specify a Web Service end point reference URL on different servers. The element embeds `#PCDATA` which is a URL. The attribute named `type` is covered later. Example:

```xml
<accessuri>
```
An example of service element with description and accessuri added:

```
<service>
  <name>Demo Service</name>
  <description>
    <constraint>
      <cpuLoad>load gt 0.01</cpuLoad>
      <memory>memory geq 5MB</memory>
      <swapmemory>swapmemory leq 3KB</swapmemory>
      <starttime>0700</starttime>
      <endtime>2200</endtime>
    </constraint>
  </description>
  <accessuri>
    http://exergy.sdsu.edu:8080/Adder/addService
    http://romulus.sdsu.edu:8080/Adder/addService
  </accessuri>
</service>
```

An entry in PublishToRegistry.xml would look as follows:

```
<root>
  <action type="publish">
    <organization>
      <name>DemoOrganization</name>
      <description>
        Organization published for demonstration purpose only.
      </description>
      <postaladdress>
        <streetnumber>5500</streetnumber>
        <street>Campanile Drive</street>
        <city>San Diego</city>
        <state>CA</state>
        <country>US</country>
        <postalcode>92182</postalcode>
        <type>TYPE-US</type>
      </postaladdress>
      <telephone>
        <countrycode>1</countrycode>
        <areacode>619</areacode>
        <number>594-5200</number>
        <type>OfficePhone</type>
      </telephone>
      <service>
        <name>Demo Service</name>
        <description>
          <constraint>
            <cpuLoad>load gt 0.01</cpuLoad>
          ```
As in case of publishing, to modify registry contents, details about the object that is to be modified are specified in a XML file. As stated before, RegistryAccess.dtd governs the syntax of all the elements that are part of this XML file. The basic structure of a XML file used to modify registry contents looks as follows:

```
<root>
  <action type ="modify">
    <organization>
      <name></name>
    </organization>
  </action>
</root>
```

The only difference here is the value of attribute `type`: it is changed from `publish` to `modify`.

All the rules stated above regarding action, organization, service elements, and others hold true in this case. At the present time of writing, the AccessRegistry API supports modification to only the most essential elements. Fields that can or cannot be modified are listed in Table 3.6.

Note that in order to carry out any modification an organization must be first published to the registry. Similarly, any modifications to a Web Service that is not published would lead to an error. A Web Service can be added to an organization that has been published before.

As listed in Table 3.6, an organization can be deleted and its description can be modified. In order to delete an organization, the type attribute of element `organization` is set
Table 3.6. List of Fields That Can or Cannot Be Modified

<table>
<thead>
<tr>
<th>Fields that can be modified</th>
<th>Fields that cannot be modified</th>
</tr>
</thead>
<tbody>
<tr>
<td>A organization can be deleted</td>
<td>To add a organization it must be published.</td>
</tr>
<tr>
<td>A organization description can be added, deleted, and edited</td>
<td>An organization name, postaladdress, and telephone number cannot be modified.</td>
</tr>
<tr>
<td>A Web Service can be added and deleted</td>
<td>A Web Service name cannot be changed once added or published.</td>
</tr>
<tr>
<td>A Web Service description and accessuri can be added, deleted, and edited (i.e. overwritten)</td>
<td></td>
</tr>
</tbody>
</table>

Note: individual description elements are not editable. The entire description needs to be specified so that it can be rewritten.

to delete. Once an organization is deleted, all the services that are associated with it are also deleted from the registry.

```xml
<!ELEMENT organization (name ,description? ,postaladdress?, telephone?, service*)>
<!ATTLIST organization type CDATA #IMPLIED>
<!-- type="delete" only this works -->

Example:

```xml
<root>
  <action type ="modify">
    <organization type ="delete">
      <name>OrganizationDemoDeleteOrganization</name>
    </organization>
  </action>
</root>
```

An organization’s description can be modified using the type attribute of description element. Substitute type with add, modify, or delete to add new description, or to modify or delete existing description respectively.

```xml
<!ELEMENT description (#PCDATA|constrain)*>
<!ATTLIST description type CDATA #IMPLIED>
<!-- type= add|modify|delete -- >

Example:

