THE COMMUNICATION MIDDLEWARE
FOR
COLLABORATIVE LEARNING OBJECT REPOSITORY
NETWORK

by

Ty Mey Eap
B.Sc. (Information Technology, TechBC), Simon Fraser University, 2003

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

In the School
of
Interactive Arts and Technology

© Ty Mey Eap 2006

SIMON FRASER UNIVERSITY
Summer 2006

All rights reserved. This work may not be reproduced in whole or in part, by photocopy or other means, without permission of the author.
# APPROVAL

<table>
<thead>
<tr>
<th>Name</th>
<th>Ty Mey Eap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Degree</td>
<td>Master of Science</td>
</tr>
<tr>
<td>Title of Thesis</td>
<td>The Communication Middleware for Collaborative Learning Object Repository Network</td>
</tr>
</tbody>
</table>

## Examing Committee:

<table>
<thead>
<tr>
<th>Chair</th>
<th>Dr. Christopher Shaw</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Associate Professor of School of Interactive Arts and Technology</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dr. Marek Hatala</th>
</tr>
</thead>
<tbody>
<tr>
<td>Senior Supervisor</td>
</tr>
<tr>
<td>Assistant Professor of School of Interactive Arts and Technology</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Vive Kumar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supervisor</td>
</tr>
<tr>
<td>Assistant Professor of School of Interactive Arts and Technology</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Matei Ripeanu</th>
</tr>
</thead>
<tbody>
<tr>
<td>External Examiner</td>
</tr>
<tr>
<td>Assistant Professor of Electrical &amp; Computer Engineering</td>
</tr>
<tr>
<td>University of British Columbia</td>
</tr>
</tbody>
</table>

## Date Defended/Approved:  July 20, 2006
DECLARATION OF
PARTIAL COPYRIGHT LICENCE

The author, whose copyright is declared on the title page of this work, has granted to Simon Fraser University the right to lend this thesis, project or extended essay to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users.

The author has further granted permission to Simon Fraser University to keep or make a digital copy for use in its circulating collection, and, without changing the content, to translate the thesis/project or extended essays, if technically possible, to any medium or format for the purpose of preservation of the digital work.

The author has further agreed that permission for multiple copying of this work for scholarly purposes may be granted by either the author or the Dean of Graduate Studies.

It is understood that copying or publication of this work for financial gain shall not be allowed without the author's written permission.

Permission for public performance, or limited permission for private scholarly use, of any multimedia materials forming part of this work, may have been granted by the author. This information may be found on the separately catalogued multimedia material and in the signed Partial Copyright Licence.

The original Partial Copyright Licence attesting to these terms, and signed by this author, may be found in the original bound copy of this work, retained in the Simon Fraser University Archive.

Simon Fraser University Library
Burnaby, BC, Canada
ABSTRACT

Web Service Description Language (WSDL) is designed for well-defined services. This requirement is essential for Web Services (WS) to ensure interoperability throughout a service lifecycle, support service-oriented architecture, and provide solid framework for business transactions. However, this constraint makes WSDL unsuitable for defining common interface functions for a collaborative system. This thesis argues that XML-based messaging middleware infrastructure is more effective and suitable for providing communication protocol for an evolving and collaborative system. The infrastructure can benefit from the well-established WS framework without depending on the WS technologies. We underpin our claim through the design and implementation of the eduSource Communication Layer (ECL). We show how XML-based messaging middleware can expand WS technologies to provide communication framework for an ongoing developing environment of learning object repository networks. Finally, we confirm the advantage of our approach using comparative analysis and use our experiences to write guidelines and recommendations for future development.

Keywords:

Learning object metadata, learning object, e-learning management, Web Services, software design, and XML-based messaging middleware
DEDICATION

This thesis is dedicated to my wife for all her love, support, and encouragement.
ACKNOWLEDGEMENTS

I am honoured and delighted to acknowledge the many people whose counsel, support, and encouragement have contributed immensely to the completion of my thesis.

My first and foremost gratitude goes to my senior supervisor, Dr. Marek Hatala. Without his continual support and guidance, this thesis would not have been possible. I am grateful for the research opportunities and appreciate his openness to listen and explore my research ideas. I also would like to thank him for helping me editing this thesis.

I would like to thank my thesis committee for their guidance and feedbacks. Thanks to Dr. Vive Kumar, my supervisor, for his support and the time we shared working in the LORNET project. I appreciate Dr. Matei Ripeanu for accepting to be my external examiner and for bringing new perspectives to this research thesis.

My sincere appreciation goes to Dr. Griff Richards. I am thankful for the time I shared working with him in various research projects and his incessant encouragement to pursue my academic goals.

I would like to thank the research team at the SFU Surrey. My appreciation goes to Dr. Dragan Gašević for the numerous hours of discussions in search of new ideas and his help for reviewing this thesis. I would like to thank Ashok Shah, Jordan Williams, and Jean-Francois Eap for their supports in the development of the ECL infrastructure.
Finally, I would like thank all the colleagues working in the eduSource, LORNET, and LionShare research projects for the debates, discussions, and feedbacks on the ECL infrastructure. Many improvements of the ECL infrastructure were possible due to their contributions.
# TABLE OF CONTENTS

Approval ............................................................................................................................. ii
Abstract ............................................................................................................................. iii
Dedication .......................................................................................................................... iv
Acknowledgements ......................................................................................................... v
Table of Contents ............................................................................................................ vii
List of Figures .................................................................................................................. x
List of Tables ................................................................................................................... xi
List of Examples And Programs ..................................................................................... xii
Glossary ........................................................................................................................... xiv

Chapter 1: Introduction ................................................................................................... 1
  1.1 Background ............................................................................................................. 1
  1.2 Motivation ............................................................................................................. 5
  1.3 Objectives ............................................................................................................. 6
  1.4 Scope ..................................................................................................................... 7
  1.5 Research Question ............................................................................................... 9
  1.6 Related Works ..................................................................................................... 10
  1.7 Thesis Structure ................................................................................................... 12

Chapter 2: The reviews of E-Learning Management and Web Services Technologies .................................................................................................................. 14
  2.1 E-Learning Technology ......................................................................................... 14
  2.1.1 Metadata ......................................................................................................... 15
  2.1.2 Metadata Query Language (MQL) ................................................................. 18
  2.1.3 IMS Content Package Protocol (IMS CPP) ................................................... 22
  2.1.4 Sharable Content Object Reference Model (SCORM) .................................... 23
  2.1.5 IMS Digital Repositories Interoperability (DRI) ............................................. 25
  2.1.6 Summary of E-Learning Technology ............................................................... 27
  2.2 Web Services Technologies .................................................................................. 27
  2.2.1 Simple Object Access Protocol (SOAP) ....................................................... 28
  2.2.2 Web Services Description Language (WSDL) ............................................. 29
  2.2.3 RPC Style Web Services ............................................................................... 31
  2.2.4 Document-style Web Services ...................................................................... 33
  2.2.5 The Usage of Type Definition: Literal versus Encoded .............................. 34
  2.2.6 Limitations and Constraints of Web Services .............................................. 35
  2.2.7 Next Generation of Web Services ................................................................. 38
<table>
<thead>
<tr>
<th>2.2.8</th>
<th>HTTP vs. Web Services</th>
<th>43</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2.9</td>
<td>Universal Description, Discovery, and Integration (UDDI)</td>
<td>44</td>
</tr>
<tr>
<td>2.3</td>
<td>Summary</td>
<td>45</td>
</tr>
</tbody>
</table>

### Chapter 3: The Design and the Implementation of the ecl infrastructure 49

<table>
<thead>
<tr>
<th>3.1</th>
<th>Design Principles</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>Selecting Implementation Platform for ECL Infrastructure</td>
<td>51</td>
</tr>
<tr>
<td>3.3</td>
<td>Architecture Design</td>
<td>52</td>
</tr>
<tr>
<td>3.3.1</td>
<td>Middleware</td>
<td>52</td>
</tr>
<tr>
<td>3.4</td>
<td>ECL Messaging Protocol</td>
<td>57</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Message Anatomy</td>
<td>58</td>
</tr>
<tr>
<td>3.4.2</td>
<td>Search Service Protocol</td>
<td>60</td>
</tr>
<tr>
<td>3.4.3</td>
<td>Gather Service Protocol</td>
<td>68</td>
</tr>
<tr>
<td>3.4.4</td>
<td>Submit Service Protocol</td>
<td>80</td>
</tr>
<tr>
<td>3.4.5</td>
<td>Request Service Protocol</td>
<td>84</td>
</tr>
<tr>
<td>3.4.6</td>
<td>Asynchronous Messaging Protocol</td>
<td>88</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
<td>89</td>
</tr>
</tbody>
</table>

### Chapter 4: The Integration Strategies of the ECL infrastructure 90

<table>
<thead>
<tr>
<th>4.1</th>
<th>ECL UDDI Registry Service</th>
<th>90</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Security Infrastructure</td>
<td>93</td>
</tr>
<tr>
<td>4.3</td>
<td>Algorithms</td>
<td>95</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Service Invocation Algorithm</td>
<td>95</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Service Dispatching Algorithm</td>
<td>96</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Domain Specific Query Algorithm</td>
<td>98</td>
</tr>
<tr>
<td>4.4</td>
<td>The Integration of the ECL Infrastructure with other Systems</td>
<td>99</td>
</tr>
<tr>
<td>4.5</td>
<td>Summary of the ECL Integration</td>
<td>103</td>
</tr>
</tbody>
</table>

### Chapter 5: The analysis of the ECL Infrastructure 105

<table>
<thead>
<tr>
<th>5.1</th>
<th>Architectural Comparative Analysis</th>
<th>106</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.1</td>
<td>SQI Architecture</td>
<td>110</td>
</tr>
<tr>
<td>5.1.2</td>
<td>OKI Architecture</td>
<td>118</td>
</tr>
<tr>
<td>5.1.3</td>
<td>SRU Architecture</td>
<td>121</td>
</tr>
<tr>
<td>5.1.4</td>
<td>SRW Architecture</td>
<td>124</td>
</tr>
<tr>
<td>5.1.5</td>
<td>Summary of the Architecture Analysis</td>
<td>127</td>
</tr>
<tr>
<td>5.2</td>
<td>Integration Requirements Analysis</td>
<td>128</td>
</tr>
<tr>
<td>5.2.1</td>
<td>SQI Integration</td>
<td>129</td>
</tr>
<tr>
<td>5.2.2</td>
<td>OKI Integration</td>
<td>131</td>
</tr>
<tr>
<td>5.2.3</td>
<td>SRU Integration</td>
<td>134</td>
</tr>
<tr>
<td>5.2.4</td>
<td>SRW Integration</td>
<td>135</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Summary of the Integration Analysis</td>
<td>136</td>
</tr>
<tr>
<td>5.3</td>
<td>Interoperability Comparison Analysis</td>
<td>137</td>
</tr>
<tr>
<td>5.3.1</td>
<td>The Interoperability between ECL and Other Protocols</td>
<td>138</td>
</tr>
<tr>
<td>5.3.2</td>
<td>The Interoperability of SQI</td>
<td>139</td>
</tr>
<tr>
<td>5.3.3</td>
<td>The Interoperability of SRU/SRW</td>
<td>140</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Summary of the Interoperability Analysis</td>
<td>141</td>
</tr>
<tr>
<td>5.4</td>
<td>Discussion</td>
<td>141</td>
</tr>
</tbody>
</table>
Guidelines for Developing LOR Communication Infrastructure

Protocol .................................................................................................................................................. 145

Choosing a Suitable Protocol Infrastructure ...................................................................................... 145
Implementation Guidelines .................................................................................................................. 146

Chapter 6: Conclusion .......................................................................................................................... 148

6.1 Future Work ................................................................................................................................... 149

Appendices ......................................................................................................................................... 151

Bibliography ....................................................................................................................................... 175
LIST OF FIGURES

Figure 2.1: IMS Content Package Framework .................................................... 23
Figure 2.2: Basic IMS DRI Architecture ......................................................... 26
Figure 2.3: BPEL4WS Execution Control ......................................................... 40
Figure 2.4: OWL-S Upper Ontology .............................................................. 43
Figure 3.1: An Overview of ECL Architecture .................................................. 54
Figure 3.2: RPC versus Document Style Web Service ..................................... 55
Figure 4.1: The ECL Security Infrastructure ..................................................... 95
Figure 4.2: An Overview of the Integration of ECL Infrastructure ..................... 99
Figure 4.3: The Client Gateway Architecture ................................................. 101
Figure 4.4: The Server Gateway Architecture for ECL Clients ....................... 102
Figure 4.5: The Gateway Architecture for Clients of other Protocols ............. 103
Figure 5.1: An Overview of the SQI Architecture ........................................... 110
| Table 2.1: | CQL Query Examples ........................................................................................................... 21 |
| Table 2.2: | SCORM Main Topics ............................................................................................................ 24 |
| Table 4.1: | The Search Service Invocation Algorithm ................................................................. 96 |
| Table 4.2: | The Search Service Handling Algorithm ....................................................................... 97 |
| Table 4.3: | The Domain Specific Query (DSQ) Utility Algorithm .................................................. 98 |
| Table 5.1: | The Evaluation of the ECL Architecture ..................................................................... 109 |
| Table 5.2: | The Evaluation of the SQI Architecture ...................................................................... 117 |
| Table 5.3: | The ECL/OSID Implementation Issues and Solutions .................................................. 119 |
| Table 5.4: | The Evaluation of the OKI Architecture ...................................................................... 121 |
| Table 5.5: | The Evaluation of the SRU Architecture ..................................................................... 124 |
| Table 5.6: | The Evaluation of the SRW Architecture .................................................................... 127 |
| Table 5.7: | The Evaluation Analysis Matrix of the Architecture Supported Features ...................... 128 |
| Table 5.8: | The Evaluation of the SQI Integration Analysis ......................................................... 131 |
| Table 5.9: | The Evaluation of the OKI Integration Analysis ......................................................... 133 |
| Table 5.10: | The Evaluation of the OKI Integration Analysis ......................................................... 135 |
| Table 5.11: | The Evaluation Analysis Matrix of the Integration Criteria ......................................... 136 |
| Table 5.12: | The Evaluation Analysis Matrix of the Interoperability Requirements ......................... 141 |
LIST OF EXAMPLES AND PROGRAMS

Sample 2.1: An Example of a WSDL file for RPC/encoded Web Services.......................... 32
Sample 2.2: An Example of Document Style WSDL............................................................. 33
Sample 2.3: A Skeleton Sample of a BPEL4WS Process Definition................................. 40
Sample 3.1: A Skeleton of the ECL Message......................................................................... 60
Sample 3.2: An Example of the ECL Search Request Messaging Protocol......................... 63
Sample 3.3: An XQuery Example ......................................................................................... 64
Sample 3.4: A HashMap Query Example ............................................................................ 65
Sample 3.5: A Skeleton of the Search Expose Messaging Protocol.................................... 66
Sample 3.6: Search Service Handler Interface Function.................................................... 66
Sample 3.7: The ECL Client Code for Building Query and Invoking an ECL Search Service ................................................................................................................. 67
Sample 3.8: An Example of the Gather Identity Request Protocol .................................. 71
Sample 3.9: An Example of the Gather Identity Expose Protocol ..................................... 71
Sample 3.10: An Example of the Gather GetRecord Request Protocol ......................... 72
Sample 3.11: An Example of the Gather GetRecord Expose Protocol ............................... 72
Sample 3.12: An example of the Gather ListIdentifiers Request Protocol ...................... 74
Sample 3.13: An Example of the Gather ListIdentifiers Expose Protocol ........................ 74
Sample 3.14: An Example of the Gather ListMetadataFormats Request Protocol ........... 75
Sample 3.15: An Example of the Gather ListMetadataFormats Expose Protocol ............ 75
Sample 3.16: An Example of the Gather ListRecords Request Protocol .......................... 76
Sample 3.17: An Example of the Gather ListRecords Expose Protocol ............................. 77
Sample 3.18: An Example of the Gather ListSets Request Protocol .................................. 78
Sample 3.19: An Example of the Gather ListSets Expose Protocol .................................... 78
Sample 3.20: Gather Service Handler Interface Functions................................................ 79
Sample 3.21: A Client Code Example for Invoking an ECL Gather Service for all Six Actions ...................................................................................................................... 79
Sample 3.22: A Skeleton of the ECL Submit Request Protocol ......................................... 82
Sample 3.23: A Skeleton of the ECL Store Protocol ........................................................... 83
Sample 3.24: Submit Service Handler Interface Function .................................................... 83
Sample 3.25: Client Code Sample for Invoking ECL Submit Service ................................. 84
Sample 3.26: An Example of the ECL Request Protocol..................................................... 86
Sample 3.27: A Skeleton of the ECL Deliver Structure ....................................................... 87
Sample 3.28: Request Service Handler Interface Function................................................ 87
Sample 5.1: A Sample of the SRW Request as Shown on SRW Web site (No Binding SOAP Message) ......................................................... 126
Sample 5.2: An Example of a SOAP Message with Binding ........................................ 126
Sample 5.3: A Sample of a SRW Response .................................................................. 126
GLOSSARY

BPEL4WS  Business Process Execution Language for Web Services
CanLOM  Canadian Learning Object Metadata Repository
CAREO  Campus of Alberta Repository of Educational Objects
CORBA  Common Object Request Broker Architecture
CPP    Content Packaging Protocol
CQL    Common Query Language
DAML   DARPA Mark-up Language
DC     Dublin Core
DCMI   Dublin Core Metadata Initiative
DCOM   Distributed Component Object Model
DOI    Digital Object Identifier
DRI    Digital Repositories Interoperability
ECL    The eduSource Communication Layer
EdNA   Education Network Australia
FIRE   The EUN Federation of Internet Resources for Education
HTTP   Hypertext Transfer Protocol
JMS    Java Message Service
LCMS   Learning Content Management System
LMS    Learning Management System
LO     Learning object
LOM    Learning Object Metadata
LOR    Learning object Repository
LORNET  Canadian Learning Object Repository NETwork
MERLOT Multimedia Educational Resource for Learning and Online Teaching
MIME   Multipurpose Internet Mail Extensions
MQL    Metadata Query Language
OAI    Open Archive Initiative
OASIS  Organization for the Advancement of Structured Information Standards
OKI    Open Knowledge Initiative
OSID   Open Source Interface Definitions
OWL-S  OWL-based Web Service Ontology
PIF    Package Interchange File
RDF    Resource Description Framework
RMI    Remote Method Invocation
RPC    Remote Procedure Call
SCO    Shared Content Object
SCORM  Sharable Courseware Object Reference Model
SMETE  Science, Mathematics, Engineering and Technology Education
SMTP  Simple Mail Transfer Protocol
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>SQI</td>
<td>Simple Query Interface</td>
</tr>
<tr>
<td>SRU</td>
<td>Search/Retrieval via URL</td>
</tr>
<tr>
<td>SRW</td>
<td>Search/Retrieval Web Service</td>
</tr>
<tr>
<td>TCP/IP</td>
<td>Transmission Control Protocol/Internet Protocol</td>
</tr>
<tr>
<td>UDDI</td>
<td>Universal Description, Discovery and Integration</td>
</tr>
<tr>
<td>URN</td>
<td>Uniform Resource Name</td>
</tr>
<tr>
<td>W3C</td>
<td>World Wide Web Consortium</td>
</tr>
<tr>
<td>WS</td>
<td>Web Services</td>
</tr>
<tr>
<td>WSDL</td>
<td>Web Services Description Language</td>
</tr>
<tr>
<td>XML</td>
<td>Extensible Mark-up Language</td>
</tr>
<tr>
<td>XQuery</td>
<td>XML Query Language</td>
</tr>
<tr>
<td>XSD</td>
<td>XML Schema Definitions</td>
</tr>
<tr>
<td>XSLT</td>
<td>Extensible Style sheet Language Transform</td>
</tr>
</tbody>
</table>
CHAPTER 1: INTRODUCTION

1.1 Background

The growing popularity and necessity of online education has researchers anticipating an abundance of learning materials. Without a doubt, making these learning materials available for reuse will revolutionize the learning industry. The abundance of learning resources will allow instructors to choose the learning materials that are the most suitable for their students. Moreover, if the learning materials are granular and self-contained instructional units, then it will be possible to use them to compose training, courses, or more complex instructional units [5, 23, 26, 50, 59, 79, 90]. Anticipating that learning materials will be designed as self-contained instructional units, people have begun to call any learning material a learning object (LO). The potential of learning objects is enormous, but achieving this potential requires progress in all areas of learning technology research.

Since the late nineties, there have been many initiatives to enable the sharing of LOs. The first initiative was the definition of learning object metadata (LOM), which laid the foundation of e-learning management. LOM bridges human knowledge to machine knowledge. It enables learning designers to describe a LO so that it is understandable to a machine [24, 34, 74]. As a result, machines can perform operations such as searching or classifying LOs based on the associated LOM records. The term learning object repository (LOR) refers, primarily, to a LOM management system. The next emergent technology was the content packaging of definitions and specifications.
The technology provides definitions and guidelines for learning designers to package learning materials into an instructional unit [5, 23, 26], which is a LO at a coarser level. This makes LOs easier to manage, and highly interoperable. Finally, to further promote the development of LO-based learning technology and exchange of LOs, organizations around the world began working on the infrastructure for LOR networks. The infrastructure allows end-users to access resources on any repository across the LOR network.

This leads to the subject of this thesis, which is the development of communication infrastructure protocol that bridges individual LORs into a LOR network. Based on our studies of the LOR network within the eduSource project (http://www.edusource.ca/english/home_eng.html) we have proposed an XML-based messaging middleware infrastructure called the eduSource Communication Layer (ECL). The eduSource project is an 18-month Canadian LOR network initiative that started in 2002. The work continued in the Canadian Learning Object Repository NETwork (LORNET) [51]. It was apparent from day one that the eduSource LOR network has to be designed as an evolving collaborative network. LORs that formed the initial eduSource network were either new repositories or still under development. The repositories were evolving rapidly as they tried to keep pace with changing technologies, and each repository is a unique entity. In a LOR network, these independent repositories have to collaborate in providing a common interface for end-users to access resources on the network. We stipulated that XML-based messaging middleware infrastructure has to provide stability for an evolving collaborative network such as the eduSource LOR network.
The decision to use XML-based messaging protocol was easy, due to the maturity of XML technology. Because it was already the mainstream technology, its usage did not add additional technology to the pre-existing base used by LORs. However, XML-based messaging communication expands the flexibility of the Web Services (WS) technologies. The first aspect of flexibility is providing a mapping for ECL messages with respect to metadata query languages and different LOM formats. The mapping strategy allows calling-application as well as LOR developers to use any type of known query languages and LOM formats. The second aspect of flexibility is making the service generic and configurable. Since an ECL request message is XML-based, ECL infrastructure can process the message and redirect the message to the appropriate service handler. On the Java platform, ECL can load a service handler, a Java-class implementation, provided by LOR developers from a configuration file and use it to process incoming ECL requests. In addition, XML-based messaging communication enables the development of different strategies that make ECL highly interoperable, adaptive, and maintainable. The first important strategy we employed was to use the middleware and the UDDI registry to coordinate message mapping, software updates, and service discovery. The second important strategy was the modularization of the ECL infrastructure design. The middleware and the UDDI registry strategy takes care of the changes within the ECL infrastructure, while the modularization of the ECL infrastructure tolerates the changes of technologies used in the building of ECL.

ECL was not the only proposed infrastructure for a LOR network. Other initiatives have also been working on their infrastructure solutions. A comparative analysis with other approaches towards the building of a communication infrastructure
such as ECL should prove the validity of the arguments put forward when building the ECL. The selected alternative protocols and infrastructures are Search/Retrieval Web Service (SRW), Search/Retrieval by URL (SRU), Simple Query Interface (SQI), and Open Source Interface Definitions (OSID). SRU and SRW are standard search and retrieve protocols based on Z39.50 semantics [83, 84]. SRU differs from SRW only in the transport protocol: SRU uses a standard HTTP form-based request while SRW uses a SOAP message-based request. Both use Common Query Language (CQL), which is a formal language for representing Web indexes, bibliographic catalogues, and museum collections [82]. SRW/SRU provides protocol definition to allow LOR developers to implement SRW/SRU services, which are accessible by calling applications using SRW/SRU protocol. Simple Query Interface (SQI) is a European initiative led by ARIADNE Foundation [4], EUN Federation of Internet Resources for Education (FIRE) [91], and ELENA project [22]. SQI is focusing on providing a common query interface for LOR networks, which is a communication infrastructure similar to ECL. The last protocol we examine is OSID, which is a collection of interface definitions that enable the independent implementation of calling applications and service providers [16]. OSID has been proposed by the Open Knowledge Initiative (OKI) [89].

These four approaches have many characteristics that are similar to ECL. All of them target collaborative LOR networks, which is the main criterion that qualifies them for the comparative analysis. Each solution has some advantages and disadvantages, depending on the type of network. The particular view of the LOR network characteristics influences the selection of technologies and the design of the communication infrastructure. The comparative analysis provides a better understanding
of the requirements of the LOR networks, WS technologies, and the technologies for e-
learning management in general. This understanding can help in the future development
of a communication infrastructure for a collaborative and evolving environment similar
to the LOR network.

1.2 Motivation

Sharing information is a winning business model and provides a strategic
advantage [12, 13, 14]. The Global Learning Objects Brokered Exchange (GLOBE)
initiative (http://globe.edna.edu.au/globe/go) is a good example of strategic alliances.
ARIADNE, EdNA, LORNET, MERLOT, and National Institute of Multimedia
Education (NIME) in Japan (http://www.nime.ac.jp/index-e.html) have committed to
work collaboratively on a shared vision of ubiquitous access to quality educational
content. As online learning becomes predominant in the learning environment, the LOR
network is an attractive solution for providing faster, cheaper, and better learning
resources. Furthermore, learning communities do not have to wait for the return on their
investments. The benefits of a LOR network can be immediate. Although most of
available learning materials are not self-contained instructional units, learning designers
and students are still able to use the material as a reference or as an example. In some
cases, learning designers can integrate, with some modification, discovered learning
material in their instructional unit. Moreover, the LOR network infrastructure stimulates
and eases the development of learning technologies.

Undoubtedly, the most valuable learning technology to a LOR network is the
development of reusable LOs. Reusability is the central idea of the LOR network and is
the key to a collaborative environment that promotes the design and development of
reusable LOs. The research in learning technology, especially in LO design, clearly supports this view. There is research currently underway to develop tools that assist learning designers in developing reusable LOs and in enhancing LO quality [62]. Meanwhile, previous research already anticipated the LO to be a self-contained instructional unit [5, 23, 50, 59, 79, 90], and there is ongoing research into the methodology of building specialized training or courses from reusable LOs, based on learners’ competencies and academic background [23, 64]. The previous research shows that learning technology is a network of research endeavours, and the success of LOR network is central to their success as it provides an integrated environment for new research ideas to grow. However, at the same time, the development of the LOR network depends on other research contributions to provide tools such as automatic generation of metadata or packaging learning materials into reusable LOs. Providing an integrated environment abstracts the complexity of the technologies and hence, is a key to accepting the technology. Based on this view, this research proposes a flexible, expandable, and maintainable communication infrastructure that provides stability to the collaborative and still-evolving environment of a LOR network.

1.3 Objectives

The first objective is to develop the ECL infrastructure to enable existing repositories to cooperate in a LOR network and to encourage the development of new repositories. To achieve this objective, ECL has to pay close attention to small and medium repositories, which do not have the resources or materials to maintain as the repository of a large institution [19]. When joining a network, some repositories have concerns about copyrights, access policies, and security. ECL must find a way to deal
with these concerns. The benefits of sharing and reusing LOs may not be sufficient to motivate LOR developers to join a LOR network. As with all investors, developers have to evaluate the benefits and costs. In the early stages of development, the costs are typically higher than the benefits. Therefore, ECL must be cost-effective. Hence, the integration of the ECL onto an existing or new LOR should be easy. In addition, ECL can provide incentive benefits to the LOR developers such as the interoperability with other mainstream protocols, and potentially, the usage of the ECL for other purposes.

The primary goal of ECL is to provide a communication interface for accessing a LOR network, but a secondary objective is to design the technology at the generic level and export it for other uses. ECL should be versatile and adaptable for other uses such as providing for an e-portfolio, user modelling, or competency services. Therefore, modular design is important in ensuring that parts of the technology are exportable and interchangeable.

Finally, the research goal is to conduct a comparative analysis of the infrastructure against the four selected infrastructure protocols to validate the merit of the research arguments, and use this analysis to provide guidelines for future development and improvement of the communication infrastructure.

1.4 Scope

This research is only concerned with the technological aspect of the LOR network. We built the requirements and specifications for the design of ECL infrastructure from discussions with the former eduSource partners and the types of repositories they operate. Since the LOR network was, and still is, a new technology, we
had to make some assumptions on the needs and incentive requirements for developers to join such a network. Enabling learning materials in public institutions such as libraries is not in the scope of this research thesis. Large institutions are stable and have well-established protocols for metadata tagging and classifying of their materials. These are mostly archive-type repositories. The materials are well defined and self-contained (i.e., books, journals, proceedings, articles, etc.) Making these materials accessible is not a challenge. A simple search-and-retrieve protocol is sufficient to make the materials available to the public. DSpace is one such successful example. As claimed by DSpace developers [6], DSpace is an institutional repository system that captures, stores, indexes, preserves, and distributes digital research materials. Many libraries have already used DSpace to facilitate the management of research papers.

In addition, an institutional repository needs considerable financial support and must have a significant amount of learning materials. Most LORNET partners do not have the resources to set up an institutional repository. For example, the Canadian Learning Object Metadata (CanLOM) Repository had to halt operation at the end of 2005. Campus of Alberta Repository of Educational Objects (CAREO) and Explor@-2 of LICEF Research Centre of the Télé-université du Québec are the two sole institutional repositories from the original working group that are still in operation.

