

Facilitating Web Service Discovery in Distributed Web Service Registries

Submitted By

Afiya S. Kassim, B.Eng.

A Thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment

of the requirements for the degree of

Master of Applied Science

in

Electrical Engineering

Ottawa-Carleton Institute for Electrical and Computer Engineering

Faculty of Engineering

Department of Systems and Computer Engineering

Carleton University

Ottawa, Ontario, Canada, K1S 5B6

January, 2008

© Copyright

2008, Afiya S. Kassim

The undersigned recommend to the Faculty of Graduate Studies and Research
acceptance of the thesis

**Facilitating Web Service Discovery in Distributed Web
Service Registries**

Submitted by Afiya S. Kassim, B.Eng

in partial fulfillment of the requirements for the Degree of Master of Applied Science

Thesis Co-Supervisor

Professor Babak Esfandiari

Thesis Co-Supervisor

Professor Shikharesh Majumdar

Chair, Department of Systems and Computer Engineering

Professor Victor C. Aitken

Carleton University

January 2008

Abstract

The management of distributed Web Service registries is an important issue in the context of a Web Services based distributed system. The current registry management systems and proposals for distributed service registries are investigated. Through this investigation, it was discovered that none of the existing proposals satisfy all the requirements of scalability, availability, ease of management and flexibility in terms of the management of the registries. Therefore, a Meta-Directory based registry management system is introduced in this thesis. The system uses distributed Meta-Directory nodes for the management of the service registries. The system also provides flexibility as the routing used for inter-connecting the distributed Meta-Directory nodes can be selected during deployment to accommodate system requirements. The distributed Meta-Directory architecture enhances system performance by allowing the user to choose an appropriate network model, during configuration, based on the system state such as bandwidth available and maximum allowable delay per query message.

Acknowledgements

First and foremost I would like to thank God for giving me the opportunity to go down this road as well as the strength and perseverance to finally be in a position of finishing my Master's degree.

I am extremely grateful to the Ontario Centres of Excellence and Alcatel-Lucent for providing operational funding for my research. I would also like to thank Bill St. Arnaud and Darcy Quesnel from CANARIE for their help in setting up the PlanetLab environment in which the experiments were performed.

I would like to extend my gratitude to my supervisors, Dr. Esfandiari and Dr. Majumdar, for guiding me and always being there to offer assistance and answering my 3:00 AM emails. Without your support and expertise this would not have been possible.

This thesis is dedicated to my parents, Mzee Kassim and Mama Kassim, who have always supported me and instilled in me the belief that anything is possible as long as I work hard at it. Nothing defines who I am or limits my abilities in anything I pursue other than my willingness and determination in achieving it. I love you and thank you for raising me to be the person I am today.

I would also like to thank my siblings, Ally, Zubeida, and Kamaria, for listening to my complaints, reminding me that everything happens for a reason, and assuring me that everything will work out in the end. You were always there and made me feel that you were always right next to me even though we are continents apart.

Last but not least, this goes out to all my friends who have been my pillar of support and my second family. Thank you for always being there and your constant check ups to make sure I am still sane while working through the challenges this thesis presented.

Table of Contents

ABSTRACT.....	iii
ACKNOWLEDGEMENTS.....	iv
LIST OF TABLES.....	x
LIST OF FIGURES.....	xi
LIST OF ACRONYMS.....	xvi
CHAPTER 1 INTRODUCTION.....	1
1.1 BACKGROUND.....	1
1.2 PROBLEM MOTIVATION AND THE PROPOSED SOLUTION	4
1.3 CONTRIBUTIONS	6
1.4 ORGANIZATION.....	7
CHAPTER 2 LITERATURE REVIEW	9
2.1 DEPLOYMENT CLASSIFICATIONS.....	10
2.1.1 <i>Centralized Architecture</i>	10
2.1.2 <i>Decentralized Architecture</i>	15
2.1.3 <i>Meta-Directory Architecture</i>	21
2.1.3.1 Centralized Meta-Directory Architecture	23
2.1.3.2 Distributed Meta-Directory Architecture.....	28
2.2 DISCUSSION	32
2.2.1 <i>Availability</i>	32
2.2.2 <i>Scalability</i>	34
2.2.3 <i>Ease of Management</i>	36

2.2.4	<i>Transparency</i>	37
2.2.5	<i>Meta-Directory System</i>	39
2.2.5.1	Centralized Meta-Directory Approach	40
2.2.5.2	A Network of Meta-Directories	41
2.3	AN INTRODUCTION TO THE SOLUTION: PROPOSED META-DIRECTORY BASED APPROACH.....	43
2.3.1	<i>Configurable Routing Framework</i>	47
CHAPTER 3 SYSTEM DESIGN.....		49
3.1	DESIGN MODELS.....	49
3.1.1	<i>Client Residing on the Service Registry</i>	49
3.1.2	<i>Client residing on the Service Provider</i>	51
3.2	USE CASES.....	53
3.3	META-DIRECTORY SYSTEM DESIGN	56
3.3.1	<i>CHORD (DHT) Model</i>	56
3.3.2	<i>Hash Table Structure</i>	58
3.3.3	<i>Publishing Service Information</i>	61
3.3.4	<i>Deleting Service Information</i>	64
3.3.5	<i>Querying Service Information</i>	66
3.4	CONFIGURABLE ROUTING FRAMEWORK	66
3.4.1	<i>Network Models</i>	67
3.4.1.1	Fully Connected Model.....	68
3.4.1.2	Fully Connected (DHT) Model.....	70
3.4.1.3	Super Peer Model.....	73

3.4.2	<i>Performance Analysis of Network Models</i>	75
3.4.2.1	Total Number of Messages Exchanged	75
3.4.2.2	Total Number of Hops	80
3.4.2.3	Maintenance Overhead	83
CHAPTER 4	IMPLEMENTATION	88
4.1	META-DIRECTORY ARCHITECTURE	88
4.2	META-DIRECTORY SYSTEM IMPLEMENTATION.....	90
4.2.1	<i>Meta-Directory Sequence Diagrams</i>	94
CHAPTER 5	PROTOTYPE TESTING	104
5.1	FUNCTIONAL TESTING	104
5.1.1	<i>CHORD (DHT) Network</i>	105
5.1.1.1	Network Setup	105
5.1.1.2	Publishing Service Information	108
5.1.1.3	Querying Service Information	108
5.1.2	<i>Fully Connected Network</i>	110
5.1.2.1	Network Setup	111
5.1.2.2	Publishing Service Information	112
5.1.2.3	Querying Service Information	113
5.1.3	<i>Super Peer Network</i>	115
5.1.3.1	Network Setup	115
5.1.3.2	Publishing Service Information	117
5.1.3.3	Querying Service Information	118
5.2	PERFORMANCE ANALYSIS.....	121

5.2.1	<i>Test Plan for the Performance Analysis Experiments</i>	122
5.2.2	<i>Performance Analysis Approaches</i>	124
5.2.3	<i>Performance Analysis using PlanetLab</i>	125
5.2.3.1	Experimental Overview	126
5.2.3.2	Experimental Parameters	126
5.2.3.3	Experimental Results	128
5.2.3.3.1	Effect of the Size of the Network	129
5.2.3.3.2	Effect of the Number of Attributes Used in a Query	131
5.2.4	<i>Performance Analysis using P2PSim</i>	133
5.2.4.1	Simulation Overview	134
5.2.4.2	Simulation Parameters	135
5.2.4.3	Simulation Results	137
5.2.4.3.1	Memory Overhead	138
5.2.4.3.2	Network Delay	138
5.2.4.3.3	Path Length	140
5.2.4.3.4	Network Bandwidth	141
CHAPTER 6 ADAPTABLE ROUTING FRAMEWORK		143
6.1	ADAPTABLE ROUTING ALGORITHMS	145
6.1.1	<i>Changing the Maintenance State of the System</i>	147
6.1.2	<i>Transforming the Network from an SPFC to an FC Instance</i>	147
6.1.3	<i>Transforming the Network from an SPCHORD to a CHORD (DHT) Instance</i> 149	
6.1.4	<i>Transforming the Network from an FC to an SPFC Instance</i>	152

6.1.5	<i>Transforming the Network from a CHORD (DHT) to an SPCHORD Instance</i>	155
6.1.6	<i>Transforming the Network from a CHORD (DHT) to an FCDHT</i>	158
6.1.7	<i>Transforming the Network from an FCDHT to a CHORD (DHT)</i>	159
6.1.8	<i>Algorithms Specific to the SP Nodes</i>	159
6.1.9	<i>Algorithm Specific to the Entry Nodes</i>	160
6.1.10	<i>Algorithm used by the RT</i>	160
6.2	ANALYSIS OF TRANSFORMATION ALGORITHMS	161
CHAPTER 7 CONCLUSIONS		165
7.1	SUMMARY	165
7.1.1	<i>Fully Connected Network</i>	165
7.1.2	<i>Fully Connected (DHT) Network</i>	166
7.1.3	<i>CHORD (DHT) Network</i>	167
7.1.4	<i>Super Peer Network</i>	167
7.2	DISCUSSION	168
7.2.1	<i>Availability</i>	168
7.2.2	<i>Scalability</i>	168
7.2.3	<i>Ease of Management</i>	169
7.2.4	<i>Transparency</i>	169
7.3	LIMITATIONS	171
7.4	FUTURE WORK	171
REFERENCES		173
APPENDIX A: DERIVATION OF TRANSFORMATION COMPLEXITIES		177

List of Tables

TABLE 1 EVALUATION OF SYSTEMS.....	38
TABLE 2 ATTRIBUTES HASHED BY THE META-DIRECTORY SYSTEM.....	60
TABLE 3 HASH TABLE CONTENTS.....	61
TABLE 4 PERFORMANCE ANALYSIS METRICS	122
TABLE 5 PERFORMANCE ANALYSIS PARAMETERS	123
TABLE 6 EXPERIMENTAL PARAMETERS AND VALUES.....	127
TABLE 7 SIMULATION PARAMETERS AND VALUES	137
TABLE 8 TIME COMPLEXITY OF ALGORITHMS.....	162
TABLE 9 LIST OF INVARIANTS FOR EVERY NETWORK TRANSFORMATION.....	163

List of Figures

FIGURE 1 CONCEPTUAL VIEW OF WEB SERVICE COMPONENTS	2
FIGURE 2 THE EXTRANET SERVICE ARCHITECTURE	4
FIGURE 3 AN EXAMPLE OF A CENTRALIZED ARCHITECTURE [25].....	11
FIGURE 4 UDDI CORE TYPES [4].....	13
FIGURE 5 AN EXAMPLE OF A DISTRIBUTED ARCHITECTURE [25].....	16
FIGURE 6 VIRTUAL SPACE BASED SYSTEM [17]	19
FIGURE 7 THE AD-UDDI DISTRIBUTED ARCHITECTURE [6]	25
FIGURE 8 METEOR COMMUNICATION LAYER OVERVIEW [26]	24
FIGURE 9 WEB SERVICES SYNDICATION OVERVIEW [13]	27
FIGURE 10 CLUSTERING OF WEB SERVICES [18].....	29
FIGURE 11 SPIDER ARCHITECTURE [16].....	30
FIGURE 12 DISTRIBUTED HASH TABLE ARCHITECTURE [1]	32
FIGURE 13 CENTRALIZED APPROACH	41
FIGURE 14 SERVICE PUBLISHING [1].....	43
FIGURE 15 DISTRIBUTED META-DIRECTORY APPROACH.....	44
FIGURE 16 SERVICE PUBLISHING IN PROPOSED META-DIRECTORY APPROACH.....	45
FIGURE 31 META-DIRECTORY SYSTEM USE CASE DIAGRAM	53
FIGURE 17 CHORD (DHT) RING	57
FIGURE 18 META-DIRECTORY INTERFACE.....	59
FIGURE 19 SERVICE PUBLISHING IN META-DIRECTORY NODES.....	62
FIGURE 20 SERVICE PUBLISHING AND MESSAGE DISTRIBUTION.....	64
FIGURE 21 DELETING SERVICE INFORMATION FROM META-DIRECTORY NODES.....	65

FIGURE 22 SERVICE DISCOVERY IN DISTRIBUTED META-DIRECTORY NODES	66
FIGURE 23 FULLY CONNECTED MODEL	68
FIGURE 24 LOCAL DISTRIBUTION MODEL.....	69
FIGURE 25 FULLY CONNECTED (DHT) MODEL	71
FIGURE 26 DHT DATA DISTRIBUTION MODEL	72
FIGURE 27 SUPER PEER MODEL.....	73
FIGURE 28 EFFECT OF NETWORK SIZE ON THE TOTAL NUMBER OF MESSAGES EXCHANGED PER QUERY	79
FIGURE 29 EFFECT OF NETWORK SIZE ON THE TOTAL NUMBER OF HOPS.....	83
FIGURE 30 EFFECT OF NETWORK SIZE ON THE PERIODIC MESSAGES EXCHANGED.....	86
FIGURE 32 META-DIRECTORY ARCHITECTURE.....	89
FIGURE 33 SERVICE PUBLISHING AND DISCOVERY IN META-DIRECTORY COMPONENTS...	90
FIGURE 34 META-DIRECTORY PACKAGES	92
FIGURE 35 UML CLASS DIAGRAM FOR THE COM.AKASSIM.OVERLAY PACKAGE.....	93
FIGURE 36 SETTING UP A FULLY CONNECTED NETWORK	95
FIGURE 37 CREATING A SUPER PEER NODE	95
FIGURE 38 JOINING OR CREATING A DHT NETWORK	96
FIGURE 39 ADAPTING THE NETWORK BETWEEN FULLY CONNECTED (DHT) AND CHORD (DHT).....	97
FIGURE 40 PUBLISHING A SERVICE IN A FULLY CONNECTED NETWORK SEQUENCE DIAGRAM.....	98
FIGURE 41 PUBLISHING A SERVICE IN A DHT NETWORK SEQUENCE DIAGRAM	99
FIGURE 42 DELETE SERVICE IN A FULLY CONNECTED NETWORK SEQUENCE DIAGRAM .	100