```xml
<root>
  <action type ="modify">
    <organization>
      <name>OrganizationDemoDeleteOrganization</name>
    </organization>
  </action>
</root>
```
As listed in Table 3.6, a service object can be added or deleted. Service elements such as description and accessuri can be modified.

To add a new service to an already existing organization the type attribute of a service element is set to add. Example:

```xml
<root>
  <action type="modify">
    <organization>
      <name>OrganizationDemoTwo_AddService</name>
      <service type="add">
        <name>Adder_AddNew</name>
        <!-- details omitted -->
      </service>
    </organization>
  </action>
</root>
```

Similarly, to delete a service, the attribute type is set to delete. Example:

```xml
<root>
  <action type="modify">
    <organization>
      <name>OrganizationDemoTwo_AddService</name>
      <service type="delete">
        <name>Adder_AddNew</name>
      </service>
    </organization>
  </action>
</root>
```

In case of an organization description, a user can add, modify, or delete a service description. As mentioned before, organization and service names should be specified as they are mandatory.

Example:

```xml
<root>
  <action type="modify">
    <organization>
      <name>OrganizationDemoTwo_AddService</name>
    </organization>
  </action>
</root>
```
Service access URIs can either be added or modified using the type attribute of an accessuri element.

```
<!ELEMENT accessuri (#PCDATA)>  
<!ATTLIST accessuri type CDATA #IMPLIED>
```

Example:

```xml
<organization>
  <name>OrganizationDemoTwo_AddService</name>
  <service>
    <name>Adder_AddNew</name>
    <accessuri type="add">
      http://exergy.sdsu.edu:8080
      /NodeMemorySize/memorysizeService
    </accessuri>
  </service>
</organization>
```

The basic structure of the XML file used to access the registry contents is same except type attribute of action is changed to “access”.

```xml
<root>
  <action type ="access">
    <organization>
      <name></name>
      <service>
        <name></name>
      </service>
    </organization>
  </action >
</root>
```

Names of both organization and its service element should be specified. Note that a service should always be enclosed by its parent organization element. Just providing a
service name without an organization name to which it belongs would lead to an error.

Example:

```
<root>
  <action type="access">
    <organization>
      <name>OrganizationDemoTwo_AddService</name>
      <service>
        <name>Adder_AddNew</name>
      </service>
    </organization>
  </action>
</root>
```

3.4.5 Developing Sample Programs Using the AccessRegistry API

The input to a program that uses the AccessRegistry API to connect to freebXML registry are two XML files: one that contains the connection details and another that contains object details along with the action to be carried out. The parser which is a part of AccessRegistry parses these files into appropriate Java objects. Next, the connection APIs establishes a connection with the registry using user credentials. Both the parser and the connection API works in the background. In order to use the AccessRegistry API a user needs to familiarize him or herself with the Registry Java class.

```java
public class Main{
    public static void main(String[] args) {
        if(args.length < 2) return;
        Registry connect = new Registry(args[0],args[1]);
        //The file names can be hard coded as shown below
        //Registry connect = new
        //Registry("connection.xml","PublishToRegistry.xml");
    }
}
```

Class Registry acts as a wrapper that internally parses the user input, connects to a registry, and handles all other registry operations. A developer begins by first creating an object of class `Registry` and pass to it pathnames to two xml files. In this case the files are passed as command line parameters or can be hardcoded as shown below.

```java
Registry connect = new Registry(args[0],args[1]);
Registry connect = new
Registry("connection.xml","PublishToRegistry.xml");
```

Once an instance of class `Registry` is created, a user then needs to invoke method `execute()` on the Registry object:
The `execute()` method is responsible for executing the action specified in the XML files. Whenever any registry object is modified or published, `execute()` returns the *Organization Id* of the organization to which the modified or published object belongs. However, `execute()` returns list of Access URIs when an organization and its Web Services are accessed.

`execute()` returns a ArrayList which in-turn contains a ArrayList. Figure 3.51, helps us understand this better. The outer result list holds different lists that store results of individual operations.

![Diagram](image.png)

**Figure 3.51.** Execute() returns a arraylist which holds the results of different operation in different arraylist.

It can be recalled that instructions to publish, modify, and access can be specified as a part of a single xml file. For example:

```xml
<root>
  <action type ="publish">
    ...
  </action>
</root>
```
Using a XML file with structure similar to one shown above, three different operations are invoked: publish, modify and access. As stated before, execute() returns the Organization Id of the organization to which the modified or published object belongs and Access URIs if a Web Service is accessed. execute() internally rearranges these results into three different lists, one each for published objects, one for modified objects and one for Access URI. However, as we know in Java a method or function can return only one object so it is not possible for execute() to return all the three lists individually. So these three lists are added to another container object and that container object is returned by execute() as shown in Figure 3.51. In this case we use ArrayList as the container, to which the other three lists are added.