Nevertheless, successful institutional repositories do exist. These repositories are typically large and have sustainable funding. Education Network Australia (EdNA) [21] has generated a good community of practice. According to an EdNA report on April 2006, in 2005 EdNA obtained support from 500 online communities, and had over 29,000 Web resources [20]. Two other successful repositories are Multimedia
Educational Resource for Learning and Online Teaching (MERLOT) [55] and Science, Mathematics, Engineering and Technology Education Open Federation (SMETE) [80]. Each has more than 15,000 LOs.

However, the number of available LOs is rather small despite the considerable effort learning communities have put into building LORs. The quality is still below standard. The resources needed for an institutional repository is substantial. The cost for maintenance is high, compared to those of a LOR network that consists of small repositories. An institutional repository has to continually promote the LOR, for example, by developing a social network for the LOR, which is the approach EdNA has taken. This shows that if there is a steady funding, an institutional LOR is an excellent way to promote the reuse of LOs.

1.5 Research Question

The underlying research question is whether XML-based messaging middleware can support a sustainable communication framework for an evolving, collaborative environment such as a LOR network. This research discusses the limitations of current WS technologies, the flexibility of messaging protocol, benefits of XML-based messaging technology, and different models of the LOR networks. The research gives a better understanding of LOR network requirements and Internet communication technology. This understanding allows us to formulate guidelines and recommendations for future development and improvement of the LOR network communication infrastructure.
1.6 Related Works

This section discusses related works that also research on solutions for e-learning management systems. The approaches of these related works do not have features for comparative analysis with the ECL. Hence, they are not selected for comparative analysis of this thesis.

The first related work that can create an impact on e-learning management systems is the Content Object Repository Discovery and Registration/Resolution Architecture (CORDRA), an open standards-based model. The CORDRA initiative has its roots in the US Department of Defense's Advanced Distributed Learning (ADL) initiative (http://www.adlnet.org), well known for the Sharable Content Object Reference Model (SCORM) [44]. CORDRA aims to identify and specify (not develop) appropriate technologies and existing interoperability standards that can be combined into a reference model that will enable learning content to be found, retrieved, and re-used (http://cordra.net/). The model assumes a set of locally managed repositories, augmented by an object identifier infrastructure, common services and applications, and three kinds of repositories: a master catalogue that registers the content held in the federation, a registry of participating repositories, and a system registry [44]. According to Kraan’s report on the First International CORDRA™ Workshop on March 15, 2005 [44], the stakeholders in e-learning management had raised numerous issues on CORDRA approaches. Consequently, the workshop organizer, Dan Rehak, emphasized in his summary that CORDRA would not be finished for a while yet and the workshop was only the first step. Finally, Kraan concluded in his report that for CORDRA to be
successful, CORDRA must determine the minimal components that are technically feasible.

The next related work is GLOBUS, a community of users and developers who collaborate on the use and development of open source software as well as associated documentation for distributed computing and resource federation [27]. GLOBUS released GLOBUS Toolkit (GT) in the late 1990s, a set of libraries and programs addressing common problems that occur when building distributed system services and applications. Once installed, a GT becomes a grid cell that can be configured to communicate with other GT grids that are part of a GLOBUS grid ecosystem. GT is another example of a middleware that uses WS technologies its core components. The first two steps for implementing a GT Web service are similar to the implementation of a Web service from scratch without using the SOAP utility: defining the service’s interface by writing a WSDL file for the service and implementing the service in Java. Then, the developers must prepare the service by writing the configuration parameters into a configuration file and compiling everything into Grid Archive or GAR file. Finally, developers can use GT tool to deploy the service into the grid ecosystem.

There were some interests in using GLOBUS framework for e-learning management system. Reklaitis et al. proposed in 2003, the mapping of e-learning components onto a set of standard GLOBUS services to accommodate the distribution of e-learning applications on the grid and to view the applications as typical grid services [67, 68]. However, Reklaitis admitted that he and his team needed to solve many questions before they can realize an e-learning implementation according to the grid service specifications. In the last two years, the interest in the GLOBUS for e-learning
management system seems to disappear. GLOBUS is a close system. Each e-learning application must be implemented as a GLOBUS service. GLOBUS is suitable for managing resources of an organization, but it is unreasonable to impose the installation and the maintenance of a GLOBUS system on an existing repository.

Finally, LionShare and SPLASH deserve some considerations as related works. Both are distributed peer-to-peer network of object repositories, and both had integrated the ECL infrastructure to access the ECL network. LionShare is a secure peer-to-peer network sitting on top of the Limewire infrastructure, which is a peer-to-peer system based on Gnutella protocol (http://www.limewire.com/english/content/home.shtml). SPLASH was a project funded in part by the CANARIE Learning Program [69]. SPLASH was originally built on Sun Microsystems JXTA platform but latter replaced JXTA with its own peer-to-peer system that is more efficient and capable of communicating across firewalls. SPLASH was our exploratory research. We had gained valuable experiences in the development of SPLASH, notably in the metadata and metadata query mapping, and had transcended these experiences onto the development of the ECL infrastructure.

1.7 Thesis Structure

This section discussed the need for a flexible communication infrastructure and the origin of this research thesis. Chapter 2 reviews several key protocol standards and technologies used in e-learning management systems and in Internet communication. The review looks inside the technologies, at the history, the advantages, and the limitations of a given technology. Chapter 3 provides detailed discussions and descriptions of the ECL infrastructure implementation as well as messaging protocols used in different types of
communications. The discussion of ECL integration takes place in Chapter 4. The integration includes a brief discussion of other ECL implementations such as the ECL security and the ECL discovery service. Chapter 5 focuses on the comparative analysis of ECL against four selected infrastructures. The chapter concludes with the discussion of the analysis and guidelines. Finally, Chapter 6 provides the summary and the conclusions of this research thesis.
CHAPTER 2: THE REVIEWS OF E-LEARNING MANAGEMENT AND WEB SERVICES TECHNOLOGIES

This chapter reviews the technologies in the area of e-learning management and WS technologies. The goal of this review is to understand the technologies and to verify the research statements. The review begins with the emergent technologies in e-learning management: Learning Object Metadata (LOM), Metadata Query Language (MQL), Content Packaging Protocol (CPP) specifications, and IMS Digital Repository Interface (DRI). The development of these technologies supports the ECL view of the LOR network, and it shows that ECL infrastructure is a complementary technology. The review also discusses various strategies to resolve problems that come with these technologies. For example, multiple LOM standards are one of the issues. Most LOR developers choose a standard that is suitable and cost-effective for them. The lack of interoperability is the price to pay in a heterogeneous collaborative network, and the question is, how should ECL deal with this issue? The MQL poses a similar problem. LORs use different MQLs. What are the possible strategies for dealing with the MQL issue? Finally, the review unearths the WS technologies, and tries to verify the statement of the research thesis that WS technologies are best suited for well-defined services, and are not suitable for evolving environments such as a LOR network.

2.1 E-Learning Technology

The first part of this section discusses the trends and history of the metadata development. Metadata standard is the first technology designed to enable the
management of LOs. Next is the review of MQL. Complex data structures found in metadata require complex query languages. XQuery appeared to be the query language for metadata [19], but the complexity of the language forced LOR developers to find other alternatives. Currently, there is an initiative headed by ARIADNE, FIRE, and ELENA to develop common MQL standard for SQI [78]. The outcome of the research may produce a new MQL that the ECL infrastructure will need to handle. Finally, the last two sections contain reviews of IMS DRI interface functions and the CPP standards. These standards continue to have new amendments to support the new development of learning technology. It is important to understand the development trend of these standards in order to design a framework that can adapt to the changes.

2.1.1 Metadata

W3C defines metadata as machine-understandable information for the Web [86]. Metadata provides a structure and encoded information, typically in XML format, about a LO, which enables a machine to classify, sort, and search a collection of metadata [86]. LOR is primarily a repository of LOM records. LOs can be located in the same or another server. Providing that the system has adequate security clearance, the system should be able to access the LO based on the information in the LOM record. Since each LOM record is associated with a LO, the system can perform all operations on the collection of LOM records. Hence, LOM is the foundation for e-learning management.

Dublin Core Metadata Initiative (DCMI) is one of the pioneers in metadata standardization efforts [35, 88]. DCMI proposed DC metadata at the second International World Wide Web Conference in October 1994 [3]. The first DC metadata standard has a dozen basic elements that are sufficient to identify an object. Since then, there have been
few amendments to the original standard. However, DC remains the smallest set of metadata definitions, and because of its simplicity, DC is the most commonly used metadata format in the industry. In 1998, IMS [1] and ARIADNE submitted a joint proposal and specifications to IEEE LOM. Both organizations continued to develop their standards until the effort culminated in July 2002 with the publication of IEEE LOM (IEEE 1484.12.1) [93]. EdNA also presented a standard in 1997 and has been developing and using it ever since [93]. The usage of metadata standards is far from unanimous. Some research projects decide to create their own metadata format because the standards do not meet their requirements, while others are expending standards to meet their needs [63, 64, 76]. CanCore, a Canadian metadata initiative, redefined IEEE LOM by chopping down unused or not-yet-needed metadata elements. In the meantime, semantic communities insist on representing the standards in RDF format so that a semantic Web tool can use the metadata as it is. It is fair to assume that this trend will continue for two reasons. First, reusable LOs are not yet at the stage where developers can accurately define the required metadata elements. Second, it is impossible to provide a single definition and specification to describe all types of LOs.

As a result, any infrastructure dealing with metadata should be able to handle multiple metadata standards, and perform metadata mapping when possible. This solution enables developers to use their preferred metadata, which is the approach taken by ECL.

Problem 1: ECL must deal with the issue of multiple metadata formats.
2.1.1.1 Metadata Mapping

Mapping between two metadata standards can result in the loss of important information. For a mapping to be successful, there must be a one-to-many or one-to-one condition. Currently, mapping from DC to IEEE LOM does not lose any information, but mapping from IEEE to DC loses a considerable amount of data. Many IEEE LOM elements are undefined in DC. Some may argue that the loss of these data is not significant since DC users do not use the additional data in the IEEE LOM. This is true in most cases, but in some specific cases the loss is significant (e.g., the loss of IEEE LOM identifier references). The mapping guidelines advise mapping IEEE identifier and IEEE location to DC identifier [94]. This is many-to-one mapping, which is impossible to map without further processing. Another problem is the catalogue entry in the IEEE LOM identifier. IEEE LOM intends to provide flexibility by associating catalogue and identifier for multiple identifiers. However, this flexibility creates a problem in metadata mapping. An IEEE LOM repository may use a generic URL for location, and to request a LO. The repository requires the use of both identifier and location. Hence, the metadata mapping needs additional information. To resolve such ambiguity the two systems must agree on the handling of identifiers (i.e., combining location with identifier). ECL can resolve this problem by using UDDI registry to coordinate the agreements between two systems.

These are solvable problems. Mapping is possible if an element in the metadata specification A has a direct or indirect reference to the element in the metadata specification B; otherwise, no computer system can do the mapping automatically. As a result, 100 percent interoperability is not always possible. A calling application that
works on a limited number of metadata elements should be able to access all repositories on the LOR network, while the calling application using special metadata can only access repositories that support similar metadata requirements.

Solution 1: ECL supports multiple metadata formats with metadata mapping and hiding non-interoperable repositories strategy. When searching a UDDI registry, ECL only presents compatible services to the end-users.

2.1.2 Metadata Query Language (MQL)

MQL is an area that needs more research. One of the problems is the incomplete specifications and requirements of the metadata query. However, this cannot be completed until the e-learning technology becomes more mature. The second problem is the complex structure of the metadata format, especially IEEE LOM. Although supporting multiple languages, identifiers, taxonomies, etc. accommodate a wide audience, this flexibility requires a complex query language. Early in the development of LOR network research, many researchers thought that XQuery [7] was a good candidate for a metadata query language. It made good sense since metadata is represented in the XML format. As a result, eduSource adopted XQuery as the query language for ECL [19]. However, XQuery never caught on. XQuery language is too complex, and most XQuery databases are not recognized. The design of ECL had anticipated the XQuery problem, and developed an internal query structure that transforms a common query into XQuery, and proposes a restricted XQuery syntax for the eduSource.
Problem 2: ECL must deal with the issue of multiple query languages.

To manage complex queries, ECL separates the query into two parts: text query, and domain-specific query. Text query is applicable to selected text elements of the metadata such as title, description, or keywords. Text query is for human end-users. Domain-specific query determines the characteristics of the LOs (e.g., education levels, age range, object format, etc.) By separating the query into two parts, ECL is able to isolate the domain-specific query, the area that needs attention. Although the text-based search is not adequate in some LORs, all LORs support some kind of text search. Note that a text query based on the relational database may be not adequate for a text search. LOR developers should use a text search engine to enrich their text query capability. Next, to deal with a domain-specific query, ECL uses its internal common query structure to handle the query, and supplies a domain-specific query engine that can perform a query on the search response from a LOR.

Solution 2: ECL supports multiple query languages with an internal common query structure, and splits metadata query into text query and domain-specific query.

A domain-specific query needs more research. New requirements will emerge as researchers continue the integration of the LOR network into the learning design environment. The immediate concern is the text query. This section reviews three dominant text search languages: Google, Common Query Language (CQL) and Lucene.
Google has gained tremendous popularity with its simple, tolerant text search. The Google search query is a combination of keywords called term. The order of keywords is a search criterion as well, and can influence the search results. By default, Google search ignores stop words such as “on” or “of” and considers all non-alphanumeric characters with the exception of some special characters (e.g., double quote mark ("), plus sign (+), minus sign or hyphen (-), and ampersand (&)) as white spaces. Google only considers the “OR” operator, while the “AND” operator is implicit. In addition, Google offers simple rules for additional filters such as filtering language with “<fr>” or search all text with “(all)” [29]. Note that if ignoring filter syntax, Google text search syntax can be mapped to a relational database SQL query (i.e., if the user input is “the text filter”, then the Google query would be “text filter”, and SQL query argument would be “LIKE ‘%text%filter%’”).

Some repositories such as SMETE and ECL-POOL use the Lucene text search engine for the text search [66]. Lucene is open source software, and freely available for download, which can be a factor in the development of LOR network technology. In addition, Lucene can perform a text search on tabular data. The syntax consists of terms, fields, and operators. The terms are regular expression (e.g., “The Right Way”, “Goo~”, “G*gle”, or “Goo?le”). The fields are column names (e.g., title, description, etc.) Finally, the operators are “OR” or “AND”. An example of Lucene query can be following:

```
```

Undoubtedly, Lucene can be used to store metadata’s text elements, and be used as a primary basic metadata query.

CQL has very strong support from repositories such as library and government document archives. As mentioned earlier, it is a formal language for representing queries
to information retrieval systems such as web indexes, bibliographic catalogues, and museum collection information [82]. The Library of Congress is currently in charge of maintaining the CQL standard, which explains the overwhelming support of CQL from librarians. CQL is the core query protocol of SRW/SRU. The query is a combination of words and operators. Table 2.1 shows some examples of CQL query.

Table 2.1: CQL Query Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>title all &quot;complete dinosaur&quot;</td>
<td>Title contains all of the words: &quot;complete&quot;, and &quot;dinosaur&quot;</td>
</tr>
<tr>
<td>title any &quot;dinosaur bird reptile&quot;</td>
<td>Title contains any of the words: &quot;dinosaur&quot;, &quot;bird&quot;, or &quot;reptile&quot;</td>
</tr>
<tr>
<td>(caudal or dorsal) prox vertebra</td>
<td>A proximity query: either &quot;caudal&quot; or &quot;dorsal&quot; near &quot;vertebra&quot;</td>
</tr>
<tr>
<td>subject any/relevant &quot;fish frog&quot;</td>
<td>Find documents that would seem relevant either to &quot;fish&quot; or &quot;frog&quot;</td>
</tr>
<tr>
<td>subject any/rel.lr &quot;fish frog&quot;</td>
<td>Same as previous, but use a specific relevance algorithm (linear regression)</td>
</tr>
<tr>
<td>date within &quot;2002 2005&quot;</td>
<td>Select date within 2002 and 2005</td>
</tr>
</tbody>
</table>

Source: Based on the examples shown on SRW/SRU website: http://www.loc.gov/standards/sru/cql/index.html

Text query targets text elements in a specific language. Most metadata standards support multi-languages elements. For example, an IEEE LOM may have titles or descriptions in different languages. To perform a query, the system must know the target language for the elements. In addition, the language for the LO and the language for the LOM record describing the LO can be different, which can cause some confusion in the metadata query. To deal with this problem, ECL adds the target language of the metadata into the query structure. ECL automatically sets the default language based on the
language of the calling application operating system. The LO language falls into the
domain specific query.

| Problem 3: ECL must deal with the metadata standard support of multi-languages. |
| Solution 3: ECL adds target metadata language requirements into its query structure. |

The diversity of the MQLs is a reflection of the complexity of the collaborative network. Without an infrastructure such as ECL, each LOR developer would have to deal with the MQL individually.

2.1.3 IMS Content Package Protocol (IMS CPP)

IMS CPP defines an interoperable framework for packaging a collection of learning materials into an aggregated LO. The IMS CPP definitions facilitate the import, export, aggregation, and disaggregation of the IMS content package [60]. There has been a series of updates since the first release of IMS CPP version 1.0 in June 2000. IMS CPP has become the core reference of SCORM® 2004 content packaging definitions [73].

The file format of IMS content package is a zip file. The package framework has two parts: a machine-readable manifest file that describes the organizations and resources of the content package, and physical content entries, which are the data files.
2.1.4 Sharable Content Object Reference Model (SCORM)

While e-learning communities focus their attention on defining reusable learning objects, developing new content models and learner assessment models, SCORM, a specification issued by the Advanced Learning Initiative (ADL), continues to develop the technical foundations via standardization [73]. SCORM is a collection of specifications and standards for building interoperable learning content. SCORM groups the specifications under three main topics: the Content Aggregation Model (CAM), the Runtime Environment (RTE), and Sequencing and Navigation (SN). The SCORM CAM defines responsibilities and requirements for building content aggregations. It provides a common means for composing learning content from discoverable, reusable, sharable,
and interoperable sources. The SCORM versions 1.2 and later adopted IMS CPP standard for CAM. The SCORM RTE describes the Learning Management System (LMS) requirements for managing the run-time environment. It ensures interoperability between sharable object-based learning content and LMS. Finally, the SCORM SN defines content sequence conformity indicating how LMS must follow in order to present a specific learning experience.

<table>
<thead>
<tr>
<th>SCORM Element</th>
<th>Concepts Covered</th>
<th>Key Technologies</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAM</td>
<td>Assembling, labelling and packaging of learning content</td>
<td>Shared Content Object (SCO), Asset, Content Aggregation, Package, Package Interchange File (PIF), Metadata, Manifest, Sequencing Information, Navigation Information</td>
</tr>
<tr>
<td>RTE</td>
<td>LMS management of the RTE, which includes launch, content to LMS communication, tracking, data transfer and error handling</td>
<td>API, API Instance, Launch, Session Methods, Data Transfer Methods, Support Methods, Temporal Model, Run-Time Data Model</td>
</tr>
<tr>
<td>SN</td>
<td>Sequencing content and navigation</td>
<td>Activity Tree, Learning Activities, Sequencing Information, Navigation Information, Navigation Data Model</td>
</tr>
</tbody>
</table>

Source: Based on the examples from [73]

SCORM is complex, which may make it unsuitable for some systems. Fortunately, SCORM does not enforce the compliance of all three components. Researchers can choose to apply CAM, RTE, or SN in any combination.

Development of SCORM and IMS encourages the production of self-contained LOs. There is much research on the SCORM integration [10, 39, 47]. Other researchers attempt to make SCORM more user-friendly by creating tools for building SCORM
content package [40, 75, 77]. Evidently, the use of SCORM has gained some momentum since early 2000. Is this a sign that IMS- or SCORM-compliant content packages will become LO standard? What would be the implication for a communication infrastructure such as ECL? The use of SCORM or IMS content packages does not affect ECL infrastructure. With middleware technology, ECL is able to provide utility to extract information from the SCORM or IMS content package. In addition, ECL eases the usage and the integration of SCORM.

It is important to note that SCORM refers to the content package as an aggregated learning content. This is a SCORM technical term. Currently, there is no consensus on the definition of the LO, but there is clear evidence that learning communities want LOs to be self-contained instructional units [5, 23, 26, 50, 59, 79, 90].

2.1.5 IMS Digital Repositories Interoperability (DRI)

IMS Digital Repositories Interoperability (DRI) Specification defines common interface functions for accessing a repository [70]. The specification refers to existing IMS metadata and content package standards, which are now part of the IEEE LOM and SCORM respectively.

- *Search/Expose* – is the interface function for search, which has two message structures: search for query and expose for the response.

- *Gather/Expose* – is the interface function for harvesting metadata, which has two message structures: gather for request and expose for the response.

- *Submit/Store* – is the interface function for submitting a learning object or metadata to a repository, which has two message structures: submit for request and store for the response.
- **Request/Deliver** – is the interface function for requesting a learning object from a repository, which has two message structures: request for request and deliver for the response.

- **(Alert/Expose)** – is an interface function for requesting a notification. It is an asynchronous message used to request a notification of some kinds of information such as the availability of new LOs. It is an option interface function; the two message structures are alert for the request and expose for the response.

Figure 2.2: Basic IMS DRI Architecture

Note that search, gather, and alert expose are different from each other. Search expose is the response of a search query, which contains a list of metadata. Gather expose can contain a list of LOM identifiers, sets, or metadata.

Source: Based on IMS DRI document available at: [http://www.imsglobal.org/drafts/registrations/driv/p0/imdr_info/p0.html](http://www.imsglobal.org/drafts/registrations/driv/p0/imdr_info/p0.html)

ECL infrastructure adopts IMS DRI as its basic service protocol. However, ECL only requires repositories joining the ECL LOR network to implement Search/Expose service if the LO is reachable through a URL link. Otherwise, the repository should implement Request/Deliver service to enable access to the LOs. Alert/Expose is not a key function. It is an asynchronous function and is difficult to implement between Web applications or WS and end-user calling applications. Furthermore, the Alert/Expose
function is for two service providers to exchange information, and is not essential to calling application. Currently, there is no need for an Alert/Expose interface function. Therefore, ECL does not provide support for Alert/Expose service protocol.

2.1.6 Summary of E-Learning Technology

The development of e-learning technology clearly supports the idea of “Reusable LO as the foundation for large-scale collaboration among educational organizations” [23, 76], which is also the premise of this thesis. It begins with the development of LOM standards, which bridge the gap between machine and human knowledge. IMS and IEEE LOM include definitions such as education level, time required learning, relationship to other LOs, etc. These definitions use specific vocabularies so that a machine can understand. Clearly, the action of IMS and IEEE LOM designers envision the use of LOM for a greater purpose such as LO composition. CPP standards supply a method for packaging a collection of learning materials into a self-contained learning unit so that it can be manipulated by a machine. Finally, IMS DRI captures the above technologies to provide interface functions for accessing the LOR.

2.2 Web Services Technologies

W3C defines a Web Service as a software system identified by a URI, whose public interfaces and bindings are defined and described using XML (i.e. WSDL). Its definition can be discovered by other software systems (i.e. UDDI). These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by Internet protocols (i.e. SOAP) [11, 92]. SOAP, UDDI, and WSDL are WS technologies well-supported by the industry. Other WS technologies such
as XML-RPC (Remote Procedure Call), RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture), and DCOM (Distributed Component Object Model) are negligible. These technologies ceased to evolve. Researchers typically select WS technologies due to the strong industrial backing of the technologies [8, 17, 32, 52, 57, 85], and there is a huge amount of work put into WS development. Utilities such as Apache Axis abstract the complexity of SOAP, and make SOAP easy to use. WS technologies offer many advantages, but there are also some constraints. This section reviews the advantages and disadvantages of WS technologies. The review begins with SOAP and WSDL, which are the main WS technologies. SOAP is the standard for WS messaging protocol, and WSDL provides definitions for describing a Web service. Next, the review looks at the advantages and disadvantages of each WS type. Finally, this section ends with a discussion of the limitations when using WS technologies to define common interface functions for a collaborative system and the approaches that attempt to resolve the limitations of WS technologies.

2.2.1 Simple Object Access Protocol (SOAP)

SOAP stands for Simple Object Access Protocol. Soon after XML became a standard, Microsoft began working on SOAP, a standardized messaging protocol based on XML that is platform-agnostic, flexible, and general-purpose. In March 2000, Microsoft received backing from IBM, and together they developed and released SOAP1.1 [56]. The SOAP specification has since been adopted and maintained as a W3C specification. SOAP defines an XML messaging envelop structure, data type encoding rules, service components, and the binding convention for the SOAP message.
2.2.2 Web Services Description Language (WSDL)

WSDL, a W3C standard, is an XML format for describing network services based on SOAP protocol as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information. The WSDL abstractly describes the operations and the messages, and then bonds those to a concrete network protocol and message format to define an endpoint [11]. WSDL does not introduce a new type of definition language. It merely extends the already supported XML Schema specifications (XSD) as its canonical system [92]. However, since it is unreasonable to expect a single system grammar that can describe all current and future messages, WSDL allows for other definition languages via its built-in extensibility. It is important to understand WSDL components in order to discuss the advantages and disadvantages of the WSDL and WS types. Below we review the six major elements of WSDL definitions:

- **types** – this element provides data type definitions used to describe the messages exchanged. The example below shows two user-type definitions used in the WSDL: GetTradePrice and TradePriceHeader. Note that XSD provides standard definitions for primitive types such as string, short, int, long, etc. The usage of these primitive types in the WSDL does not require an explicit definition.

```xml
<types>
  <schema targetNamespace="http://example.com/stockquote.xsd"
           xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="GetTradePrice">
      <complexType>
        <all>
          <element name="tickerSymbol"
                   type="xsd:string"/>
          <element name="time" type="xsd:timeInstant"/>
        </all>
      </complexType>
    </element>
    <element name="TradePriceHeader" type="uriReference"/>
  </schema>
</types>
```
- *message* - represents an abstract definition of the data being transmitted. The message consists of logical parts, each of which is associated with a definition within some type of system. The following example shows the usage of the two user-defined types `GetTradePrice` and `TradePriceHeader` in the message:

```xml
<message name="GetTradePrice">
  <part name="body" element="xsd1:GetTradePrice"/>
  <part name="tradePriceHeader" element="xsd1:TradePriceHeader"/>
</message>
```

- *portType* - is a set of abstract operations. Each operation refers to an input message and output message. In the following example, the output type is undefined, hence, assumed to be an XML element if the binding is a document style binding:

```xml
<portType name="StockQuotePortType">
  <operation name="GetTradePrice">
    <input message="tns:GetTradePrice"/>
  </operation>
</portType>
```

- *binding* - specifies concrete protocol and data format specifications for the operations and messages defined by a particular portType. Note that the binding is set for document-style binding:

```xml
<binding name="StockQuoteSoap" type="tns:StockQuotePortType">
  <soap:binding style="document"
    transport="http://example.com/smtp"/>
  <operation name="GetTradePrice">
    <input message="tns:GetTradePrice">
      <soap:body parts="body" use="literal"/>
      <soap:header message="tns:GetTradePrice"
        part="tradePriceHeader" use="literal"/>
    </input>
  </operation>
</binding>
```

- *port* - specifies an address for a binding, thus defining a single communication endpoint. The ‘port’ element is used in the service element below:

```xml
<port name="StockQuotePort" binding="tns:StockQuoteSoap">
  <soap:address location="mailto:subscribe@example.com"/>
</port>
```

- *service* - is used to aggregate a set of related ports:

```xml
<service name="StockQuoteService">
  <port name="StockQuotePort" binding="tns:StockQuoteSoap">
```
2.2.3 RPC Style Web Services

The XML Schema specifications were developed to allow ordinary data that is usually locked up in a proprietary format to be described in an open format that is human-readable, self-describing, and self-validating [53]. When using RPC WS message formatting, the XML schema describes the method and the encoded parameters used for the method call. Hence, all input and output data-types of each operation must be clearly defined (See sample 2.1) in the WSDL [11]. With the available SOAP library, RPC WS are easy to implement and are widely used. SOAP engines such as Apache Axis automatically generate a WSDL file when developers configure and deploy a Web service on the Web server.

If using user-defined types, RPC WS implementation must be relatively static since calling applications must generate stub code to invoke the service. Any changes to the service interface would break the communication between the service and the calling applications. If a service is widely distributed, then it is likely that a large number of applications have already produced stub code from its original WSDL document. Changing the WSDL would cause all of the applications that rely on a specific method signature to break. Good design dictates that the method signature of an RPC message service should never change [53]. Clearly, these constraints are not ideal for an environment under development such as a LOR network.
Sample 2.1: An Example of a WSDL file for RPC/encoded Web Services

Axis SOAP utility constructs SOAP request based on a WSDL file such as this example. However, developers can only use this WSDL file as a reference and must manually implement the service. The WSDL file for the service, usually generated by the SOAP utility, will be slightly different from the example shown here (e.g. the endpoint, namespace, etc).

```xml
<definitions name="StockQuote" ...>

<message name="GetTradePriceInput">
    <part name="tickerSymbol" element="xsd:string"/>
    <part name="time" element="xsd:timeInstant"/>
</message>

<message name="GetTradePriceOutput">
    <part name="result" type="xsd:float"/>
</message>

<portType name="StockQuotePortType">
    <operation name="GetTradePrice">
        <input message="tns:GetTradePriceInput"/>
        <output message="tns:GetTradePriceOutput"/>
    </operation>
</portType>

<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
    <soap:binding transport="http://schemas.m1soap.org/soap/http">
        <operation name="GetTradePrice">
            <input>
                <soap:body use="encoded" namespace="http://example.com/stockquote" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </input>
            <output>
                <soap:body use="encoded" namespace="http://example.com/stockquote" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
            </output>
        </operation>
    </soap:binding>
</binding>

<service name="StockQuoteService">
    <port name="StockQuotePort" binding="tns:StockQuoteBinding"/>
</service>
</definitions>
```
2.2.4 Document-style Web Services

Document style WS makes full use of XML and does not require a rigid contract. SOAP passes an XML document as a parameter in the SOAP message [53]. RPC message can also include an XML document as a string parameter, but the validation is the responsibility of the service and calling application implementation [11]. In the WSDL file example below (Sample 2.2), if developers do not want SOAP to validate the input XML document, they can use ‘ANY’ for TradePriceRequest and TradePrice type. However, the advantage of document style WS over using string message RPC WS ceases to exist since there is no validation. It would be simpler just to use RPC, and pass an XML document parameter as a string. Another argument used to sell the merit of document style WS is the ability to enhance and change a Web service without affecting the calling applications. Clearly, this is true only if the calling application can continue to build the XML document input, and process the XML document output. This constraint limits the advantage of document style WS. However, a slight advantage when implementing a Web service from document style WSDL might be that developers have more control over the service WSDL file.