FIGURE 43 DELETE SERVICE IN A DHT NETWORK SEQUENCE DIAGRAM	101
FIGURE 44 SERVICE DISCOVERY IN A FULLY CONNECTED NETWORK SEQUENCE DIAGRAM	102
FIGURE 45 SERVICE DISCOVERY IN A DHT NETWORK SEQUENCE DIAGRAM	103
FIGURE 46 NETWORK SETUP.....	105
FIGURE 47 CREATING A DHT NODE	106
FIGURE 48 CONNECTING TO A DHT NETWORK.....	106
FIGURE 49 DHT ADMINISTRATIVE CONSOLE	107
FIGURE 50 PUBLISHING A SERVICE ON THE ADMIN CONSOLE.....	108
FIGURE 51 REGISTRY LOCATIONS.....	109
FIGURE 52 RESPONSE FROM THE REGISTRIES QUERIED	109
FIGURE 53 REGISTRY LOCATION RETURNED BY NARROWING THE SCOPE OF THE QUERY	110
FIGURE 54 RESPONSE FROM THE REGISTRY	110
FIGURE 55 FULLY CONNECTED ADMINISTRATIVE CONSOLE	111
FIGURE 56 FULLY CONNECTED NETWORK INSTANCE.....	111
FIGURE 57 SERVICE PUBLISHING IN A FULLY CONNECTED NETWORK.....	112
FIGURE 58 SUCCESSFUL SERVICE PUBLISHING IN A FULLY CONNECTED NETWORK.....	112
FIGURE 59 QUERY ISSUED BY THE NODE LISTENING ON PORT 6348.....	113
FIGURE 60 REQUEST RECEIVED BY NODE LISTENING AT PORT 6347.....	113
FIGURE 61 REQUEST RECEIVED BY NODE LISTENING ON PORT 6346	114
FIGURE 62 RESPONSE RECEIVED BY REQUESTING NODE LISTENING ON PORT 6348.....	115
FIGURE 63 SUPER PEER ADMINISTRATIVE CONSOLE	116
FIGURE 64 SUPER PEER NETWORK INSTANCE.....	116

FIGURE 65 SERVICE PUBLISHING IN SP CLUSTER.....	117
FIGURE 66 SUCCESSFUL SERVICE PUBLISHING IN SP CLUSTER.....	118
FIGURE 67 REQUEST RECEIVED BY ENTRY NODE.....	119
FIGURE 68 REQUEST RECEIVED BY SUPER PEER NODE AT PORT 6346.....	119
FIGURE 69 REQUEST RECEIVED BY SUPER PEER NODE AT PORT 6347.....	120
FIGURE 70 REQUEST RECEIVED BY THE ENTRY NODE N7E.....	120
FIGURE 71 RESPONSE RECEIVED BY NODE N7B.....	121
FIGURE 72 SEQUENCE DIAGRAM FOR PLANETLAB EXPERIMENTS.....	130
FIGURE 73 EFFECT OF THE SIZE OF THE NETWORK ON THE QUERY RESPONSE TIME.....	131
FIGURE 74 EFFECT OF THE NUMBER OF ATTRIBUTES ON THE QUERY RESPONSE TIME....	132
FIGURE 75 EFFECT OF THE SIZE OF NETWORK ON THE MEMORY OVERHEAD.....	138
FIGURE 76 EFFECT OF THE SIZE OF THE NETWORK ON THE LOOKUP DELAY.....	139
FIGURE 77 EFFECT OF THE SIZE OF THE NETWORK ON THE PATH LENGTH.....	140
FIGURE 78 EFFECT OF THE SIZE OF THE NETWORK ON THE BANDWIDTH USED PER QUERY	142
FIGURE 79 RUNTIME RE-CONFIGURATION.....	145
FIGURE 80 ACTIONS PERFORMED BY A SYSTEM ADMINISTRATOR WHEN THE NETWORK IS MODIFIED.....	147
FIGURE 81 TOGGLING THE STATE OF THE SYSTEM.....	147
FIGURE 82 CHANGING AN SPFC NETWORK TO AN FC NETWORK.....	148
FIGURE 83 EXAMPLE OF TRANSFORMATION FROM SPFC TO FC.....	149
FIGURE 84 CHANGING AN SPCHORD NETWORK TO A CHORD (DHT) NETWORK.....	150
FIGURE 85 EXAMPLE OF TRANSFORMATION FROM SPCHORD TO CHORD (DHT).....	151

FIGURE 86 CHANGING AN FC INSTANCE TO A SPFC INSTANCE.....	153
FIGURE 87 EXAMPLE OF TRANSFORMATION FROM FC TO SPFC.....	154
FIGURE 88 CHANGING A CHORD (DHT) INSTANCE TO AN SPCHORD INSTANCE.....	156
FIGURE 89 EXAMPLE OF TRANSFORMATION FROM CHORD (DHT) TO SPCHORD.....	157
FIGURE 90 CHANGING A CHORD (DHT) INSTANCE TO A FCDHT INSTANCE.....	158
FIGURE 91 CHANGING A FCDHT INSTANCE TO A CHORD (DHT) INSTANCE.....	159
FIGURE 92 GET THE REFERENCE OF ALL THE CLUSTERS IN A SUPER PEER NETWORK	159
FIGURE 93 PUBLISH KEY-VALUE PAIRS TO A CLUSTER THROUGH THE SUPER PEER NODE	160
FIGURE 94 DELETE KEY-VALUE PAIRS FROM A NODE.....	160
FIGURE 95 CREATE A POINT-TO-POINT CONNECTION WITH REMOTE NODES IN LIST	161

List of Acronyms

CORBA	Common Object Request Broker Architecture
DARPA	Defense Advanced Research Projects Agency
DAML-S	DARPA Agent Markup Language for Services
DHT	Distributed Hash Table
DUNS	Data Universal Numbering System
FC	Fully Connected
FCDHT	Fully Connected (DHT)
GLN	Global Location Number
JiST	Java in Simulation Time
P2P	Peer-to-Peer
PC	Primary Cluster
RDF	Resource Description Framework
Sax	Simple API for XML
SC	Secondary Cluster
SFC	Hilbert Space-Filling Curve
SOAP	Simple Object Access Protocol
SPCHORD	Super Peer model (CHORD (DHT))
SPFC	Super Peer model (Fully Connected)
SPFCDHT	Super Peer model (Fully Connected (DHT))
UDDI	Universal Description, Discovery and Integration
URI	Universal Resource Identifier
UUID	Unique Universal Identifier

W3C	World Wide Web Consortium
WS	Web Service
WSDL	Web Services Description Language
WSMX-DB	Web Service Modeling Execution Environment Database
XML	Extensible Markup Language
XSD	XML Schema Definition

CHAPTER 1 INTRODUCTION

Web Service Registries are used to facilitate the discovery of Web Services in a distributed system. The problem of management of distributed registries is a topic that has started attracting attention from a number of researchers. In a distributed registry network, we want to provide a single registry view such that the user does not need to know the locations of all the registries they want to query. Requests are sent to a single interface that handles the propagation of the requests to multiple registries without requiring any other input from the user. The responses are received from the multiple registries and forwarded back to the requester from the same interface. While providing this service, the underlying configuration between the registries should not be modified so that the solution can be applied to an already existing network of distributed registries.

An introduction to Web Services is covered in this chapter along with the introduction of an Extranet architecture. The Extranet architecture is used as an example where the distributed registry management system proposed in this thesis can be applied. Discussions on the motivation of this thesis as well as its overall contributions are also covered in this chapter.

1.1 Background

Web services are a paradigm that allow applications to communicate directly regardless of the language or platform in a distributed architecture [4]. Web services employ a Service Oriented Architecture [3] where software is no longer a product but is offered as a service. The applications are decomposed into distributed services and offered over the distributed system. The main components required for the invocation of

a web service are the Universal Resource Identifier (URI) of the web service as well as the interface definitions on how the web service should be invoked. The interface is described using a machine-processable format such as the Web Service Description Language (WSDL) [5], and the messages are represented using Extensible Markup Language (XML) with Simple Object Access Protocol (SOAP) as the communication protocol.

In order for web services to be shared over the distributed system, the service providers publish the required components in a service registry whose location is known to service requesters. This interaction is shown in detail in Figure 1.

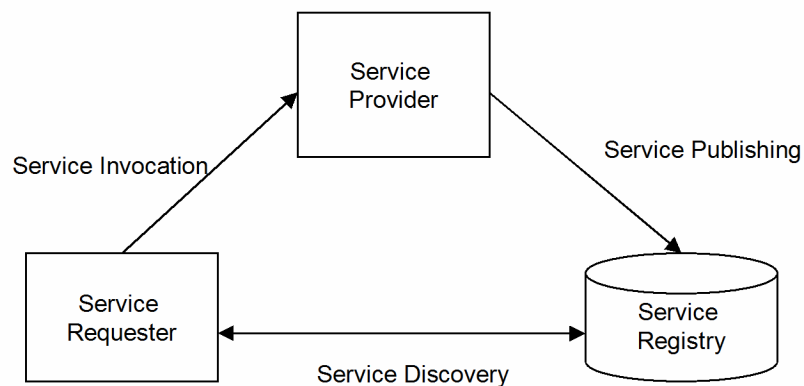


Figure 1 Conceptual View of Web Service Components [7]

The service provider is the entity that provides a Web Service that can be invoked in a distributed network. The service registry stores the location where the Web Service can be invoked as well as binding details on how the Web Service can be invoked. The service requester is the entity that wants to use a service that is provided over the distributed system. Figure 1 illustrates the communication model when there is only one service registry.

In this thesis, an Extranet system [15] is used for illustration of the distributed Web Service discovery system. Enterprises want to conduct businesses with their customers, suppliers, and or business partners electronically. An Extranet achieves this objective by providing a private network that allows enterprises to share part of the enterprise's information and operations as seen in Figure 2. Many Extranets are expected to be Web Service based due to the multi-million dollar investment in Web Service technologies by major software companies. Enterprises would also like to minimize their costs in terms of management of the Extranet system. Thus there is a requirement for facilities that would offer an Extranet service to enterprises where Extranets of multiple enterprises operate without interfering with each other. Such a facility will support many logically independent Extranets, one for each enterprise and its partners, which are active concurrently in the system.

In an Extranet system, a Web Service (WS) Gateway is the entry point to an enterprises services and registries. A Service Requester must send all query messages through the WS Gateway. The WS Registry is where service information is stored and the Server is where the service can be invoked by a Service Requester. A Remote Service Requester is a WS requester who is located outside the local network of an enterprise.

As can be seen in Figure 2, the architecture of an Extranet introduces multiple distributed registries in the network that interconnect enterprises. When there are multiple registries, a service requester needs to know the location of all the registries it needs to query when the requester wants to invoke a service. This poses a problem as there can be a lot of registries in the network.

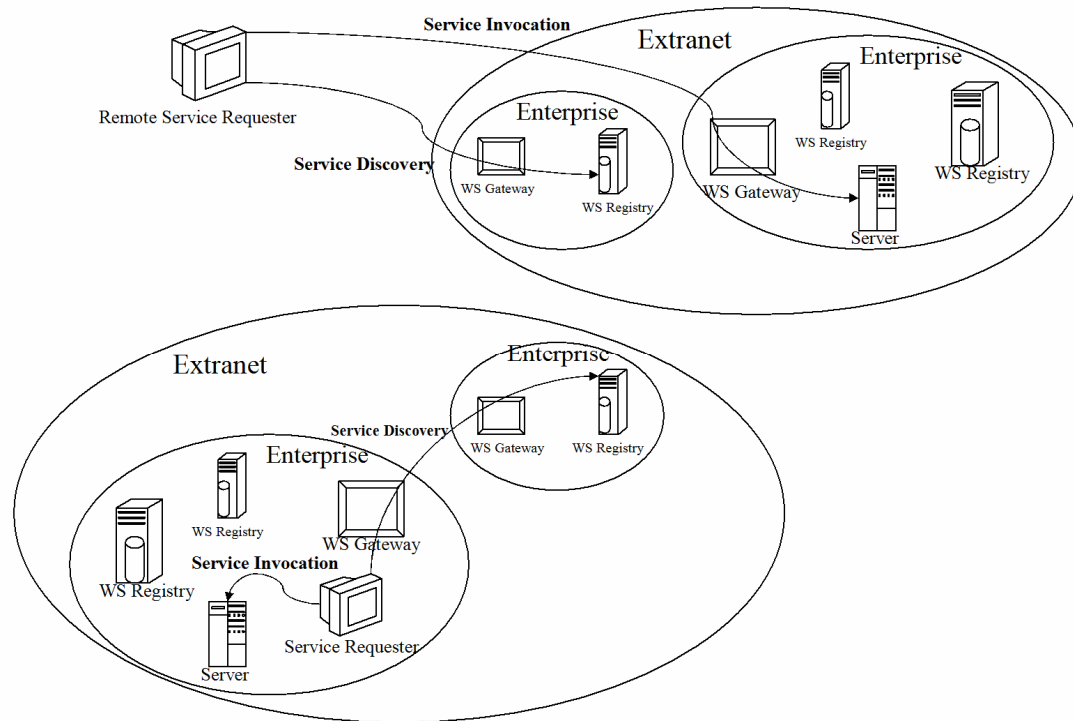


Figure 2 The Extranet Service Architecture

1.2 Problem Motivation and the Proposed Solution

In order for an Extranet to perform effectively, a system that manages the distributed registries and offers the service requester a single interface to the network while the system handles the communication among the multiple registries needs to be devised. By providing the service requester with a single discovery interface, such a system provides a centralized registry view to the users while distributing the load in multiple registries. Another important concern is the effective management of the network through query delay minimization and an effective utilization of the network bandwidth.

In a distributed registry environment, it is paramount that the management of the system is seamless to the user. A *Meta-Directory approach* is used in this thesis to

manage the system such that the current distributed registry architecture already existing in the companies does not have to be modified. This approach can be applied to any distributed multiple registry-based system without incurring a high installation overhead. A Meta-Directory node stores information about the distributed registries in the network. Whenever a new service is published in the network, the attributes that can be used in a query along with the location of the registry is forwarded to the Meta-Directory node.

When a service discovery request is received, a search is first performed in the Meta-Directory system in order to find the registry that has the information, and then the request is forwarded to the appropriate registry. This message exchange among the nodes in the Meta-Directory system and the registries is transparent to the service requester. From the service requester's point of view, the request is sent to one node in the system and the response is received from the same node.

In a distributed environment it is also important to minimize the delay incurred during a query as well as minimize the network bandwidth usage by a query. For improved network performance, this research looks at a configurable network model for the Meta-Directory nodes. The routing between the Meta-Directory nodes can be chosen based on network parameters, such as the number of Meta-Directory nodes, to provide better system performance.

As we can see, there are a number of problems associated with distributed registry systems. In order to facilitate web service discovery, a distributed registry management system has to address all these issues. The following section provides and discusses the overall contributions of the Meta-Directory approach proposed in this thesis.

1.3 Contributions

Distributed Web Service registries offer an interesting area of research in terms of the management of the registries. This thesis proposes a system that supports large scale service discovery in distributed Web Service registries.

A Meta-Directory architecture which manages the effective discovery of Web Services is introduced. The Meta-Directory nodes store the location of the Web Service registry that can answer a query in hash tables so as to facilitate searching. The important characteristics of the Meta-Directory system are described.

- Since information about the services that are stored in the service registries are saved in the Meta-Directory nodes, during a query only the service registries that have the information requested will be queried. Therefore, the query does not have to be propagated to all the distributed registry nodes.
- The Meta-Directory system allows the service provider to publish service information in any of the local registries and forward the registry information to any of the Meta-Directory nodes in the system. This decouples the registries from the Meta-directory nodes so that the local registries are always available for service publishing and discovery.
- The Meta-Directory system does not change the underlying communication model of any of the already existing Web Service components, i.e. the service provider, requestor and registry.
- The framework allows the existence of multiple distributed Meta-Directory nodes.