```java
outerList = registry.execute(...);
```

As shown above, outerList is the outer container that holds the three lists. Each of these three lists can be accessed individually using either a index as shown below:

- outerList[0] will contain Organization Ids corresponding to published organizations.
- outerList[1] will contain Organization Ids corresponding to modified organizations.

The following is a sample program that demonstrates how to use AccessRegistry API, in this program we are trying to publish a dummy organization name DemoOrganization to freebXML registry.

```java
public class Main{
    public static void main(String[] args) {
        if(args.length <2 ) return;
        Registry connect = new Registry(args[0],args[1]);
        ArrayList result = connect.execute();
        Iterator itr = result.iterator();
        while(itr.hasNext()){
            ArrayList organization = (ArrayList) itr.next();
```
Iterator itrOrga = organization.iterator();
while(itrOrga.hasNext()){
    System.out.println("Organizationid:"+
    itrOrga.next().toString());
}
}

Program output:

    Organization id :- urn:uuid:59bd7041-781f-4c57-b985-f0293588642b

As a part of the program output we have printed the organization id of the organization that was published to freebXML using AccessRegistry API. The search result for the organization that was published is shown in Figure 3.52. The Figure 3.52 shows DemoOrganization listed in the list of organizations.

![Image of ebXML Registry Repository](image-url)

Figure 3.52. A search for DemoOrganization on freebXML confirms that the organization was published successfully using AccessRegistry API.
Next we will try performing different operations on freebXML registry contents using AccessRegistry API. The idea here is to publish some organizations and Web Services that belong to these organizations. We then try modifying these published organizations and Web Service through AccessResigtry API. This exercise will give us an idea how AccessRegistry API can be used to modify registry contents. To start with, using the code in Table 3.7 and action.xml following organizations and Web Services are published to the registry.

Table 3.7. List of Organization and Services from PublishToRegistry.xml

<table>
<thead>
<tr>
<th>Organization</th>
<th>Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>DemoOrg_DeleteOrganization</td>
<td>DemoService_Delete</td>
</tr>
<tr>
<td>DemoOrg_AddDescription</td>
<td>DemoSrv_DeleteService</td>
</tr>
<tr>
<td>DemoOrg_ModifyService</td>
<td>DemoSrv_AddDescription</td>
</tr>
<tr>
<td></td>
<td>DemoSrv_EditDescription2</td>
</tr>
<tr>
<td></td>
<td>DemoSrv_AddAccessUri</td>
</tr>
<tr>
<td></td>
<td>DemoSrv_DeleteAccessUri</td>
</tr>
</tbody>
</table>

Listed below is the program output. The output prints out the organization ids of all the organizations that are published.

```
The Organization ids are returned as the output
Organization id:-urn:uuid:87b517af-8125-4789-85f8-d09759f0c236
Organization id:-urn: uuid:03e2868a-3b16-4c08-819b-b844b71067cc
Organization id:-urn:uuid:e9a61cb8-1b82-412f-a3f6-bd8719ac4dfe
BUILD SUCCESSFUL (total time: 11 seconds)
```

We can confirm this by searching the registry. Figure 3.53 shows the listing of all the organizations published. Figure 3.54 list the Web Services.

Now we try to modify the contents that we published. To do this we modify action.xml Table 3.8 lists elements that are modified and the expected result of this modification.