Sample 2.2: An Example of Document Style WSDL

To implement WS from this WSDL file on Axis, the developer would implement Java class StockQuoteService.class that has a method “Element GetLastTradePrice(Element TradePriceRequest);”, which is easier if TradePriceRequest is an RPC encode type:

```xml
<definitions name="StockQuote" ...>
  <types>
    <schema ...>
      <element name="TradePriceRequest">
        <complexType>
          <all>
            <element name="tickerSymbol" type="string"/>
          </all>
        </complexType>
      </element>
    </schema>
  </types>
</definitions>
```
2.2.5 The Usage of Type Definition: Literal versus Encoded

There is little difference between literal and encoded type definition. When using literal (use="literal"), it means that the type definition literally follows an XML Schema definition in the WSDL. The usage of "encoded" style means that there is an additional set of rules outside of the WSDL. The calling application must know the encoding definition in order to make the call. Consequently, most WS developers use SOAP
encoding definitions. The usage of “literal” is predominant in the document style WS, which is natural since the developers want to have control over the XML document schema. However, the usage of “literal” in RPC can bridge the difference between RPC and document WS [11]. For example, if in sample 2.1, the SOAP body of RPC WS input was “literal”, then the GetTradePriceInput declaration could be replaced with XML Schema complex type definition of an XML document.

2.2.6 Limitations and Constraints of Web Services

When developers want to integrate a Web service into their application, the key problem is to understand, both semantically and syntactically, the services and the service operations [25]. WSDL is just a specification for constructing a SOAP message. It does not provide the semantics of the WS. Developers must read the service documentation and interpret, to the best of their abilities, the input and response parameters. Clearly, this is not a task for the machine. Developers must do the integration. Therefore, changes or enhancements to the service’s WSDL file require the intervention from calling application developers, as the stub code generated from the previous WSDL file is no longer compatible with the new Web service.

Implementing a Web service from a WSDL file is not a simple task. Developers must implement code to handle the data structures as well as the operation that performs the service. Using a WSDL file as reference to develop a Web service may affect interoperability and is error prone. Developers are human being. Each developer may have a different interpretation of the WSDL file. The level of knowledge of the service, semantically and syntactically, is much greater than just invoking the service from an application. Furthermore, a WSDL file is specific for an endpoint [11]. Two service
providers implementing the same service produce two distinct WSDL files. This is not scalable for LOR network communication infrastructure. A calling application must manage hundreds of WSDL files.

Although a calling application can continue to access a document style Web service and retrieve the output XML document after the service has made the enhancements, there is no guarantee that the calling application will continue to work. The calling application must be able to process the XML documents. If the structure or the semantic of XML documents has changed, the code implemented to process the XML documents can no longer process them. Therefore, the calling application would break down.

XML-RPC emerged to make a statement against the complexity of SOAP. No doubt, SOAP has a lot of overhead. XML-RPC provides a cleaner and simpler protocol for making a remote procedure call. XML-RPC was quite successful until the widespread use of XML and the emergence of SOAP libraries. The SOAP libraries (e.g., Apache Axis) provide utilities for developers to develop and use WS without knowing how to write a WSDL file or build a SOAP message. SOAP libraries know how to write and interpret a Web service’s WSDL file. For example, developers can use the Axis utility to deploy a Java class as a Web service. Axis configures and automatically creates a WSDL file for the deployed Web service. Client-applications can use SOAP utility (i.e. Axis, PHP, Perl, and ASP) to access a Web service by merely providing the required parameters and a SOAP endpoint. For a complex a Web service that has user-defined data-types, developers may use a SOAP utility to produce stub code from the service’s WSDL file to invoke the Web service. Depending on the complexity of the data structure,
writing a WSDL file for a Web service can be a daunting task. Having a SOAP engine automatically build the WSDL file saves time and simplifies the implementation of a Web service. Evidently, libraries abstract the complexity of SOAP, and make SOAP easy to use.

When using document style WS, the WS infrastructure becomes merely a transportation protocol. The WS support for interoperability is not significant since developers are responsible for handling the XML documents. Developers who know how to process XML documents should know how to validate them as well. Hence, document style WS’s support for XML document validation is not very significant. Even from WS framework point of view, document style WS does not have much advantage over RPC WS using XML string as its parameter, a simplest form of Web service that is easy to implement and can be replaced by an Internet communication protocol. From a middleware point of view, RPC WS with XML string as parameter is more flexible and stable. It is flexible because the middleware can validate the XML document and perform data transformation outside of the WS framework, which enables the middleware to support the service modification without violating WS requirements. It is stable because the middleware can define and use the same service throughout the middleware lifecycle. Furthermore, since a string is a primitive type, developers can use a SOAP utility to invoke services without having to generate stub code.

Evidently, WS technologies focus on providing interoperability for conducting business transactions. The design of WSDL is for a service provider to write a disclaimer for a Web service. A WSDL file is a service specification. It is a binding contract that all clients should be able to rely upon. Hence, WSDL is not suitable for describing interface
function definitions for a LOR network where many service providers provide a common interface for the LOR network clients. Although Web services that provide common interface functions are similar, the specifications of the services are slightly different from one service provider to another. Our design philosophy is not to be constrained by the technology but use the technology to our advantage. If WSDL is used, we have to limit our approach to what the technology can offer. However, by using XML-based messaging middleware, we are able to design a communication infrastructure that meet the requirement of the WS technologies and make the infrastructure highly adaptive to the changing environment. We reduce WS technologies to a mere transport protocol, which we can replace with another Internet transport protocol such as HTTP, JMS, plain socket connection, etc.

2.2.7 Next Generation of Web Services

WS technologies work well for well-defined services. A WSDL file is the service specification that developers rely on to integrate the service with the calling application. Once developers publish the WSDL file for a Web service, they should never change the WSDL file regardless of whether the service is a document or RPC style. The WSDL file is a binding contract. Therefore, changing the WSDL file would break the contract. The calling application would no longer be able to construct the service call or process the output. This section discusses two proposed solutions that attempt to expand the current WS capabilities: Business Process Execution Language for WS (BPEL4WS) proposed by IBM [2] and OWL-based WS Ontology (OWL-S) proposed by the OWL coalition [52]. The core concept for both research directions is a service composition. The main theory is that basic individual services are more stable, and the composition and the behaviour of
a complex service or a business process depend on the outcome of an execution of the individual basic services. Finally, this section reviews the simplistic approach taken by SRU. SRU is an HTTP-based service and represents an alternative to the current WS technologies.

2.2.7.1 Business Process Execution Language for Web Services (BPEL4WS)

BPEL4WS is an XML based workflow definition language, which describes the connection specifications of individual services when joining a business process [25]. BPEL4WS defines business processes as an executable business process that behaves as a participant in a business interaction and business protocols that use process descriptions to exchange messages without revealing their internal behaviour. The process descriptions for business protocols are abstract processes [2]. BPEL4WS extends WSDL specifications to define the common core, and adds the essential extensions required for its use. The deployment of BPEL4WS service requires the BPEL4WS engine, which typically wraps over the WS engine, such as Apache Axis. To deploy a service, the BPEL4WS engine interprets the descriptions in the BPEL4WS extended WSDL file and BPEL4WS process description file. If successful, the engine creates a standard WSDL file that any calling application can use to invoke the service. The composition takes place at the server side to hide the complexity from the client. As shown in the example skeleton below (Sample 2.3), the BPEL4WS process defines the service interactions and the necessary logical state that is required to achieve the business goal. The deployment of a business process is complex and needs careful planning, but the advantage of BPEL4WS is the ability to create complex services with the existing granular services.
Figure 2.3: BPEL4WS Execution Control

The diagram below illustrates an example of the BPEL4WS execution control and shows how many business processes are working together to achieve a business goal.

Source: Based on BPEL4WS specifications [2]

Sample 2.3: A Skeleton Sample of a BPEL4WS Process Definition

This is a typical definition of BPEL4WS process. As shown in the sample, BPEL4WS provides specifications for variables, flow control, and fault handler. The partnerLink element holds the service information. To deploy a business process, BPEL4WS engine needs BPEL4WS extended WSDL file and this process description file. Note that BPEL4WS extended WSDL file is not a standard WSDL file. It requires BPEL4WS additional parameters.

```xml
<process name="purchaseOrderProcess" ...>
  <partnerLinks>
    <partnerLink name="purchasing" ... />
    <partnerLink name="invoicing" ... />
    ...
  </partnerLinks>
  <variables><!-- variable declarations --></variables>
  <faultHandlers>
    <catch faultName="lns:cannotCompleteOrder" ...>
      <reply partnerLink="purchasing" ... />
    </catch>
  </faultHandlers>
</process>
```
The success of BPEL4WS depends heavily on the granular services that make up the business process [2]. To ensure the success of a business process, developers must subject the business process to rigorous testing. However, it is not easy to test a composite WS. The process must have proper error handling for the testing to be successful [46]. Furthermore, composing a business process needs some strategies in selecting the best services for the composition [87]. Clearly, BPEL4WS absolutely depends on the well-defined services.

2.2.7.2 Semantic Web Services

The release of the OWL-S (formerly DAML-S) fuels a lot of research in the area of the semantic WS [18, 25, 37, 38, 48, 54, 81, 95]. The structure ontology of OWL-S describes three essential types of knowledge about the service: Service Profile, Service Model, and Service Grounding. Service Profile provides the service definitions and description. Service Model describes the service usage. Service Grounding provides the technical specifications required to deploy and access the service [52].
The upper ontology begins with class Service, which provides a simple means for organizing the parts of a Web service semantic description. One instance of class Service exists for each distinct published service. The services are described by (describedBy) profiles, models, and groundings, which are captured by ServiceProfile, ServiceModel, and ServiceGrounding sub-classes respectively. The details of profiles, models, and groundings may vary widely from one instance of Service to another, but each of these three sub-classes should provide respective essential information about the service.

Semantic WS use the same service composition idea as BPEL4WS, but instead of specifying WS in terms of their execution logic or control flow, it uses ontology to describe the details of the WS composition. With the ontology, the semantic WS can take advantage of the reasoning engine to automatically discover the required service or its approximate, invoke the service by locating and assisting the computer program or agent to make the call, and compose the service from a collection of atomic services.
Figure 2.4: OWL-S Upper Ontology

The upper ontology provides a framework for semantic Web definition, and all definitions should start with this ontology.

Source: based on OWL-S Upper Ontology available at: http://www.ai-uei.com/xml/services/owl-s/1.2/overview/

2.2.8 HTTP vs. Web Services

SOAP and WSDL are complex technologies. It is a tedious task to create and deploy a Web service without the assistance of a SOAP utility. Each WSDL file is specific for a service endpoint. It is the service specifications that a service provider gives to the clients to invoke the service. Therefore, WSDL is not designed for a service provider to implement a service. SRU works around WS constraints by offering a URL-based service. It is a simple HTTP form-based request protocol sent to a Web application over HTTP protocol. In turn, the Web application responds with the output of an XML document. The response is identical to the SRW response and uses the same XML Schema. The XML document is a list of metadata that match the query request. As a Web application, a SRU service is easy to implement [83] and easy to maintain. SRW also dropped its WSDL specifications. Currently, the SRW protocol is just a simple SOAP
messaging protocol over HTTP. The protocol does not require a SOAP utility. Developers are responsible for building the request and the response of SRW SOAP messages, which is a simple process of pasting SRW XML document into the SOAP body.

SRW/SRU has broken WS tradition. It only uses technologies for its benefit and does not try to comply with WS technologies, which is a design philosophy that ECL has adopted.

2.2.9 Universal Description, Discovery, and Integration (UDDI)

UDDI, an OASIS\textsuperscript{1} standard, specifies protocols for creating a registry for WS, methods for controlling access to the registry, and a mechanism for distributing or delegating records to other registries [28, 92]. The UDDI is an interoperable framework that ensures the effective interaction of services in a service-oriented architecture. It provides means for developers to define business policies and technical specifications.

UDDI is a key factor for building a flexible and maintainable infrastructure. The cooperation between multiple services requires coordination of a registry that houses advertisement entries for services. ECL extends UDDI functionality and uses it to coordinate the communication between service providers and calling applications. There is little to say about the UDDI technology at this stage, except to expose the technology for later discussion.

\textsuperscript{1} Organization for the Advancement of Structured Information Standards (OASIS) is a non-profit, international consortium that drives the development, convergence, and adoption of e-business standards.
2.3 Summary

This chapter reviewed essential technologies commonly used in the development of a LOR network. The analysis of each technology supports the stipulations and assumptions of the research thesis. The first part of the review indicates a strong commitment of the learning communities to developing reusable LOs. The commitment is reflected in the development of the supported technologies for reusable LOs such as LOM, IMS CPP, and SCORM as well as in increasing research to integrate reusable LO technology. The second part of the review demonstrated the constraints of WS technologies. This shows that there is a need for an infrastructure such as ECL that addresses those constraints in the context of developing a collaborative network.

The purpose of developing metadata standards is to enable the distribution of learning materials, and serves as the foundation of e-learning management. Metadata is a LO machine-understandable classified ad. It enables LO creators to describe their LO in a fashion that a machine can understand and process the information. The review has also shown that reusability requires a LO to be a self-contained instructional unit [5, 23, 50]. Self-contained LO simplify many problems in e-learning management as well as in learning design, such as the delivery and the reuse of LOs. Currently, there are learning materials that are self-contained by nature (e.g. research papers, books, journals, articles, etc.), but many more are not. The development of IMS and SCORM CPP specifications attempts to resolve this problem by providing specifications and frameworks for the packaging of dependent learning materials into a self-contained LO or, in a technical term, an aggregated learning content. SCORM may be too complex. However, there are researchers working on making SCORM more user-friendly [40, 75, 77]. Consequently,
the LOR communication infrastructure should anticipate the increasing usage of SCORM, and provide utilities to support SCORM, or at least support IMS CPP. Finally, IMS DRI captures these technologies in the definitions of functional interface for accessing LOR. The definitions cover basic LOR interface functions for managing LOR network as well as for servicing the end-users. These are common interface functions for accessing LOR. Therefore, communication infrastructure such as ECL should support them.

The main objective of metadata standards, IMS and SCORM content packages, and IMS DRI is to enable the sharing of reusable LOs. These technologies are evolving and adapting to emergent technologies. One of the adaptation strategies is to provide varieties of specifications. The strategy is reflected in the metadata development. IEEE LOM includes custom elements for future definition, and user-defined elements. These flexibilities can affect interoperability and make the technologies too complex and too far ahead of what the end-users can do. The quality of current metadata is poor, and most metadata are not machine-understandable. Researchers attempt to rectify this problem by creating a framework to help generate metadata as well as to construct SCORM-compliant packages [10, 39, 40, 47, 64, 65, 75, 76, 77]. These tools abstract the complexity of the technology from the end-users while allowing the system to use the technology to its full potential. For example, metadata tagging is time-consuming and error-prone. Researchers are working on tools to generate or assist in the metadata tagging task [9, 36, 45, 61, 71, 72]. This integration process necessitates some changes in the e-learning management technologies, which is one of the arguments of this research thesis.
SOAP is the key component of WS technologies, and the widespread use of SOAP is mainly due to the development of SOAP utility libraries. Nowadays, it is rare that developers implement a Web service without some use of the SOAP library. SOAP libraries make WS technologies easy to use. Developers can use the technology without knowing the structure of the SOAP message or the schema of the WSDL. The utility abstracts the complexity of WS technologies and automatically enforces WS standards to ensure interoperability. The review reveals some of the WS constraints that can affect the implementation of a communication infrastructure. WS technologies insist on well-defined services. A published WSDL file should remain unchanged during the service lifecycle. Changes to WSDL file can result in integration failure. If using RPC style WS, calling applications may have created stub code from the WSDL file. If the new WSDL file has a different signature, the stub code is no longer compatible with the new Web service. When using document style WS, some enhancements are acceptable if the calling application can continue to construct the input and interpret the output of XML documents. On the other hand, rigorous specifications and constraints enforced by WS technologies may be the reason behind industrial support. Business transactions require well-defined services, and need the constraints to ensure interoperability. Semantic WS and BPEL4WS count on WS reliability to develop a programmable WS framework. Both technologies embrace WS composition methodology. Semantic WS and BPEL4WS theorize that complex services are combinations of basic services, which means that basic services can appear in different business processes or complex services.

Finally, the HTTP-based service protocol of SRW/SRU merits some consideration. Why should LOR developers bother with WS if they can provide service
over HTTP? However, WS technologies are a well-established framework. When wrapping with a middleware such as the ECL middleware infrastructure, the WS framework can be a commodity. ECL is able to provide security infrastructure based on WS technologies and use Apache SOAP Axis utility library to provide API for developers to deploy and access the ECL services.
CHAPTER 3: THE DESIGN AND THE IMPLEMENTATION OF THE ECL INFRASTRUCTURE

This chapter discusses the design and the implementation of the ECL infrastructure. The work on ECL began in December, 2002 when we undertook the development of the communication protocol for the eduSource network. The first task was to convince the eduSource members of the validity and the advantage of ECL proposal to use the XML-based messaging middleware infrastructure protocol. After a few demonstrations of the ECL infrastructure design and approach, the eduSource members accepted the proposal. The first release of ECL 0.3\(^2\) was in June, 2003. ECL 0.3 was a complete implementation of the core ECL infrastructure based on the stipulations and the design principles of this research thesis. Over the last three years, there were two major upgrades. The first upgrade was ECL 0.4, released in January, 2005, with the integration of the ECL UDDI registry. The second upgrade was ECL 0.5 in December, 2005, with the integration of the ECL security infrastructure. In between, there were small upgrades for several client gateways such as EdNA, SMETE, SQI of ARIADNE, and SQI of FIRE, SRU/SRW of RDN, and the secured server gateway for ECL clients to access the LionShare peer-to-peer network. The core idea of ECL has remained practically unchanged since the first release. The upgrades were add-on components, which were possible thanks to the modular design of ECL.

---

\(^2\) ECL 0.1 and ECL 0.2 were incomplete and were merely used for testing the requirements of the ECL infrastructure.
The description of ECL implementation in this section may be lengthy to some readers. However, this research is a real project. The implementation detail helps LOR developers to see the advantages of ECL and helps them to choose the right communication infrastructure for their LOR network. The proven advantages of ECL for an evolving collaborative network are the supporting arguments of this research thesis.

3.1 Design Principles

This section describes the four principles that influenced the design and implementation of ECL. The first two principles are the redefined principles commonly used in software design. The third principle is a defined principle for ECL targeting the stability of the collaborative network. The fourth and final principle was established as a policy by the eduSource project steering committee. It centres on the idea that the integration of the communication infrastructure should be simple, and does not require repositories to implement a huge amount of code, or install additional software.

The Abstraction Principle – The principle of the abstraction is common in object-oriented design, but in this research, it refers to the abstraction of new protocol standards and new technologies. The infrastructure design shall attempt to hide the complexity of new technologies and protocol standards from the developers by providing simple interface functions or APIs to ease the integration of the infrastructure.

The Modularity Principle – The principle of modularity enforces the grouping of similar technology components in order to reduce dependency on the technologies. In a developing environment, technologies change frequently. Grouping technologies in
modules permits the substitution of the technologies without affecting the whole infrastructure.

**The Maintainability Principle** – The principle of maintainability forces the development of strategies that makes the infrastructure adaptive to change. For example, a calling application using the old ECL version should be able to invoke the ECL service implemented on a new ECL infrastructure. Similarly, the protocol infrastructure should be able to assimilate new protocol standards and adaptive to the changes of existing standards.

**The Enable-and-not-Require Principle** – This principle dictates the usage of middleware that can perform self-configuration and self-installation of the infrastructure. However, the middleware technology is platform-dependent. To achieve this principle, middleware users must be able to get the middleware implementation for their platforms. Therefore, a middleware such as ECL should consider the interoperability among ECL implementations on different platforms.

### 3.2 Selecting Implementation Platform for ECL Infrastructure

The main purpose of the current implementation of ECL is to prove the feasibility and the design theory. The Java platform has an abundance of Java open-source libraries, which simplifies the implementation of the ECL infrastructure. Java is a selected platform, but any platform having SOAP, UDDI, and XML libraries should have a head start on the implementation job. Nevertheless, any chosen platform should work for the ECL implementation. The only condition for interoperability among the ECL implementations is RPC WS for passing ECL messages. Since WS technologies are
platform-independent, WS technologies guarantee interoperability among the implementations of the ECL.

As a proof of the interoperability, the ECL search protocol was implemented in Python. The implementation enables a calling application to invoke an ECL search service deployed on a Java platform. This example shows the interoperability among the implementations. With Java implementation of the ECL serving as implementation guidelines, the implementation is quite simple. It took approximately three days for an experienced programmer to learn Python and implement the search client for ECL. Any Python application can use this implementation of the ECL.

3.3 Architecture Design

3.3.1 Middleware

The enable-and-not-require and abstraction principles call for the usage of middleware technology. Apache Axis (http://ws.apache.org/axis/) is a good example of a successful middleware that obeys these two principles. Axis abstracts from SOAP complexity and makes SOAP easy to use. As mentioned in the previous chapter, a service provider only has to implement a Java class that handles the operations of the deployed Web service. Axis utility takes care of deployment of the service as well as creating and publishing the WSDL file. A calling application can immediately use the WSDL file to invoke the service. Axis does a lot of work behind the scene for the developers.

The ECL infrastructure wraps over Axis and takes full advantage of the Axis utility. The deployment of ECL is a simple WAR file, which allows Apache Jakarta Tomcat (http://tomcat.apache.org/) to deploy the ECL infrastructure automatically. LOR
developers can use a Web application to configure ECL, such as loading search handler class and entering the repository information. ECL uses the Axis utility to deploy the service as well as publishing the service WSDL file. In addition, ECL performs the service registration to the UDDI registry and continues to update the registry throughout the service lifecycle to ensure that the service information is always up-to-date. The ECL API at the calling application has the correct service information to perform the necessary message transformation when invoking the service. Finally, ECL middleware supplies developers with a number of utilities to ease the integration of the learning management technologies such as the IMS content packaging, metadata mapping, metadata query mapping, etc. As a result, developers can use their supported metadata and metadata query language formats. In conclusion, the middleware allows ECL to obey its design principles and provide a stable and intelligent infrastructure that can perform auto-diagnostic and self-configuration.

3.3.1.1 Modularity

To avoid dependency on a particular technology, ECL architecture groups technologies into independent modules. Each module has specific interface functions. The implementations of two modules are equivalent if both implement the module interface functions. This makes the substitution of an implemented module possible. As shown later in the integration section, the modularity design of ECL allows the reuse of ECL core modules on the gateway architecture, which enables ECL to interoperate with other protocols.
3.3.1.2 The Transport Module

The transport module is responsible for exchanging messages and files between the calling applications and the service providers. Currently, the ECL transport module wraps around Apache Axis SOAP engine. ECL uses SOAP for its benefits such as using the Axis utility to deploy and invoke ECL service without additional configuration and implementation. ECL selected RPC WS to transport the ECL message. As discussed in the previous chapter, if the WS only uses XSD built-in types, RPC WS offers better performance and interoperability. Since the ECL needs to perform message transformation outside of the SOAP infrastructure, a document-style WS does not offer any advantage. Using a document-style WS requires more work, and has more constraints, such as the requirement of an individual WSDL file for each service. RPC WS keeps the protocol simple, and reduces the transport module to a mere carrier of the
ECL message in a serialized XML string format. A calling application can use a SOAP client utility to invoke ECL services without having to generate the sub code. This eliminates the need for a widespread update when releasing an upgrade of the ECL implementation. The transport module has a single interface function. Note that the file attachment is the Axis defined type and is a defined type for most SOAP utilities:

```java
String processEclMessage(String eclMessage, DataHandler fileAttachment);
```

The task of the transport module is to carry a string message and a file attachment from the calling application to the dispatcher module. Any Internet protocol (i.e. HTTP form request and response, TCP/IP socket connection, or JMS) can support such a task.

Figure 3.2: RPC versus Document Style Web Service

If using document style WS (top diagram), the ECL message must be predefined. The architecture is constrained by the WS technologies. The RPC architecture below is much simpler and more stable since changes can occur outside of the SOAP infrastructure.
3.3.1.3 Transformer Module

The transformer module is a utility module that takes care of data transformation as well as supplying necessary utilities for converting foreign protocol standards from/to the local protocol standards (e.g., metadata query language and metadata standards). Its first task is to convert local messages from/to common message protocol. For example, a calling application builds a search request using XQuery ECL specified syntax. The transformer module transforms the search request to ECL internal common query structure before passing the message to the transport module. Similarly, the transformer module is responsible for converting the response metadata to the format used by the calling application. On the calling application side, the transformer needs UDDI assistance to determine the protocol version, security, and metadata format of the ECL service. Essential service information is kept on the UDDI registry, which is accessible to all ECL clients. The information instructs the transformer how to perform the message transformation (see Service Invocation Algorithm in section 4.3 for more information).

3.3.1.4 Dispatcher Module

The dispatcher module is responsible for dispatching an ECL message to an appropriate service handler. When a message reaches the dispatcher module, the dispatcher determines where the message should go, based on the service URN. The URN identifies the service handler class that developers configured into the ECL infrastructure. If the URN matches a service handler class in the configuration, the dispatcher loads the class to process the message. Otherwise, it throws a "Service Not Found" exception. ECL extends four IMS DRI interface functions: Search/Expose, Gather/Expose, Request/Deliver, and Submit/Store. It predefines these four services in
the infrastructure and provides data-structure classes to developers to handle the data. The data-structure classes are also responsible for serialization of the XML document. ECL also provides a generic service handler interface that developers can use for other services. The data structure is a standard XML element. Developers can upload XML document schema used for a generic ECL service to an ECL UDDI registry, which become accessible to the clients for validating the XML document parameter.

3.4 ECL Messaging Protocol

The ECL messaging protocol follows IMS DRI specifications closely. As mentioned earlier, ECL defines a service protocol for the four main interface functions of IMS DRI. This section discusses in detail the definitions of all the service protocols. Each of the service protocol functions has special characteristics that the ECL infrastructure must handle. Search/Exposé requires a communication protocol that can handle input and output transformations. The metadata query language and metadata format can be different from one calling application/service provider to another, and the building of the metadata query itself can be difficult for developers. Therefore, the protocol requires special handling. Next, the Gather/Exposé function needs different types of actions. In this service protocol, ECL introduces the notion of action and resumption token. An action is an additional instruction to perform a specific operation. Resumption token, as well as an action, is not an IMS DRI definition. Resumption is a technique borrowed from the OAI protocol [58] for handling large data transfers. Finally, Request/Deliver and Submit/Store require a service protocol to handle file download and upload, which can be complex when using WS and when having to deal with the security.
The discussion in this section does not cover the Alert/Expose interface function. It is an optional interface function of IMS DRI. Developers can use the ECL generic service to implement the Alert/Expose service. The ECL messaging protocol has the definition for adding call-back information in the sender specifications. This versatility is one of the key features of the ECL infrastructure. It is the ability to expand and support other types of services.

Submit/Store and Request/Deliver may need security. The ECL infrastructure handles security separately, and abstracts the security from the messaging protocol. Although ECL 0.5 had integrated security module in the distribution [31], the discussion of ECL security is not in the scope of this thesis. Please see the ECL security infrastructure document for more information [31].

3.4.1 Message Anatomy

The ECL protocol has two parts: the header and the body. It is a typical messaging protocol. The header contains essential information for the middleware infrastructure to convey messages between a calling application and a service provider. The body contains a payload, which is the message designated for the service handler or calling application depending on the direction of the communication.

3.4.1.1 Header

- **type** – identifies the ECL message type (e.g., search, gather, request, submit, etc.).
- **cid** – is a global unique identifier for each message.
- **protocol** – identifies the ECL protocol.
- **version** – identifies the ECL protocol version.
senderInfo:

- **mode** – is the return mode: push or pull; push assumes that the endpoint is a valid TCP/IP socket server while pull assumes the endpoint is an identifier.

- **endpoint** – is a TCP/IP endpoint for a call-back. It must be an endpoint that can accept a socket connection to receive an ECL message containing search expose, gather expose, deliver, or store message as a payload wrapped in SOAP message similar to SRW³.

- **receiverServiceUrn** – is the identifier of the service being requested.

- **errorMessage** – is the message that explains an occurrence error.

- **secToken**
  - **certificate** – is a digital certificate of the message sender.
  - **assertion** – is a SAML assertion used to access a repository resource.

- **status** – is a status of the response message: ERROR or OK.

### 3.4.1.2 Body

The ECL message body carries a payload. It can take any XML document. Unless it is a user-defined messaging protocol designated for ECL generic service, the XML document obviously should be an ECL protocol message. The dispatcher module extracts the request XML document from the ECL message, and passes the request to the service handler. The dispatcher module is responsible for validating and de-serializing the XML document, which is a process of instantiating a correct data structure handler. During initialization, ECL automatically registers the schema of the XML document, data

---

³ Basically, it is the same SOAP message as the synchronous ECL response, except that it has no binding information. The calling application must extract the ECL message from the SOAP body and use ECLMessageImpl data structure to de-serialize the message. From there, the ECL message is the same as a synchronous message.
structure class, and any other information required for correct data transformation onto the UDDI registry to ensure that calling applications send a correct ECL message to the service handler. Similarly, the ECL client utility is responsible for de-serializing the response XML document. The information that the ECL uploads to the UDDI registry during the configuration of an ECL service instructs ECL clients how to perform data transformation.