- The underlying distributed architecture is transparent to the user that is provided with a single registry view. The client only needs to know the location of one of the Meta-Directory nodes and the distributed system is transparent to the user.
- The service requester does not need to know the location of any of the service registries.
- To minimize the inter-communication delay among these distributed Meta-Directory nodes, a configurable network model is introduced that can be used when the system is being deployed. The system performance for a given network depends on the proper choice of a network model. An appropriate protocol that gives rise to good performance for a given system can thus be chosen by the system administrator during the configuration of the system.
- The Meta-Directory system ensures that local business registries are accommodated hence creating a network of registries.
- The distribution of the Meta-Directory nodes also ensures that there is no performance bottleneck and the system does not introduce a single point of failure as the information in the Meta-Directory nodes is replicated.

1.4 Organization

This thesis outlines the Meta-Directory architecture that can be used in a Web Service based distributed system that includes the Extranet environment. The configurable network model of the system is described as well. Chapter 2 reviews the existing literature on management of distributed registry systems and introduces the Meta-Directory approach proposed in this thesis. Chapter 3 introduces the Meta-Directory system design as well as the configurable framework that allows the system

administrator to choose the appropriate network model during system configuration. The system implementation is described in chapter 4. Chapter 5 introduces the evaluation mechanisms followed in evaluating the Meta-Directory system as well as presents and analyses the results. Chapter 6 provides an overview of the adaptable framework and provides the complexity and invariants for the algorithms. Chapter 7 concludes the thesis, discusses limitations of the Meta-Directory design and outlines future directions for research.

CHAPTER 2 LITERATURE REVIEW

This chapter looks at the existing Web Service registry architectures for distributed service systems. The problem this thesis is addressing is the management of distributed service registries where the user is provided with a single registry view and the user is oblivious to the existence and communication among the distributed registries. Since the system we are looking at consists of distributed service registries, the existing literature will be analyzed in terms of scalability, availability, ease of management and transparency. In this thesis, scalability is defined as the ability of the architecture to handle increasing load, as in service requests, as well as increasing number of web service registry entries. Availability is the ability of the registry architecture to continue with normal operation despite the presence of faults. Ease of management is in terms of the administration of the architecture and transparency is the ability of the architecture to manage the system without modifying the architecture of the existing service components.

These architectures are described in detail and an overall analysis on scalability, availability, ease of management, and transparency is performed at the end of the section in order to discuss and compare their performances.

The deployment of distributed service systems can be divided into three main categories: the centralized, the decentralized and the Meta-Directory architectures.

- Centralized architectures incorporate the use of a single centralized registry which stores all the service descriptions of the distributed service providers.
- Decentralized architectures often involve the existence of peer registries which introduces autonomous behavior. The peers have equal responsibilities and the

architecture is dynamic as peers can join and leave the network without disrupting service.

- The Meta-Directory architecture involves the introduction of nodes that store meta-information regarding the distributed registries. In this architecture, the peers do not have equal responsibilities as the decentralized architecture. In this case, some of the peers have extra responsibilities as they provide management of the distributed peer registries.

This section provides an overview of these web service registry architectures followed by examples of each type of architecture.

2.1 Deployment Classifications

This sub-section describes three deployment classifications for distributed systems. These architectures are the centralized, decentralized and the Meta-Directory architectures. A number of papers are analyzed that introduce and apply these architectures and finally an overall analysis on these architectures based on scalability, availability, ease of management and transparency is performed.

2.1.1 Centralized Architecture

In a centralized architecture, each service provider would publish their service descriptions in a centralized web service registry server. Service clients would then contact the well known centralized registry for service information. This is analogous to a client/server architecture where the centralized registry acts as the server. An example of a centralized registry architecture is depicted in Figure 3 where there exists a single registry. Information on the location of the web services is stored in the Web Service

Modeling Execution Environment Database (WSMX-DB) [25]. After a service requester receives the list of providers from the WSMX-DB indicating the providers' locations, the requester is then able to contact the web service providers directly.

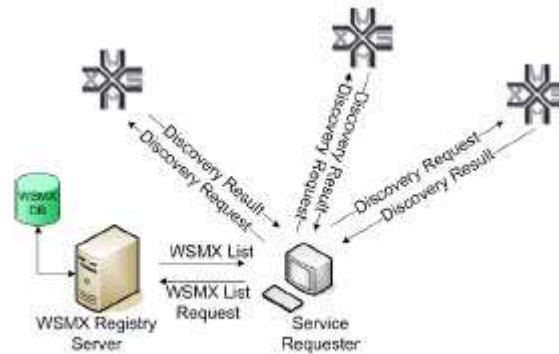


Figure 3 An Example of a Centralized Architecture [25]

This section provides an overview of the Universal Description, Discovery and Integration (UDDI) [4] registry that is used for publishing and searching of web service descriptions. UDDI is a standard for both the specification of web services and businesses. UDDI also supports the description and discovery of the interfaces that can be used to access the specified services. The web service descriptions are not part of the UDDI specification but they can be referenced by using tModels. A web service description can be provided by using Web Service Description Language (WSDL) [5]. UDDI uses standard technologies that are on top of an operating stack. These technologies are:

- XML for service definitions and querying,
- SOAP which is an XML communication protocol consisting of three parts: an envelope defining a framework that describes the contents of a message and how

it should be processed, a set of encoding rules for expressing application specific data types, and a convention for representing remote calls and responses.

- HTTP over TCP/IP is used as the transport protocol.

In order for a service to register with the UDDI registry, they must first define an interface description document using WSDL and then deploy it in a public Internet location. The publisher must then send a SOAP publication message to the registry that also indicates the location of the WSDL using a URI. The service requester would query the UDDI registry using either the web service name, or the business name and the UDDI registry will respond with the location of the service and description document.

The UDDI data model [4] consists of data structures that are hierarchically organized. The information about a web service is described in several categories where each category provides a more detailed information than the one before it. A Unique Universal Identifier (UUID) is used to identify each entity in the data model. There are six data structures in the UDDI version 3 of the specification: businessEntity, businessService, bindingTemplate, tModels, publisherAssertion, and operationalInfo structures. The four core types and their relationships are shown in Figure 4.

The business entity module provides information about the provider that has published the service. These business entity structures encapsulate information about Web services. They contain the following elements: names and descriptions, contacts of the people associated with the business entity, categories that represent the business entity's features, identifiers such as a department number and discovery URI's which are links to any additional documents describing the business entity.

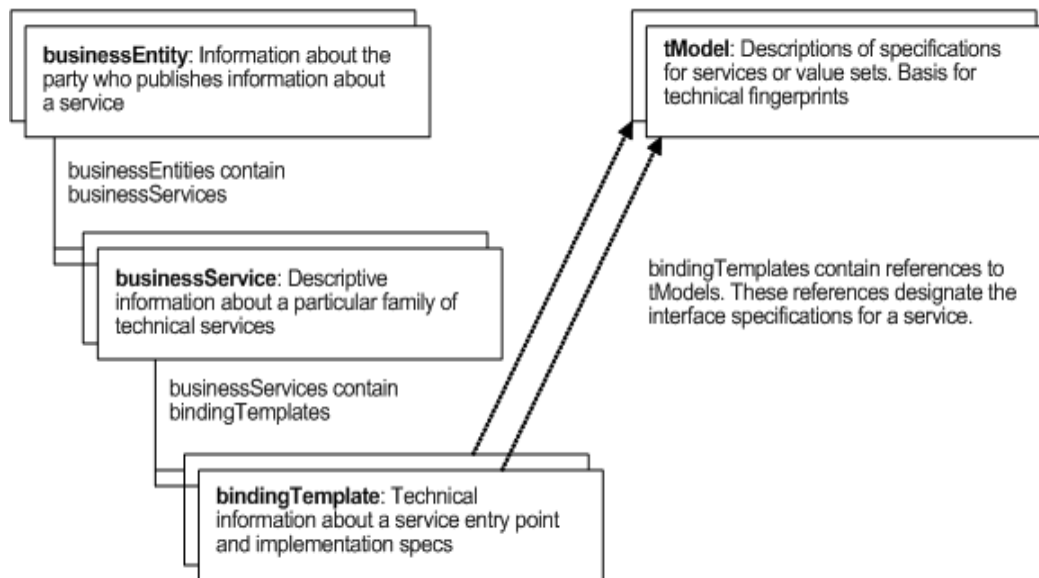


Figure 4 UDDI Core Types [4]

A business service module represents the resources provided by the business entities in more detail. A business entity can represent multiple business services. A business service contains the names and descriptions as well as the set of categories that represent the business service features and qualities such as a version number. It should be noted that a business service does not necessarily represent a Web service, and other services such as Common Object Request Broker Architecture (CORBA) [27] based services can also be represented.

A binding template contains technical information about the services offered. The binding template contains information about the web service entry point and references to tModels. A business service can contain one or more binding templates while the binding template structure has a single logical business service parent.

The tModel is used to represent each distinct specification, transport, protocol or namespace. tModels enable the interoperability of Web Services and examples include

those based on WSDL, XML Schema Definition (XSD), and other documents that specify the interface that a Web Service may choose to comply with [4].

Many businesses have diverse descriptions and they might not be able to be encapsulated in one business entity and hence raised the need of defining the publisher assertion structure. This structure provides a reference to the two related business entities by using fromKey and toKey elements. The fromKey element indicates the location of the business entity that created the publisher assertion while the toKey element points to the location of the business entity which has accepted the relationship. This structure is created by a service provider and if the two business entities are owned by different providers, the other service provider needs to accept the relationship otherwise the publisher assertion will not be created. If the business entities are owned by the same service provider, the publisher assertion is created without requiring an acknowledgment.

The operationalInfo structure provides information about the UDDI core data structures, that is the businessEntity, businessService, bindingTemplate, and the tModel structures. This information includes the date and time that the data structure was created and modified, the identifier of the UDDI node that is storing the information, and the identity of the publisher.

The UDDI centralized architecture is discussed as it provides the foundation for extension to distributed architectures and as such will be referred to in this thesis but will not be discussed any further.

The main disadvantage of using a centralized registry node to store all the service descriptions is that the centralized registry node provides a single point of failure. If the centralized node is not available, service requesters will not be able to discover any web

service information. In the Meta-Directory architecture proposed in this thesis, the system ensures availability as it provides multiple distributed Meta-Directory nodes. Another disadvantage of this architecture is that the system is not scalable as the number of services that can be discovered is limited by the amount of memory available in the centralized registry node. In our proposed Meta-Directory architecture, the system is highly scalable as new Meta-Directory nodes can be added to the system as the number of services provided increases. Due to the limitations of the centralized architecture, researchers started looking at deploying distributed registries. The following sections look at the deployment architectures when a system has distributed service registries.

2.1.2 Decentralized Architecture

The decentralized architectures discussed in this section such as [15] use peer-to-peer (P2P) approaches in setting up communication among distributed registries. All the peers in the system have equal responsibilities and they act as both service providers and registries at the same time as depicted in the example in Figure 5. This is because each peer has a local registry (for example the local WSMX (Web Service Execution Environment) registry) for storage of web service descriptions. In this example (Figure 5) the service requester sends the request to one peer and then the peer forwards the request to another peer when it does not have a response. The peer that can handle that request then sends the response back to the requester. These architectures eliminate the single registry bottleneck as well as improve scalability and ensure availability.

Hoschek [10] proposes a web service discovery architecture that is built on top of a grid based architecture. The discovery layer is composed of four interfaces as well as a

tuple based universal data model. These four interfaces, called the Presenter, Consumer, MinQuery and XQuery, are used for interaction among the peers.

- The Presenter interface is used by clients to retrieve the service description by the use of HTTP Get requests.
- The Consumer interface allows the provider to publish a content link which enables the consumer to retrieve the current content. The content link is only a dynamic pointer to the location of the service description.
- The MinQuery interface provides the simplest possible query support that returns the full set of available tuples in the tuple space.
- The XQuery interface on the other hand provides XQuery support by executing a given XQuery to cover the available tuple space. XQuery is a query standard developed by the XML Query working group of the World Wide Web Consortium (W3C) that is designed to query collections of XML data. It is a means to extract and manipulate data from XML documents or any data source that uses a tree-structured model such as relational databases or office documents.

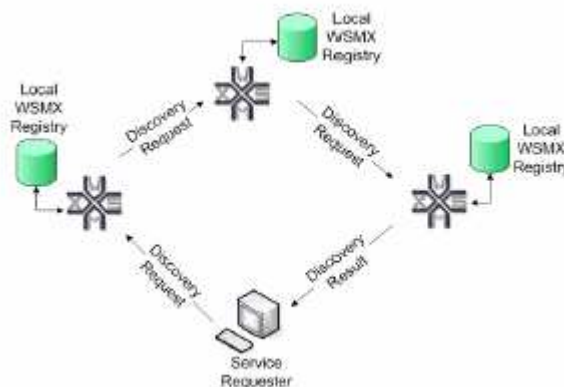


Figure 5 An Example of a Distributed Architecture [25]

Each peer can either support all or a subset of the available interfaces. A peer that only searches for web services might only support the Consumer interface while a publisher peer might only implement the Presenter interface. The registry peer on the other hand may implement all four interfaces. This approach ensures availability as there is no single point of failure but it incurs a high set-up and maintenance overhead on the four different interfaces.

Schmidt and Parashar [20] propose an indexing scheme that associates each data element with a sequence of keywords and uses the Hilbert Space-Filling Curve (SFC) approach in mapping the data element keyword space to the index space in order to preserve locality. This is similar to existing data lookup systems such as the CHORD [22] lookup protocol in that it enhances the lookup protocol by providing flexibility in keyword searches. In a DHT, the keys are partitioned among participating nodes and messages can be efficiently routed to the unique owner of any given key. The CHORD architecture treats nodes as points in a circle where the keys are evenly distributed within the nodes. Usually, in such distributed systems that are based on hash tables, a hash function is applied to the key and maps element identifiers to indices without the notion of locality.

The authors of [20] go further in performing simulations of their architecture. The simulation results show that their approach provides scalability as well as minimizes message overhead and number of nodes involved during query processing as the nodes have a sense of locality. The main disadvantage to this approach is the extra processing required to set up the Hilbert's SFC on top of managing the CHORD overlay P2P network.

Banaei-Kashani et al. [11] introduce a fully decentralized and interoperable discovery service with semantic-level matching. Each peer is composed of two components, the local query engine and the communication engine:

- The communication engine provides an interface to the user and peers and handles requests from the users and its neighbors in the peer-to-peer network.
- The local query engine handles the queries received from the communication engine and searches for matching services within the local site or registry.

The authors also introduce a time-to-live value that is used when queries are forwarded to the neighboring peers. This time-to-live value is set by the peer where the request originates from, and is decremented when the request is forwarded to another peer. The peers stop forwarding the request when the time-to-live value reaches zero. DAML-S is used for semantically enriching the web service descriptions. This is used in the communication engine when it receives responses from the local query engine and only returns the relevant responses based on semantics.