```
Below is the program output. The debug messages are added for understanding purpose.
```
Figure 3.53. Organization search results after publishing from PublishToRegistry.xml.

// Web Services associated with DemoOrg_DeleteOrganization are deleted first.
Service is Deleted
key was urn:uuid:e18d09ed-4655-4df7-b327-853d1a0c3405
Organization is deleted
key was urn:uuid:e9a61cb8-1b82-412f-a3f6-bd8719ac4dfe

// Organization id for DemoOrg_AddDescription is returned as new description is added to it.
Organization Modified
key was urn:uuid:87b517af-8125-4789-85f8-d09759f0c236

// DemoSrv_AddDescription Web Service is modified new service description is added.
Organization Modified
key was urn:uuid:03e2868a-3b16-4c08-819b-b844b71067cc
ServiceDescription Added
key was urn:uuid:e827f81b-0d88-4e1e-925e-fff5ec30f3ec
<constrain>
<cpuLoad>load gt 0.01</cpuLoad>
</constrain>

// DemoSrv_EditDescription2 Web Service is modifier the existing service description is modified.
ServiceDescription Added
key was urn:uuid:4fe59708-0c8c-43fa-901f-8057d2802927

// New access URI is added to Web Service DemoSrv_AddAccessUri.
Figure 3.54. Service search results after publishing from PublishToRegistry.xml.

Table 3.8. List of Elements That Are Modified Using ModifyRegistry.xml

<table>
<thead>
<tr>
<th>Registry Objects</th>
<th>Action</th>
<th>Expected Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>1   DemoOrg_DeleteOrganization</td>
<td>Deleted</td>
<td>Services associated with the organization should get deleted.</td>
</tr>
<tr>
<td>2   DemoOrg_AddDescription</td>
<td>Organization description is added.</td>
<td></td>
</tr>
<tr>
<td>3   DemoSrV_AddDescription</td>
<td>Service description is added</td>
<td></td>
</tr>
<tr>
<td>4   DemoSrV_EditDescription2</td>
<td>Service description is edited</td>
<td></td>
</tr>
<tr>
<td>5   DemoSrV_AddAccessUri</td>
<td>Access URI is added</td>
<td></td>
</tr>
<tr>
<td>6   DemoSrV_DeleteAccessUri</td>
<td>Access URI is deleted</td>
<td></td>
</tr>
<tr>
<td>7   DemoSrV_DeleteService</td>
<td>Service is deleted</td>
<td></td>
</tr>
</tbody>
</table>
ServiceBinding is added
  key was urn:uuid:e076b5c6-bfdb-4e05-b058-9dfde089d30a

// Access URIs are deleted from DemoSrv_DeleteAccessUri
ServiceBinding is deleted
  key was urn:uuid:fe31b15a-2d5f-47fd-a1e8-841de20fe1d9

// Web Service DemoSrv_DeleteService is deleted
Service is Deleted
  key was urn:uuid:e547e085-5015-494c-99c5-8797125c6003

// Finally organization ids of all the organizations that were modified are returned.
Organization id :-urn:uuid:87b517af-8125-4789-85f8-d09759f0c236
Organization id :-urn:uuid:e9a61cb8-1b82-412f-a3f6-bd8719ac4dfe
Organization id :-urn:uuid:03e2868a-3b16-4c08-819b-b844b71067cc
BUILD SUCCESSFUL (total time: 12 seconds)

We verify the authenticity of the program output above by searching the registry. Below are some of the screen shots of the search results. The search results for organization names starting with DemoOrg does not list DemoOrg_DeleteOrganization, also description is added to DemoOrg_AddDescription in the Figure 3.55.

Further the search results for service names starting with DemoSrv do not list services DemoSrv_DeleteService and DemoService_Delete also new description is added to DemoSrv_AddDescription refer Figure 3.56. The existing service description of Web Service DemoSrv_EditDescription2 is edited see Figure 3.57.

Now that we have seen how registry contents can be published or modified we will now see how access URIs can be fetched using AccessRegistry API. To start with we modify action.xml to fetch access URIs associated with service DemoSrv_AddAccessUri and DemoSrv_DeleteAccessUri belonging to organization DemoOrg_ModifyService. Program Output

http://romulus.sdsu.edu:8080/Adder/addService
http://exergy.sdsu.edu:8080/Adder/addService
http://exergy.sdsu.edu:8080/Adder/addService
http://exergy.sdsu.edu:8080/Adder/addService
http://romulus.sdsu.edu:8080/Adder/addService

Note that access URI http://romulus.sdsu.edu:8080/Adder/addService was added to DemoSrv_AddAccessUri while http://exergy.sdsu.edu:8080/Adder/addService was deleted from DemoSrv_DeleteAccessUri
3.4.6 JUnit Test Cases

As a part of the development work of the AccessRegistry API we developed couple of JUnit test cases. JUnit [33] is a Java framework for performing unit test case. JUnit test case provides a quick and effective way to evaluate if the changes made to the code work and also that the changes made do not break previously tested and working code.