Sample 3.1: A Skeleton of the ECL Message

(See Appendix A1 for the ECL message XML Schema)

```
<ecl xmlns="http://www.edusource.ca/xsd/ecl_v1p1">
  <header>
    <type>search</type>
    <cid>trbf8e8025</cid>
    <protocol>ECL</protocol>
    <version>0.5</version>
    <receiver>SearchService</receiver>
    <sender>
      <mode>PUSH</mode>
      <endpoint>some TCP/IP</endpoint>
    </sender>
  </header>
  <body>
    <payload>
      <!-- XML document - see following protocols: Search/Expose, Gather/Expose, Request/Deliver, Submit/Store, and etc -->
    </payload>
  </body>
</ecl>
```

3.4.2 Search Service Protocol

Search/Expose is an IMS DRI interface function for metadata query. ECL supports this interface function in the ECL search service protocol. To become a member of a LOR network, a LOR must provide a search service. Note that SRU, SRW, and SQI only support the search interface function. Providing search service is sufficient if all the LOs are URL-accessible. However, if the LOs are not accessible via URL, a LOR must
also support a Request/Deliver service to deliver LOs to the calling applications. ECL Search/Expose service protocol has two data structures that encapsulate ECL search-request and search-expose messaging protocols Search and SearchExpose. Search is responsible for mapping between foreign query protocol and local query protocol and provides a simple API for building a metadata query. SearchExpose takes care of the metadata mapping, as well as providing API for constructing the search response message.

3.4.2.1 Search Request Protocol

Based on the search strategy described in the technology review (Chapter 2), search request protocol contains two sub-query structures: text and domain-specific query. Text query is language-specific. It targets the metadata elements that match a specified language. By default, ECL loads the query language from the calling application operating system and sets the text search mode to a simple text search based on Google text search syntax (see Metadata Query Language, section 2.1.2 for more information). Normally, the language setting of the operating system is usually the language used on the interface of the calling application, which should be the working language of the end-users. The calling application can override the default setting by specifying the target language of the metadata elements in the metadata query. Note that the ECL sub-query structure is able to accommodate any type of XML queries, and can convert from/to any type of known metadata query languages. The following describes the six elements of ECL common search request protocol:
- **pattern** – is the LOM structure code that identifies the LOM as a complete or summary LOM. A summary LOM contains only four basic elements: title, description, location, and format.

- **language** – is the language code that identifies the language of the metadata to be queried.

- **text-search-mode** – is the value that identifies text-search mode; by default text-search mode is simple text search.

- **basic** – is the sub-query structure of text query. Currently, ECL assumes that all searches begin with a text query. In IEEE LOM, text query is applicable to title, description, and/or keyword elements.

- **advanced** – is the sub-query structure of domain-specific query that is applicable to the rest of IEEE LOM elements.

- **resumptionToken** – is the token for the calling application to get the next metadata records if the token exists (optional).

Note: When requesting an XQuery from the search-request data structure, the ECL API combines the text- and domain-specific query to build the XQuery. However, the LOR developer can extract the text query from the search-request data structure independently, perform the text query, and then apply domain specific query on to the result of the text query.
Sample 3.2: An Example of the ECL Search Request Messaging Protocol

(See Appendix A2a for the ECL search request messaging protocol XML Schema)

```xml
<search xmlns="http://www.edusource.ca/xsd/ecl_search_v1p1">
  <pattern>0</pattern>
  <language>en</language>
  <text_search_mode>en</text_search_mode>
  <basic>
    <or>
      <or>
        <complex xpath="lom/general/title/string">
          <simple op="contains" caseSensitive="false" text="java" />
        </complex>
        <complex xpath="lom/general/description/string">
          <simple op="contains" caseSensitive="false" text="java" />
        </complex>
        <complex xpath="lom/general/keyword/string">
          <simple op="eq" caseSensitive="false" text="java" />
        </complex>
      </or>
    </or>
    <advanced>
      <complex xpath="lom/general/technical/format">
        <simple op="contains" caseSensitive="false" text="html" />
      </complex>
    </advanced>
  </basic>
</search>
```

A search request messaging protocol using resumption token:

```xml
<search xmlns="http://www.edusource.ca/xsd/ecl_search_v1p1">
  <resumptionToken expirationDate="2006-05-22T14:21:16.025" completeListSize="1000" cursor="100">
    bf8e8222
  </resumptionToken>
</search>
```

The sub-query structure has four elements. Nested elements are possible, but a sub-query shall contain only one element:

- **or** – is the structure that holds an “OR” operation query.
- **and** – is the structure that holds an “AND” operation query.
• complex – is the structure that holds a complex query structure of the XML element specified by a given x-path
  - xpath – is an X-path value specified a child element to be queried.

• simple – is the structure that holds the data to perform the query operation
  - op – is a comparative operator.
  - caseSensitive – is a value specified if the query is case sensitive; the default value is true.
  - text – is the string that holds the keywords or the terms of the query
  - attName – is the attribute name; if an attName is present, the query is performed on the value of the attribute; otherwise the query is performed on the text content of the XML element.

This common query structure was introduced in the release of ECL 0.3. The structure enables ECL to implement a query transformation utility that can transform this common query structure to different types of metadata query languages (e.g., FIRE query structure, EdNA, SRU/SRW, or SMETE). For example, in the first release, ECL was equipped with the utility to transform the common query to XQuery and HashMap, as shown in the two following examples:

Sample 3.3: An XQuery Example

```xml
<results>{
 FOR $lom in document("loms.xml")/lom WHERE
 ((contains($lom/general/title, "java") or
 contains($lom/general/description, "java")) or
 contains ($lom/general/keyword, "java")) AND equals
 ($lom/general/format, "html")
 RETURN {$lom}
 }</results>
```
Sample 3.4: A HashMap Query Example

\{
  and=
   or=
     contains={xpath="lom/general/title", value="java"}
     contains={xpath="lom/general/description", value="java"}
     contains={xpath="lom/general/keyword", value="java"}
   
   eq={xpath="technical/format", value="html"}
\}

The eduSource project imposed XQuery as the query language used in the ECL. The popularity of XQuery had declined over the years, and it is too complex for LOR developers. Some eduSource LOR developers preferred using the HashMap structure that ECL provides. Nevertheless, the ability of ECL to support different types of queries proves that the implementation follows the abstraction principle.

3.4.2.2 Search Expose Protocol

The ECL search expose protocol accepts a list of LOM records and an optional resumption token.

- **advanced** – is a token indicating that a domain-specific query has been applied to the search results. LOR developers should turn the advanced search value to *true* if they perform a domain-specific query.

- **resumptionToken** – is a token for the calling application to get the next set of metadata records if all the search results cannot be included in the response (optional).
Sample 3.5: A Skeleton of the Search Expose Messaging Protocol

(See Appendix A2b for the search expose messaging protocol XML Schema)

```xml
<search_expose xmlns="http://www.edusource.ca/xsd/ecl_search_ex_vlp1" advanced="true">
  <lom xmlns="http://ltsc.ieee.org/xsd/LOM">
    ...
  </lom>
  ...
  <resumptionToken expirationDate="2006-05-22T14:21:16.025" completeListSize="1000" cursor="100">bf8e8222
  </resumptionToken>
</search_expose>
```

3.4.2.3 Search Handler Implementation

To simplify the ECL integration, ECL encapsulates the search service handler in an abstract class called SearchServiceHandler for LOR developers to connect with their LOR to the network. The processSearch method is the only method LOR Developers have to implement. Developers get the search interface API (Search class) as an input parameter to the processSearch method. They can extract their preferred query structure from the API and use ECL search expose API (SearchExpose class) to build the response. The search expose API can take any type of known LOM formats and convert the LOM records to the calling application’s preferred LOM format.

Sample 3.6: Search Service Handler Interface Function

(See Appendix B1)

```java
public abstract class SearchServiceHandler implements ServiceHandler {
  public abstract SearchExpose processSearch(Search request) throws SearchException;
}
```
3.4.2.4 Search Client Utility

The ECL Search Client utility provides an easy API for a calling application to invoke an ECL search service so that developers do not have to build the stub code. To use the ECL search client, developers only have to provide the service key and the search request to the API. Developers can add an interface to their calling application to allow the end-users query ECL services on the UDDI registry. Each service has a service key as a unique identifier. The calling application can cache the service keys in the users' preferences and use them for search. If calling application developers do not have any query language, they can use ECL common query and the API provided by the ECL infrastructure. The search request API is intuitive, and it is easy to construct a metadata query. The API can also take known metadata query input and continue to add new API to allow developers to input a metadata query using different types of query languages (i.e., setXQuery(String xquery)). The search client returns an instance of SearchExpose, which can return LOM records in any known format. The following code is an example of how to invoke the ECL search service:

Sample 3.7: The ECL Client Code for Building Query and Invoking an ECL Search Service

```java
String serviceKey = null; // a UDDI search service key

// create an instance of UDDI registry
UddiQuery uddi = new UddiQuery();

SearchClient sc = new SearchClient();
Search s = new SearchImpl();

// Basic Search is a text query. All searches should begin with
// text query, and text query should be performed on title
// and/or description and/or keyword. Note setBasicSearch uses OR
// operator for all three elements
s.setBasicSearch("java");

CompType ct = new CompType("lom/general/technical/format",
new SimpleType(SimpleType.EQ, "html", false));
```
s.setAdvanceSearch(ct);

// Print out search request XML string
System.out.println(s.toXmlString());

// Search the repository
SearchExpose se = sc.search(uddi, serviceKey, s);

// Transform IEEE LOM to DC LOM
se.setPrefferedNS(LomConstants.DC_NS_URI);

// Print out search expose. The metadata is in DC format.
System.out.println(XmlUtil.getXmlString(se.toXmlElement()));

3.4.3 Gather Service Protocol

The gather service protocol is the representation of Gather/Expose interface functions of the IMS DRI for harvesting metadata. As instructed in IMS DRI specifications, ECL defines gather service protocol based on OAI harvesting. As in OAI harvesting protocol, the gather service protocol has six actions (verbs): Identify, GetRecord, ListIdentifiers, ListMetadataFormats, ListSets, and ListRecords. As in all service protocols, ECL provides simple API (GatherRequest and GatherExpose data structure classes) for calling applications, as well as LOR developers to handle the gather service. Each action requires specific arguments and specific responses. Gather protocol service reflects the flexibility of the messaging protocol. Normally, the service needs six different operations. The messaging protocol can use a single interface function (processEclMessage) of the transport protocol. In this case, the service handler is responsible for parsing the actions. However, ECL hides the processing from LOR developers and exposes the simple interface in the GatherSearchHandler to the developers.
The following sections discuss in detail all possible Gather/Expose actions. Each action has a pair of gather-request and gather-expose messaging protocols. The XML schemas for these protocols are in Appendix A3a and A3b.

3.4.3.1 Who are the Gatherers or Harvesters?

A gatherer can be a researcher who collects metadata for a research, or a portal repository that gathers metadata from internal and protected repositories and makes the metadata available to the public through a main portal. Usually, institutions use gather function to reorganize institutional resources.

3.4.3.2 Selective Gathering

Selective gathering allows gatherers to make gather requests for specific types of metadata. However, selective gathering needs some extensions. OAI supports a set spec⁴, metadata prefix, and date for record filtering. Repositories using IEEE LOM do not have such date and set definitions, and the metadata prefix is IEEE LOM. ECL will pass these request parameters to the gather service handler. It is up to LOR developers to handle these parameters.

3.4.3.3 Date Format

There must be an agreement on the date format. ECL adopts Universal Time Central (UTC) for all the date usage in the ECL infrastructure. ECL supports the following date formats: YYYY-MM-DDThh:mm:ss or YYYY-MM-DDThh:mm:ss.SSS.

⁴ A set spec is a set identifier. OAI lets repository developers define their set organization and suggests using colons to separate the set hierarchy (i.e. sciences:mathematics, sciences:physics, etc). ECL suggests using the Dewey classification as the set organization and the Dewey numeric identifier as the set spec.
3.4.3.4 Identify Action Protocol

The Identify action is a request protocol for retrieving the repository information. ECL adopts OAI identify protocol for the Identify action. Repositories may also employ the Identify to return additional descriptive information about the repository such as security, copyrights, and licenses.

Arguments:

- \textit{verb} – identifies the action, and the value of this action is "Identify".

Error and Exception Conditions:

- \textit{badArgument} – is caused by incorrect "action" value.

Expose Parameters:

- \textit{responseDate} – is the response date.
- \textit{repositoryName} – is a human-readable name for the repository.
- \textit{baseURL} – is the base URL of the repository.
- \textit{protocolVersion} – is the version of ECL protocol supported by the repository.
- \textit{earliestDatestamp} – similar to OAI, a UTC date time that is the guaranteed lower limit of all date-stamp recording changes, modifications, or deletions in the repository; ECL provides default value based on the installation date of ECL.
- \textit{deletedRecord} – not yet defined in ECL (may use OAI vocabulary)
- \textit{granularity} – not yet defined in ECL (may use OAI vocabulary)
- \textit{description} – is any information the repository wants to provide the client.
Sample 3.8: An Example of the Gather Identity Request Protocol

```
<gather xmlns="http://www.edusource.ca/xsd/ecl_gather_v1p1">
  <verb>Identify</verb>
</gather>
```

Sample 3.9: An Example of the Gather Identity Expose Protocol

```
<gather_expose
  xmlns="http://www.edusource.ca/xsd/ecl_gather_ex_v1p1">
  <responseDate>2006-05-22T14:21:15.744</responseDate>
  <Identify>
    <repositoryName>Pool Test Repository</repositoryName>
    <baseURL>http://pool.iat.sfu.ca/pool</baseURL>
    <protocolVersion>ECL</protocolVersion>
    <deletedRecord>persistent</deletedRecord>
    <granularity>creation</granularity>
    <adminEmail>root@home.com</adminEmail>
    <compression>deflate</compression>
    <description>This is a test repository.</description>
  </Identify>
</gather_expose>

3.4.3.5 GetRecord Action Protocol

The GetRecord is a request protocol for retrieving an individual metadata record from a repository. Since the ECL utility can convert metadata, ECL does require a metadata prefix as required in the OAI protocol. Repositories only have to return the original metadata, and ECL deals with the transformation.

Arguments:

- **verb** – identifies the action, and the value of this action is “GetRecord”.
- **identifier** – is a required argument that specifies the unique identifier of the item in the repository from which the record must be disseminated.

Error and Exception Conditions:

- **badArgument**
  - **verb** – is caused by incorrect “action” value.
- idDoesNotExist – indicates that the error is caused by the unknown identifier or illegal identifier in the repository.

Expose Parameters:

- responseDate – is the response date.

- Record:
  
  - header: is the header structure that holds the identifier, date stamp, and a set spec.

  - metadata: is the structure that holds the IEEE LOM.

Sample 3.10: An Example of the Gather GetRecord Request Protocol

```xml
<gather xmlns="http://www.edusource.ca/xsd/ecl_gather_v1p1">
  <verb>GetRecord</verb>
  <identifier>lobf8e8025</identifier>
</gather>
```

Sample 3.11: An Example of the Gather GetRecord Expose Protocol

```xml
<gather_expose
  xmlns="http://www.edusource.ca/xsd/ecl_gather_ex_v1p1">
  <responseDate>2006-05-22T14:21:15.794</responseDate>
  <GetRecord>
    <record>
      <header>
        <identifier>lobf8e8025</identifier>
        <datestamp>
          2006-01-22T14:21:15.804
        </datestamp>
        <setSpec>chem100</setSpec>
      </header>
      <metadata>
        <lom xmlns="http://ltsc.ieee.org/xsd/LOM">
          ...
        </lom>
      </metadata>
    </record>
  </GetRecord>
</gather_expose>
```
3.4.3.6 ListIdentifiers Action Protocol

The ListIdentifiers action is a request protocol for retrieving record headers from a repository. OAI calls ListIdentifiers an abbreviation of ListRecords because the action is identical to ListRecords, except that the response has only the headers of the metadata records. Usually, developers use this action to check for updates to metadata.

Arguments:

- **verb** – identifies the action, and the value of this action is "ListIdentifiers".

- **from** – is an optional argument with a UTC date time value, which specifies a lower bound for selective harvesting.

- **until** – is an argument with a UTC date time value, which specifies an upper bound for selective harvesting.

- **setSpec** – is an optional argument of a set spec, which specifies set criteria for selective harvesting.

Error and Exception Conditions:

- **badArgument**
  
  - **verb** – indicates that the "action" parameter is incorrect.

Expose Parameters:

- **responseDate** – is the response date.

- **ListIdentifiers**:
  
  - **header**– is the header structure that holds the identifier, date stamp, and set spec.

  - **resumptionToken** – is a token for the calling application to get the next identifiers if all the identifiers cannot be included in the response.
Sample 3.12: An example of the Gather ListIdentifiers Request Protocol

```xml
<gather xmlns="http://www.edusource.ca/xsd/ecl_gather_v1p1">
  <verb>ListIdentifiers</verb>
  <from>2006-05-22T22:04:36.065</from>
  <until>2006-05-23T14:21:16.065</until>
</gather>
```

Sample 3.13: An Example of the Gather ListIdentifiers Expose Protocol

```xml
<gather_expose
   xmlns="http://www.edusource.ca/xsd/ecl_gather_ex_v1p1">
  <responseDate>2006-05-22T14:21:16.065</responseDate>
  <ListIdentifiers>
    <header>
      <identifier>1obf8e8025</identifier>
      <setSpec>chem100</setSpec>
    </header>
    <resumptionToken expirationDate="2006-05-22T14:21:16.065" completeListSize="100"
      cursor="50">reb8e8111</resumptionToken>
  </ListIdentifiers>
</gather_expose>
```

3.4.3.7 ListMetadataFormats Action Protocol

This ListMetadataFormats action is a request protocol for retrieving available metadata formats of a LOM record from a repository. Since ECL takes care of metadata mapping, currently developers should not need this action, but this research also supports the OAI view on the metadata standards. It is impossible to provide definitions for every LO. Hence, in the future, this action may be needed.

Arguments:

- **verb** – identifies the action, and the value of this action is "ListMetadataFormats".
- **identifier** – is a required argument that specifies the unique identifier of the item in which the metadata format is being requested.

Error and Exception Conditions:

- badArgument
- verb – indicates that the “action” parameter is incorrect.

- idDoesNotExist – indicates that the error is caused by the unknown identifier or illegal identifier in the repository.

**Expose Parameters:**

- `responseDate` – is the response date.

- **ListMetadataFormats:**
  - `metadataFormat` is the structure that holds the metadata prefix, the schema, and the namespace of the record.

**Sample 3.14: An Example of the Gather ListMetadataFormats Request Protocol**

```xml
<gather xmlns="http://www.edusource.ca/xsd/ecl_gather_v1p1">
  <verb>ListMetadataFormats</verb>
</gather>
```

**Sample 3.15: An Example of the Gather ListMetadataFormats Expose Protocol**

```xml
<gather_expose
  xmlns="http://www.edusource.ca/xsd/ecl_gather_ex_v1p1">
  <responseDate>2006-05-22T14:21:16.185</responseDate>
  <ListMetadataFormats>
    <metadataFormat>
      <metadataPrefix>ieee</metadataPrefix>
      <schema>http://ecl.iat.sfu.ca/lom.xsd</schema>
      <metadataNamespace>
        http://ltsc.ieee.org/xsd/LOM
      </metadataNamespace>
    </metadataFormat>
  </ListMetadataFormats>
</gather_expose>
```

**3.4.3.8 ListRecords Action Protocol**

This ListRecords action is a request protocol for gathering metadata records from a repository. Optional arguments permit the selective gathering of records based on a set and/or a date stamp. The required arguments are the same as in the ListIdentifiers action protocol.
Arguments:

- *verb* – identifies the action; the value of this action is “ListRecords”.

- *from* – is an optional argument with a UTC date time value, which specifies a lower bound for selective harvesting.

- *until* – is an optional argument with a UTC date time value, which specifies an upper bound for selective harvesting.

- *setSpec* – is an optional argument, which specifies set criteria for selective harvesting.

Error and Exception Conditions:

- *badArgument*
  - *verb* – indicates that the “action” parameter is incorrect.

Expose Parameters:

- *responseDate* – is the response date.

- *ListRecords*:
  - *Record* – is the record structure that holds header and metadata.
  - *resumptionToken* – is a token for the calling application to get the next records if all the records cannot be included in the response.

Sample 3.16: An Example of the Gather ListRecords Request Protocol

```xml
<gather xmlns='http://www.edusource.ca/xsd/ecl_gather_v1p1'>
  <verb>ListRecords</verb>
  <from>2006-05-22T22:04:36.015</from>
  <until>2006-05-23T14:21:16.015</until>
</gather>
```
Sample 3.17: An Example of the Gather ListRecords Expose Protocol

```xml
<gather_expose
  xmlns="http://www.edusource.ca/xsd/ecl_gather_ex_v1p1">
  <responseDate>2006-05-22T14:21:16.015</responseDate>
  <ListRecords>
    <record>
      <header>
        <identifier>lobf8e8025</identifier>
        <setSpec>chem100</setSpec>
      </header>
      <metadata>
        <lom xmlns="http://ltsc.ieee.org/xsd/LOM">
          ...
        </lom>
      </metadata>
    </record>
  </ListRecords>
</gather_expose>
```

3.4.3.9 ListSets Action Protocol

The ListSets action is a request protocol for retrieving a list of support sets from the repository. A set organization is useful for selective gathering. IEEE LOM repository does not use sets. However, the set is a good way to classify LOs. ECL recommends the Dewey Decimal Classifications for set organization.

Arguments:

- `verb` - identifies the action, and the value of this action is “ListSets”.

Error and Exception Conditions:

- `badArgument`
  - `verb` - indicates that the “action” parameter is incorrect.
Expose Parameters:

- `responseDate` - is the response date.
- `ListSets`:
  
  - `set` - is a structure that holds the set spec, name, and description.

Sample 3.18: An Example of the Gather ListSets Request Protocol

```xml
<gather xmlns="http://www.edusource.ca/xsd/ecl_gather_v1pl1">
  <verb>ListSets</verb>
</gather>
```

Sample 3.19: An Example of the Gather ListSets Expose Protocol

```xml
<gather_expose
  xmlns="http://www.edusource.ca/xsd/ecl_gather_ex_v1pl1">
  <responseDate>2006-05-22T14:21:16.205</responseDate>
  <ListSets>
    <set>
      <setSpec>chem100</setSpec>
      <setName>Chemistry</setName>
      <setDescription>
        Chemistry Description
      </setDescription>
    </set>
    <set>
      <setSpec>bio100</setSpec>
      <setName>Biology</setName>
      <setDescription>
        Biology Description
      </setDescription>
    </set>
  </ListSets>
</gather_expose>
```

3.4.3.10 Gather Handler Implementation

Like all ECL service handlers, to provide a gather service, LOR developers only have to implement GatherServiceHandler class and add the implemented class into the ECL infrastructure. GatherServiceHandler is an ECL abstract class for the gather service handler. ECL provides examples to implement each action of the gather service as well as data structures to help LOR developers in the implementation of a gather service.
Sample 3.20: Gather Service Handler Interface Functions

(See Appendix B2)

```java
public abstract GatherExpose getRecord(GatherRequest request) throws GatherException;
public abstract GatherExpose identify(GatherRequest request) throws GatherException;
public abstract GatherExpose getRecord(GatherRequest request) throws GatherException;
public abstract GatherExpose listIdentifiers(GatherRequest request) throws GatherException;
public abstract GatherExpose listMetadataFormats(GatherRequest request) throws GatherException;
public abstract GatherExpose listRecords(GatherRequest request) throws GatherException;
public abstract GatherExpose listSets(GatherRequest request) throws GatherException;
```

3.4.3.11 Gather Client Utility

As shown in the code example below (Sample 3.21), ECL provides simple API for calling application to construct parameters and invoke gather services.

Sample 3.21: A Client Code Example for Invoking an ECL Gather Service for all Six Actions

```java
String serviceKey = null; // a UDDI gather service key
EclSecurity eclSecurity = null; // ECL security

// create an instance of UDDI registry
UddiQuery uddi = new UddiQuery();

// get ECL service information from UDDI
EclServiceInfo serviceInfo = uddi.getServiceInfo(serviceKey);

// making Identify request
GatherRequest request = new GatherRequestImpl();
request.setVerb(GatherRequest.IDENTIFY);
GatherExpose gatherExpose = GatherClient.gather(serviceInfo,
    request, eclSecurity);

// making GetRecord request
request = new GatherRequestImpl();
request.setVerb(request.GETRECORD);
request.setParameter(request.IDENTIFIER, "lobf8e8025");
gatherExpose = GatherClient.gather(serviceInfo, request, eclSecurity);

// making ListIdentifiers request
request = new GatherRequestImpl();
request.setVerb(request.LISTIDENTIFIERS);
```
3.4.4 Submit Service Protocol

ECL represents the Submit/Store interface function of IMS DRI in the submit service protocol. The interface function is for uploading LOs or LOM records to a repository. In this protocol, the infrastructure must be able to handle file attachments. As the message traverses from one module to another, the protocol must find ways to deliver the file attachment from a calling application to the submit service handler. ECL assumes there are two types of submit service: submitting one or more LOM records or submitting an IMS or a SCORM package to a repository. ECL supports the submission of LOM records to a repository, assuming that each LOM record contains a valid URL for the referenced LO. The transport module is responsible for the file upload. Apache Axis has
a built-in support for file attachments, which simplifies the implementation of the ECL. However, if the implementation uses the HTTP protocol, the transport module can use the HTTP MultiPart/Form encoding, which is a standard encoding for Web browsers. The encoding is a simple protocol that allows the transmission of HTTP form data such as string parameters and files to the Web application server. This discussion only wants to point out the ability of the ECL to substitute implemented modules and technologies. Although such implementation took place in another project related to ECL, this document does not cover the detail of HTTP MultiPart/Form encoding protocol.

3.4.4.1 Submit Request Protocol

The IMS DRI does not provide specifications on how to handle ownership, permissions, or resource management. However, LOR network developers should develop a strategy for security. The submit protocol of the ECL defines three types of permissions: public, private, and group-accessible. The following is the list of the submit request parameters.

Submit Parameters:

- **action** – is the value that indicates the submission action: delete or save; delete action takes one or more LOM/LO identifiers.
- **content_type** – is the value that identifies the content type of the submit: **ims** for IMS content package, **scorm** for SCORM content package, and **mds** for a list of LOM records.
- **username** – is a unique username that identifies the owner of the resource; the username must be the same as the username used in the ECL security module.
- **groupname** – identifies the group name if the permission is set to 760.
• **permission** – is the value of the permission based on the UNIX scheme: 766 for public ownership; 760 for group ownership; 700 for private ownership.

• **content** – is the value that identifies (1) a filename of the IMS or SCORM package or (2) the XML string that holds a collection of LOM records.

Sample 3.22: A Skeleton of the ECL Submit Request Protocol

(See Appendix A4a for the ECL submit request protocol XML Schema)

```xml
<submit xmlns="http://www.edusource.ca/xsd/ecl_submit_v1p1">
  <action>save</action>
  <username>johndoe</username>
  <permission>766</permission>

  <!-- Submitting an IMS content package -->
  <content_type>ims</content_type>
  <content>ims_content_filename.zip</content>

  <!-- Submitting a list of metadata -->
  <!-- mds element holds the collection of IEEE LOM records in XML string. -->
  <content_type>mds</content_type>
  <content>... </content>
</submit>
```

3.4.4.2 Store Protocol

The store protocol is an acknowledge response to the submit request. Store could contain a list of successful save or delete LOM identifiers or a list of errors.

**Store Parameters:**

• **identifiers** – is a list of LOM identifiers that have been successfully saved or deleted.

• **errors** – is a list of LOM identifiers that cannot be saved or deleted.
Sample 3.23: A Skeleton of the ECL Store Protocol

(See Appendix A4b for the ECL store protocol XML Schema)

```xml
<store xmlns="http://www.edusource.ca/xsd/ecl_store_v1p1">
  <identifiers>
    <identifier>midbf8e8025</identifier>
  </identifiers>
  <errors />
</store>
```

3.4.4.3 Submit Handler Implementation

SubmitServiceHandler is the ECL abstract class for submit service. It has only one method for LOR developers to implement, which is the processSubmit method. The data structures API are Submit and Store classes. LOR developers get an instance of Submit data structure and a filename of a temporary content package zip file as parameters. ECL also provides API for manipulating IMS and SCORM content packages, and developers can use Store data structure to construct the response. If LOR developers have configured ECL security, the transport module would pass the submit request to the security module to do the validation and store the security token in the instance of Submit data structure that was passed as a parameter. The security token contains a digital certificate and/or a SAML assertion of the requesting client as well as other parameters such as the username.

Sample 3.24: Submit Service Handler Interface Function

(See Appendix B3)

```java
public abstract class SubmitServiceHandler implements ServiceHandler {
    public abstract Store processSubmit(Submit request, String zpath) throws SubmitException;
}
```
3.4.4 Submit Client Utility

The code example below (Sample 3.25) shows that the calling application can submit LOM records or a LO packaged in an IMS content package to a repository. To invoke a submit service, developers can use the Submit data structure to construct the request parameters, and Submit client utility can take care of the request on behalf of the developers.

Sample 3.25: Client Code Sample for Invoking ECL Submit Service

```java
String serviceKey = null; // a UDDI submit service key
EclSecurity eclSecurity = null; // ECL security
UddiQuery uddi = new UddiQuery();
EclServiceImpl serviceInfo = uddi.getServiceInfo (serviceKey);