The main advantage of this approach is that it reduces the flooding of messages in the network by ensuring that only semantically matched responses are returned. But since the architecture is based on the Gnutella [9] peer-to-peer framework, this framework introduces a high level of message overhead because the queries are forwarded throughout the network and only the number of responses are minimized by using semantics. The time-to-live mechanism limits the number of responses received hence it is possible to miss relevant responses.

Sapkota et al. [17] propose the use of distributed shared virtual spaces (in the form of tuple spaces) for the publishing and discovery of web service descriptions, as illustrated

in Figure 6, in addition to the web service repositories such as UDDI. The web service descriptions and user's requirements are described using a Resource Description Framework (RDF) data model. They propose a number of components that will be used within the shared space of the discovery architecture. These components include the discovery manager which handles interaction with the providers and clients through an interface, the query parser, the virtual space reader and writer, the result filter or matchmaker, which matches the results obtained with the user's request returning only the most appropriate responses and finally the storage space itself which stores the resource descriptions.

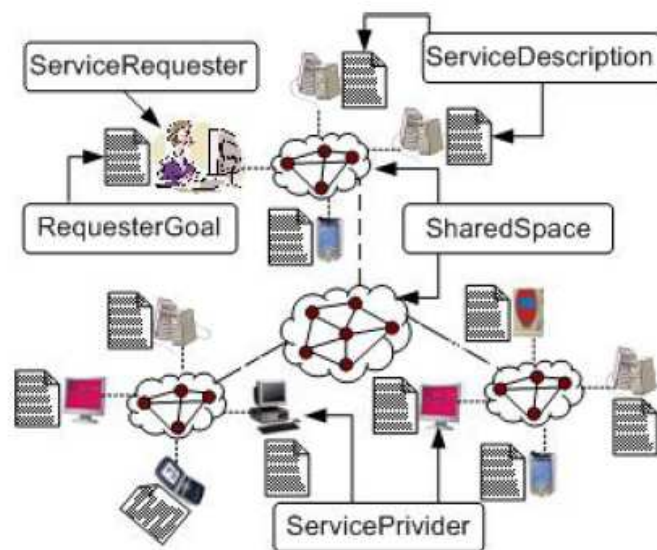


Figure 6 Virtual Space Based System [17]

In Figure 6 a service requester sends a request goal to the shared space using the discovery manager. In this case the shared space does not have the information for the required service and it forwards the request to the neighboring shared space, which then forwards the request to its neighbors. Once the required information is found, the response is sent back to the requester.

The authors claim that their approach is scalable and reliable as well as provides support for resource limited devices such as mobile phones without providing any experimental nor simulation results. The main drawback of this approach is the overhead incurred in the creation and maintenance of the virtual shared spaces on top of the maintenance of the information within the registries themselves.

Toma et al. [25] propose an architecture where all the registries act as peers with equal responsibility. Clusters are created based on the concepts of service and domain ontologies. The topology within the cluster heads is based on a Hypercube P2P approach that is adopted from [19].

This topology addresses the message routing overhead where the nodes are arranged in a hypercube which is a generalization of a 3-cube to n (the number of nodes) dimensions. Intra-cluster, they employ a keyword based search approach that is forwarded to all peers while P2P search algorithms are deployed inter-cluster due to the Hypercube topology. The cluster heads are organized in a Hypercube architecture.

The approach appears promising however test results would be required to better judge the efficacy. The disadvantage is that they use a timeout value on requests, which is set by the peer where the request originates from, and is decremented when the request is forwarded to another peer. The peers stop forwarding the request when the timeout value reaches zero. This limits the discovery scope hence the best services may not be discovered. Also, the underlying P2P architecture leads to limited scalability due to the flooding nature of queries and cannot include resource limited devices. The underlying communication model between the service registries also has to be modified.

Overall, there are three main drawbacks to the decentralized architectures described here. All nodes in the network have equal responsibility and hence the architecture does not allow the existence of heterogeneous nodes such as nodes with less computing power. The second disadvantage to this approach is that it incurs a high communication overhead in large networks as messages might be potentially sent to all the peers. In the Meta-Directory system proposed in this thesis, requests are only forwarded to the service registries that have the requested information. The third disadvantage to these architectures is that they all incur a high maintenance overhead. Our proposed Meta-Directory system does not incur maintenance overhead as the data between the Meta-Directory nodes is managed seamlessly without requiring any other input from the user after system set-up.

Due to the drawbacks of the centralized and decentralized approaches, researchers started looking at providing architectures that incorporate features of both the centralized and decentralized mechanisms so as to inherit the advantages of both architectures. This is done by introducing super peers or root peers that provide a form of infrastructure within the nodes while peer-to-peer operations are maintained within the clusters. The following section discusses the literature that incorporates root peers for management of distributed service registries.

2.1.3 Meta-Directory Architecture

Meta-Directory approaches combine both the centralized and decentralized approaches so as to inherit the advantages of both architectures. These architectures introduce the notion of root peers or super peers which contain meta-information

regarding the registries. In the literature that has been reviewed for this thesis, three types of indexing were performed.

- In the first category, all the distributed registries forward their locations to the super peer node. Whenever a new node joins, they register with the root node and the root node informs them of the locations of the rest of the registries in the network. In this distribution, requests are forwarded to all the registries in the network as there is no specification to the services offered by each registry. The root nodes merely store the locations of the distributed registries.
- In the second category, the registries are categorized into groups where each service registry only stores information for a specific category. In this model, requests are forwarded to the registries that provide the type of services associated with the query.
- In the third category, the registries forward keywords associated to the services that they offer to the root nodes. In this type of indexing, the requests are only forwarded to the nodes that provide the service that is being queried.

The super peers provide transparent registry access while the web service provider and requester are not aware of the distributed nature of the system. “The process of registration and discovery is similar to the one done in a centralized scheme, except in this case communication overhead is incurred among the super peers when the search includes several distributed registries” [7] . These hybrid architectures are proposed so as to reduce the bottleneck of using one public UDDI, ensure availability, and improve scalability when compared to centralized approaches. When compared to decentralized approaches, the hybrid architectures reduce management overhead as now the

management cost is only applied to the super peers and since all the nodes do not have equal responsibility, this architecture allows the existence of nodes with less computing power.

In this section, two categories of Meta-Directory architectures are discussed. One category depends on one dedicated super peer to ensure that the architecture can provide all the required functionality while the other architectures introduce the notion of multiple super peers.

2.1.3.1 Centralized Meta-Directory Architecture

In a Meta-Directory architecture, the system has root registries that index information regarding the distributed registries. This section discusses literature that uses a single dedicated node to store the information regarding the multiple registries.

The *Managing End-To-End Operations using Semantics METEOR-S* [26] project provides an implementation of a distributed registry structure composed of four different types of peers as seen in Figure 7. The gateway peer manages the access to the peer-to-peer network for new registry operations by indexing all web service registries. It is a central entity that also informs the other peers in the network as soon as registry updates are necessary. In this case, the gateway peer indexes the locations of the service registries, so that when a new registry peer joins the network, they can find out about all the other registries in the network.

Registry updates occur whenever an operator peer joins or leaves the network. The operator peer controls a local registry as well as acts as a provider of the registry ontology. The registry ontology captures the relationships among registries based on their affiliations and domains. The auxiliary peers only act as providers of registries ontology

but do not have control over any registries. Finally the client peers are only instantiated to allow users to use the capabilities of the METEOR-S architecture.

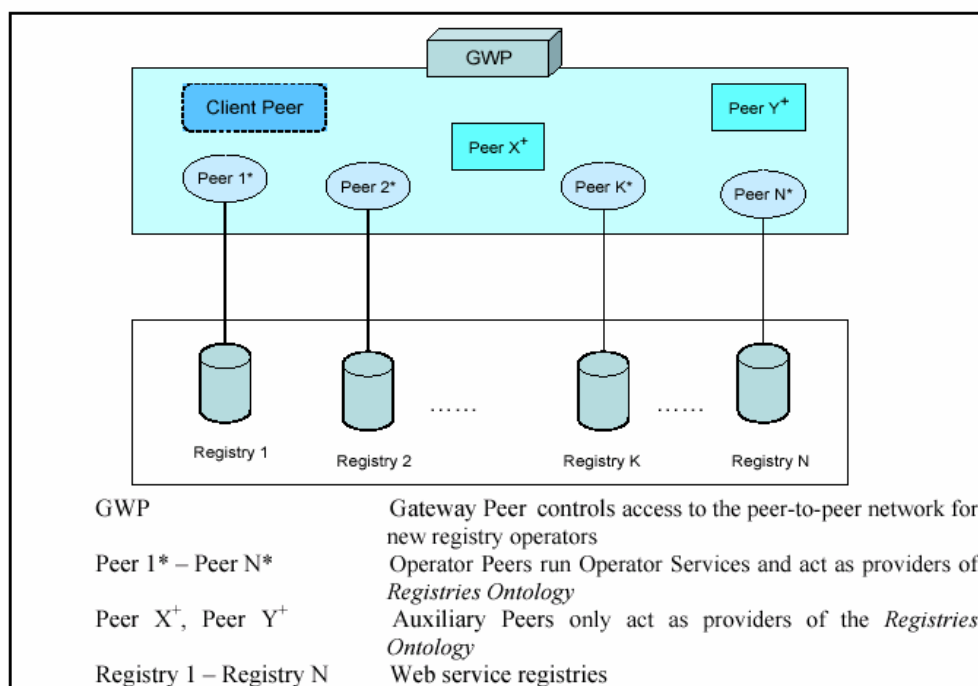


Figure 7 METEOR Communication Layer Overview [26]

This architecture is scalable as only the gateway peer is responsible for managing the system. The gateway peer unfortunately also acts as a single point of failure, even though it does not impact the publishing and discovery of web services, new registries will not be able to join the network if the gateway peer fails. Also, since the registry ontology needs specific maintenance and management, the organization of the registries is not trivial.

Another drawback of this approach is that when a request is received by the network, the query is propagated to all the registries as there is no classification on the types of services offered by each registry. This drawback is addressed by the following papers where they provided a classification for the types of services handled by each registry.

Du et al. [6] propose an active and distributed (Ad-UDDI) registry architecture as seen in Figure 8, whereby the registry information is distributed among multiple registries. Their architecture is based on a root registry in the Root Registry Layer, which is in charge of managing the active and distributed registries and does not store any data related to the services themselves. The root registry in this case stores the type of services that each registry is providing.

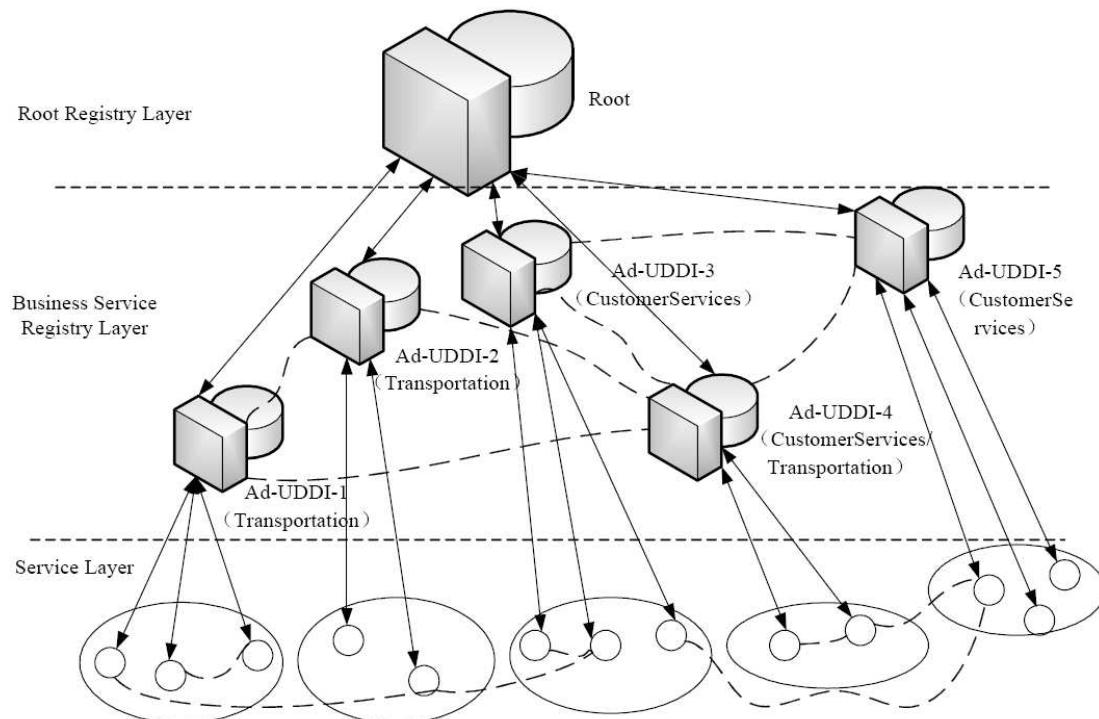


Figure 8 The Ad-UDDI Distributed Architecture [6]

The business services are handled by the intermediate registries in the Business Service Registry Layer which are classified based on the business service they handle, as can be seen in Figure 8 where there are two business services: Transportation and Customer Services. These intermediate registries have to register with the root registry as a web service, and they establish neighborhood relationships with other intermediate

registries which handle the same business classification. In Figure 8, Ad-UDDI-5 has a relationship, signified by the dotted lines, with Ad-UDDI-3 and Ad-UDDI-4 since they all deal with Customer Services. The Service Layer is composed of the web service providers and requesters. They also propose an active monitoring system on the registries which monitors and updates the registry entries. They argue that passive registries based on UDDI are prone to outdated information hence it is better to periodically ping the service providers on their registry information. Their simulations confirm that Ad-UDDI performs better than using a single UDDI registry.

Their approach ensures the real time validity of the registry information. But it still suffers from a single point of failure, as new UDDI registries are not able to join the network if the root registry is not available.

Papazoglou et al. [13] propose a service syndication protocol whereby related businesses form groups of interest with their own UDDI peer registries that operate in a decentralized peer-to-peer fashion (Figure 9). They also define the existence of super peers which store a sub directory of a UDDI business registry using the Syndication UDDI where each service peer would publish their service description. These super peers register with a central UDDI registry which handles discovery among the super peers. In this case, the meta-information that is stored in the central UDDI registry provides the type of services that are offered in the service syndications.

The super peer manages the communication among service peers and is responsible for the joining and leaving of peers of service syndications. These service peers are the actual web service providers. They also provide an event mechanism for publish-subscribe features which allow service peers to function in an independent way. This

enables the formation of peer acquaintance groups which consist of peers having the same interests, and each knowing every member in the group. The peers in the same group can propagate web services within their own group without involving the super peer.

Once again, this architecture provides a single point of failure since if the super peer crashes, new nodes will not be able to join the peer groups. Another drawback is that no mechanism is available for getting the real time status of services.