The JUnit test cases can be found in *TestPackages* in the AccessRegistry project. The XML files needed for the unit test cases are stored in a folder named *TestFiles*, while the source code for the test cases is placed in a folder called *registry* within folder *TestPackages*, as shown in Figure 3.58. It is highly recommended that in order for these tests to work the XML files should not be modified.
The test cases test the API’s ability to publish, modify, and access different registry objects to the registry such as organizations and Web Services. Different test cases that are generated are listed in Table 3.9.

The result of these test run shown below and in Figure 3.59.

Testsuite: registry.AccessTest
execute
access
Inside Access
*****TestWebServiceService
Test Organization
http://eon.sdsu.edu:8080/TestWebService/TestWebServiceService
Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 8.67 sec
Figure 3.57. Service DemoSrv_EditDescription2.

------------- Standard Output -------------
execute
access
Inside Access
****TestWebServiceService
Test Organization
http://eon.sdsu.edu:8080/TestWebService/TestWebServiceService

------------- ---------------- ---------------
Testsuite: registry.ModifyTest
eexecute
modify
Inside modify
ServiceBinding is added
key was urn:uuid:b1b95164-3670-4a25-bf5e-93326168aeea
Organization Modified
key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Figure 3.58. JUnit test cases within the AccessRegistry API.

Organization saved
key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979

execute
modify
Inside modify
ServiceBinding is deleted
key was urn:uuid:883b6233-f08d-40c9-ab48-8d263a5965a4
Organization Modified
key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization saved
key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
execute
### Table 3.9. Test Case Description

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>testGetBusinessLifeCycleManager</td>
<td>Part of RegistryTest.java this tests for the availability of BusinessLifeCycleManager</td>
</tr>
<tr>
<td>testGetBusinessQueryManager</td>
<td>Part of RegistryTest.java; tests for the availability of BusinessQueryManager</td>
</tr>
<tr>
<td>testExecute</td>
<td>Part of PublishTest.Java; tests the APIs that publish registry objects to the registry.</td>
</tr>
<tr>
<td>testExecute_AddAccessURI</td>
<td>Part of ModifyTest.Java; tests the part of APIs that add access URI to a Web Service.</td>
</tr>
<tr>
<td>testExecute_DeleteAccessURI</td>
<td>Part of ModifyTest.Java; tests the part of APIs that deletes access URI from a Web Service.</td>
</tr>
<tr>
<td>testExecute_DuplicateAccessURI</td>
<td>Part of ModifyTest.java; tests for duplication of access URI.</td>
</tr>
<tr>
<td>testExecute_AddService</td>
<td>Part of ModifyTest.java; tests the part of the API that adds a Web Service to an already existing organization.</td>
</tr>
<tr>
<td>testExecute_AddServiceDescription</td>
<td>Part of ModifyTest.java; tests the part of the API that adds a Web Service description to an already existing Web Service.</td>
</tr>
<tr>
<td>testExecute_DeleteService</td>
<td>Part of ModifyTest.java; tests the part of the API that deletes a Web Service to an already existing organization.</td>
</tr>
<tr>
<td>testExecute_DeleteOrg</td>
<td>Part of ModifyTest.java; tests the part of the API that deletes an already existing organization.</td>
</tr>
<tr>
<td>testExecute</td>
<td>Part of AccessTest.java; tests the part of the API fetches the access URI for the Web Service from the registry.</td>
</tr>
</tbody>
</table>

modify
Inside modify
ServiceBinding is added
  key was urn:uuid:b1b95164-3670-4a25-bf5e-93326168aeea
Organization Modified
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization saved
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979

execute
modify
Inside modify
Organization Modified
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization saved
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979

execute
modify
Inside modify
<constraint>
<cpuLoad>load ls 1.0</cpuLoad>
<memory>memory geq 5MB</memory>
Figure 3.59. JUnit test results screen shot showing all the test cases were executed successfully.