// create submit structure
Submit submit = new SubmitImpl();
submit.setAction(Submit.SAVE);
submit.setContentType(Submit.IMS);
submit.setObjectAccessPermission("766");
submit.setImsContentName("ims_content_filename.zip");

// instantiate submit client
SubmitClient submitClient = new SubmitClient(serviceInfo,
eclSecurity);

// do the submit request
Store store = submitClient.submitRequest(submit);
```

3.4.5 Request Service Protocol

ECL represents the Request/Deliver interface function of IMS DRI in the request service protocol. IMS DRI anticipates the need for LO delivery through a service. ECL supports IMS DRI view. In an integrated environment, a computer system can request the LOs. If the LOs are protected, the system must negotiate security clearance with the LO providers. Implementing such a process in a service is much simpler than working with HTTP, which is user-interface oriented.
The protocol must enable the download of a LO from a repository. The delivery of a LO through a service is necessary if the LOR wants to enforce security or the LO request is a computer system. WS technologies do not provide support for the file attachment in the response. WS technologies are Web-based. There is no reason for WS technologies to provide such a support. A client should be able to access the file through a URL. As a result, ECL develops a generic strategy for file download using the resumption token technique. ECL downloads the file in chunks via the ECL messaging pipe.

3.4.5.1 Request Protocol

To enable a chunked download of a file, ECL adds the resumption token in the request protocol. ECL calling API continues to request the remaining data of the requested object until it gets all the data and saves the data into a file before presenting the response to the developers.

Request Parameters:

- **type** – is the value that identifies the document type.
- **loid** – is the value that identifies the learning object.
- **resumptionToken** – is a token for the calling application to get the next chunk of data.
Sample 3.26: An Example of the ECL Request Protocol
(See Appendix A5a for the ECL request XML Schema)

```xml
<request xmlns="http://www.edusource.ca/xsd/ecl_request_v1p1">
  <data>
    <type>lo</type>
    <loid>lobf8e8025</loid>
  </data>
  <resumption_token>
    <cursor>65536</cursor>
    <expirationDate>2006-05-22T22:04:36.065</expirationDate>
    <completeListSize>1024000</completeListSize>
    <resumptionTokenId>bf8e8025</resumptionTokenId>
  </resumption_token>
</request>
```

3.4.5.2 Deliver Protocol

To enable the chunked download of a file, ECL adds the resumption token in the delivery protocol. ECL request service handler API takes care of delivering the LO to the calling application. LOR developers only have to supply the filename where the downloaded file is stored.

**Request Parameters:**

- `bytes` – is the data encoded in base64.
- `resumptionToken` – is a token for the calling application to get the next chunk of data if all the data cannot be included in the response.
Sample 3.27: A Skeleton of the ECL Deliver Structure

(See Appendix A5b for the ECL deliver XML Schema)

```xml
<deliver xmlns="http://www.edusource.ca/xsd/ecl_deliver_v1p1">
  <bytes><![CDATA[bytes encoded in Base64]]></bytes>
  <resumption_token>
    <cursor>65536</cursor>
    <expirationDate>2006-05-22T22:04:36.065</expirationDate>
    <completeListSize>1024000</completeListSize>
    <resumptionTokenId>bf8e8025</resumptionTokenId>
  </resumption_token>
</deliver>
```

3.4.5.3 Request Handler Implementation

The abstract class for request service is RequestServiceHandler, which only has one abstract method (processRequest) for developers to implement. The Request service may need to process security similar to the submit service. If applicable, developers would find the security token in the Request object instance passed as a processRequest parameter. Deliver class can take care of the file download. Developers only have to provide the filename.

Sample 3.28: Request Service Handler Interface Function

(See Appendix B4)

```java
public abstract class RequestServiceHandler implements ServiceHandler {
  public abstract Deliver processRequest(Request request) throws RequestException;
}
```

3.4.5.4 Request Client Utility

The ECL request client API simplifies the invocation of the ECL request service. As shown in the code below (Sample 3.29), requesting a LO from a repository requires only a few lines of code.
Listing 3.29: Client Code Sample for Invoking Request Service

```java
String serviceKey = null; // a UDDI request service key
EclSecurity eclSecurity = null; // ECL security
UddiQuery uddi = new UddiQuery();
EclServiceInfo serviceInfo = uddi.getServiceInfo(serviceKey);

// create Request
Request request = new RequestImpl();
request.setLoIdentifier("lobf8e8025");
request.setRequestType(Request.LO);

// invoke Request service
Deliver del = RequestClient.request(serviceInfo, request, eclSecurity);
```

3.4.6 Asynchronous Messaging Protocol

ECL does not explicitly support asynchronous messaging. However, asynchronous messaging is possible in the ECL infrastructure. Sender information allows a calling application to add information for an asynchronous response. There are two possible modes of asynchronous messaging on ECL: push and pull. In pull, the calling application is responsible for pulling the responses from the service provider. The service provider can use the endpoint in sender information as an identifier of the resumption token to identify the results stored temporarily in the service provider's storage. A calling application that resides behind a firewall must use this mode since there is no guarantee that service providers can reach the client behind the firewall. In a push mode, the service provider opens the TCP/IP connection to the calling application to return the responses, which are the ECL messages containing payload responses as in the synchronous messaging (i.e., search expose, gather expose, deliver, etc.) In this case, the endpoint of the sender information is a TCP/IP of a server that can accept ECL messages. The calling application must reside outside of the firewall. Currently, there is no integration example of the asynchronous messaging. ECL elected to use a gateway strategy described in the
next chapter to deal with asynchronous connections such as peer-to-peer network (i.e., LionShare).

3.5 Summary

This chapter has shown the inside of the design and implementation of the ECL. The discussion begins with the design principles and strategies, which influence the design of the ECL architecture. Achieving the design objectives is possible thanks to the flexibility of the XML-based messaging middleware protocol. The middleware technology allows ECL to abstract new technologies and protocol standards from developers by encapsulating them in the middleware. For example, ECL hides the existence of multiple metadata and query language standards from developers and allows developers to use their preferred metadata formats and query languages. The gateway strategy simplifies the task for ECL to interoperate with other infrastructure protocols while other software update strategy permits the compatibility among ECL versions. Finally, the modularity design makes the ECL architecture highly reusable and cross-platform interoperable. Therefore, the design and implementation of ECL support the flexibility argument of the XML-based messaging middleware protocol stipulated by this research thesis.
CHAPTER 4: THE INTEGRATION STRATEGIES OF THE ECL INFRASTRUCTURE

This chapter discusses the integration strategies of the ECL infrastructure and ties the discussion with the description of the algorithms used for service invocation, service dispatching, and domain-specific query filtering. To assist in the integration process, ECL uses a UDDI registry to coordinate the software synchronization as well as the assimilation of new technologies and provide the security infrastructure to enable LORs to share their resources at different levels of security. The discussion begins with the ECL UDDI registry and the ECL security infrastructure. An overview of these two components helps in understanding the algorithm descriptions. The algorithms show how all the components work together within the infrastructure. Finally, this chapter ends with the discussion of various strategies and real-life examples of the ECL integrations with other technologies.

4.1 ECL UDDI Registry Service

The UDDI registry service is essential in a distributed system such as a LOR network to provide a rendezvous point where calling applications can get the latest information about the network and available services. It provides discovery service and coordinates the service invocation as well as the service deployment. Service discovery is the primary task of the UDDI registry as it enables end-users to find suitable services. The ECL discovery service uses the Dewey Decimal Classification system to classify LOR services based on the content held in the repository. Furthermore, a single LOR can
provide more than one search service with different classifications. The strategy speeds up the query process and reduces Internet traffic. The end users only search repositories that are most likely to have the content closely related to their queries. The ECL design has made the process and the communication between the ECL API and the UDDI registry invisible for the LOR developers. For example, a calling application can cache LORs' service keys in the user's preferences. During service invocation, the ECL API always gets the latest service information from the UDDI registry to invoke the service. By being able to update the latest information to the clients, a LOR can update software or reconfigure its services without compromising the interoperability with existing clients.

The ECL UDDI search service allows searching the registry based on security types, LOR classification, and other technical requirements. The goal is to provide a basic searching capability for end users to narrow down their searches. The security criteria and technical requirements eliminate services that end-users cannot access, while classification criteria permit the selection of suitable repositories. A registry such as the ECL UDDI registry does not need fine grain classifications. Related categories may contain materials that match the search criteria. In addition, the Dewey Classification System is hierarchical and numeric. The search system can widen the search by extending the input classification. For example, Dewey groups related areas logically, using a decimal numbering scheme (i.e. 005--Computer programming, 005.1--Programming, 005.7--Data in computer systems, 005.8--Data security, etc.) When an end user enters a field of studies such as "005.7--Data in computer systems", the search system can expand the search to "005--Computer programming". This is only a basic search mechanism.
Further research may expand the search to distance-related classifications such as “510--Mathematics”, “004--Data processing”, etc.

The service invocation and service deployment are hand in hand. Both must deal with security and version control. A software update can affect the service invocation. Similarly, a newly deployed ECL service must inform calling applications about the ECL version, and a calling application using an older version of the ECL must be able to communicate with newly deployed ECL services. ECL uses the UDDI registry to coordinate the flow of this information. To be successful, a service invocation must have the following information: (1) the protocol and version; (2) the requirements of the data conversion; and (3) the security requirements.

The algorithm sections below describe the incorporation of the above information. The strategy of data transformation permits a calling application using ECL version 0.3 to call a service using ECL version 0.5. Using the information in the UDDI registry, a calling application can download input/output XSLT files from a server (i.e. the UDDI registry server) and apply the XSLT transformation on the input and output of the ECL message. Older implementations of the calling application can continue to process the ECL messages as it used to process before, and the data conversion can take place within the ECL infrastructure without the intervention of the calling application developers. During the deployment of the ECL service, ECL automatically registers the technical requirements along with information provided by the LOR developers into the UDDI registry, and continues to update any changes throughout the service lifecycle to ensure that the updated information is always in the registry.
4.2 Security Infrastructure

This section provides an overview of ECL security infrastructure [31]. The integration of security infrastructure reduces the overhead cost of the security, but more importantly, it enables LORs to share information with different security requirements. The ECL security strategy may encourage LORs to free up some licensed resources to end users while having the ability to protect licensed resources. The ECL security is an implementation based on the Shibboleth security (http://shibboleth.internet2.edu/), a federated Single-SignOn and attribute exchange framework that provides extended privacy functionality allowing a user who is authenticated at their home site to access resources on service providers of other organizations. Using Shibboleth security technology, ECL security is able to protect the message integrity and the users' privacy, as well as managing trust for the LOR network. The strength of ECL security is comparable to the security used in industry.

There were some challenges when incorporating the Shibboleth technology. Shibboleth uses XMLSec library to serialize and de-serialize the XML string while Axis continues to use the standard XML parser provided in the Java platform to perform the serialization of the XML. When inserting a security token (SAML assertion) onto the SOAP header and encrypting the body of the SOAP message, Axis cannot perform the serialization correctly. However, due to the flexibility of the XML-based messaging protocol and modularity design of the ECL, the incorporation of the Shibboleth technology was possible. ECL uses WS security specifications to convert the security

---

5 Single sign-on (SSO) is mechanism whereby a single action of user authentication and authorization can permit a user to access all computers and systems where he has access permission, without the need to enter multiple passwords.
token into a string token and insert the string token into the SOAP header. Encryption and signing of the ECL message take place outside of the SOAP body, and the information is put into the security token.

An alternative approach to WS security is to send SOAP messages over a secure connection using SSL (Secured Socket Layer) or TLS (Transport Layer Security). The advantage of these techniques is the use of the existing HTTPS infrastructure, which is already supported by most Web servers. A calling application can establish an anonymous SSL connection with a service provider the same way a browser connects to a secure Web server. However, SSL is a point-to-point technology and operates on the lowest level of interoperability [42]. Furthermore, it only provides message integrity and message privacy protections while ECL security provides full protections including trust management and identity protections.
The overview of the security infrastructure shows the interaction of LOR using different identity providers. This is an example of a trust network where users belonging to organization A can access resources on a LOR of organization B, and because both organizations trust each other, users of organization A do not have to reveal their identities to organization B.

4.3 Algorithms

This section describes the algorithms used in the service invocation, service dispatching, and domain-specific query filtering. The algorithms take advantage of the ECL XML-based messaging protocol features such as dynamic mapping, security handling, and version control.

4.3.1 Service Invocation Algorithm

The service invocation algorithm shows how the technologies and protocol standards work together in an integrated environment to provide a simple solution to the
calling application developers. This algorithm assumes that the caller has the service key from the ECL UDDI registry.

Table 4.1: The Search Service Invocation Algorithm

<table>
<thead>
<tr>
<th>Sending Request:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. If not initialized then</td>
</tr>
<tr>
<td>1.1 Load the basic configuration, service caches, and security configuration</td>
</tr>
<tr>
<td>1.2 Create an instance of the ECL security manager</td>
</tr>
<tr>
<td>End if</td>
</tr>
<tr>
<td>2. If the UDDI update time is expired then</td>
</tr>
<tr>
<td>2.1 Update the service information</td>
</tr>
<tr>
<td>End if</td>
</tr>
<tr>
<td>3. Construct the ECL common message</td>
</tr>
<tr>
<td>4. If the service uses different protocol then</td>
</tr>
<tr>
<td>4.1 Transform the message using the input XSLT provided by the UDDI registry</td>
</tr>
<tr>
<td>End if</td>
</tr>
<tr>
<td>5. If the service requires security then</td>
</tr>
<tr>
<td>5.1 Request a security token from the ECL security manager</td>
</tr>
<tr>
<td>5.2 If the signature and/or encryption is required then</td>
</tr>
<tr>
<td>5.2.1 Encrypt and/or sign the message</td>
</tr>
<tr>
<td>End if</td>
</tr>
<tr>
<td>5.3 Construct a security token based on the collected information</td>
</tr>
<tr>
<td>5.4 Insert the security token into the SOAP header</td>
</tr>
<tr>
<td>End if</td>
</tr>
<tr>
<td>Processing Response:</td>
</tr>
<tr>
<td>6. Invoke the service using the endpoint and the service name in the service info</td>
</tr>
<tr>
<td>7. If the message was signed and encrypted then</td>
</tr>
<tr>
<td>7.1 Verify the signature and decrypt the message</td>
</tr>
<tr>
<td>End if</td>
</tr>
<tr>
<td>8. If the service uses a different protocol then</td>
</tr>
<tr>
<td>8.1 Transform the response message using the output XSLT provided by the UDDI registry</td>
</tr>
<tr>
<td>End if</td>
</tr>
<tr>
<td>9. Convert the response from the ECL common message to the local format</td>
</tr>
<tr>
<td>10. Do the domain specific query filter to remove all LOM records that do not meet the search criteria and present the result to the caller</td>
</tr>
</tbody>
</table>

4.3.2 Service Dispatching Algorithm

The service-dispatching algorithm below (Table 4.2) describes the message-handling process. Since this algorithm occurs on the service side, it assumes ECL has
loaded the service configuration, updated the required information to the UDDI registry, and created an instance of the ECL security if the LOR uses any type of security. The algorithm starts when the ECL transport module receives an ECL message from the calling application. The message traverses from the transport module to the dispatcher module and to the service handler.

Table 4.2: The Search Service Handling Algorithm

1. If the ECL security is not null then
   1.1 Pass the message handle to the ECL security to extract the security token, verify the signature, and decrypt the message
   1.2 Place the security token into the message context (where it is accessible by the dispatcher module)
   End if
2. If the service handler is not loaded then
   2.1 Load the service handler
End if
3. Construct the request data structure from the ECL Module message
4. Extract the security token from the message context and insert the token into the request data structure
5. Convert the request to the local format and call the service handler to process the request
6. If the metadata format of the response is not the proffered metadata format then
   6.1 If can convert to the proffered metadata format then
   6.1.1 Convert the result to the request format
   Else
   6.1.2 Convert the result to the ECL common IEEE LOM
   End if
   End if
7. Construct the response message
8. If the ECL security is not null and the signature and/or encryption are required then
   8.1 sign and/or encrypt the message
End if
4.3.3 Domain Specific Query Algorithm

Currently, there is no agreement for a metadata query language standard protocol, and many LORs do not support advanced search. ECL resolves this problem by providing a search filter API to perform the advanced search on the returned search results directly in ECL search client before presenting the results to the end-users.

Table 4.3: The Domain Specific Query (DSQ) Utility Algorithm

This is a recursive algorithm due to the tree structures of the LOM record and the query structure. The algorithm returns true if a LOM record meets query criteria.

<table>
<thead>
<tr>
<th>Function doDSQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. If the query element is a &quot;complex&quot; query then</td>
</tr>
<tr>
<td>1.1 Extract the LOM child element based on the given X-path</td>
</tr>
<tr>
<td>1.1.2 Return doDSQ (LOM child element, child query)</td>
</tr>
<tr>
<td>1.2 Else if the query element is an &quot;OR&quot; or an &quot;AND&quot; operator then</td>
</tr>
<tr>
<td>1.2.1 For each child query element do</td>
</tr>
<tr>
<td>1.2.1.1 If doDSQ () and the query element is an &quot;OR&quot; operator then</td>
</tr>
<tr>
<td>1.2.1.1.1 Return true</td>
</tr>
<tr>
<td>1.2.1.2 Else if NOT doDSQ () then</td>
</tr>
<tr>
<td>1.2.1.2.1 Return false</td>
</tr>
<tr>
<td>1.2.2 End if</td>
</tr>
<tr>
<td>1.2.2.1 If the query element is an &quot;AND&quot; operator then</td>
</tr>
<tr>
<td>1.2.2.1.1 Return true</td>
</tr>
<tr>
<td>1.2.2.2 Else</td>
</tr>
<tr>
<td>1.3.1 If the attName not null then</td>
</tr>
<tr>
<td>1.3.1.1 Extract the attribute value from the element</td>
</tr>
<tr>
<td>1.3.1.2 Else</td>
</tr>
<tr>
<td>1.3.2.1 Extract the text value from the element</td>
</tr>
<tr>
<td>1.3.2.2 End If</td>
</tr>
<tr>
<td>1.3.3 If the extracted value eq/gt/lt/contains keyword then</td>
</tr>
<tr>
<td>1.3.3.1 Return true</td>
</tr>
<tr>
<td>1.3.4 Return false</td>
</tr>
<tr>
<td>1.3.5 End If</td>
</tr>
<tr>
<td>End Function</td>
</tr>
</tbody>
</table>
4.4 The Integration of the ECL Infrastructure with other Systems

In October 2003, the edusource partners met for an eight-hour workshop to integrate the ECL with the edusource repositories: CAREO, CanLOM, Explora, and Athabasca’s repository. The workshop was successful. The integration was easy, as expected. The workshop attests to the interoperability and flexibility of the ECL implementation. Figure 4.2 below illustrates an overview of the integrated environment of the ECL infrastructure with an existing LOR. As shown in the figure, the requirements for developers to integrate ECL are kept to a bare minimum.

Figure 4.2: An Overview of the Integration of ECL Infrastructure

The calling application developers do not have to worry about the complexity of the ECL protocol. They just use the ECL API to find and invoke ECL services. Similarly, LOR developers only have to concern themselves with the service handler class and the method they have to implement.
After the workshop, the work on the ECL integration with other protocol infrastructures continued. ECL has two integration strategies to enable ECL clients to access resources on other networks using their protocols. The first strategy is the amendment of client gateways into the ECL distribution to allow a calling application to invoke services using other protocols directly. The second strategy is to install gateway services as proxies to other protocol infrastructures. These strategies also enable users from other networks to access resources in the ECL LOR network. In this case, the protocol translation process is reversed.

Figure 4.3 is an overview of the client gateway architecture. Adding gateways to the ECL client distribution enables ECL calling applications to invoke services using other protocols as if the services were ECL services. Non-ECL repositories do not have to intervene to make their repositories ECL-compatible. The disadvantages are the size of the ECL distribution and the instability of these protocols. SRW, SQI, OKI, EdNA, and SMETE are specific for each particular installation endpoint (e.g., two SRW Web services require two different sets of stub code). Hence, each service needs a set of stub code. Clearly, this is not scalable. Furthermore, these protocols require frequent updates. As a result, the client gateway architecture only works if the number of repositories is limited, and the repositories have stable services. For this reason, ECL is only able to maintain a small number of selected client gateways: ARIADNE (SQI), EdNA, SMETE, and RDN (SRU and SRW).
A client gateway translates the ECL messaging protocol to the protocol of the service provider, invokes the service, and transforms the response back to ECL messaging protocol.

Figure 4.4 is the server gateway architecture. The gateway accepts requests from ECL calling applications, converts the requests to the foreign protocol requests such as SRU, SRW, SQI, OKI, EdNA, or SMETE requests, and invokes the services on behalf of the calling applications. Then the gateway converts the responses from other protocol services to ECL responses before returning the response to the calling applications. To ECL clients, a server gateway is just another ECL service. This strategy reduces the size of the ECL distribution and provides some flexibility for protocol updates. The downside of this strategy is the bottleneck at server gateways and the maintenance of the gateways. Processing protocol transformation and relaying requests have a high cost in execution time. In addition, due to the resource constraints, a server gateway usually hosts more than one gateway service. The strategy works fine for a small volume of repositories, but large repositories such as EdNA, SMETE, and RDN may cause huge bottlenecks on server gateways. While load balancing can resolve the bottleneck problem, it requires more resources to maintain a gateway server. For this reason, ECL uses both gateway strategies for large repositories. From the UDDI registry information, an ECL client
determines if it should use a client gateway or a server gateway by looking at the required ECL protocol in the registry information. If the ECL client does not have the required version of the client gateway, it makes the request through the server gateway. This strategy allows ECL to deal with the frequent updates in foreign protocol infrastructures.

Figure 4.4: The Server Gateway Architecture for ECL Clients

The server gateway transforms the ECL message protocol to another protocol, relays the call to the designated service provider, and transforms the response back to the ECL message protocol before returning the response to the ECL client.

The above discussion covers concerns about allowing ECL clients to access resources on networks using other protocols. Figure 4.5 is an overview of the gateway architecture for other protocols to invoke ECL services. For performance purposes, calling applications using other protocols should use this architecture as a client gateway. However, it is possible to use this architecture as a server gateway for other protocols. ECL uses this architecture to implement OSID for OKI/OSID consumers.

102
Currently, there are integrated gateways for SRU and SRW of Resource Discovery Network (RDN), SQI of ARIADNE and FIRE, SMETE, EdNA, and LionShare. SMETE is a repository that uses its own protocol and is a Web service that supports Lucene text search for metadata query. EdNA is another repository that has its own protocol, while LionShare is a bi-directional-server-gateway that bridges to a secure peer-to-peer network based on trust and responsibility. Users on the LionShare network can impose access restrictions to the resources, i.e., the resources are shared only with authorized users satisfying those permissions. Hence, the integration with LionShare involves dealing with security and the usage of the ECL request service. The integration cases described above demonstrate the flexibility of the ECL infrastructure and its ability to interoperate with other protocols.

4.5 Summary of the ECL Integration

The XML-based messaging and middleware features of the ECL infrastructure make ECL easy to integrate with other protocol infrastructures. To interoperate with a system such as the LionShare demands a highly flexible infrastructure that is able to meet
the security requirements, deal with asynchronous messaging, and interoperate with a peer-to-peer protocol. Apart from providing a simple integration with a service handler implementation for existing LORs and different types of gateway strategies, the ECL infrastructure provides a mechanism for coordinating vertical compatibility software updates, service configuration, and service diagnostics. Older implementations of the ECL can interoperate with new implementations via the data transformation. ECL automatically uploads updated information of installed ECL services to the UDDI registry, which immediately makes the information available to the calling applications. Similarly, calling applications can digest the service information obtained from the UDDI registry, make necessary data transformations, and is able to invoke the service without the intervention of the calling application developers. Finally, the flexibility of the ECL infrastructure permits the addition of new components such as the ECL security infrastructure. Evidently, the integrations discussed in this chapter support the flexibility of the XML-based messaging argument stipulated by the research thesis.
CHAPTER 5: THE ANALYSIS OF THE ECL INFRASTRUCTURE

This chapter continues the discussion with a comparative analysis of the ECL infrastructure against the four select protocol infrastructures. The goal of this analysis is to see the advantages and disadvantages of the infrastructure protocols. The design of ECL focuses on an evolving collaborative network, while other protocols have their merits and offer better advantages for other types of networks. For example, if the view of the LOR network is a publicly accessible repository that only needs text search, then SRU/SRW is the best protocol. In the first part, we analyse the architecture design. The structure and technologies used in the architecture as well as the design decisions have an impact on the infrastructure flexibility. The architecture analysis should reveal characteristics that may or may not be desirable for a collaborative LOR network. Next, we analyse the infrastructure integration. To incorporate a service, developers must understand the service, syntactically and semantically [25]. The analysis approximates the complexity of the infrastructure integration based on the number of methods and parameters of the services as wells as the protocol structure. Last is the analysis of the interoperability. The analysis concentrates on the amount of work that is required to make an infrastructure protocol interoperate with another protocol, and focuses on the interoperability in one direction at a time. This analysis identifies work needed to improve the interoperability between two protocols.
5.1 Architectural Comparative Analysis

Based on the literature review, it appears that there are two distinct aspects to architecture evaluation analysis: performance and the quality assurance of the architecture. Performance is largely a function of the frequency and nature of inter-component communication and is predictable by the study of the architecture [15]. A performance analysis focuses mainly on fine-tuning the architecture performance that all organizations should do before releasing their software. This research is only concerned with the quality assurance of the architectures, and considers whether the architectures meet the requirements of an evolving collaborative network. Over the past decade, there have been a large number of proposed architectural analysis methods, and each focuses on different aspects of the architecture [41]. The method depends on the criteria chosen to conduct the analysis. In other words, a software architecture analysis is an ad hoc analysis. For example, Lindvall et al. proposed an evaluation based on the analysis of maintainability. The method is the restructuring of the architecture by adding new components and analyzing the discrepancies between the new and old architectures [49]. Meanwhile, Zhu et al. proposed an architecture evaluation using an analytic hierarchy process, which identifies critical tradeoffs and sensitive points in the decision-making process [96]. The architecture trade-off-analysis method is a scenario-based evaluation method. Quality scenarios are gathered through stakeholder workshops and the requirement analysis.

This research uses a similar method proposed by Kontio et al. in evaluating reusable software components. The method consists of defining factors, goals, and criteria for evaluating a reusable component [43]. Of course, factors, goals, and criteria in
this research analysis are based on the need of the collaborative and developing environment of a LOR network. As Kazman et al. have indicated, the intent of the criteria is to determine how well the architecture satisfies a set of business goals, which are the design objectives of this research thesis. The following criteria are essential for analyzing and evaluating a LOR network. These criteria would demonstrate the superiority of the XML-based messaging middleware infrastructure of the ECL over other protocols that do not take advantage of the XML-based messaging and the middleware technology.

1. *Multi-Query Language* – is the support of multiple query languages. These criteria affect the protocol interoperability with other protocols and the integration of existing LORs.

2. *Common Query* – is the protocol support of a common query language. While the multiple query language support allows LOR developers to use their favourite query languages, calling application developers should only have to deal with a single query language. Hence, having a common query for calling applications is an advantage for a communication protocol.

3. *Query Mapping* – is the ability of the protocol to map a metadata query between different query languages. A good query mapping strategy is an obvious advantage for an infrastructure protocol.

4. *LOM Mapping* – refers to the protocol capability of providing strategy for metadata mapping. Metadata mapping is not always straightforward. As shown in the interoperability analysis section below, metadata mapping can result in a significant loss of data. The protocol must have strategies to increase the quality of the metadata mapping.

5. *Version Compatibility* – is the ability and advantages of a protocol to maintain compatibility with its own protocol versions.

6. *Multi-purpose Framework* – is the protocol capability to expand and support other types of communications.
7. **Platform Independence** – refers to the adaptation of the protocol to work with different platforms.

8. **Discovery service** – is essential in a LOR network. This criterion identifies the strategy of the protocol to form a LOR network dynamically.

9. **Asynchronous** – is the capability of the protocol to support asynchronous messaging. Asynchronous messaging ability can be useful for certain types of services and is one of the SQI focuses.

10. **Interoperability** – is the potential capability to interoperate with other protocols. The analysis attempts to determine whether the protocol has strategies to interoperate with other protocols.

The evaluation factors are Fully Supported (FS), Partially Supported, (PS), Barely Supported (BS), Not Supported (NS), and Not Applicable (NA). The ECL is designed to provide full support to all these criteria. Therefore, if a small protocol such as SRU/SRW is able to provide full or partial support, SRU/SRW may be considered a better option than ECL. However, this last decision is the LOR network developers’ decisions.
Table 5.1: The Evaluation of the ECL Architecture

<table>
<thead>
<tr>
<th>No</th>
<th>Criteria</th>
<th>Level</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Multi-Query Language</td>
<td>Fully supported</td>
<td>Previous discussion has shown ECL provides strong support and an abstraction level to deal with multiple query languages.</td>
</tr>
<tr>
<td>2</td>
<td>Common Query</td>
<td>Fully supported</td>
<td>ECL strongly supports common query language for the calling applications. This is the abstraction principle of the ECL design.</td>
</tr>
<tr>
<td>3</td>
<td>Query Mapping</td>
<td>Fully supported</td>
<td>This is an ongoing work of ECL, and it is the ECL design objective.</td>
</tr>
<tr>
<td>4</td>
<td>LOM Mapping</td>
<td>Fully supported</td>
<td>ECL uses UDDI to coordinate and improve the quality of metadata mapping, and it is an ongoing effort of the ECL to provide better mapping algorithm.</td>
</tr>
<tr>
<td>5</td>
<td>Version Compatibility</td>
<td>Fully supported</td>
<td>This is also another ECL design objective.</td>
</tr>
<tr>
<td>6</td>
<td>Multi-purpose Framework</td>
<td>Fully supported</td>
<td>Throughout this thesis, there are significant proofs that the ECL infrastructure can support extensions.</td>
</tr>
<tr>
<td>7</td>
<td>Platform Independence</td>
<td>Fully supported</td>
<td>ECL is aiming for platform independence. The implementations of ECL in different platforms can interoperate with each other. Hence, ECL is platform-independent.</td>
</tr>
<tr>
<td>8</td>
<td>Discovery Service</td>
<td>Fully supported</td>
<td>ECL uses UDDI to allow calling applications to find new services. In addition, the end-users can perform searches on the UDDI registry to select suitable LORs.</td>
</tr>
<tr>
<td>9</td>
<td>Asynchronous</td>
<td>Fully supported</td>
<td>ECL can support asynchronous messaging without extra definition, and the ECL can deal with firewalls, which is a problem in today's network infrastructure.</td>
</tr>
<tr>
<td>10</td>
<td>Interoperability</td>
<td>Fully supported</td>
<td>ECL has developed gateway strategies to interoperate with other protocols, which makes ECL highly interoperable.</td>
</tr>
</tbody>
</table>
5.1.1 SQI Architecture

SQI is a collaborative effort of the CEN/ISSS (Comité Européen de Normalization / Information Society Standardization System) Learning Technologies Workshop, to achieve interoperability between learning object repositories, and the Information Society Technology programme-funded PROLEARN project to coordinate some of the practical experiments with the SQI specifications (http://ariadne.cs.kuleuven.ac.be/vqwiki-2.5/jsp/Wiki?LorInteroperability). The first workshop meeting took place in Madrid in Spain, in January 26, 2004, seven months after the release of the ECL 0.3, and the first SQI document was released in May, 2005 [78]. As mentioned earlier, SQI only supports the search service, so it is only possible to compare one of the four ECL services. The discussion in this section is mainly drawn from the integration of the ECL with SQI/ARIADNE and SQI/FIRE. Both use SQI to provide search service interoperability.

Figure 5.1: An Overview of the SQI Architecture

Please note that FIRE is able to implement this architecture since it is a middleware infrastructure while ARIADNE must provide additional utilities in order to implement this type of infrastructure. This type of infrastructure cannot fit in a WSDL framework.

Source: This is a simplified version of SQI architecture found in SQI documentation [78].
Technically, SQI/FIRE implementation can follow the SQI architecture shown in figure 5.1. FIRE is a middleware that wraps over the Java Message Service (JMS) framework, a messaging standard that allows application components based on the Java 2 Platform to create, send, receive, and read messages [30]. JMS is a centralized messaging system that has a provider to coordinate all message deliveries and transactions. Calling applications as well as service providers are clients of the JMS provider, and all JMS clients must maintain an open connection to a JMS provider. As a result, FIRE must package JMS in a middleware infrastructure similar to ECL. FIRE inherits the same advantages from the middleware technology as ECL does. It can hide the complexity of the SQI protocol and provides simple Java interface functions for developers to implement SQI services and client API to access SQI services. Meanwhile, ARIADNE opts for document style WS, and uses its WSDL file as a reference for other developers to implement SQI services. To follow the architecture diagram in figure 5.1, ARIADNE must provide additional utility APIs to the developers. The ARIADNE approach requires additional effort from the developers who have to understand the syntax of the WSDL file, the semantic of the SQI service, and any utility that comes with the SQI/ARIADNE. Note SQI uses the term “source” for calling application and “target” for service provider.

The messaging architectures of the ECL and SQI are somewhat similar. Both are XML-based messaging protocols. Like ECL, SQI can use any transport protocol (i.e., XML RPC, Java RMI, JMS, or SOAP) [30]. The main different between SQI and ECL messaging protocols are the interface functions. SQI explicitly supports open connection for asynchronous messaging while ECL provides asynchronous support implicitly, and strongly emphasizes synchronous messaging. To support an open connection
asynchronous messaging, SQI must support session management, and provide interface functions that allow calling applications to interact with the service providers. As a result, SQI allows a calling application to set the query language, maximum results, and source location (the calling application endpoint) prior to making the metadata query. This makes SQI protocol more complicated than the ECL protocol. The following shows some interface functions of the SQI messaging protocol:

- **setQueryLanguage** – is the operation for setting the query language.

  **Request Message:**
  
  \[
  \langle \text{setQueryLanguage} \rangle \\
  \langle \text{targetSessionID} \rangle \\ 
  \langle \text{queryLanguageID} \rangle \\ 
  \langle \text{setQueryLanguage} \rangle 
  \]

  **Response Message:**
  
  A message string response or a SQI fault that identifies an occurrence error

- **synchronousQuery** – is the operation for making a query request

  **Request Message:**
  
  \[
  \langle \text{synchronousQuery} \rangle \\
  \langle \text{targetSessionID} \rangle \\
  \langle \text{querystatement} \rangle \\
  \langle \text{startResult} \rangle \\
  \langle \text{synchronousQuery} \rangle 
  \]

  **Response Message:**
  
  A message string response containing listing of LOM records in which the format the calling application had specified earlier in setResultsFormat or a SQI fault that identifies an occurrence error

---

6 These examples are typical message structures according to SQI version 1.0. Readers must be aware that there have been many changes since version 0.4, and the message structures may be different in the new SQI version.
• *setSourceLocation* – is the operation for setting a call back service endpoint for the service provider to call back; this operation must be called prior to making the asynchronous query.

**Request Message:**