All the proposals that provide a single dedicated root registry node for storing information about the services provide a single point of failure. This limitation was also seen by a number of researchers who then proposed a distributed network of root registries that provide meta-information about the distributed service registries. These proposals are discussed in detail in the following section.

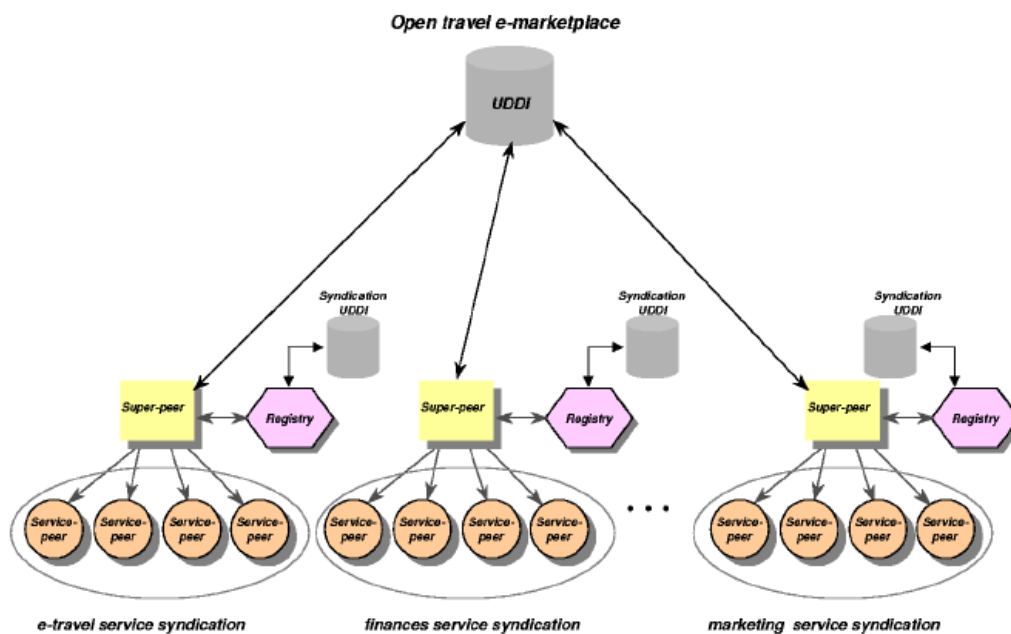


Figure 9 Web Services Syndication Overview [13]

2.1.3.2 Distributed Meta-Directory Architecture

Due to the limitations of a centralized Meta-Directory architecture, a number of researchers proposed an architecture that involves a distributed network of Meta-Directories.

The authors in [18] propose a clustering approach (see Figure 10) based on similar web service descriptions. Example of clusters are indicated by the light dotted circles in Figure 10 where the services are classified in terms of bus service, hotel service, air service, and train service. The clusters are maintained by what they refer to as “super nodes”, indicated by the heavy dotted circle in the figure. These super nodes are dynamically elected based on their availability, available storage and processing power.

The registries within the cluster register with the super node and the super node maintains communication within its cluster and with other cluster heads. In this paper, the information stored in the super nodes indicates the locations of the registries that provide the same service. The authors also propose a goal decomposition mechanism which is handled by the cluster heads. If a client’s request needs to be handled by multiple clusters, as the client might require multiple services, this request is decomposed by a cluster head and the sub query instead of the full query is then forwarded to the appropriate cluster head(s). This reduces the message overhead in large networks.

The authors claim that their approach enables load balancing, discovers web services faster, requires less invocations, and is more scalable [18]. The main disadvantage of this approach is that it incurs maintenance overhead during the dynamic creation and maintenance of cluster heads. Also, a particular service provider can only be a member of one cluster and cannot provide services in more than one category.

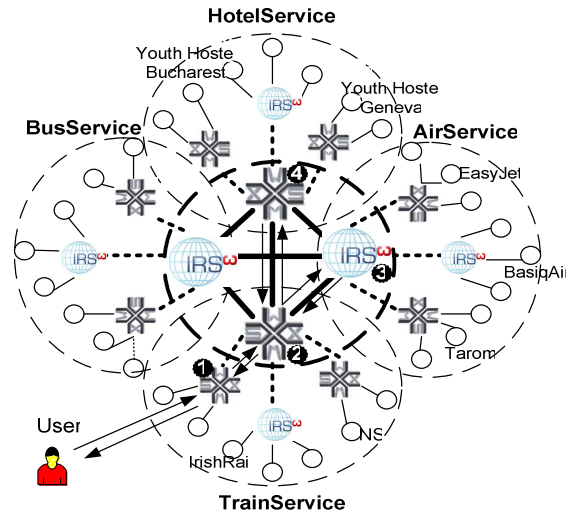


Figure 10 Clustering of Web Services [18]

The *Sahin et al.* [16] approach is based on a super peer network that is built on top of CHORD (the CHORD distribution model will be discussed in detail in Section 3.3.1) as the distributed hash system (Figure 11). Here, the super peers are elected based on availability and computing power allowing the existence of mobile nodes with limited resources as the client peers. Therefore, this model allows the existence of heterogeneous nodes. The communication among the super peer nodes is handled by the CHORD system and it ensures that nodes are located in $O(\log N)$ hops, where N is the number of nodes.

The client peers are directly connected to the super peers. Services are advertised using keywords and information on locating the service provider, the keywords are hashed and forwarded to the super peer responsible for storing that keyword. This approach stores the keywords for all the services offered by the client peers. Queries are sent from the client peers to super peers who then forward it within super peers using the CHORD hash function which ensures that the request is forwarded to the appropriate

super peer, this in turn reduces the message routing overhead that would be incurred in peer networks without any form of infrastructure such as [20]. The architecture does not require a central administration as the peers are selected dynamically.

This architecture has two main advantages, since only the super peers form the CHORD ring; the routing and joining/leaving costs within the ring are reduced. The second advantage is that it can handle nodes with heterogeneous resources as only the super peers are expected to be able to handle routing and message forwarding capabilities. The main disadvantage of this approach is that it incurs maintenance overhead during the dynamic creation and maintenance of super peers. Also, there is no discussion on the number of super peers required per network size nor a discussion on the number of client peers per cluster as there is no form of cluster classification like the one we have seen in [18] where clusters are based on similar services. Another drawback to this approach is that the service providers themselves form the CHORD ring hence the underlying communication model between the service providers needs to be modified. This limitation is addressed by [1] where the underlying communication model between the service entities is not modified.

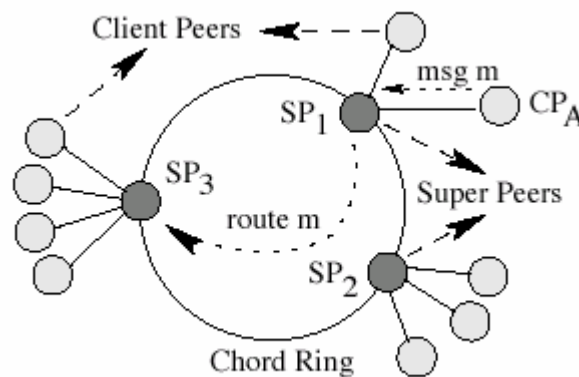


Figure 11 SPiDER Architecture [16]

Banerjee et al. [1] propose an architecture that is based on grid services and UDDI registries. The scalability of the UDDI registries is improved by using Distributed Hash Tables (DHT) as a mediation among multiple UDDI registries. A distributed hash table maintains a collection of key-value pairs that are used for publishing and discovery. For the deployment in [1], the key is a hash of a keyword from a service name or description. There are multiple values for a single key, one for each service containing the keyword. Nodes are given an identifier by hashing either their IP address and port number, or public key.

They propose an architecture that is composed of proxy registries (Figure 12) that mediate the communication between the local UDDI registry and the DHT service. “The DHT service is the glue that connects the proxy registries together and facilitates searching across registries.” [1] The proxy registry is in charge of publishing, deleting and searching information in the UDDI registries. The clients communicate with the proxy registries which then delegate the request to the DHT service. The DHT provides information about the relevant registries hence query operations are only propagated to relevant registries. The process is accomplished by simply looking up the hashed key instead of using a set of search parameters.

This architecture is able to support large scale discovery of web services as the search operation is only done on the UDDI hash values returned by the DHT service instead of searching in all UDDI registries. The main disadvantage of this architecture is that the proxy services provide a single point of failure since if they break down; no operations are possible within the network.

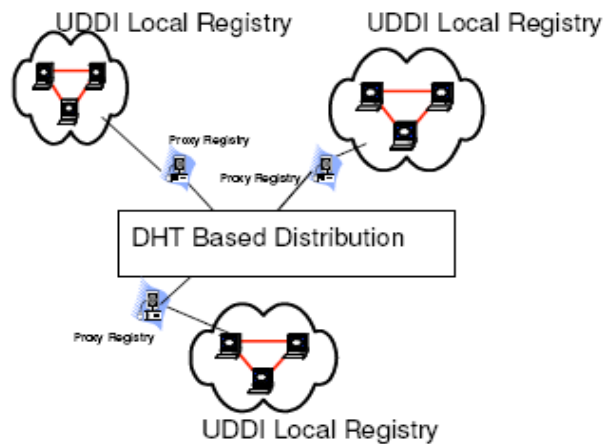


Figure 12 Distributed Hash Table Architecture [1]

Architectures that have nodes which are required to be available provide a single point of failure while the ones that dynamically select super peers ensure availability regardless of which nodes leave the network. Cluster based approaches on the other hand do not provide a single point of failure and also provide an added advantage to the other approaches as they allow the existence of nodes with limited computing power such as mobile nodes. The following section discusses the properties in terms of availability, scalability, ease of management and transparency of the centralized, decentralized and hybrid architectures in more detail.

2.2 Discussion

This section provides a discussion on the previously described web service architectures in terms of availability, scalability, ease of management and transparency.

2.2.1 Availability

Availability in terms of fault tolerance, which is the ability of the web service architecture to continue with normal operation despite the presence of faults, is discussed.

- Centralized registry architectures are simple but they give rise to a single point of failure as well as a performance bottleneck since all requests must be handled by the single registry. To solve the problem of fault tolerance, a replication strategy can be implemented. In the paper by *Sun et al.* [23] two replication schemas: UDDI replication specification and Middleware replication, were implemented and compared. The UDDI replication protocol can be seen as a lazy replication strategy as the nodes periodically exchange status information and if the receiving node is missing some information then it sends a record request to the sender. The middleware replication strategy employs a group communication system that supports group maintenance and reliable multicast. Whenever a registry is updated, it will immediately multicast the update request to all sites. Further discussion of these strategies is beyond the scope of this paper.
- Decentralized registry architectures ensure the availability of the service as all the registry nodes have equal responsibility. The nodes can also join and leave the network dynamically without disrupting the service. Failure of a registry peer does not affect any other peer as each peer acts as a registry node.
- The Meta-Directory approaches inherit the single point of failure disadvantage due to the existence of super peers as well as retain some peer functionalities since the other nodes in the network maintain peer-to-peer communication. The Ad-UDDI [6], METEOR-S [26], and the federated approach [13] depend on a centralized node for the management of the system. With these architectures, if this root registry fails, publish and discovery operations will still be available but new registries will not be able to join the network and the other nodes will not be

informed if a registry fails or decides to remove itself from the system. The distributed hash table approach using registry proxies proposed by *Banerjee et al.* [1] on the other hand would cause the partitioning of the network if the proxy registries fail, since if a proxy registry fails it causes the isolation of that part of the network. The cluster approach proposed by *Sapkota et al.* [18] and the SPiDer approach [16] are more flexible as the super peers are dynamically selected. Hence they do not provide a single point of failure as a new super peer will be dynamically selected by the system if the current super peer fails.

Hence, the architecture that provides the best availability is based on the decentralized approach. But there is a trade-off between availability and communication overhead when using the decentralized approach. The Meta-Directory architecture in which the super nodes can be dynamically selected also provides good availability.

2.2.2 Scalability

Scalability is discussed in terms of the ability of the architecture to handle increasing load as well as an increase in the number of web service registry entries. The load in this case is the increase in the arrival rate for search queries per unit time. This section compares the scalability of the three types of architectures discussed.

- The centralized approach provides limited scalability as all the web service descriptions are stored in a single entity. The number of entries directly depends on the storage capacity of the registry entity. Similarly, the query arrival rate that the architecture can handle at a time also depends on the computing capacity of the registry entity. A replication schema can be implemented to solve the problem of scalability by having several servers which offer a replicated strategy. The

drawback to this approach is that replication requires data consistency to be maintained and introduces an administrative overhead.

- Decentralized approaches offer a scalable solution as the load of the registry entries are distributed among the peers leading to increased performance when the web service application grows. There is a trade-off between scalability and the communication overhead in decentralized architectures. This is because in some decentralized architectures, when a query is sent out, it is potentially broadcast to all the nodes and the results have to be propagated from all the nodes back to the originating query node. However, if the result is in the local registry it reduces the search time when compared to the centralized approach as there are fewer entries to search from since the registries are smaller.
- Meta-Directory architectures also offer a scalable solution since they inherit the properties of the decentralized architecture. Communication overhead is reduced due to the existence of dedicated super nodes. Hence, during a global search, the search queries are only forwarded to the super peers, each of which then decides if its cluster is involved with the query. If not, the query request is not forwarded to the peers within the cluster. In the case where the clusters are arranged according to the type of web service descriptions stored (see [18] for example), the communication overhead is reduced further as the query requests are only forwarded to the appropriate super peers.

Even though decentralized approaches provide the most scalable solution, there is a trade-off with communication overhead as requests are forwarded to multiple nodes in the peer network. The Meta-Directory architectures provide a better solution in terms of

scalability as they reduce the communication overhead but in turn they introduce an additional administrative overhead.

2.2.3 Ease of Management

We look at the ease of management in terms of administration of the architecture:

- Ensuring that important entities of the architecture are always available;
- Maintenance of business federations in terms of managing the clusters and syndications;
- Routing and communication between the peers and super peers in case of hybrid architectures discussed that use super peers.

Centralized architectures are simple to set up as only one entity is required to be monitored. If there is no replication strategy involved, there is also no coordination or replication of information exchanged between registries hence greatly reducing the administrative costs.

Decentralized architectures that are discussed in this report incur no administrative overhead as registries can join or leave the network without requiring any registration. A registry does not need to know about a central registry or super peers, it only needs to know an arbitrary peer in order to publish or search for a web service. Although decentralized architectures do not require any management, there are communication overhead costs that have to be dealt with. These communication overhead costs are incurred when a search query is executed since potentially the message might be forwarded to the whole network.

Meta-Directory architectures on the other hand require administrative costs as registries need to know about the network topology, i.e. the location of the super peers

and which cluster they belong to. These super peers also need to be monitored to ensure that they are always available, otherwise some if not all web service functionalities will not be available.