```
<swapmemory>swapmemory geq 1GB</swapmemory>
<starttime>0700</starttime>
<endtime>2200</endtime>

</constraint>
ServiceDescription Added
key was urn:uuid:82c1e70f-9126-4a46-8a1c-cf2e37fe7b66
Organization Modified
key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization saved
key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
execute
modify
Inside modify
Service is Deleted
key was urn:uuid:b1b95164-3670-4a25-bf5e-93326168aeea
key was urn:uuid:82c1e70f-9126-4a46-8a1c-cf2e37fe7b66
Organization is deleted
key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization Modified
key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
```
Organization saved
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Tests run: 6, Failures: 0, Errors: 0, Time elapsed: 19.998 sec

------------- Standard Output ---------------
execute
modify
Inside modify
ServiceBinding is added
  key was urn:uuid:b1b95164-3670-4a25-bf5e-93326168aeea
Organization Modified
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization saved
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979

execute
modify
Inside modify
ServiceBinding is deleted
  key was urn:uuid:883b6233-f08d-40c9-ab48-8d263a5965a4
Organization Modified
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization saved
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979

execute
modify
Inside modify
ServiceBinding is added
  key was urn:uuid:b1b95164-3670-4a25-bf5e-93326168aeea
Organization Modified
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization saved
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979

execute
modify
Inside modify
Organization Modified
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization saved
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979

execute
modify
Inside modify
<constraint>
  <cpuLoad>load 1s 1.0</cpuLoad>
  <memory>memory geq 5MB</memory>
  <swapmemory>swapmemory geq 1GB</swapmemory>
<starttime>0700</starttime>
<endtime>2200</endtime>
</constraint>
ServiceDescription Added
  key was urn:uuid:82c1e70f-9126-4a46-8a1c-cf2e37fe7b66
Organization Modified
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization saved
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979

eexecute
modify
Inside modify
Service is Deleted
  key was urn:uuid:b1b95164-3670-4a25-bf5e-93326168aeea
  key was urn:uuid:82c1e70f-9126-4a46-8a1c-cf2e37fe7b66
Organization is deleted
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization Modified
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979
Organization saved
  key was urn:uuid:594d132d-7c44-45d9-88f3-95481e398979

------------- ---------------- ---------------
Testsuite: registry.PublishTest
execute
publish
Inside Publish
Organization saved
  key was urn:uuid:c0f43634-68f8-4044-956d-afc3fb00f3ed
Tests run: 1, Failures: 0, Errors: 0, Time elapsed: 5.971 sec

------------- Standard Output -------------
execute
publish
Inside Publish
Organization saved
  key was urn:uuid:c0f43634-68f8-4044-956d-afc3fb00f3ed

------------- ---------------- ---------------
Testsuite: registry.RegistryTest
access
Inside Access
*****TestWebServiceService
getBusinessLifeCycleManager
access
Inside Access
*****TestWebServiceService
getBusinessQueryManager
Tests run: 2, Failures: 0, Errors: 0, Time elapsed: 1.19 sec
---------- Standard Output ----------
access
Inside Access
*****TestWebServiceService
getBusinessLifeCycleManager
access
Inside Access
*****TestWebServiceService
getBusinessQueryManager
---------- ---------------- ---------------
test-report:
test:
BUILD SUCCESSFUL (total time: 46 seconds)
CHAPTER 4

RESULTS

4.1 PUBLISH ORGANIZATION AND WEB SERVICE

Listed here are the results for some common case scenarios involving manipulating freebXML registry contents.

The command line given below is used for this result. SampleProject is a Java project that uses code shown in Section 3.4.5 to access freebXML registry. The input to this program are two xml files action.xml and connection.xml.

```
java SampleProject "action.xml" "connection.xml"
```

The action.xml below is used here to publish contents to freebXML registry.

```
<root>
  <action type="publish">
    <organization>
      <name>San Diego State University (SDSU)</name>
      <description>
        San Diego State University (SDSU), founded in 1897 as San Diego Normal School, is the largest and oldest higher education facility in the greater San Diego area (generally the City and County of San Diego), and is part of the California State University system.
      </description>
      <postaladdress>
        <streetnumber>5500</streetnumber>
        <street>Campanile Drive</street>
        <city>San Diego</city>
        <postalcode>92182</postalcode>
        <state>CA</state>
        <country>US</country>
      </postaladdress>
      <telephone>
        <countrycode>1</countrycode>
        <areacode>619</areacode>
        <number>5945200</number>
        <type>OfficePhone</type>
      </telephone>
    </organization>
    <service>
      <name>NodeStatus</name>
      <description>
```

Service to monitor node status
</description>
<accessuri>
http://thermo.sdsu.edu:8080/NodeStatus/NodeStatusService
http://exergy.sdsu.edu:8080/NodeStatus/NodeStatusService
</accessuri>
</service>
</organization>
</action>
</root>

From the screen shot below we can verify that the contents were published to the registry successfully. See Figure 4.1.

![Screen shot of ebXML Registry Repository](image)

**Figure 4.1.** Search results showing organization and Web Service published.

### 4.2 Add Web Service

The action.xml is now modified to added Web Service *ServiceAdder* to organization *San Diego State University*.