```xml
<setSourceLocation>
  <targetSessionID>!--session id--</targetSessionID>
  <sourceLocation>
    <!-- endpoint that implements SQI query listener service -->
  </sourceLocation>
</setSourceLocation>
```

**Response Message:**

A message string response or SQI fault that identifies an occurrence error

• *asynchronousQuery* – is the operation for making asynchronous query request; setSourceLocation must be called first.

**Request Message:**

```xml
<synchronousQuery>
  <targetSessionID>!--session id--</targetSessionID>
  <queryStatement>
    &lt;simpleQuery&gt;&lt;term&gt;&quot;java&quot;&lt;/term&gt;&lt;/simpleQuery&gt;
  </queryStatement>
  <queryID>!--query id--</queryID>
</synchronousQuery>
```

**Response Message:**

A message string response or SQI fault that identifies an occurrence error

• *queryResultsListener* – is the operation for the service provider to return the query result.

**Request Message:**

```xml
<queryResultsListener>
  <queryID>
    <!-- query id used in asynchronousQuery -->
  </queryID>
  <queryResults>
    <!-- XML string containing listing of LOM records in which the format the calling
```
The listings above are only a part of the SQI operations. In total, SQI has nine operations. However, developers do not have to implement all the SQI operations. For example, if they do not want to support an asynchronous query, developers can implement an asynchronous query operation that returns a SQI fault "QUERY_MODE_NOT_SUPPORTED". With this number of operations, SQI has not yet covered the query language. FIRE has defined an advanced query structure that supports language for LO and age range, which is not part of the SQI discussion.

Compared to the ECL search service, SQI defines additional interface functions to provide flexibility, but at the same time the protocol becomes more complex. The ECL design objective is to make the interface as simple as possible for developers. The ECL search service only has one interface function -- the search function -- and embeds all other SQI operations for synchronous query as parameters in the request message. Ideally, the response should be in a single message. However, ECL does provide the resumption token technique to return addition results of a search request to the calling application. ECL lets LOR developers limit the maximum results in a response, since LORs developers should know what a suitable size of response is, based on the usability study and the speed of their servers.

SQI protocol allows an open connection between the calling application and the service provider. This feature is easy to implement in an infrastructure such as FIRE, but
can be difficult on a WS infrastructure. Furthermore, most of the calling applications are behind firewalls and are not reachable by the service providers. Due to the complexity, ECL does not explicitly support asynchronous messaging. However, LOR developers can use ECL in an asynchronous mode. A calling application can indicate to the service providers that it wants the results to be returned through a TCP/IP endpoint or call back to get the results using a pull mechanism. The structures of the messages are the same as in synchronous messaging.

In addition, the implementation of SQI/ARIADNE is more difficult. The review has shown that to implement a Web service from a WSDL file, developers need a greater understanding of the service semantics. Based on the ECL integration experience, developers must construct data structure classes with respect to the WSDL file and implement a service class that can process SQI messages. This needs some time for developers to digest the information. Another problem, when implementing services based on a WSDL file, the SOAP utility used to implement the service may produce a WSDL file that is not identical to the referenced WSDL file. The SOAP utility can add XML namespace to identify certain WSDL elements. As a result, the integration with a calling application requires more attention. Furthermore, the support of multiple metadata query languages without query language mapping constrains developers to support the only query languages they use. Developers have to reject query languages that they cannot support. This implies that calling application developers must know what query languages and result formats the service providers can support. If a calling application chooses a wrong query language or metadata format, the call to the service would fail. In contrast, ECL abstracts query complexity from developers and takes care of query
language mapping behind the scene to allow developers to use whatever query language or metadata format they want to use. Finally, since each WSDL file is unique for each SQI service, calling applications have to generate stub code for each service. This means that they have to integrate each SQI service individually. This is a disadvantage compared to the one-for-all integration of ECL or FIRE.
<table>
<thead>
<tr>
<th>No</th>
<th>Criteria</th>
<th>Level</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Multi-Query Language</td>
<td>Fully supported</td>
<td>Calling application as well as the service provider can choose to use or support any query language.</td>
</tr>
<tr>
<td>2</td>
<td>Common Query</td>
<td>Barely supported</td>
<td>This may change with the current effort to define a common query for SQI, which SQI expects to release soon, possibly in the fall of 2006.</td>
</tr>
<tr>
<td>3</td>
<td>Query Mapping</td>
<td>Barely supported</td>
<td>This may change as well. The architecture diagram shows the SQI has a wrapper used for mapping purposes. SQI may provide mapping with the release of SQI common query.</td>
</tr>
<tr>
<td>4</td>
<td>LOM Mapping</td>
<td>Partially supported</td>
<td>Similar to query language, developers have the choice of metadata formats.</td>
</tr>
<tr>
<td>5</td>
<td>Version Compatibility</td>
<td>Not supported</td>
<td>SQI does not have a strategy for version compatibility.</td>
</tr>
<tr>
<td>6</td>
<td>Multi-purpose Framework</td>
<td>Not supported</td>
<td>There is no such strategy.</td>
</tr>
<tr>
<td>7</td>
<td>Platform Independence</td>
<td>Fully supported</td>
<td>If not considering FIRE (JMS is platform-dependent), SQI/ARIADNE is platform-independent.</td>
</tr>
<tr>
<td>8</td>
<td>Discovery Service</td>
<td>Partially supported</td>
<td>SQI/ARIADNE only defines a WSDL file; hence, the discovery is manual; however, FIRE/JMS has a type of peer-to-peer discovery service.</td>
</tr>
<tr>
<td>9</td>
<td>Asynchronous</td>
<td>Fully supported</td>
<td>Asynchronous SQI design objective; however, SQI does not seem to deal with firewalls.</td>
</tr>
<tr>
<td>10</td>
<td>Interoperability</td>
<td>Partially supported</td>
<td>SQI can use server gateway strategy to interoperate with other protocols.</td>
</tr>
</tbody>
</table>
5.1.2 OKI Architecture

OKI has defined and promoted a set of interface functions for service consumers (calling application) called Open Source Interface Definitions (OSID). The goal of OKI/OSID is to provide general software contracts between service consumers and service providers [89]. OSID provides a wide range of definitions for common functions in the learning environments such as the assessment, grading, filling, authentication, authorization, logging, etc. This section is only concerned with the Digital Repository OSID and its repository manager. The analysis is based on ECL/OSID implementation, which was a part of the join project with LionShare (http://lionshare.its.psu.edu/). ECL implements OSID functionality using a gateway strategy for other protocols [33].

According to OKI, application developers can rely on OSID to provide a constant and well-defined interface function. They can use the functions without worrying how service providers implement these functions. This may work in theory. In reality, service provider developers have to make many difficult decisions when implementing OSID functions. There are numbers of interface functions that are not applicable to a repository. Despite the effort and the wide range of ECL interface functions, ECL can only implement approximately 60-70 percent of OSID repository manager functions, and throws “unimplemented exception” for the remaining functions. This is common OKI/OSID implementation. What are the effects of these unsupported functions? No doubt calling application developers must know how to deal with unimplemented interface functions, which can be a big issue if the calling application uses more than one OSID implementation. Clearly, the same interface function implemented by two different service providers is not the same. How can a calling application deal with such a
discrepancy? Clearly, the only solution is the coordination between the developers of a calling application and the developers of a service provider.

Several decisions made during the implementation of OKI/OSID can be described as questionable. Table 5.3 lists the examples of the ECL/OSID implementation issues and the resolutions taken to resolve these issues.

<table>
<thead>
<tr>
<th>Issues</th>
<th>Explanation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Updating Data (LOM or LO) to Repository</td>
<td>OSID assumes directly operation on the repository, and any change should reflect the repository. ECL accesses a remote repository. It must know when and what data should be saved.</td>
<td>ECL added monitoring points for editing on all the data structures to raise a flag if a change has occurred. When the data is no longer used, ECL issues a save command to the remote repository if there has been a change.</td>
</tr>
<tr>
<td>Redundancy of Asset Name and Incompatibility of Data Structure</td>
<td>For ECL, the asset(^7) name is redundant since the asset contains only one record, and ECL only has one name to map to record name and asset name. The problem is due to data structure incompatibility. There is no direct mapping.</td>
<td>ECL uses metadata title for asset name while all record names are set to “IEEE LOM Metadata”. This is the only mapping that can be made from two incompatible types.</td>
</tr>
<tr>
<td>OKI/OSID does not have Definitions for IEEE LOM Record Type</td>
<td>OSID has a few definitions for record types but does not have IEEE LOM definitions. However, OSID does allow Out-of-Band Agreement definitions.</td>
<td>ECL implemented Out-of-Band Agreement definition for IEEE LOM record type.</td>
</tr>
</tbody>
</table>

\(^7\) The asset is the term given to a collection of dependent LOs, which can have multiple records.
<table>
<thead>
<tr>
<th>Issues</th>
<th>Explanation</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Language Support</td>
<td>IEEE LOM standard supports multiple languages. OSID does not support multi-languages. This becomes an issue when trying to convert IEEE LOM to OSID record.</td>
<td>ECL attempts to select parts in IEEE LOM that match the local default language or use the first entry. This is not a perfect solution.</td>
</tr>
<tr>
<td>Mapping IEEE LOM to OSID</td>
<td>Mapping a tree structure with multiple values of IEEE LOM to a flat structure of OSID parts defined by OKI is difficult. For example, the Annotation part of IEEE LOM has multiple contributors and dates.</td>
<td>ECL simply maps the first contribution date of the annotation. Other contributors are not omitted.</td>
</tr>
</tbody>
</table>

Currently, there are several OKI/OSID implementations: SRW/OSID, SQI/OSID, LionShare/OSID, ECL/OSID, etc. Hence, to interoperate with these implementations, calling applications must deal with the discrepancies among implementations. This could be a difficult task for developers to deal with, and can result in a significant loss of data. Normally, the calling application developers can only use common features among all the implementations without additional coding. For example, if all OSID implementations return title as a string, then the calling applications can use it without any problems. However, if some OSID implementations return title as an XML string, then they must implement the code to deal with this discrepancy.
Table 5.4: The Evaluation of the OKI Architecture

<table>
<thead>
<tr>
<th>No</th>
<th>Criteria</th>
<th>Level</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Multi-Query Language</td>
<td>Not applicable</td>
<td>OKI depends on the underlying protocol that implements the OSID; hence, OKI has no such strategy.</td>
</tr>
<tr>
<td>2</td>
<td>Common Query</td>
<td>Fully supported</td>
<td>It is the OKI goal to provide a single interface.</td>
</tr>
<tr>
<td>3</td>
<td>Query Mapping</td>
<td>Not applicable</td>
<td>Similar to criterion 1; however, an implementation of OKI/OSID to access multiple repositories must have some kind of query mapping.</td>
</tr>
<tr>
<td>4</td>
<td>LOM Mapping</td>
<td>Not applicable</td>
<td>It is similar to criterion 3.</td>
</tr>
<tr>
<td>5</td>
<td>Version Compatibility</td>
<td>Not applicable</td>
<td>However, the goal of OKI is to provide a stable interface.</td>
</tr>
<tr>
<td>6</td>
<td>Multi-purpose Framework</td>
<td>Not supported</td>
<td>OKI provides all types of definitions, but one implementation of OSID only serves one purpose.</td>
</tr>
<tr>
<td>7</td>
<td>Platform Independence</td>
<td>Barely supported</td>
<td>OKI can be implemented on any platform, but the implementations are not interoperable.</td>
</tr>
<tr>
<td>8</td>
<td>Discovery Service</td>
<td>Not applicable</td>
<td>It is similar to criterion 1.</td>
</tr>
<tr>
<td>9</td>
<td>Asynchronous</td>
<td>Not applicable</td>
<td>It is similar to criterion 1.</td>
</tr>
<tr>
<td>10</td>
<td>Interoperability</td>
<td>Barely supported</td>
<td>As shown in the analysis, calling application should only have to use one implementation of OKI/OSID.</td>
</tr>
</tbody>
</table>

5.1.3 SRU Architecture

SRU is a simple search and retrieve operation between a Web server and a calling application. SRU search has an HTTP request form. SRU plays with the input parameters
and the output XML document to provide a basic protocol for an institutional LOR. This simplicity makes SRU one of the most attractive protocols. The following describes SRU parameters and explains how SRU uses these parameters to provide SRU functionalities.

- A calling application can specify what version it is using by adding “version” parameter into the request. A service provider can choose to support a version by processing the query or not to support a version by returning an unsupported version exception. This does not constitute automatic backward version compatibility. LOR developers must take it upon themselves to support backward compatibility by continuing to support older versions.

- The parameter “query” is the query input based on CQL. To claim CQL conformance, repository must at least support first level (level 0) of the three CQL levels (0-2). To support level 0, the repository should be able to process a term-only query. The term can be a single keyword or combination of keywords (e.g., “Great Barrier Reef”).

- The parameters “startRecord”, “maximumRecords”, and “resultSetTTL” allow calling applications to control the flow of the results.

- The parameters “recordPacking”, “recordSchema”, “recordXPath”, and “sortKeys” allow calling applications to control the format of the results.

- The parameter “stylesheet” enables certain capabilities for metadata mapping. However, it is up to the repository to support style sheets.

- The parameters “extraRequestData” and “operation” provide certain provisions for protocol expansion. For example, the parameter “operation” enables LOR developers to support additional operations other than “search/retrieve”. Currently, “search/retrieve” is the only supported operation.

The response of SRU is in XML format, which contains a set of response parameters including “records”, which houses a list of DC records. SRU only mandates
“version” and “numberOfRecords”. The rest of the parameters are optional. Obviously, these parameters allow repositories to package and manage the return of the results.

If repositories are archive and institutional repositories, SRU is probably the best candidate. The query language may be limited, but the expansion of the SRU query should be simple. However, the integration of the SRU protocol into a calling application can be an issue. As repositories implement their version of the SRU, each repository is free to support its own set of request criteria. Therefore, calling application developers must integrate each repository individually to ensure that they use the parameters supported by an individual repository.
### Table 5.5: The Evaluation of the SRU Architecture

<table>
<thead>
<tr>
<th>No</th>
<th>Criteria</th>
<th>Level</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Multi-Query Language</td>
<td>Not supported</td>
<td>SRU is a single query language.</td>
</tr>
<tr>
<td>2</td>
<td>Common Query</td>
<td>Fully supported</td>
<td>By contrast to criterion 1</td>
</tr>
<tr>
<td>3</td>
<td>Query Mapping</td>
<td>Not supported</td>
<td>With respect to criterion 1</td>
</tr>
<tr>
<td>4</td>
<td>LOM Mapping</td>
<td>Partially supported</td>
<td>It is the responsibility of the repository to provide XSLT style sheet for the mapping.</td>
</tr>
<tr>
<td>5</td>
<td>Version Compatibility</td>
<td>Partially supported</td>
<td>SRU allows calling application to specify version; however, it is up to service provider to manage the version compatibility.</td>
</tr>
<tr>
<td>6</td>
<td>Multi-purpose Framework</td>
<td>Barely supported</td>
<td>SRU is extendable but with limited purpose.</td>
</tr>
<tr>
<td>7</td>
<td>Platform Independence</td>
<td>Fully supported</td>
<td>SRU can be implemented on any platform.</td>
</tr>
<tr>
<td>8</td>
<td>Discovery Service</td>
<td>Partially supported</td>
<td>SRU can use UDDI, but a calling application needs to integrate a SRU service individually.</td>
</tr>
<tr>
<td>9</td>
<td>Asynchronous</td>
<td>Not supported</td>
<td>SRU has no definition for asynchronous messaging.</td>
</tr>
<tr>
<td>10</td>
<td>Interoperability</td>
<td>Partially supported</td>
<td>SRU has no definition, but can use similar server gateway strategy as SQI and ECL to support other protocols.</td>
</tr>
</tbody>
</table>

#### 5.1.4 SRW Architecture

SRW uses the same messaging protocol as the SRU, but instead of conveying the message through an HTTP request form, SRW literally wraps the XML request message in a SOAP message and sends the SOAP message using HTTP protocol with no specific
binding as required for RPC or document WS. In earlier versions, SRW was a real WS with a SOAP binding and a complete CQL protocol schema but had dropped the requirement in the latest version. Normally, as shown in sample 5.2, the body of the binding SOAP message contains an operation element identified by the operation name and the service URN while in sample 5.1, and the SRW SOAP message does not have any binding information. As a result, a SOAP engine such as Axis cannot process a SRW SOAP message.

To provide SRW service, developers just have to open an HTTP POST portal (i.e., implement a Web application) and manually process the SOAP message. However, the SRW message is simple. The request message is an XML element that holds a list of SRU/SRW parameters. To invoke a SRW service, calling application developers just have to build and insert the SRW request message into a standard SOAP message body. Since SRW does not require “SOAPAction” to appear in the HTTP header, developers can use any URL connection utility (e.g., Java API HttpURLConnection) to send the SOAP message. Note that no requirement for “SOAPAction” in the HTTP header is also another indication that SRW is not RPC or document WS. The response message is not complicated either, as it is also an XML element. Clearly, SRW chooses this simple method for sending its messaging protocol to keep the SRW protocol similar to the SRU and to eliminate the need for a WSDL file. Calling applications can call any SRW service without having to create stub code, and the implementation of a SRW service becomes very simple.
Sample 5.1: A Sample of the SRW Request as Shown on SRW Web site (No Binding SOAP Message)

This is not a RPC or document style SOAP request

```xml
<SOAP:Envelope
xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <!-XML document type (searchRetrieveRequestType)
defined in SRU/SRW type schema -->
    <SRW:searchRetrieveRequest
      xmlns:SRW="http://www.loc.gov/zing/srw/">
      ...
    </SRW:searchRetrieveRequest>
  </SOAP:Body>
</SOAP:Envelope>
```

Sample 5.2: An Example of a SOAP Message with Binding

A typical RPC or Document style SOAP request

```xml
<SOAP:Envelope
xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <ns1:searchRetrieve
      xmlns:ns1="urn:searchRetrieveService">
      <SRW:searchRetrieveRequest
        xmlns:SRW="http://www.loc.gov/zing/srw/">
        ...
      </SRW:searchRetrieveRequest>
    </ns1:searchRetrieve>
  </SOAP:Body>
</SOAP:Envelope>
```

Sample 5.3: A Sample of a SRW Response

This is not a RPC or Document style SOAP response