2.2.4 Transparency

The availability, scalability and management properties of the registry architectures have been covered in the previous sections. This section explores the possibility of an architecture to provide transparency. Transparency in this case is the ability of the architecture to manage the system without having to modify the existing architecture within the registry entities. In this section, we will also introduce the notion of configurability which is the ability of the architecture to allow different configurations of the system during system set-up. By different configurations, we are referring to the communication model between the entities used in the management of the distributed service registries.

The centralized architecture provides transparency as the underlying architectures of the registry entities (i.e. service provider, service requester, and service registry) are not modified since there is only one service registry in the network.

In the decentralized architecture, the underlying architectures of the service registries need to be modified as the service registries now must perform peer-to-peer operations between them. This increases the management overhead between the service registries. The underlying architectures of the service requesters and providers on the other hand are not modified as they are provided with a single registry view by any one of the peer registries.

In the Meta-Directory architecture, the only approach that provides transparency is the one by *Banerjee et al. [1]*, where the meta-information is stored in a DHT distribution. The rest of the approaches require the service registries to register themselves with a root registry as well as maintain peer-to-peer operations between themselves. The approach by *Sahin et al. [16]* goes further by modifying the communication between the service providers themselves.

In terms of configurability, no single proposed architecture that has been reviewed provides the functionality that has been described.

Table 1 provides a summary of the characteristics of all the previously discussed existing approaches to the management of distributed Web Service registries, in terms of availability, scalability, ease of management and transparency. The X's indicate the property supported by the respective approach.

Table 1 Evaluation of Systems

Approaches	Availability	Scalability	Ease of Management	Transparency
Centralized			X	X
Decentralized	X	X		
Meta-Directory:				
<i>Centralized</i>		X		
<i>Distributed :</i>				
Sahin et al. [16]	X	X		
Sapkota et al. [18]	X	X		
Banerjee et al. [1]		X	X	X

As we can see from Table 1, the approach that provides most of the properties ideal for a system managing a network of distributed service registries is the proposal by *Banerjee et al. [1]* which is implemented using the Meta-Directory approach. The Meta-Directory systems introduce nodes that store meta-information on the distributed

registries. Due to these advantages, the registry management approach used in this thesis uses the Meta-Directory approach.

The Meta-Directory approach used in this thesis offers scalability, adaptability, ease of management, as well as flexibility. The system proposed in this thesis also allows the choice of the right configuration during system set-up for handling a given network state such as the size of the network, the system load, and ease of management. Another advantage of the proposed Meta-Directory approach is that the underlying architecture of the service entities is not modified. The following section discusses in more detail the state-of-the-art that has some similarities with our Meta-Directory approach and highlights the differences and the advantages of the system proposed in this thesis to the state-of-the-art.

2.2.5 Meta-Directory System

The previous section discussed the approaches currently used in the management of distributed registries. The approaches discussed by *Papazoglou et al.* [13], *Du et al.* [6], *Verma et al.* [26], *Banerjee et al.* [1], *Sapkota et al.* [18], and *Sahin et al.* [16] in the state-of-the-art introduce the Meta-Directory idea where information regarding the distributed registries is stored in order to facilitate the querying of distributed registries.

The Meta-Directory systems reviewed in this thesis store meta-information regarding the distributed registries in three different forms. The meta-information can either be on the location of the distributed registries such that when new registries join the system, they can get the location of all the registries in the network. The second form of meta-information is on the types of services offered by the distributed registries. This type of distribution is used in a network where the registries are classified based on the services

that they offer. Finally, the third type of meta-information is based on the actual services stored in each registry. In this system, the keywords associated with each service are advertised to the root registries thereby facilitating querying based on the actual service.

There are two architectures of Meta-Directory systems which are the centralized system that uses only one Meta-Directory node and the distributed system which uses multiple decentralized Meta-Directory nodes. This section analyzes these approaches in detail and points out their disadvantages.

2.2.5.1 Centralized Meta-Directory Approach

In a centralized Meta-Directory approach, the literature introduces a centralized node that provides information regarding the distributed registries. The approaches discussed by *Papazoglou et al.* [13] and *Du et al.* [6] introduce a centralized UDDI node where each of the distributed registries have to publish itself in a centralized node indicating the type of services that are offered by the registry. The approach by *Verma et al.* [26] also uses a centralized node; but in this approach the centralized node only stores the location of the registry without any notion of the type of services offered by the registry. Figure 13 illustrates the basic structure used in these approaches. The main disadvantage of this approach is that the root registry provides a single point of failure. This is because if the root registry is not available, new UDDI nodes will not be able to join the network. Also if the root registry is not available, the appropriate UDDI node cannot be located and as such all the UDDI nodes will have to be queried during a service discovery.

Due to the single point of failure limitation, researchers started looking towards solutions that involve more than one meta-directory. The following section summarizes the literature that use distributed meta-directories to store meta-information.

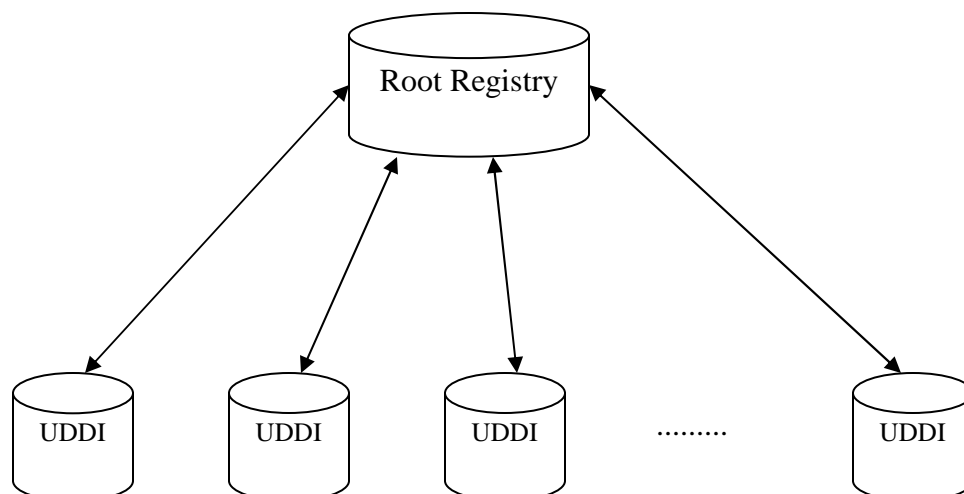


Figure 13 Centralized Approach

2.2.5.2 A Network of Meta-Directories

Due to the single point of failure drawback of the centralized Meta-Directory approach, researchers proposed the use of a network of Meta-Directory nodes. The approach introduced by the authors in [16] uses super peers whereby the client peers publish the services that they offer. The Super Peers in [16] are the actual service providers in the distributed Web Service Discovery Mechanism shown in Figure 11. This means that the underlying communication model used by the service providers has to be modified to support the indexing algorithms provided by CHORD.

The authors in [16] abstract away from using any service registry. In [16], the service providers advertise their services to the Super Peers that index the service keywords in the DHT (as discussed in detail in Section 2.1.3.2) This forces the approach in [16] to be a specific solution to a service discovery mechanism and cannot be applied to an already existing system that uses UDDI compliant service registries for example.

The authors in [18] on the other hand introduce a clustering approach where registries that provide the same type of service form a cluster and register themselves with a super peer. The super peer in this case only knows of the location of the service registries and there is no indication as to the actual services offered by the registries but only knows of the class of services that are offered. The main drawback of this approach is that it incurs a high maintenance overhead as the super peers are dynamically selected and a registry can only provide service information for one category.

The proxy registry approach introduced by *Banerjee et al.* uses the Bamboo (DHT) system [1]. In [1] each local registry is tied with a single proxy registry that links the UDDI registries with the DHT service. This proxy registry handles all the requests to the underlying local registry. This one on one relationship forces as many proxy registries as there are service registries in the network.

The main disadvantage of the approach in [1] is that the service provider must forward a publish request to the proxy registry associated with the local registry as shown in Figure 14. This property makes the approach in [1] provide a single point of failure because if a proxy registry is not available, information cannot be published in the local registry that is associated with the proxy registry.

As we have seen from the discussion on the proposals that use Meta-Directories, even though these approaches have drawbacks, they still provide better solutions in terms of scalability, ease of management, and flexibility compared to the rest of the literature in the management of distributed registries. Due to the advantages of the Meta-Directory approaches, our approach also uses Meta-Directory super nodes where we overcome the disadvantages of the reviewed literature. The following section introduces the Meta-

Directory system proposed in this thesis and discusses how we overcome the drawbacks of the existing research.

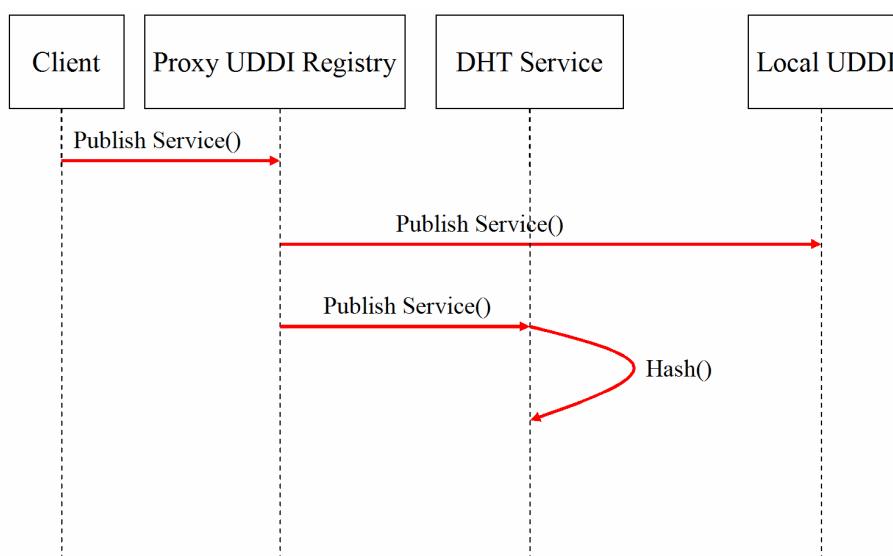


Figure 14 Service Publishing in Proxy Registry Approach [1]

2.3 An Introduction to the Solution: Proposed Meta-Directory Based Approach

Our approach introduces a Meta-Directory layer that has information regarding the services and registries that an enterprise wants to be publicly available in the network. The architecture used in this research is illustrated in Figure 15. In this example, each Meta-Directory node may be located on a Gateway in the distributed network. It should be noted that the Meta-Directory nodes can be located anywhere in the distributed network as they do not rely on any services from the Gateways.

A Service Provider is an entity that provides the Web Service in the distributed system while the Service Requester is the client that wants to invoke the Web Service.

The Gateway is the entry point to an enterprise's Service Registries. Information on how the Web Services can be invoked is stored in the Service Registries.

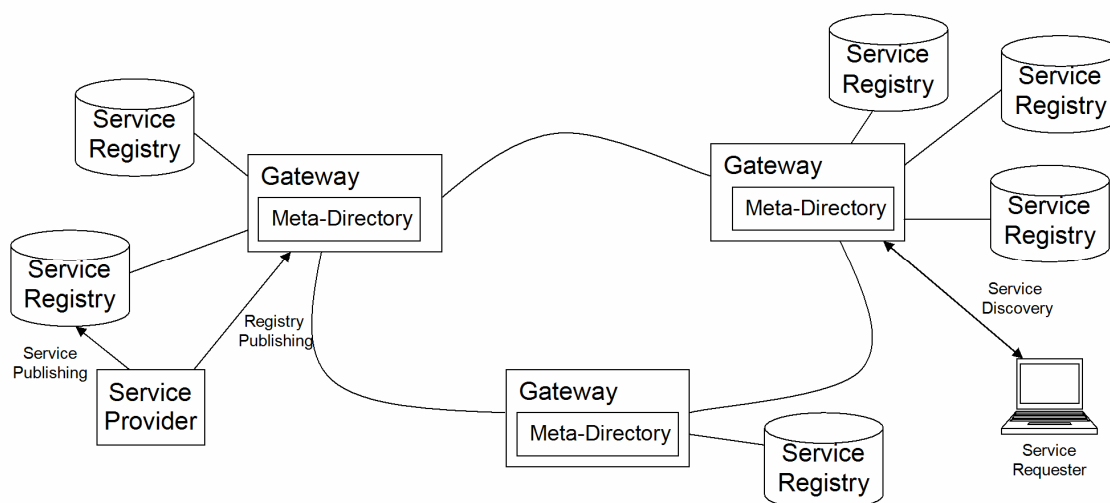


Figure 15 Distributed Meta-Directory Approach

When a Service Provider wants to publish Web Service information, the Service Provider first forwards the publish request to the Service Registry. Once the information is stored in the Service Registry, the Service Provider then forwards the publish request as well as the location of the Service Registry to one of the Meta-Directory nodes as shown in Figure 16. During a Web Service Discovery, the Service Requester only needs to send a service request message to one of the Meta-Directory nodes. The Meta-Directory nodes then collaborate and return the requested information to the Service Requester. The collaboration among the Meta-Directory nodes is determined by the data distribution model and the forwarding algorithm employed by the Meta-Directory nodes.

The distributed Meta-Directory approach is proposed because a centralized Meta-Directory approach introduces a single point of failure similar to the schemes introduced

by *Papazoglou et al.* [13], METEOR-S [26] and *Du et al.* [6] where the service information is published in a centralized UDDI registry.

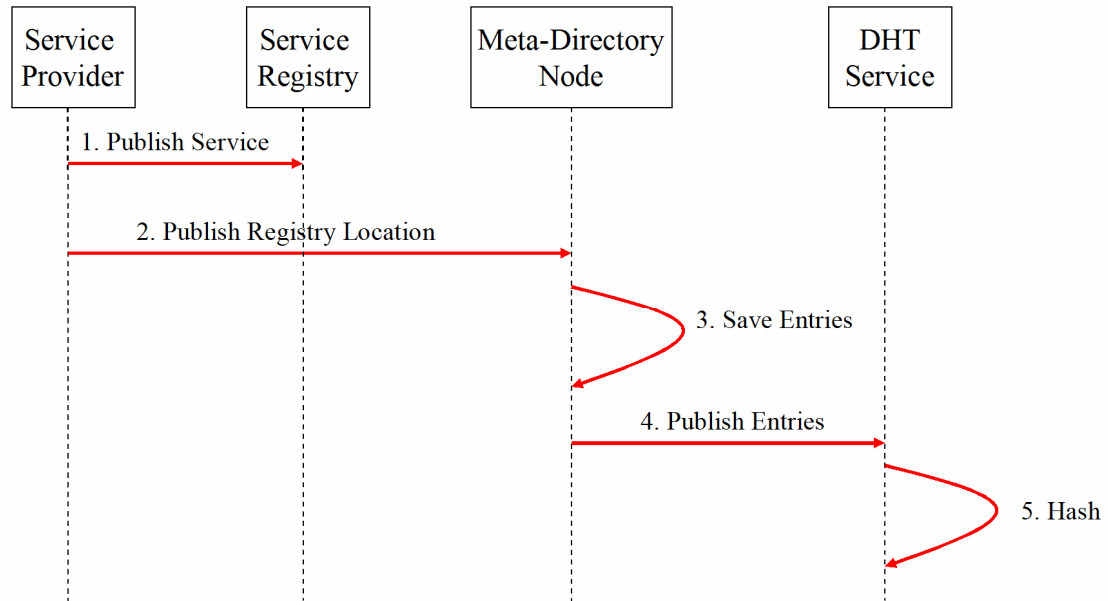


Figure 16 Service Publishing in Proposed Meta-Directory Approach

Compared to the SPiDer approach [16], the Meta-Directory approach on the other hand does not modify the underlying architectures of any of the Web Service discovery components (i.e. the service provider and the service registry). The Meta-Directory approach can be applied to any existing system that uses service registries which is not possible with the system described in [16].