```xml
<!DOCTYPE root SYSTEM 'RegistryAccess.dtd'>
<root>
  <action type="modify">
    <organization>
```
Screen shot in Figure 4.2 that ServiceAdder is now added to San Diego State University.

Figure 4.2. Search result after adding ServiceAdder to organization San Diego State University.
4.3 EDIT WEB SERVICE DESCRIPTION

The action.xml is now modified to edit service description of Web Service ServiceAdder. Below is the command line.

```
java SampleProject "action.xml" "connection.xml"
<root>
  <action type="modify">
    <organization>
      <name>San Diego State University (SDSU)</name>
      <service type="edit">
        <name>ServiceAdder</name>
        <description type ="edit">
          <constraint>
            <cpuLoad>load ls 1.0</cpuLoad>
          </constraint>
        </description>
      </service>
    </organization>
  </action>
</root>
```

The result is displayed in Figure 4.3 we can see description “load ls 1.0” added to ServiceAdder.

Figure 4.3. Search result after editing Web Service description.
4.4 DELETE A WEB SERVICE

The action.xml is now modified to delete Web Service ServiceAdder. Below is the command line.

```xml
<root>
  <action type="modify">
    <organization>
      <name>San Diego State University (SDSU)</name>
      <service type="edit">
        <name>ServiceAdder</name>
        <description type ="edit">
          <constraint>
            <cpuLoad>load ls 1.0</cpuLoad>
          </constraint>
        </description>
      </service>
    </organization>
  </action>
</root>
```

Results are shown in Figure 4.4.

Figure 4.4. Search result after deleting Web Service ServiceAdder.
4.5 **DELETE ORGANIZATION**

The action.xml is now modified to delete organization *San Diego State University (SDSU)*. Below is the command line.

```xml
<root>
  <action type="modify">
    <organization>
      <name>San Diego State University (SDSU)</name>
      <service type="delete">
        <name>ServiceAdder</name>
      </service>
    </organization>
  </action>
</root>
```

The result is shown in Figure 4.5.

![Image of ebXML Registry Repository](image)

**Figure 4.5.** Search result after deleting organization San Diego State University (SDSU).

4.6 **ACCESS WEB SERVICE**

Finally, action.xml is now modified to retrieve the access URI associate with Web Service *ServiceAdder*. Below is the command line.
Listed below are the access URIs associated with Web Service ServiceAdder.

http://thermo.sdsu.edu:8080/Adder/addService
http://exerby.sdsu.edu:8080/Adder/addService
CHAPTER 5

CONCLUSION

5.1 SUMMARY OF WORK

In this thesis we presented a load balancing scheme for ebXML registries such that a URI returned by a registry resolves to a host that satisfies different system constrains like current CPU load, physical memory, swap memory, and time of day. The proposed scheme is unique as it extends the existing freebXML registry implementation by adding load balancing features along with other QoS capabilities. The proposed scheme is completely transparent to an end user as no significant code changes are required by a user to utilize this load balancing scheme. It is possible to implement a MTC application using distributed Web Services in an effective way across multiple hosts where the CPU load and system memory is uniformly maintained.

5.2 FUTURE DIRECTIONS

The load balancing scheme suggested here uses a Web Service to determine the load and other parameters on a host machine. Any registry with the capability to publish and invoke a Web Service can be modeled similarly to incorporate load balancing feature. Parameters such as network delay can be added as one of the constraints used to rank the access URIs. Network delay takes into account network traffic and packet latency, thus access URIs for a Web Service are ranked on an estimated time required to access a particular Web Service deployed on multiple hosts.
BIBLIOGRAPHY