```xml
<SOAP:Envelope
xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP:Body>
    <!-XML document type (searchRetrieveResponse) defined
in SRU/SRW type schema -->
    <SRW:searchRetrieveResponse ...
    </SRW:searchRetrieveResponse>
  </SOAP:Body>
</SOAP:Envelope>
```
Table 5.6: The Evaluation of the SRW Architecture

<table>
<thead>
<tr>
<th>No</th>
<th>Criteria</th>
<th>Level</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Multi-Query Language</td>
<td>Not supported</td>
<td>SRW is a single query language.</td>
</tr>
<tr>
<td>2</td>
<td>Common Query</td>
<td>Fully supported</td>
<td>By contrast to criterion 1</td>
</tr>
<tr>
<td>3</td>
<td>Query Mapping</td>
<td>Not supported</td>
<td>With respect to criterion 1</td>
</tr>
<tr>
<td>4</td>
<td>LOM Mapping</td>
<td>Partially supported</td>
<td>Although SRW cannot use style sheet, a calling application can set metadata schema, and the service provider can do the transformation before sending the result.</td>
</tr>
<tr>
<td>5</td>
<td>Version Compatibility</td>
<td>Partially supported</td>
<td>Similar to SRU, SRW allows a calling application to specify version; however, it is up to the service provider to manage the version compatibility.</td>
</tr>
<tr>
<td>6</td>
<td>Multi-purpose Framework</td>
<td>Barely supported</td>
<td>SRW is extendable but with limited purpose.</td>
</tr>
<tr>
<td>7</td>
<td>Platform Independence</td>
<td>Fully supported</td>
<td>SRW can be implemented on any platform.</td>
</tr>
<tr>
<td>8</td>
<td>Discovery Service</td>
<td>Partially supported</td>
<td>SRW can use UDDI, but calling application needs to integrate a SRW service individually.</td>
</tr>
<tr>
<td>9</td>
<td>Asynchronous</td>
<td>Not supported</td>
<td>SRW has no definition for asynchronous messaging.</td>
</tr>
<tr>
<td>10</td>
<td>Interoperability</td>
<td>Partially supported</td>
<td>SRW has no definition, but can use similar server gateway strategy as SQI and ECL to support other protocols.</td>
</tr>
</tbody>
</table>

5.1.5 Summary of the Architecture Analysis

The results in Table 5.7 clearly show the superiority of the ECL. Without the support of a middleware infrastructure, other architectures cannot meet the criteria.
required for an evolving and collaborative LOR network. For example, in the LOM
mapping (row 4), SQI, SRU, and SRW have the potential to support LOM mapping, but
without a middleware, LOR developers have to handle metadata mapping themselves
while ECL take care of the mapping behind the scene for the developers.

Table 5.7: The Evaluation Analysis Matrix of the Architecture Supported Features
(FS – Fully Supported, PS – Partially Supported, BS – Barely Supported, NS – Not
Supported, NA – Not Applicable)

<table>
<thead>
<tr>
<th>Support Description</th>
<th>ECL</th>
<th>SQI</th>
<th>OKI</th>
<th>SRU</th>
<th>SRW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Query Languages</td>
<td>FS</td>
<td>FS</td>
<td>NA</td>
<td>NS</td>
<td>NA</td>
</tr>
<tr>
<td>Common Query</td>
<td>FS</td>
<td>BS</td>
<td>FS</td>
<td>FS</td>
<td>FS</td>
</tr>
<tr>
<td>Query Mapping</td>
<td>FS</td>
<td>BS</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>LOM Mapping</td>
<td>FS</td>
<td>PS</td>
<td>NA</td>
<td>PS</td>
<td>PS</td>
</tr>
<tr>
<td>Version Compatibility</td>
<td>FS</td>
<td>NS</td>
<td>NA</td>
<td>PS</td>
<td>PS</td>
</tr>
<tr>
<td>Multi-purpose Framework</td>
<td>FS</td>
<td>NS</td>
<td>BS</td>
<td>BS</td>
<td>BS</td>
</tr>
<tr>
<td>Platform Independent</td>
<td>FS</td>
<td>FS</td>
<td>BS</td>
<td>BS</td>
<td>BS</td>
</tr>
<tr>
<td>Discovery Service</td>
<td>FS</td>
<td>PS</td>
<td>NA</td>
<td>PS</td>
<td>PS</td>
</tr>
<tr>
<td>Asynchronous</td>
<td>FS</td>
<td>FS</td>
<td>NA</td>
<td>NS</td>
<td>NS</td>
</tr>
<tr>
<td>Interoperability</td>
<td>FS</td>
<td>PS</td>
<td>BS</td>
<td>PS</td>
<td>PS</td>
</tr>
</tbody>
</table>

5.2 Integration Requirements Analysis

This section examines the integration requirements of each protocol infrastructure
and attempts to make a comparison based on the real life experience of the ECL
integration with other protocols. This analysis only considers the integration of the search
service, which is common to all the comparing protocol infrastructures.

To access ECL LOR network, calling application developers only have to
integrate the ECL client API. Developers need to know how to query the ECL UDDI
registry, build the query, and process the results. Let’s assume that developers know how
to process XML documents such as building search requests and processing results. What
is the level of difficulty for developers to integrate, adapt, update, and maintain a protocol
service? We select the four criteria based on the integration experiences of ECL with the
comparing protocol infrastructures: integration, adaptation, update, and maintenance. We
conduct the analysis using the same method as in the architecture analysis, and assume
that each LOR has a Web server, and the maintenance for a Web application or a WS is
easy. Overall Integration considers the overall difficulty for a service provider as well as
for a calling application to integrate the evaluating protocol infrastructure. Protocol
update focuses on the requirements for updating the implementation of the evaluating
protocol. Adaptation refers to the effort of an existing system using another protocol
standard to adapt and use the evaluating protocol. Finally, maintenance considers the
need to maintain the evaluating protocol service. These criteria demonstrate the flexibility
of the XML-based messaging middleware, which eases the integration and the
maintenance of a ECL service. The analysis uses four levels of difficulty: easy, moderate,
difficult, very difficult, as well as not applicable. The design objective of the ECL is a
simple integration, and the integration workshop as well as the integration of the ECL
with other protocols reflects the simplicity of the ECL. Therefore, the levels of
difficulties of the ECL integration for all the above criteria should be easy.

5.2.1 SQI Integration

If using FIRE, the integration of the SQI is quite simple. However, it is still more
complex than the ECL due to the asynchronous nature of FIRE. The ECL/FIRE
integration of a service provider, the “target”, requires the implementation of four Java
classes: SQITarget, FireResults, FireQuery, and FireSession. The integration for the
calling application, the “source”, needs a similar amount of work. Both the service provider and the calling application are JMS clients. Hence, the calling application also has to keep an open connection to the JMS provider. Therefore, the maintenance of FIRE is difficult for an existing LOR hosted on a Web server.

However, if ECL developers were to implement SQI/ARIADNE based the SQI WSDL file, the task would be much harder. First, developers must be able to understand the SQI Web service. Let’s assume that a developer only wants to implement the basic search service for SQI, the synchronous-query search service. The developer must implement a minimum of six methods: setQueryLanguage, setMaxQueryResults, setResultsFormat, setResultsSetSize, synchronousQuery, and getTotalResultsCount. The developer must configure the SOAP engine, construct and test a WSDL file. The developer must also understand input parameters and construct a valid response for each operation. In addition, the developer must handle unsupported operations. At calling application side, developers must generate stub code for each service, determine what query language and metadata format the service provider is using, and know how to process SQI messages.
Table 5.8: The Evaluation of the SQI Integration Analysis

<table>
<thead>
<tr>
<th>No</th>
<th>Criteria</th>
<th>Level</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overall Integration</td>
<td>Difficult</td>
<td>Due to the reason above for integration SQI/ARIADNE, the integration of SQI is difficult. Integration of FIRE is simpler, but FIRE mostly targets distributed networks. A calling application requires the same level of difficulty for a service provider to integrate with FIRE infrastructure.</td>
</tr>
<tr>
<td>2</td>
<td>Protocol Update</td>
<td>Difficult</td>
<td>SQI has no strategy for updates. Based on analysis of WS technologies, SQI/ARIADNE requires a widespread synchronization update. Similarly, with FIRE, all JMS clients must communicate with the JMS provider and therefore must use the same JMS software as well as the SQI protocol.</td>
</tr>
<tr>
<td>3</td>
<td>Adaptation</td>
<td>Moderate</td>
<td>There are some problems with interoperability, but SQI allows repositories to use their query languages and their metadata formats. This may shift the problem to the calling application side, but if SQI follows its architecture, SQI would supply the SQI wrapper for data mapping.</td>
</tr>
<tr>
<td>4</td>
<td>Maintenance</td>
<td>Easy</td>
<td>If the SQI protocol is stable, the maintenance SQI/ARIADNE is simple. It is a Web service. On the other hand, FIRE builds on JMS, which requires more maintenance.</td>
</tr>
</tbody>
</table>

5.2.2 OKI Integration

It may be unfair to analyze an integration of the OKI/OSID since OKI/OSID is merely interface definitions. On the other hand, OKI/OSID is a competing infrastructure for the LOR network interoperability. The analysis of the OKI/OSID should reveal the complexity of the OKI/OSID, and it is up to the reader to evaluate if the cost of the OKI/OSID implementation is worth the effort. This analysis assumes that the
implementation of the OKI/OSID has a supporting infrastructure such as the ECL to implement OSID functions.

OKI claims that OSID enables the development of calling applications independently from the service development environment and that OSID would ease the service integration. However, the reality is different. It is impossible for OKI to define every possible interface function, present or future. As a result, OKI permits out-of-band agreement definitions. However, the out-of-band agreement definitions can compromise the OKI/OSID interoperability. OKI/OSID has to limit the number of out-of-band agreement definitions. Therefore, only a limited number of organizations get the permission to add out-of-band agreement definitions. Of course, anyone can work on new out-of-band agreement definitions, but there is no guarantee that OKI will promote the new definitions.

In addition, some developers have already found that OKI/OSID has too many functions. Consequently, OSID permits providers to implement only the interface functions that are applicable to their services. This is where the problem begins. Unsupported functions are an issue for application developers. It is naïve to believe that the application developers can do the integration without the support of the service providers. Application developers must know what the support interface functions are. Therefore, the integration of the OKI/OSID with the calling application depends on the implementation of the OKI/OSID. Implementing OKI/OSID is even more complex. Developers must understand the OKI/OSID semantics. In addition, they must deal with the ambiguity of the definitions as well as with the functions and types that have no definitions. For example, ECL had to define LOM record and part types for IEEE LOM
as out-of-band agreement definitions. Therefore, ECL has to communicate this out-of-band agreement to the calling application developers. Normally, it is the responsibility of the OKI to promote new out-of-band agreement definitions so that everybody using OKI uses the same out-of-band agreement definitions such as IEEE LOM record and part types. However, OKI is not in a hurry to accept new out-of-band agreement definitions. Meanwhile, other developers may also have developed their out-of-band agreement definitions for IEEE LOM. This becomes a huge problem with interoperability.

Clearly, the integration of OKI/OSID is the most difficult of all the protocols. Fortunately, OKI/OSID is quite stable. Over the last three years, OKI had made one major change. Updating OKI software would require a reimplementation of the OSID.

Table 5.9: The Evaluation of the OKI Integration Analysis

<table>
<thead>
<tr>
<th>No</th>
<th>Criteria</th>
<th>Level</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overall Integration</td>
<td>Very difficult</td>
<td>A complete implementation of the OKI/OSID is definitely very difficult.</td>
</tr>
<tr>
<td>2</td>
<td>Protocol Update</td>
<td>Very difficult</td>
<td>This requires a reimplementation of the OKI/OSID. Hence, it is very difficult.</td>
</tr>
<tr>
<td>3</td>
<td>Adaptation</td>
<td>Very difficult</td>
<td>The usage of OKI/OSID requires extensive understanding of OSID. Applications using other standards will have difficulty adapting.</td>
</tr>
<tr>
<td>4</td>
<td>Maintenance</td>
<td>Not applicable</td>
<td>It is only interface functions. The service maintenance depends on the underlying system that supports the implementation.</td>
</tr>
</tbody>
</table>

The analysis of OKI/OSID integration reveals high levels of difficulty in all aspects. OKI highlights the stability of OSID. Once an application has been integrated with OSID, changes do not affect the application. This is not always true. For example, if the old interface using MIT LOM record XML returns an XML string that represents a
DC LOM record, the calling application would break if the new interface returns an XML string that represents an IEEE LOM record.

5.2.3 SRU Integration

SRU is a Web application, and a SRU request is a HTTP form request using the GET method. A calling application can construct a SRU request as a URL by composing the parameters as URL arguments using question mark and ampersand (e.g.,
http://z3950.loc.gov:7090/voyager?version=1.1&operation=searchRetrieve&query=dinosaur). Since some end-users may already know how to build a CQL query string, calling applications may not need to do extra work to educate the end-users. Simple examples would be sufficient to allow the end-users to search a SRU repository. SRU returns a response message in an XML string format. What are the requirements to implement a SRU service? The context of this analysis assumes that repositories are some kinds of Web applications. Implementing SRU is purely a process of adding another Web application that can digest SRU request parameters and return results based on the SRU XML Schema. Repositories that choose SRU are probably archive repositories such as libraries or some publishing institutions. They are probably capable of processing CQL query string. In the worse case, LOR developers have to parse CQL query string before they can use it in their query system, but this is not a big task to accomplish.

However, allowing a calling application to choose the format of metadata standard, the protocol version, and the style sheet for metadata conversion implies that the calling application must know beforehand if a repository supports the chosen values. Automatic discovery is not possible in this case. To support service discovery, SRU
needs to add a new operation to SRU service to allow a calling application to determine the version, the LOM standard, or the available style sheets.

Table 5.10: The Evaluation of the OKI Integration Analysis

<table>
<thead>
<tr>
<th>No</th>
<th>Criteria</th>
<th>Level</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Overall Integration</td>
<td>Moderate</td>
<td>Although there are some considerations such as individual integration of SRU service in the calling application, the simplicity of the protocol makes the protocol easy to implement and quite easy to integrate.</td>
</tr>
<tr>
<td>2</td>
<td>Protocol Update</td>
<td>Difficult</td>
<td>SRU service providers can support different versions. However, updating the protocol does require widespread update and reimplementation.</td>
</tr>
<tr>
<td>3</td>
<td>Adaptation</td>
<td>Moderate</td>
<td>SRU uses XML. With a little effort, applications using other protocol standards, such as the query language and metadata format, can use SRU.</td>
</tr>
<tr>
<td>4</td>
<td>Maintenance</td>
<td>Easy</td>
<td>SRU is a Web application. Hence, it should be easy to maintain.</td>
</tr>
</tbody>
</table>

5.2.4 SRW Integration

SRW has the same characteristics as the SRU. The only difference is the message format sent over HTTP: SRU uses the standard HTTP multipart/form-data encoding; and SRW uses the standard SOAP message. This means that developers must handle the SOAP message. Calling application developers have to build a SRW request XML message and insert it into a SOAP message, and when they receive the message, they have to extract the SRW response from the SOAP message. A similar process, but in reverse order, happens on the service provider side. SRW is slightly more difficult to integrate than the SRU, but it is not significant enough to change the evaluation.
5.2.5 Summary of the Integration Analysis

Table 5.11 lists the evaluation summaries of the integration analysis. Although the SRU/SRW protocol does get the best evaluations, the simplicity of the protocol deserves some consideration. Of course, with a middleware infrastructure, ECL is easy to use while other protocols demand more effort from developers to do the integrations. For example, the overall integration of the SQL protocol is difficult. Implementing an SQI service from a WSDL file is not a simple task. Developers must understand SQI protocol syntactically and semantically. It is more complicated than implementing an SRU or SRW service where the protocol is simpler and does not require the WS infrastructure support. The integration of OKI/OSID is very difficult. Developers must have an underlying infrastructure (e.g. the ECL) in order to implement the OKI/OSID, and they have to learn and make numerous decisions regarding the interface they want to implement. In addition, they have to deal with undefined functions and types, which need the backing and participation from the OKI/OSID developers.

Table 5.11: The Evaluation Analysis Matrix of the Integration Criteria

<table>
<thead>
<tr>
<th>Criteria</th>
<th>ECL</th>
<th>SQI</th>
<th>OKI</th>
<th>SRU</th>
<th>SRW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall Integration</td>
<td>Easy</td>
<td>Difficult</td>
<td>Very difficult</td>
<td>Moderate</td>
<td>Moderate</td>
</tr>
<tr>
<td>Protocol Update</td>
<td>Easy</td>
<td>Difficult</td>
<td>Very difficult</td>
<td>Difficult</td>
<td>Difficult</td>
</tr>
<tr>
<td>Adaptation</td>
<td>Easy</td>
<td>Moderate</td>
<td>Very difficult</td>
<td>Moderate</td>
<td>Moderate</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Easy</td>
<td>Easy</td>
<td>Not applicable</td>
<td>Easy</td>
<td>Easy</td>
</tr>
</tbody>
</table>
5.3 Interoperability Comparison Analysis

According to the IEEE Standard Computer Dictionary, interoperability is the ability of two systems or components to exchange information and use the exchanged information. Hence, the key to interoperability is the ability of the protocol infrastructure to map from and to a foreign protocol. The problem with mapping is the loss of information. Mapping IEEE LOM to DC LOM can lose more than 30 percent of the data. On the other hand, when mapping from DC LOM to IEEE LOM, the loss of data is practically zero. Based on this observation, an estimation of interoperability between two protocols is possible.

When estimating interoperability, other factors come into consideration. One is significant and insignificant loss of data. The loss of data when mapping IEEE LOM to DC may be insignificant because DC users do not use IEEE LOM extra data. However, in some cases, when mapping IEEE LOM and DC, there is some significant loss of data. For example, IEEE LOM has two sources (identifier and location) to map to DC identifier. Normally, the IEEE LOM location is mapped to the DC identifier. If a repository needs an IEEE LOM identifier to retrieve a LO, DC users do not have the IEEE LOM identifier to access the LO. Another problem is the different format. An IEEE identifier has two values: catalogue and entry. In this case, the interoperability requires an agreement between DC users (calling application) and IEEE repository (service provider) to create a new format for the identifier (e.g., a composite identifier using the following form: “catalog:entry”). This example illustrates the issue of mapping and interoperability. Despite the effort, avoiding the loss of significant data can be difficult in some contexts. Similarly, there is no loss of data when mapping from DC LOM to IEEE
LOM, but this does not mean 100 percent interoperability. If an application were using IEEE LOM identifiers for some purposes, the mapped IEEE LOM from DC would not have identifiers. How would the application deal with this lack of identifiers?

The following provides a brief analysis of interoperability. The analysis considers the interoperability between two protocols in one direction at a time and only considers Web-based protocols such as ECL, SQI/ARIADNE (version 1.0), and SRU/SRW (version 1.1), and to see the superiority of the XML-based messaging middleware, readers must consider the strategies ECL use to improve the level of interoperability.

5.3.1 The Interoperability between ECL and Other Protocols

This interoperability analysis considers the case where calling application uses the ECL to connect to SQI/ARIADNE or SRU/SRW.

- **ECL-calls-SQI** – Mapping ECL search request to SQI search request may lose some data. However, the loss of query data is insignificant since ECL can apply an additional domain-specific filter on the SQI response (e.g.,. The loss of the query data produces imprecise query results, but by applying an additional query filter, the results become precise and meet the query criteria. Hence, the loss of query data is insignificant). An area of concern is the text search. In the current state, neither ECL nor SQI have guidelines for the text search. ECL recommends using Google-style query syntax. SQI appears to be leaning toward CQL. Hence, due to the lack of search compatibility, the query may lose some precision. For the response from SQI, there is not loss of data. Both ECL and SQI use IEEE LOM.

- **ECL-calls-SRU/SRW** – Most SRU/SRW repositories use DC metadata format, which is reasonable since most of these repositories are archive repositories. There is practically no loss of data when mapping the SRU/SRW response to the ECL response. Currently, there is no mapping available for
mapping Google-style text query to CQL, and the mapping is only applicable to an ECL text search. ECL needs to apply a domain-specific filter on the search results to compensate for the loss of query data. If ECL were to provide SRU/SRW interoperability, ECL has to integrate SRU/SRW service individually. ECL gateway strategy is handy for this type of integration. Using the UDDI registry to supply information about the version, the level of CQL, and the metadata schema, ECL can supply a generic SRU/SRW gateway with a query mapping utility to allow a calling application to invoke any SRU/SRW service the same way the calling application would invoke an ECL search service. Nevertheless, ECL needs some additional work to achieve full interoperability. Currently, there is some loss of data in the query, and there is a need for handling metadata mapping such as IEEE LOM identifiers.

5.3.2 The Interoperability of SQI

Since SQI/ARIADNE is not a middleware infrastructure, it is the responsibility of the calling application of SQI/ARIADNE to interoperate with the ECL. It is not a good strategy for ECL service providers to provide SQI services. A better strategy would be to package SQI mapping in ECL middleware infrastructure and give it to the SQI calling applications. ECL transforms SQI query requests to ECL query requests, invokes ECL services, and transforms ECL responses to SQI responses.

How would a calling application using SQI interoperate with a SRU/SRW service? There are two possibilities. One is to build a utility for the calling applications to invoke SRU/SRW services using a client gateway strategy. Another solution is to use a server gateway strategy and implement SQI server gateway services that relay the requests to SRU/SRW services and return SQI responses. What would be the interoperability factors between SQI and ECL and between SQI and SRU/SRW?
- SQI-calls-ECL – Mapping SQI query to ECL query represents a similar problem to mapping ECL query to SQI query. Although the ECL common query can accommodate all the SQI query entries, the unresolved issue of the text search faces the same problem as converting ECL query to SQI query.

- SQI-calls-SRUISRW – There is a definite loss of data when mapping SQI query to SRU/SRW query. The lost of query data can be significant if a domain-specific query is important (e.g., typical age range or language). As mentioned above, mapping DC of SRU/SRW to IEEE LOM of SQI may need some work to ensure full interoperability.

5.3.3 The Interoperability of SRU/SRW

SRU/SRW faces a similar problem as SQI. However, SRU/SRW only has a text query, which is a common feature for all metadata query languages. Therefore, it has a greater chance to interoperate with other protocols.

- SRU/SRW-calls-ECL – The problem raised earlier about the identifier can be an issue. The ECL repository may choose to deliver LOs via a request/deliver service, which would require additional effort to make SRU/SRW interoperable with the ECL.

- SRU/SRW-calls-SQI – Since SQI does not use a request/deliver service, the location value in the metadata is a valid URL, and it is a direct mapping for the DC identifier. Hence, the loss of data in the metadata mapping may be insignificant. The mapping metadata query needs some work for mapping CQL/SRU to SQI keyword search.

---

8 This analysis is based on SQI/FIRE query schema.
5.3.4 Summary of the Interoperability Analysis

Table 5.12 provides a summary of the work required to make a protocol infrastructure interoperate with another protocol.

Table 5.12: The Evaluation Analysis Matrix of the Interoperability Requirements
The levels of requirements: No, Minor, Some, and Major.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Query Interoperability requirements</th>
<th>Metadata Interoperability requirements</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECL-calls-SQI</td>
<td>Minor work on text search mapping</td>
<td>No work</td>
</tr>
<tr>
<td>ECL-calls-SRU/SRW</td>
<td>Minor work on text search mapping</td>
<td>Minor work on the metadata mapping</td>
</tr>
<tr>
<td>SQI-calls-ECL</td>
<td>Minor work on text search mapping</td>
<td>No work</td>
</tr>
<tr>
<td>SQI-calls-SRU/SRW</td>
<td>Minor work on text search mapping</td>
<td>Minor work on the metadata mapping</td>
</tr>
<tr>
<td></td>
<td>Major work on domain-specific query</td>
<td></td>
</tr>
<tr>
<td>SRU/SRW-calls-ECL</td>
<td>Minor work on text search mapping</td>
<td>Some work on the metadata mapping</td>
</tr>
<tr>
<td>SRU/SRW-calls-SQI</td>
<td>Minor work on text search mapping</td>
<td>Minor work on the metadata mapping</td>
</tr>
</tbody>
</table>

5.4 Discussion

XML-based messaging is an obvious choice for a communication infrastructure protocol for an evolving collaborative network such a LOR network. By coupling with a middleware and a UDDI registry, ECL takes full advantage of the XML-based messaging to provide an integrated infrastructure that is adaptive, extendable, and maintainable. SQI and SRU/SRW follow a similar footstep as the ECL. Both use XML-based messaging protocols. The only difference from the ECL is the way in which they pass the message. ECL passes the message as an XML string over a predefined RPC WS that are
deployable and accessible using a SOAP utility without any configuration or implementation. Meanwhile, SQI/ARIADNE uses document style WS and SRU/SRW merely sends the message directly over HTTP. Both SQI/ARIADNE and SRU/SRW choose to provide semi-independent LOR, and care very little about the calling applications. Developers must integrate each SQI/ARIADNE or SRU/SRW individually into their calling applications. In contrast, OKI/OSID mostly concentrates on providing a stable set of interface functions for the calling application developers. In a developing and evolving environment, OKI/OSID faces many unresolved issues, which make OKI/OSID a difficult infrastructure to use.

Based on the criteria required for an evolving collaborative environment, ECL clearly comes ahead of all other protocols. It equips developers with utilities and strategies and abstracts the complexity of the technologies and the protocol standards to ease the integration of the ECL into the calling application as well as into the service provider. The protocol is generic, and is capable of supporting other types of communications. In addition, ECL has developed strategies to interoperate with other protocols. The gateway strategy of ECL enables ECL clients to invoke services such as SQI/ARIADNE or SRU/SRW services as if the services are ECL services. In comparison, a calling application is responsible for integrating each SQI/ARIADNE or SRU/SRW service individually.

There is a stereotype that favours the WS technologies against the middleware. Some researchers believe that WS are stable and reliable, and everybody knows how to implement and invoke a Web service from a WSDL file. The review and analysis in Chapter 2 shows that there are some complications when implementing a Web service

142
from a WSDL file. In addition, the Web service must be stable and well defined, and this is not the case in an evolving collaborative network. SRU/SRW has dropped conventional WS standards in favour of the XML-based messaging over HTTP. Clearly, this makes the SRU/SRW protocol more attractive, easier to integrate, and expandable. On the other hand, SQI/FIRE solves the integration problem by implementing SQI on JMS framework, which is typically a middleware. Obviously, when communication infrastructure developers want to provide better assistance to LOR developers, they must consider middleware technology or some kind of utility libraries. This shows that the XML-based messaging middleware infrastructure of the ECL is an excellent candidate for a LOR network.

Apart from providing flexibility and easy integration, the XML-based messaging middleware technology enables ECL to develop maintainability and message mapping strategies for interoperability within ECL implementations (i.e. vertical compatibility), as well as with other protocol infrastructures. Combined with the UDDI registry, the ECL infrastructure is able to feed the information on new software and protocol standards to the calling applications using the ECL. This characteristic is different from the software updates where software distributors have full control over the update. The calling application is attached to the ECL infrastructure, and its implementation is based on the API provided by the ECL. The integrated application knows how to build a particular format of input request to invoke an ECL service, and expects a particular type of response. Hence, updating a middleware implementation is more complex. The update involves message mapping. An older ECL implementation must be able to map the old
ECL message to the new ECL message used by a service provider. The mapping strategy becomes handy when trying to interoperate the ECL with other protocol infrastructures.

However, readers should not lose sight of other factors that influence the adoption of an infrastructure protocol. The best technical solution does not guarantee success. In the last three years, two forces acted against the adoption of the ECL infrastructure. First is the lack of industrial backing. ECL had become merely a research project. Several eduSource partners (CanLOM, CAREO, and Athabasca) lost their funding, and ceased to provide ECL services. In contrast, SQI has very strong support from the European Commission. SRU/SRW has the backing of the Library of Congress. EdNA, in which the infrastructure protocol is proprietary, has very strong support from Australia educational institutions and government. Similarly, SMETE has the backing of the National Science Foundation. Second, there is resistance to the dependency on the LOR network. In the early days of ECL development, a partner from Athabasca University strongly argued against the middleware infrastructure. He agreed that using ECL is very simple to connect a LOR to the eduSource network but still preferred to have a WSDL definition as guidelines for him to implement services that enable access to Athabasca LOR for the eduSource members. Clearly, the developer wants to have control of the WS and be independent from the eduSource network.

When joining a collaborative network, retaining individualism may be difficult. LOR developers must realize that they provide a service to the clients, and the clients should be able to use an API to invoke any service on the network. Individual integration of each LOR service is not scalable. Therefore, a LOR network needs some type of
coordinated infrastructure such as the ECL infrastructure to ensure the interoperability throughout the network lifecycle.

5.5 Guidelines for Developing LOR Communication Infrastructure Protocol

We draw on the finding and experience gathered over the past three years to provide some guidelines and recommendations for choosing, as well as for developing a communication infrastructure protocol for a LOR network.

5.5.1 Choosing a Suitable Protocol Infrastructure

When choosing an infrastructure protocol for a LOR network, developers should select the protocol based on the requirements of their LOR network.

Obviously, ECL is an infrastructure for an evolving collaborative LOR network for any type of LORs (i.e., small, medium, and large). ECL enables the sharing of LOs while assisting in the development of learning technologies. The ECL infrastructure is generic, can accommodate other types of services, and equips with a UDDI registry for service discovery and a multi-level security infrastructure.

SQVFIRE is a good candidate for a peer-to-peer type LOR network. It does not require a dedicated server for a LOR, which can be located behind a firewall. LORs (target) and calling applications (source) participate equally in the network and must maintain an open connection to a JMS provider. Hence, the protocol is suitable for small LORs that want to participate in a peer-to-peer network.

For a public or semi-public LOR network that consists of independent LORs where LOR developers only want to provide a common search service to access LOR
resources, the suitable protocols are SRU/SRW or SQI-WS of ARIADNE type. SRU/SRW protocol is easy to implement and is a better choice if CQL search is sufficient for all LORs in the network. Archive repositories such as libraries that already support CQL for search should select the SRU/SRW protocol. SQI-WS may be better suited for a LOR network that needs an advanced search support.

An ideal network for OKI/OSID is a homogenous LOR network. Obviously, calling applications should have only a single OKI/OSID implementation to deal with. The goal and purpose of the OSID is to provide simple and stable interface functions for calling applications. Therefore, it does not make sense to present multiple OKI/OSID implementations to the calling application developers. Consequently, OKI/OSID needs an underlying infrastructure protocol that can unify heterogeneous LORs or a homogenous LOR network.

### 5.5.2 Implementation Guidelines

*Separating text query from domain-specific query* – the strategy reduces the complexity of the metadata query and allows communication protocol developers to resolve the problem with text query before focusing on the domain specific query. For example, text query must deal with two issues: language and text search syntax. The language used on the application operating system can be the default language for text query, and the syntax can be Google, Lucene, or CQL text search syntax.

*Providing domain-specific query filter* – obviously, LORs cannot adapt quickly enough to support new requirements of a domain-specific query. A protocol
infrastructure such the ECL can apply the domain-specific query filter on the client side on the query results before presenting them to the end-users.

*Grouping LOM records and providing service based on LOM classifications* – in a large network, this strategy allows targeting search, which reduces Internet traffic and speeds up the response time for a search query. For example, instead of providing a service that supports multiple domains of knowledge (e.g., mathematics, biology, chemistry, etc.), it is preferable to provide services for each domain.

*Building a single API for accessing a LOR network* – Ideally, calling applications should only have to deal with a single connector. Currently, calling applications have to deal with each SQI/ARIADNE or SRU/SRW service individually. This is not a good strategy. A better strategy would be to provide an API that can access every SQI or SRU/SRW service.

*Using largest definition standard for common protocol* – always try to use a largest definition of standard if the size of data is not important and loss of data is important. For example, IEEE LOM is currently a good candidate for common metadata. IEEE LOM is the largest metadata definition among all mainstream metadata standards. Mapping from a smaller set of metadata definitions such as DC or IMS LOM to a larger set of metadata definitions of IEEE LOM does not lose any data. Therefore, a LOR network should develop a master LOM definition that can capture all the metadata definitions. However, calling application developers should abstract the unused definitions from the end-users.
CHAPTER 6: CONCLUSION

This research thesis has demonstrated that the LOR network is a collaborative and evolving environment, and the reusability of LOs is the fundamental driving idea behind the LOR network. Therefore, the need for a flexible but stable communication infrastructure is imminent to ensure the ongoing development of learning technologies.

Throughout the literature review, the design and implementation, the integration of the ECL infrastructure, and the comparative analysis, this thesis has shown the constraints of the WS technologies and the advantages of the XML-based messaging middleware infrastructure protocol. The current WS technologies are not tolerant of change. Certain flexibility is achievable through the WS composition techniques of the BPEL4WS and the semantic WS, but the two technologies still need a well-defined WS. This is a constraint to an evolving environment. XML-based messaging middleware infrastructure offers better flexibility and enables the development of a framework and strategies to deal with issues encountered in an evolving collaborative network. The use of multiple metadata query languages, LOM formats, and infrastructure implementation versions are parts of the issues that the ECL infrastructure has resolved. In addition, the ECL infrastructure has successfully dealt with the interoperability with other protocol infrastructures using the gateway strategies.

The advantages of XML-based messaging are also reflected in the ECL infrastructure design. The infrastructure consists of composite modules. Each module performs specific tasks and has specific interface functions. In an evolving environment,
switching technologies and adding new requirements and specifications is frequent. A modular design of the ECL infrastructure enables ECL to switch implemented modules or add new modules without affecting the network interoperability. Since the first release of the ECL, there were two planned upgrades. The first upgrade was the addition of the UDDI registry, and the second upgrade was the integration of the ECL security infrastructure. The upgrade was possible without having to redesign the whole ECL architecture.

In conclusion, this research provides a better understanding of the LOR network research and technologies and demonstrates the need for a flexible communication infrastructure protocol. Finally, it provides guidelines for choosing a suitable communication infrastructure for a particular type of LOR network, as well as guidelines for improving the infrastructure design and implementation.

6.1 Future Work

To be successful, a LOR network initiative may need to present an integrated environment to the learning designers to reuse and develop LOs. Integrating ECL into an environment such as a learning design editor is a major task, but it would provide an optimal test for the ECL infrastructure. Undoubtedly, the integration would reveal the need for additional requirements and specifications. This allows the work on the ECL to continue to strengthen the ECL design and implementation.

The next research direction will focus on security, in particular the fabric of trust. Sharing reusable LOs requires dealing with different types of security. What type of security policy as well as security infrastructure should be in place to ease the exchange
of LOs? What are the conditions that make two institutions willing to share their resources with each other? Is it possible to design a trust model for the entire network? This type of research expands the usage and allows the refinements of the ECL design.

Finally, there are several research directions that can improve a LO management system. First is the research on automatic metadata generation based on the information collected in an integrated environment such as the learning design editor and course management system. This research would attempt to capture the learning context of a LO to provide better descriptions and improve the quality of LOM records. This research is an amendment to the current research into automatic metadata generation. The second research direction would concentrate on building a system for grouping and classifying a collection of LOM records. For example, the system can look into a LOM repository collection and group the LOM records into the appropriate domains of knowledge. Similarly, the system would know in what domain of knowledge a newly built LOM record should be.
Appendix A

The XML Schemas of the ECL Messaging Protocols

A protocol is a set of formal rules that prescribe the format and transmission of data across the network. Since ECL is XML-based messaging protocol, the XML Schema is a perfect fit for the ECL messaging protocols. The schema describes the protocol in an open format that is human-readable, self-describing, and self-validating. Appendix A is a listing of the current ECL messaging protocols represented in XML Schema definitions.

Appendix A1: The XML Schema of the ECL Main Messaging Protocol

```xml
<xs:schema xmlns="http://www.edusource.ca/xsd/ecl_v1p1"
    targetNamespace="http://www.edusource.ca/xsd/ecl_v1p1"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    elementFormDefault="qualified" version="ECL 0.5">
    <xs:element name="ecl" type="eclType" />
    <xs:element name="header" type="headerType"/>
    <xs:element name="type" type="xs:string"/>
    <xs:element name="cid" type="xs:string"/>
    <xs:element name="protocol" type="xs:string"/>
    <xs:element name="version" type="xs:string"/>
    <xs:element name="receiver" type="xs:string"/>
    <xs:element name="sender" type="senderType"/>
    <xs:element name="mode" type="xs:string"/>
    <xs:element name="endpoint" type="xs:string"/>
    <xs:element name="body" type="bodyType"/>
    <xs:element name="payload" type="payloadType"/>
</xs:schema>
```
Appendix A2a: The XML Schema of the ECL Search Request Protocol