When compared to the approach by [18] which only stores the locations of the registries based on the type of services offered by the registries, the Meta-Directory approach proposed stores the keywords of the actual services offered by the registries. In [18] when a query is received by the super peer, it has to be forwarded to all the registries offering that type of service, while in our proposal, the query is only propagated to the registry that provides that particular service. Also, the Meta-Directory approach does not

modify the registries themselves while in [18] the super peers are also service registries with extra responsibilities.

The architecture proposed in this thesis overcomes the proxy registry limitation in [1] by allowing the service provider to publish in the local registry of its choice, and then publishes the registry information in any Meta-Directory node. In our proposed architecture, Meta-Directory nodes are not tied to any specific registries and therefore they can be located anywhere in the network and a service provider can publish in any one of them regardless of the UDDI registry location. The approach in [1] forces as many proxy registries as service registries in the network, while in our approach the number of service registries does not affect the number of Meta-Directory nodes since the Meta-Directory nodes are not tied to any Service Registries.

Another advantage of this approach in the context of the Extranet architecture is that since the Meta-Directory nodes are located on the Gateway, this approach allows the existence of private registries within the network. It also accommodates existing local business registries thereby forming a network of registries. The service providers only forward the information they want to share with the network to the Meta-Directory nodes. Private entries will not be forwarded to the Meta-Directory node hence this architecture provides security to internal enterprise information.

Since the Meta-Directory nodes are distributed in the network and information is replicated within the Meta-Directory nodes, as shown in the hash table structure discussed in Section 3.3.2, this approach does not have a single point of failure. Information is distributed in multiple nodes that share the system load.

The distributed Meta-Directory model abstracts the distributed nature of the system from the service provider, by providing a single interface through the Meta-Directory system that takes care of the querying and forwarding of messages among the distributed Service Registries. Therefore the Service Requester does not have to know the locations of any registries and only needs to know the location of one Meta-Directory node thereby giving the illusion of a centralized system.

For the Meta-Directory architecture to provide better performance with varying network size and available bandwidth, the routing used for interconnecting the Meta-Directory nodes is configurable during system setup. This configurable framework is introduced in the following section.

2.3.1 Configurable Routing Framework

Multiple network configurations were analyzed in terms of their performance and scalability and it was discovered that no single communication protocol is suitable for networks whose system configuration can evolve with time. A configurable routing framework is hence introduced and implemented in this thesis. With this framework, the system administrator can decide which configuration is to be used. The various possible configurations include CHORD (DHT), Fully Connected, Fully Connected (DHT) and Super Peer. These are discussed in detail in Chapter 3 .

Further analysis showed that a framework that allows the system configuration to be modified during runtime would be ideal. Due to this, an adaptable framework and its underlying algorithms are also introduced in this thesis. The adaptable framework has not been implemented but counts towards the future areas of research for this thesis.

The next chapter looks at the overall design of the distributed Meta-Directory system by discussing the Meta-Directory node design in terms of the communication model, the hash table structure and the hash table data distribution model. The configurable routing framework is also introduced and analyzed by discussing the network models that are supported. The adaptable routing framework is then introduced by providing the transformation algorithms that would allow the system to be modified during runtime.

CHAPTER 3 SYSTEM DESIGN

This chapter looks at the overall design of the distributed registry management system that uses configurable distributed Meta-Directory nodes. From the discussion in the previous chapter, we have seen that there are three types of Meta-Directory systems. The Meta-Directory system that is proposed in this thesis saves information regarding the services offered by the distributed service registries. The system stores the attributes that are used when service information is queried from a registry.

There are two possible design models for the proposed Meta-Directory system. One design involves the creation of a client on the Service Registry and in the other design the client resides on the Service Provider. The two models are discussed in the following section followed by use cases and system design diagrams, and then the configurability of the network is demonstrated by describing the supported network models.

3.1 Design Models

In Section 2.3, the Meta-Directory approach proposed in this thesis was introduced. There are two ways in which the Web Service components (Service Provider, Service Requester, and Service Registry) can interact with the Meta-directory System. The first design that involves the creation of a client component on the Service Registry is discussed below.

3.1.1 Client Residing on the Service Registry

The Meta-Directory system proposed in this thesis can be realized by creating a client on all the Service Registries. These clients will be in charge of communication

between the Meta-Directory System and the Web Service components. Figure 17(a) shows the message distribution when a service is published in this model. The Service Provider forwards a *Publish Service* request to the Service Registry. The Service Registry stores the service information and then forwards the same request to the Meta-Directory System. The Meta-Directory System then saves the information in the Meta-Directory network.

Figure 17(b) illustrates the messages exchanged during a service discovery. The Service Requester forwards a *Query Service* request to any one of the Service Registries. The service registry receiving the request forwards the query to the Meta-Directory System which then finds all the Service Registries that have the information requested. The Meta-Directory System then forwards the *Query Service* request to all the respective Service Registries which return the response to the Meta-Directory System. The Meta-Directory System then returns all the responses to the Service Registry which sent the originating request which then forwards the response to the Service Requester.

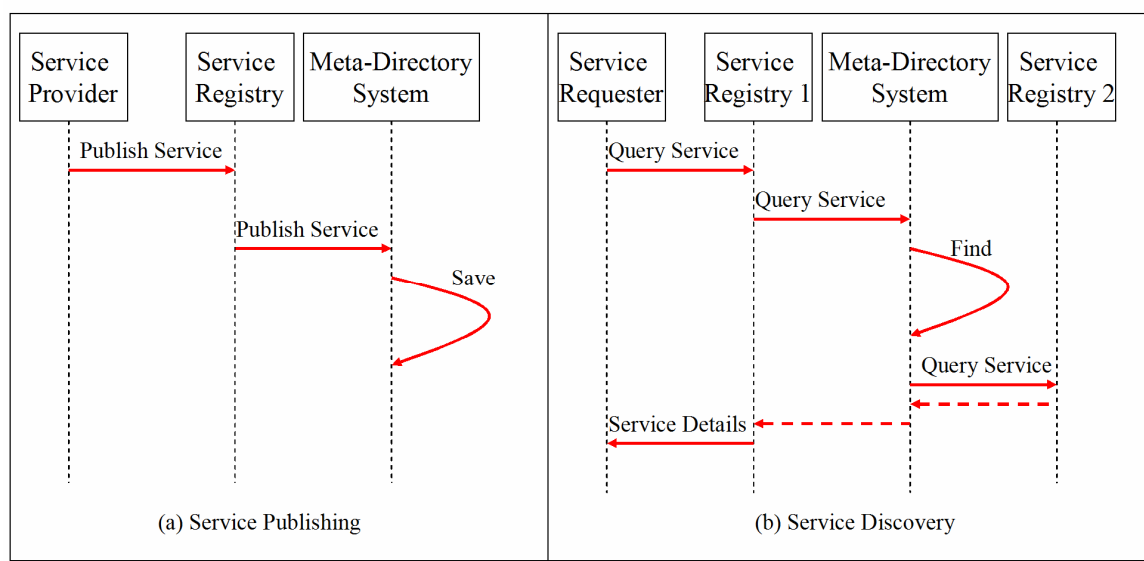


Figure 17 Sequence Diagrams for Service Registry Design

This model abstracts the existence of multiple Service Registries from the Service Provider as well as the Service Requester. The Service Provider publishes in a Service Registry of its choice and the Service Requester queries any one of the Service Registries. The following model introduces a client on the Service Provider.

3.1.2 Client residing on the Service Provider

A second model that can be used to realize the Meta-Directory system proposed in this thesis is by having a publishing client on the Service Providers instead. Figure 18(a) shows the message exchange when service information is published in this design model. The Service Provider first sends a *Publish Service* request to the Service Registry. Once a confirmation is received from the Service Registry that the information is saved, the Service Provider forwards a *Publish Registry Location* request to the Meta-Directory System. The *Publish Registry Location* message includes the original *Publish Service* request along with the location of the Service Registry the information is stored in. The Meta-Directory System then stores the information that can be used to query the service.

Figure 18(b) illustrates the message distribution during service discovery. In this model, the Service Requester forwards the *Query Service* request directly to the Meta-Directory System. Similar to the previous design, the Meta-Directory System finds the Service Registries that have the information requested and forwards the request to the Service Registries. Unlike the previous design, once the Meta-Directory System receives the response from the Service Registry, the system forwards the response directly to the Service Requester.

In this model, during Service Discovery, the query messages are only propagated the Service Registries which have responses for the query. While in the previous model, the

Service Registry that receives the request might not have the information requested but it must participate in the message exchange since the client resides on the Service Registry.

Any of these two designs can be used to analyze the performance of the proposed Meta-Directory system because once the system receives a request; the flow of messages is exactly the same. The system first finds the appropriate Service Registries and forwards the request to them. In this thesis, the model that was implemented was the one that involves the client residing on the Service Provider. This model does not require any changes on the Service Registry nor the Service Requester. The client is required on the Service Provider only when information is being published in the network. The rest of the thesis will refer to this model as the Meta-Directory system. The following section discusses the Use Cases of the proposed Meta-Directory system.

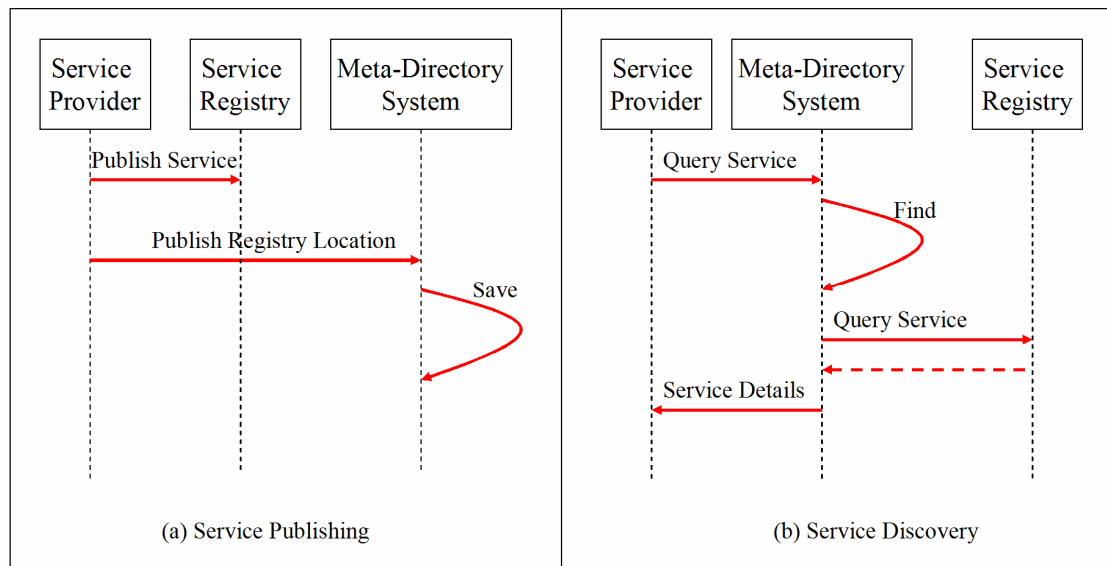


Figure 18 Sequence Diagrams for Service Provider Design

3.2 Use Cases

Figure 19 shows the use case diagram for the Meta-Directory System. There are four actors that interact with the Meta-Directory System, the ServiceRequester, ServiceProvider, SystemAdministrator and the ServiceRegistry. The ServiceRequester is the entity that is looking for a service it can invoke in the distributed network. The ServiceProvider is the company that provides a Web Service. The ServiceRegistry is the location where information on how the web service can be invoked is stored.

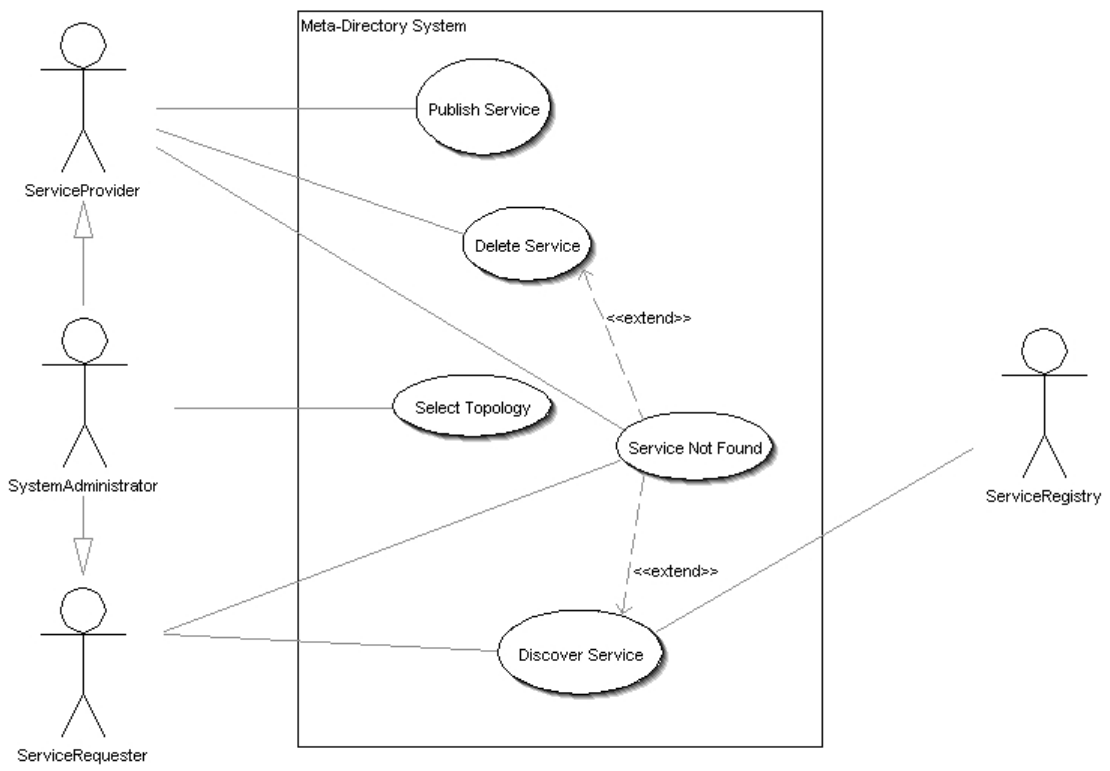


Figure 19 Meta-Directory System Use Case Diagram

The following are the Use Case descriptions for the Meta-Directory System:

Use Case Name: **Select Topology**

Actors: Initiated by the SystemAdministrator

Entry Condition: The system is being deployed and the SystemAdministrator has selected the underlying network model

Flow of Events:

1. The SystemAdministrator selects a network
2. The Meta-Directory node sets its internal overlay network model to the selected model

Exit Condition: The network model of the node has been set to that selected by the system administrator.