```xml
<xs:schema xmlns="http://www.edusource.ca/xsd/ecl_search_v1p1"
    targetNamespace="http://www.edusource.ca/xsd/ecl_search_v1p1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    elementFormDefault="qualified" version="ECL 0.5">

    <xs:element name="search" type="searchType" />
```
<xs:element name="pattern" type="patternType" />
<xs:element name="language" type="xs:language" />
<xs:element name="basic" type="singleType" />
<xs:element name="advanced" type="singleType" />
<xs:element name="or" type="compositeType" />
<xs:element name="and" type="compositeType" />
<xs:element name="complex" type="complexType" />
<xs:element name="simple" type="simpleType" />

<xs:complexType name="patternType">
  <xs:annotation>
    <xs:documentation>
      0 - full IEEE LOM; 1 - summary IEEE LOM (title, description, location, format)
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:enumeration value="0" />
    <xs:enumeration value="1" />
  </xs:restriction>
</xs:complexType>

<xs:complexType name="searchType">
  <xs:annotation>
    <xs:documentation>
      ECL Search Protocol
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element ref="pattern" minOccurs="1" maxOccurs="1" />
    <xs:element ref="language" minOccurs="1" maxOccurs="1" />
    <xs:element ref="basic" minOccurs="1" maxOccurs="1" />
    <xs:element ref="advanced" maxOccurs="1" />
    <xs:element name="resumptionToken" type="resumptionTokenType" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="singleType">
  <xs:choice>
    <xs:element name="or" minOccurs="1" maxOccurs="1" />
    <xs:element name="and" minOccurs="1" maxOccurs="1" />
    <xs:element name="complex" minOccurs="1" maxOccurs="1" />
    <xs:element name="simple" minOccurs="1" maxOccurs="1" />
  </xs:choice>
</xs:complexType>

<xs:complexType name="CompositeType">
  <xs:sequence>
    <xs:choice>
      <xs:element name="or" minOccurs="1" maxOccurs="1" />
      <xs:element name="and" minOccurs="1" maxOccurs="1" />
      <xs:element name="complex" minOccurs="1" maxOccurs="1" />
      <xs:element name="simple" minOccurs="1" maxOccurs="1" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>
<xs:choice>
  <xs:element name="or" minOccurs="1" maxOccurs="1"/>
  <xs:element name="and" minOccurs="1" maxOccurs="1"/>
  <xs:element name="complex" minOccurs="1" maxOccurs="1"/>
  <xs:element name="simple" minOccurs="1" maxOccurs="1"/>
</xs:choice>
</xs:sequence>
</xs:complexType>

<xs:complexType name="resumptionTokenType">
  <xs:sequence>
    <xs:element name="resumptionTokenId" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="cursor" type="xs:Integer" minOccurs="1" maxOccurs="1"/>
    <xs:element name="completeListSize" type="xs:Integer" minOccurs="1" maxOccurs="1"/>
    <xs:element name="expirationDate" type="xs:dateTime" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="compType">
  <xs:attribute name="xpath" type="xs:string"/>
  <xs:choice>
    <xs:element name="or" minOccurs="1" maxOccurs="1"/>
    <xs:element name="and" minOccurs="1" maxOccurs="1"/>
    <xs:element name="complex" minOccurs="1" maxOccurs="1"/>
    <xs:element name="simple" minOccurs="1" maxOccurs="1"/>
  </xs:choice>
</xs:complexType>

<xs:complexType name="simpleType">
  <xs:simpleContent>
    <xs:restriction base="xs:token">
      <xs:enumeration value="eq"/>
      <xs:enumeration value="lt"/>
      <xs:enumeration value="gt"/>
      <xs:enumeration value="contains"/>
      <xs:enumeration value="not"/>
    </xs:restriction>
  </xs:simpleContent>
</xs:complexType>
</xs:schema>
Appendix A2b: The XML Schema of the ECL Search Expose Protocol

```xml
<xs:schema xmlns="http://www.edusource.ca/xsd/ecl_search_ex_v1pl1"
    targetNamespace="http://www.edusource.ca/xsd/ecl_search_ex_v1pl1"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    elementFormDefault="qualified" version="ECL 0.5">
    <xs:import namespace="http://ltsc.ieee.org/xsd/LOM"
        schemaLocation="http://ecl.iat.sfu.ca/lom.xsd" />

    <xs:element name="search_expose" type="exposeType" />

    <xs:complexType name="exposeType">
        <xs:attribute name="advanced" type="xs:string" />
        <xs:sequence>
            <xs:element ref="lom" maxOccurs="unbounded" />
            <xs:element name="resumptionToken"
                type="resumptionTokenType" maxOccurs="1" />
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="resumptionTokenType">
        <xs:sequence>
            <xs:element name="resumptionTokenId" type="xs:string"
                minOccurs="1" maxOccurs="1" />
            <xs:element name="cursor" type="xs:integer"
                minOccurs="1" maxOccurs="1" />
            <xs:element name="completeListSize" type="xs:integer"
                minOccurs="1" maxOccurs="1" />
            <xs:element name="expirationDate" type="xs:dateTime"
                minOccurs="1" maxOccurs="1" />
        </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Appendix A3a: The XML Schema of the ECL Gather Request Protocol

```xml
<xs:schema xmlns="http://www.edusource.ca/xsd/ecl_gather_v1pl1"
    targetNamespace="http://www.edusource.ca/xsd/ecl_gather_v1pl1"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    elementFormDefault="qualified" version="ECL 0.5">

    <xs:element name="gather" type="gatherType" />

    <xs:element name="verb" type="verbType" />
    <xs:element name="verbType" type="xs:string" />
    <xs:element name="identifier" type="xs:string" />
    <xs:element name="from" type="xs:dateTime" />
    <xs:element name="until" type="xs:dateTime" />
    <xs:element name="metadataPrefix" type="xs:string" />
    <xs:element name="setSpec" type="xs:string" />

    <xs:complexType name="verbType">
        <xs:restriction base="xs:token">
            <xs:enumeration value="Identify" />
        </xs:restriction>
    </xs:complexType>
</xs:schema>
```
Appendix A3b: The XML Schema of the ECL Gather Expose Protocol

```xml
<xs:element name="gather_expose" type="exposeType" />
<xs:element name="header" type="headerType" />
<xs:element name="metadata" type="metadataType" />
<xs:element name="responseDate" type="xs.dateTime" />
<xs:element name="responseDate" type="xs.dateTime" />
<xs:element name="Identify" type="IdentifyType" />
<xs:element name="GetRecord" type="GetRecordType" />
<xs:element name="ListIdentifiers" type="ListIdentifiersType" />
<xs:element name="ListMetadataFormats" type="ListMetadataFormatsType" />
<xs:element name="ListRecords" type="ListRecordsType" />
<xs:element name="resumptionToken" type="resumptionTokenType" />
```

```xml
<xs:complexType name="exposeType">
  <xs:sequence>
    <xs:element ref="responseDate" minOccurs="1" maxOccurs="1" />
  </xs:sequence>
  <xs:choice>
    <xs:element ref="Identify" minOccurs="1" maxOccurs="1" />
    <xs:element ref="GetRecord" minOccurs="1" maxOccurs="1" />
  </xs:choice>
</xs:complexType>
```
Appendix A4a: The XML Schema of the ECL Submit Request Protocol

```xml
<xs:schema xmlns="http://www.edusource.ca/xsd/ecl_submit_v1pl1"
  targetNamespace="http://www.edusource.ca/xsd/ecl_submit_v1pl1"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  elementFormDefault="qualified" version="ECL 0.5">

  <xs:complexType name="submitType">
    <xs:sequence>
      <xs:element ref="action" type="xs:string" minOccurs="1" maxOccurs="1" />
      <xs:element ref="content_type" type="xs:string" minOccurs="1" maxOccurs="1" />
      <xs:element ref="username" minOccurs="1" maxOccurs="1" />
      <xs:element ref="groupname" minOccurs="1" maxOccurs="1" />
      <xs:element ref="permission" minOccurs="1" maxOccurs="1" />
      <xs:choice>
        <xs:element ref="ims_content" minOccurs="1" maxOccurs="1" />
        <xs:element ref="mds" minOccurs="1" maxOccurs="1" />
      </xs:choice>
      <xs:element ref="any" minOccurs="0" maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="setType">
    <xs:sequence>
      <xs:element name="setSpec" type="xs:string" />
      <xs:element name="setName" type="xs:string" />
      <xs:element name="setDescription" type="xs:string"
        maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="actionType">
    <xs:restriction base="xs:token">
      <xs:pattern value="\S+\(\S+\.)+\S+" />
    </xs:restriction>
  </xs:complexType>

</xs:schema>
```
Appendix A4b: The XML Schema of the ECL Store Protocol

```xml
<xs:schema xmlns="http://www.edusource.ca/xsd/ecl_store_v1p1"
    targetNamespace="http://www.edusource.ca/xsd/ecl_store_v1p1"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    elementFormDefault="qualified" version="ECL 0.5">

  <xs:complexType name="storeType">
    <xs:annotation>
      <xs:documentation>
        ECL Store Protocol
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:element ref="identifiers" maxOccurs="1" />
      <xs:element ref="errors" maxOccurs="1" />
      <xs:element ref="params" maxOccurs="1" />
      <xs:element ref="resumption_token" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="resumptionTokenType">
    <xs:sequence>
      <xs:element name="resumptionTokenId" type="xs:string"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="cursor" type="xs:Integer"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="completeListSize" type="xs:Integer"
        minOccurs="1" maxOccurs="1" />
      <xs:element name="expirationDate" type="xs:dateTime"
        minOccurs="1" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```
Appendix A5a: The XML Schema of the ECL Request Protocol

```xml
<xs:schema xmlns="http://www.edusource.ca/xsd/ecl_request_v1p1"
            targetNamespace="http://www.edusource.ca/xsd/ecl_request_v1p1"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            elementFormDefault="qualified" version="ECL 0.5">

  <xs:element name="request" type="requestType" />
  <xs:element name="data" type="dataType" />
  <xs:element name="identifiers" type="identifiersType" />
  <xs:element name="resumption_token" type="resumptionTokenType" />

  <xs:complexType name="requestType">
    <xs:annotation>
      <xs:documentation>
        ECL Request Protocol
      </xs:documentation>
    </xs:annotation>
    <xs:sequence>
      <xs:choice>
        <xs:element ref="data" minOccurs="1" maxOccurs="1" />
        <xs:element ref="resumption_token" minOccurs="1" maxOccurs="1" />
      </xs:choice>
      <xs:element ref="identifiers" maxOccurs="1" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="resumptionTokenType">
    <xs:sequence>
      <xs:element name="resumptionTokenId" type="xs:string">
        <!-- Details of the resumptionTokenId element -->
      </xs:element>
    </xs:sequence>
  </xs:complexType>

  <!-- Other elements and types related to the ECL Request Protocol -->
</xs:schema>
```

---

161
Appendix A5b: The XML Schema of the ECL Deliver Protocol

<xs:schema xmlns="http://www.edusource.ca/xsd/ecl_deliver_v1pl1" targetNamespace="http://www.edusource.ca/xsd/ecl_deliver_v1pl1" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" elementFormDefault="qualified" version="ECL 0.5">  
<xs:element name="deliver" type="deliverType" />  
<xs:element name="resumption_token" type="resumptionTokenType" />  
</xs:schema>

<xs:complexType name="deliverType">
  <xs:annotation>
    <xs:documentation>
      ECL Deliver Protocol
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:choice>
      <xs:element name="record" type="recordType" maxOccurs="unbounded" />
      <xs:element ref="bytes" type="xs:string" minOccurs="1" maxOccurs="1" />
    </xs:choice>
  </xs:sequence>
</xs:complexType>
</xs:choice>
  <xs:element ref="resumption_token" maxOccurs="1" />
  <xs:element ref="any" maxOccurs="unbounded" />
</xs:sequence>
</xs:complexType>

<xs:complexType name="resumptionTokenType">
  <xs:sequence>
    <xs:element name="resumptionTokenId" type="xs:string"
      minOccurs="1" maxOccurs="1" />
    <xs:element name="cursor" type="xs:Integer"
      minOccurs="1" maxOccurs="1" />
    <xs:element name="completeListSize" type="xs:Integer"
      minOccurs="1" maxOccurs="1" />
    <xs:element name="expirationDate" type="xs:dateTime"
      minOccurs="1" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="headerType">
  <xs:sequence>
    <xs:element name="modified_date" type="xs:dateTime"
      minOccurs="1" maxOccurs="1" />
    <xs:element name="permission" type="permissionType"
      minOccurs="1" maxOccurs="1" />
  </xs:sequence>
</xs:complexType>

<xs:complexType name="metadataType">
  <xs:element ref="lom" maxOccurs="unbounded" />
</xs:complexType>

<xs:complexType name="recordType">
  <xs:sequence>
    <xs:element name="header" minOccurs="1"
      maxOccurs="1" />
    <xs:element name="metadata" minOccurs="1"
      maxOccurs="1" />
  </xs:sequence>
</xs:complexType>
</xs:schema>
Appendix B

The Service Handler Interfaces

A service handler interface is a quick and easy way to implement and provide an ECL service. The SearchServiceHandler abstract class defines required methods as well as providing examples to help LOR developers in the service implementation task.

Appendix B lists the service handler classes defined for the four main IMS DRI interface functions.

Appendix B1: The SearchServiceHandler Class

```java
package ca.edusource.ecl.provider.services.handlers;

import javax.activation.DataHandler;
import ca.edusource.ecl.common.msg.*;
import ca.edusource.ecl.common.msg.search.*;
import ca.edusource.ecl.provider.services.ServiceHandler;

/**
 * SearchServiceHandler is an abstract class that lays out the required method (processSearch) for LOR developers to implement to provide the Search service. The implemented SearchServiceHandler class must be added to the configuration file (services_handlers.xml). It will be dynamically loaded to process the incoming search requests.
 */
public abstract class SearchServiceHandler implements ServiceHandler {

    // Default constructor
    public SearchServiceHandler() {} 

    /**
     * This method is invoked by the service dispatcher to handle the incoming messages. The method constructs the request data structure and relays the call to processSearch method.
     * @param message - The ECL message containing the request
     * @param dh - The data structure specified by JAF
     * @return EclMessage - The ECL message containing search results
     */
    public EclMessage digest(EclMessage message, DataHandler dh) {
        try {
            // returns the explanation and examples of the search service
            ExplainMessage explain = ExplainMsgBuilder.checkExplain(
                message.SEARCH, message);
        }
```
if (explain != null) {
    message.setPayload(explain.toXmlElement());
    return message;
}
Search request = new SearchImpl(message.getPayload());
SearchExpose response = processSearch(request);
message.setMessageType(EclMessage.EXPOSE);
message.setPayload(response.toXmlElement());
} catch (Exception e) {
    message.setMessageType(EclMessage.ERROR);
    message.setErrorMessage(e.getMessage());
    message.setPayload(null);
}
return message;

/**
 * LOR developer must implement this method to process the incoming
 * search request.
 * 
 * Usage:
 * The following is an example how to implement processSearch
 * method.
 * 
 * public SearchExpose processSearch(Search request) throws
 * SearchException {
 *     try {
 *         SearchExpose se = new SearchExposeImpl(
 *             request.getTemplateType());
 *         ArrayList searchRes = myLOR.searchRepository(
 *             resumptId, request, userid);
 *         se.setPrefferedNS(request.getPrefferedNS());
 *         se.setSupportAdvancedSearch(true);
 *         for (int i = 0; i < searchRes.size(); i++) {
 *             se.addLom((Element) searchRes.get(i));
 *         }
 *         return se;
 *     } catch (Exception e) {
 *         throw new SearchException("No result found: " +
 *             e.getMessage());
 *     }
 * }
 * 
 * @param request - The ECL Search content
 * 
 * @return SearchExpose - The Expose message for search function
 */
public abstract SearchExpose processSearch(Search request) throws
    SearchException;
Appendix B2: The GatherServiceHandler Class

```java
package ca.edusource.ecl.provider.services.handlers;

import javax.activation.DataHandler;
import ca.edusource.ecl.common.msg.*;
import ca.edusource.ecl.common.msg.gather.*;
import ca.edusource.ecl.provider.services.ServiceHandler;

/**
 * GatherServiceHandler is an abstract class that layouts all the
 * required methods that LOR developers must implement.
 * The implemented GatherServiceHandler class must be added into the
 * configuration file (services_handlers.xml). It will be
 * dynamically loaded to process the incoming gather requests.
 */
public abstract class GatherServiceHandler implements ServiceHandler {

    // Default constructor
    public GatherServiceHandler() {}

    /**
     * This method is invoked by the service dispatcher to handle the
     * incoming messages. The method parses the request, constructs the
     * request data structure, and relays the call to appropriate
     * methods.
     *
     * @param message - The ECL message containing the request
     * @param dh - The data structure specified by JAF
     *
     * @return EclMessage - The ECL response message
     */
    public EclMessage digest(EclMessage message, DataHandler dh) {
        try {
            // returns the explanation and examples of the
            // gather service
            ExplainMessage explain = ExplainMsgBuilder.checkExplain(
                    message.GATHER, message);
            if (explain != null) {
                message.setPayload(explain.toXmlElement());
                return message;
            }
            GatherExpose response = gather(new
                    GatherRequestImpl(message.getPayload()));
            message.setMessageType(EclMessage.EXPOSE);
            message.setPayload(response.toXmlElement());
        } catch (Exception e) {
            message.setMessageType(EclMessage.ERROR);
            message.setErrorMessage(e.getMessage());
            message.setPayload(null);
        }
        return message;
    }

    private GatherExpose gather(GatherRequest request) throws
```
GatherException {
String verb = request.getVerb();
if (verb.equals("GetRecord")) {
    return getRecord(request);
} else if (verb.equals("ListIdentifiers")) {
    return listIdentifiers(request);
} else if (verb.equals("ListMetadataFormats")) {
    return listMetadataFormats(request);
} else if (verb.equals("ListRecords")) {
    return listRecords(request);
} else if (verb.equals("ListSets")) {
    return listSets(request);
} else if (verb.equals("Identify")) {
    return identify(request);
} else {
    throw new GatherException(GatherException.BAD_ARGUMENT);
}
}

/**
 * LOR developers must implement this abstract method (GetRecord())
 * to handle the request for a record identified by a given
 * identifier.
 *
 * Usage:
 * The following is an example how to implement getRecord
 * method.
 *
 * Note that the usage of GatherExpose for building
 * the response for gather request.
 *
 * public GatherExpose getRecord(GatherRequest request) throws
 * GatherException{
 *    GatherExpose response = new
 *        GatherExposeImpl(request);
 *    response.addGetRecordData(lomRecord, identifier, 
 *        dateStamp, setSpec);
 *    return response;
 * }
 *
 * @param request - The GetRecord request
 *
 * @return GatherExpose - The gather expose
 */
public abstract GatherExpose getRecord(GatherRequest request)
    throws GatherException;

/**
 * LOR developers must implement this abstract method
 * (listIdentifiers ()) to handle the request for a list of
 * identifiers based on set, from, and/or until input
 * parameters.
 *
 * Usage:
 * The following is an example how to implement
 */
listIdentifiers.
public GatherExpose listIdentifiers(GatherRequest request)
throws GatherException{
    GatherExpose response = new GatherExposeImpl(request);
    response.addListIdentifiersData(identifier1, dateStamp1, setSpec1);
    response.addListIdentifiersData(identifier2, dateStamp2, setSpec2);
    response.addListIdentifiersData(identifier3, dateStamp3, setSpec3);
    // OPTIONAL: ResumptionToken could be set if there
    // are too many records.
    response.setResumptionToken(resumptionToken, expirationDate, completeListSize, cursor);
    return response;
}

@param request - The listIdentifiers request
@return GatherExpose - The gather expose
/
public abstract GatherExpose listIdentifiers(GatherRequest request)
throws GatherException;

/**
 * LOR developers must implement this abstract method
 * (listMetadataFormats()) to handle the request for metadata
 * formats of a record identified by a given identifier
 *
 * Usage:
 * The following is an example how to implement
 * listMetadataFormats.
 *
 public GatherExpose listMetadataFormats(GatherRequest request) throws GatherException{
    GatherExpose response = new GatherExposeImpl(request);
    response.addMetadataFormatData(metadataPrefix1, schema1, metadataNamespace1);
    response.addMetadataFormatData(metadataPrefix2, schema2, metadataNamespace2);
    response.addMetadataFormatData(metadataPrefix3, schema3, metadataNamespace3);
    return response;
} 

@param request - The listMetadataFormats request
@return GatherExpose - The gather expose
*/
public abstract GatherExpose listMetadataFormats(GatherRequest request) throws GatherException;

168
/**
 * LOR developers must implement this abstract method
 * (listRecords()) to handle the request for a list of records
 * based on set, from, and/or until input parameters.
 * 
 * Usage:
 * The following is an example how to implement
 * listRecords.
 * 
 * public GatherExpose listRecords(GatherRequest request)
 * throws GatherException{
 * GatherExpose response = new
 * GatherExposeImpl(request);
 * response.addListRecordsData(lomRecord1, identifier1,
 * dateStamp1, setSpec1);
 * response.addListRecordsData(lomRecord2, identifier2,
 * dateStamp2, setSpec2);
 * response.addListRecordsData(lomRecord3, identifier3,
 * dateStamp3, setSpec3);
 * // OPTIONAL: ResumptionToken could be set if there
 * // are too many records.
 * response.setResumptionToken(resumptionToken,
 * expirationDate, completeListSize, cursor);
 * return response;
 * }
 *
 * @param request - The listRecords request
 * @return GatherExpose - The gather expose
 */

public abstract GatherExpose listRecords(GatherRequest request)
throws GatherException;

/**
 * LOR developers must implement this abstract method
 * (listSets()) to handle the request for a entire list of the
 * repository's set specifications.
 * 
 * Usage:
 * The following is an example how to implement listSets.
 * 
 * public GatherExpose listSets(GatherRequest request) throws
 * GatherException{
 * GatherExpose response = new
 * GatherExposeImpl(request);
 * response.addListSetsData(setSpec1, setName1);
 * response.addListSetsData(setSpec2, setName2);
 * response.addListSetsData(setSpec3, setName3);
 * return response;
 * }
 *
 * @param request - The listSets request
 * @return GatherExpose - The gather expose
 */
public abstract GatherExpose listSets(GatherRequest request) throws GatherException;

/**
 * LOR developers must implement this abstract method
 * (identify()) to handle the request for the repository
 * information.
 * Usage:
 * The following is an example how to implement identify.
 * public GatherExpose identify(GatherRequest request) throws
 * GatherException{
 * // Note that GatherResponseImpl takes request
 * GatherExpose response = new GatherExposeImpl(request);
 * response.addIdentifyData(repositoryName, baseURL,
 * earliestDateStamp, deletedRecord,
 * granularity, adminEmail);
 * return response;
 * }
 * @param request - The identify request
 * @return GatherExpose - The gather expose
 */
public abstract GatherExpose identify(GatherRequest request) throws GatherException;

Appendix B3: The SubmitServiceHandler Class

package ca.edusource.ecl.provider.services.handlers;

import javax.activation.DataHandler;
import ca.edusource.ecl.common.msg.*;
import ca.edusource.ecl.common.msg.submit.*;
import ca.edusource.ecl.provider.services.ServiceHandler;
import ca.edusource.ecl.config.AppConfigManager;
import ca.edusource.ecl.util.FileUtil;

/**
 * SubmitServiceHandler is an abstract class that layouts the
 * required method (processSubmit) for LOR developer to implement
 * to provide the Submit service. The implemented SubmitServiceHandler
 * class must be added to the configuration file
 * (services_handlers.xml). It will be dynamically loaded to process
 * the incoming submit requests.
 */
public abstract class SubmitServiceHandler implements ServiceHandler {

    // Default constructor
    public SubmitServiceHandler() {}
This method is invoked by the service dispatcher to handle the incoming messages. The method constructs the request data structure and relays the call to processSubmit method.

@throws 

**public** EclMessage digest(EclMessage message, DataHandler dh) {
   try {
      EclMessage explain = ExplainMsgBuilder.checkExplain(
         message.SUBMIT, message);
      if (explain != null) {
         message.setPayload(explain.toXmlElement());
         return message;
      }
      Submit request = new SubmitImpl(message.getPayload());
      String zpath = null;
      if (dh != null) {
         zpath = AppConfigManager.getTempDir() +
            request.getImsContentName();
         FileUtil.copyFile(dh.getInputStream(), zpath);
      }
      Store response = processSubmit(request, zpath);
      message.setMessageType(message.STORE);
      message.setPayload(response.toXmlElement());
   } catch (Exception e) {
      message.setMessageType(message.ERROR);
      message.setErrorMessage(e.getMessage());
      message.setPayload(null);
   }
   return message;
}/**

LOR developers must implement this abstract method to handle the submit request.

Usage:
The following is an example how to implement processSubmit.

    public Store processSubmit(Submit request, String zipName)
    throws SubmitException {
        Store store = new StoreImpl();
        String username = request.getUsername();
        String action = request.getAction();
        if (action.equals(Submit.SAVE)) {
            return saveMetadata(store, username, request, zipName);
        } else if (action.equals(Submit.LIST_IDENTIFIERS)) {
            HashMap map = pr.listIdentifiers(username);
            ArrayList list = (ArrayList) map.get("identifiers");
            for (int i = 0; i < list.size(); i++) {
                store.addIdentifier((String)
list.get(i));
*
* String resumption = (String) map.get(
"resumptionToken");
* store.setParameter("resumptionToken",
resumption);
* store.setStatus("OK");
* return store;
* else if (action.equals(Submit.DELETE_LOM)) {
* 
* 
* 
* try {
* pr.deleteMetadata(username, request. 
* getParameter(Submit.IDENTIFIER));
* store.setStatus("OK");
* } catch (Exception e) {
* store.setStatus("ERROR");
* store.setParameter("ERROR_MESSAGE",
* e.getMessage());
* }
* return store;
* 
* } 
* throw new SubmitException("Unknown action");
* }
* @param request The ECL Submit content 
* @param zpath The location of the attached file 
* * 
* @return Store The IMS Expose for submit function 
*/
public abstract Store processSubmit(Submit request, String zpath) 
throws SubmitException;
}

Appendix B4: The RequestServiceHandler Class

package ca.edusource.ecl.provider.services.handlers;
import javax.activation.DataHandler;
import ca.edusource.ecl.common.msg.*;
import ca.edusource.ecl.common.msg.request.*;
import ca.edusource.ecl.provider.services.ServiceHandler;
import ca.edusource.ecl.util.Log;

/**
* RequestServiceHandler is an abstract class that outlines the 
* required method (processRequest) for LOR developer to implement 
* to provide the Submit service. The implemented 
* RequestServiceHandler class must be added to the configuration file 
* (services_handlers.xml). It will be dynamically loaded to process 
* incoming submit request.
*/
public abstract class RequestServiceHandler implements ServiceHandler {

// Default constructor
public RequestServiceHandler() {}

/**
This method is invoked by the service dispatcher to handle the incoming messages. The method constructs the request data structure and relays the call to processRequest method.

@param message - The ECL message containing the request
@param dh - The data structure specified by JAF

@return EclMessage - The ECL message containing search results

public EclMessage digest(EclyMessage message, DataHandler dh) {
    try {
        ExplainMessage explain = ExplainMsgBuilder.checkExplain(message.REQUEST, message);
        if (explain != null) {
            message.setPayload(explain.toXmlElement());
            return message;
        }
        Request request = new RequestImpl(message.getPayload());
        Deliver response = processRequest(request);
        message.setMessageType(message.DELIVER);
        message.setPayload(response.toXmlElement());
    } catch (Exception e) {
        Log.debug(this, e);
        message.setMessageType(message.ERROR);
        message.setErrorMessage(e.getMessage());
        message.setPayload(null);
    }
    return message;
}

/**
 * LOR developers must implement this abstract method to handle the request for LO or LOM.
 *
 * Usage:
 * The following is an example how to implement processRequest for request of a list of LOM records.
 *
 * public Deliver processRequest(Request request) throws RequestException {
 *     RequestException e;
 *     if (!request.getRequestType().equals(Request.LOMS)) {
 *         throw new RequestException("Request type not supported");
 *     }
 *     ArrayList list = request.getMetadataIdentifiers();
 *     String userId = request.getUsername();
 *     Deliver deliver = new DeliverImpl();
 *     for (int i = 0; i < list.size(); i++) {
 *         try {
 *             Identifier id = (Identifier) list.get(i);
 *             Metadata md = pr.getMetadata(userId, id.getEntry());
 *             if (md != null) {
 *                 deliver.addMetadata(md);
 *             } else {
 *                 deliver.addUnretrievableIdentifier((String)
*                      list.get(i));
*                      }
*                ) catch (Exception ee) {
*                      deliver.addUnretrievableIdentifier((String)
*                      list.get(i));
*                      Log.error(this, ee);
*                      }
*                return deliver;
*            }
* @param request The ECL Request content
* @return Deliver The IMS Expose for request function
*/
public abstract Deliver processRequest(Request request) throws RequestException;
}
BIBLIOGRAPHY


"History of the Dublin Core Metadata Initiative," [Online document], [2006 Jun 5], Available at WWW: http://dublincore.org/about/history/.


[70] K. Riley and M. McKell, “IMS digital repositories interoperability,” [Online document], [2006 May 20], Available at WWW: http://www.imsglobal.org/digitalrepositories/driv1p0/imsdri_info/1p0.htm.


180


"The Open Knowledge Initiative," [Online document], [2006 May 20], Available at WWW: http://okicommunity.mit.edu/.