Use Case Name: **Publish Service**

Actors: Initiated by ServiceProvider

Entry Condition: ServiceProvider has registered the service information in a ServiceRegistry.

Flow of Events:

1. ServiceProvider forwards the information published to the Meta-Directory system.
2. Meta-Directory node receiving the publish message hashes the attributes and stores the attributes.
3. Meta-Directory system informs the ServiceProvider that the information was successfully published in the system.

Exit Condition: The information is stored in the Meta-Directory system in key-value pairs.

Use Case Name: **Delete Service**

Actors: Initiated by the ServiceProvider

Entry Condition: The service information has been deleted by the ServiceProvider from the ServiceRegistry

Flow of Events:

1. ServiceProvider sends a delete information message to the Meta-Directory system
2. Meta-Directory node receiving the delete message hashes the attributes and deletes the attributes.
3. Meta-Directory system informs the ServiceProvider that the information was successfully deleted from the system.

Exit Condition: The service information is successfully deleted from the Meta-Directory system.

Use Case Name: **Discover Service**

Actors: Initiated by the ServiceRequester, communicates with the ServiceRegistry

Entry Condition: The ServiceRequester has indicated that she/he wants to discover a service.

Flow of Events:

1. ServiceRequester sends a service discovery message to the Meta-Directory system.
2. The Meta-Directory node hashes the attributes and gets the location of the ServiceRegistries.
3. The Meta-Directory node forwards the service discovery request to the ServiceRegistries.
4. The ServiceRegistries respond with the service details.
5. The Meta-Directory System forwards the service details to the ServiceRequester.

Exit Condition: The ServiceRequester has received information on how to invoke the service.

Use Case Name: **Service Not Found**

Actors: Communicates with the ServiceProvider, ServiceRequestor

Entry Condition: This is an extend Use Case that is triggered by the Discover Service Use Case and the Delete Service Use Case when the service cannot be located in the system.

Flow of events:

1. The service information is not found in the Meta-Directory system.
2. The Meta-Directory system informs the actor that sent the message that the service is not found

Exit Condition: The actor that initiated the extending Use Case has been informed that the service could not be found.

After covering the Use Cases for the Meta-Directory network proposed in this thesis, the decisions made during the design of the Meta-Directory system are discussed in the following section.

3.3 Meta-Directory System Design

The Meta-Directory system is composed of distributed Meta-Directory nodes that store the location of the service registries based on the services stored in those registries. In order to design the system, the underlying configuration between the Meta-Directory nodes as well as the structure of the data stored in the nodes needs to be discussed.

This section presents the Meta-Directory system that is deployed using the CHORD (DHT) network model. The CHORD (DHT) model is discussed and the underlying hash table data distribution model is covered, and then an analysis is provided for service publication, service deletion and service query performed on the network.

3.3.1 CHORD (DHT) Model

This forwarding algorithm is a structured algorithm whereby the nodes only know of a subset of nodes in the network. There are no hierarchies in this model, as each Meta-Directory node has the same responsibilities as the rest of the Meta-Directory nodes in the network shown in Figure 20.

A formal definition of the CHORD (DHT) model is presented [22]:

- A CHORD (DHT) network is a directed graph $G = \{V, E\}$ with vertices V , representing the nodes and edges E , representing the finger table.
- Each vertex is labeled by its ID
- The degree of each vertex in the graph = m
- The graph's size $|E| = N(\log_2 N - 0.5)$

This structured forwarding model is deployed using the CHORD [22] algorithm where each peer has a unique ID and is only responsible for a subset of the hashed values and only knows of its neighbors in the network. Each node and key is assigned an m bit

identifier using SHA-1 [22] as a base hash function. Each key is assigned to the node whose node ID is equal to or a successor to the key value. The nodes are arranged in a ring as shown in Figure 20 and each node n has m neighbors where the i^{th} entry of the routing table contains the identity of the first node that succeeds n by at least 2^{i-1} in the identifier circle [22]. The neighborhood table also known as the finger table includes both the CHORD identifier and the IP address of the relevant node and the first entry is the immediate successor on the circle.

Figure 20 shows a CHORD (DHT) ring with 8 nodes and storing 5 keys with $m = 6$. The successor of identifier 28 is node 32; therefore key 28 would be located at node 32. Similarly, keys 49 and 51 would be located at node 51, key 65 at node 72, and key 98 at node 1. Figure 20 also shows the finger table for N14.

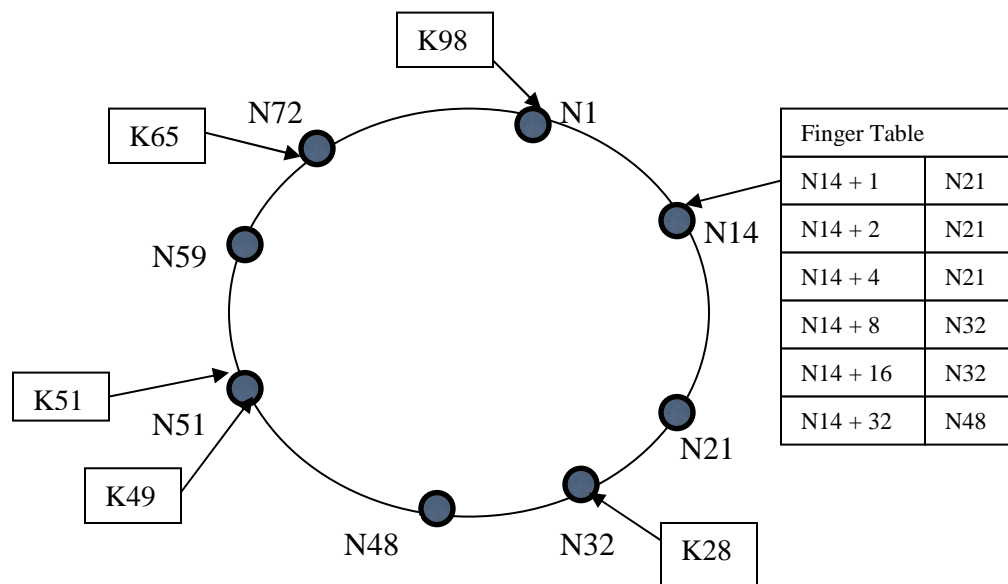


Figure 20 CHORD (DHT) Ring

When a publish request message is received by a Meta-Directory node, the Meta-Directory node hashes the attributes and the key-value pairs it is responsible for are stored locally. The Meta-Directory node then forwards the other key-value pairs to the

responsible nodes by routing the messages through its neighbors. When a query request is received, the Meta-Directory nodes follow the same forwarding algorithm as when a publish request is received.

The following section describes the key-value pair structure in the hash tables used for the CHORD (DHT) network.

3.3.2 Hash Table Structure

This section presents the structure of the hash tables used in the management of distributed registries using the Meta-Directory approach. A hash table is used in all the data distribution models as this is where the information about the Service Registry that can respond to a query is stored. Since the value in the key-value pair in the hash table provides the location of the relevant registries, query messages are only propagated to the relevant registries.

In our system, the information in the registries is stored using the Universal Description, Discovery and Integration (UDDI) [4] protocol. The UDDI protocol has a standardized query interface. This query interface facilitates the information stored in the hash tables as only the UDDI attributes that can be queried are stored in the hash table as key-value pairs. The key is the hashed value of the attribute and the value is the location of the Service Registry that has information regarding that attribute (as can be seen in the example in Table 3). It should be noted that not everything that is in the service registries is forwarded to the Meta-Directory system but only the keywords of the subset of services that are offered to the Extranet members and the locations of the registries corresponding to the services have to be sent to the Meta-Directory system.

Even though the UDDI interface is the one used in explaining the hash table architecture in this thesis, the design of the hash tables can accommodate any registry protocol because the attribute value is concatenated with the attribute name when it is hashed. This facilitates the use of any publication and query protocol that is followed by the service registry. The only additional attribute in all the publish messages is the location of the service registry where the Web Service can be invoked.

The query interface supports the functions shown in Figure 21 that are compatible with the UDDI publish and query functions. The Meta-Directory system only supports the save, delete, and find functions associated with the business, service, and tModel entities of the UDDI protocol and does not support the `get_details` function. This is because, in order to get the details of the entities, the user must know the entity ID, which in turn means the user already knows the location of the service registry and therefore does not need to route the message through the Meta-Directory system.

```
void save_business(businessName, discoveryURL, identifier,
                  category, registryLocation);
void save_service(serviceName, businessName, category,
                  registryLocation);
void save_tModel(tModelName, identifier, category,
                 registryLocation);
void delete_business(businessName, discoveryURL, identifier,
                    category, registryLocation);
void delete_service(serviceName, businessName, category,
                    registryLocation);
void delete_tModel(tModelName, identifier, category,
                   registryLocation);
String find_business(businessName, discoveryURL, identifier,
                    category);
String find_service(serviceName, businessName, category);
String find_tModel(tModelName, identifier, category);
```

Figure 21 Meta-Directory Interface

The *save* functions are used by the service provider to publish service and business information in the Meta-Directory nodes. The *delete* functions are called by the service

provider whenever they want to delete service and business information from the Meta-Directory system. The *find* functions are called by the service requesters whenever they want to find web service information and they don't know the location of the registry that has the information. All these messages can be forwarded to any of the Meta-Directory nodes in the system.

Table 2 lists that attributes from the messages in Figure 21 that are hashed in the Meta-Directory system. These attributes provide the key and the "registry location" attribute provides the value stored in the hash tables. Each attribute is hashed individually so that if only one attribute is supplied, such as the category, when a *find* function is used, the system will query all the registries that have service information in that category.

Table 2 Attributes Hashed by the Meta-Directory System

Hashed Attributes
BusinessName
DiscoveryURL
Identifier
Category
ServiceName
tModelName

When the *save* and *delete* functions are used, all the attributes are required, while one or all of the attributes can be provided during a *find* request. When more than one attribute is provided by a service requester, the system employs an AND function to the

attributes and only returns the registry locations that have information on all the provided attributes.

Table 3 shows an example of the contents of a hash table when three businesses were published as follows: `save_business(name::serviceProvider, location::www.services.com)`, `save_business(name::communications, location::www.communications.ca)`, and `save_business(name::softnet, location::www.registry.com:8080)`. The name attributes were hashed to produce the keys 342, 574, and 283 respectively as shown in Table 3.

After looking at the contents of the Hash Table, the following section illustrates how the hash table data is distributed among multiple Meta-Directory nodes when service information is published in the system.

Table 3 Hash Table Contents

Key	Value
342	<u>www.services.com</u>
574	<u>www.communications.ca</u>
283	<u>www.registry.com:8080</u>

3.3.3 Publishing Service Information

Figure 22 shows the attribute distribution when a publish message is sent to the Meta-Directory network. In this example, each Meta-Directory node in the network follows the CHORD (DHT) protocol thereby handling a subset of hash values. This ensures that regardless of the attribute being hashed, the load is evenly distributed in the network. The attributes are hashed individually with the attribute name concatenated with

the attribute value. This is done in order to distinguish attributes that have the same name. For example, there can be a company with the business name software, while at the same time another company could be providing a service under the category software. These two attributes are distinguished by concatenating the attribute name with the value when applying the hash function.

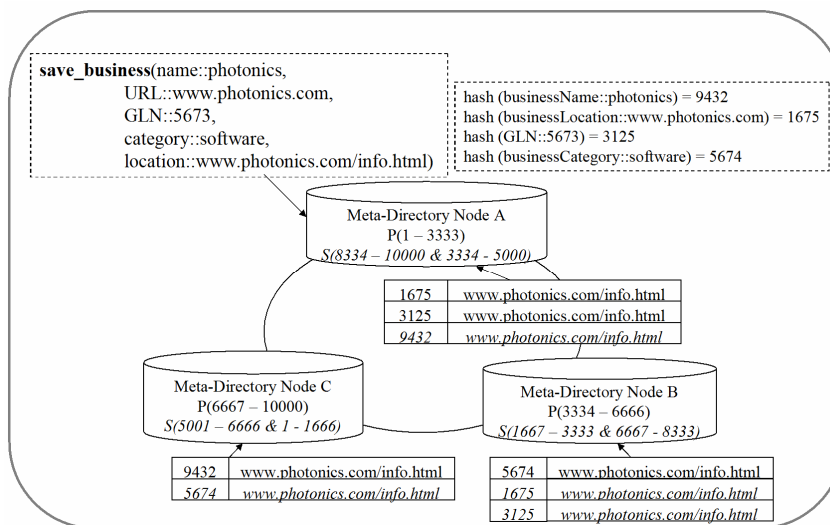


Figure 22 Service Publishing in Meta-Directory Nodes

To address the problem of introducing a single point of failure, Figure 22 also illustrates the data replication algorithm followed in the Meta-Directory system. In addition to the primary range of values that a Meta-Directory node stores, there are also secondary replicated key-value pairs that a Meta-Directory node is responsible for to avoid losing information if a Meta-Directory node dies. The secondary key-value pairs are replicated values that are introduced to ensure redundancy in the network and these are illustrated in the figure by the range of values shown in italics. The secondary keys are selected so that each Meta-Directory node has a key-value replication of the first half of values handled by the successor and the latter half of values handled by the

predecessor. This ensures that the hashed information is evenly distributed among the distributed Meta-Directory nodes.

The Meta-Directory node receiving the publish request hashes the attributes received as shown by Meta-Directory Node A in Figure 22. In this example, the attribute “businessName” is concatenated with the attribute value “photonics” and then hashed to create the hashed key number 9432, the attribute “businessLocation” is concatenated with its value “www.photonics.com” to produce the key 1675, the attribute “GLN” which indicates the unique business global location number is concatenated with the value “5673” to produce the key 3125 and the attribute “businessCategory” concatenated with the value “software” is hashed to create the key value 5674. Once all the attributes are hashed, the Meta-Directory node that received the publish request stores the key-value pairs that it is responsible for, and forwards the remaining key-value pairs to the relevant Meta-directory node.

In Figure 22, Meta-Directory Node A then stores the key-value pairs for the primary keys 1675 and 3125 and the secondary key 9432 since it is responsible for the primary keys in the range of 1 to 3333 and the secondary keys in the range of 8334 to 10000 and 3334 to 5000. The secondary keys replicate half of the range of values covered by both the predecessor and successor nodes. Also, in Figure 22, the key-value pairs for the primary key 9432 and secondary key 5674 are then forwarded to Meta-Directory Node C and that of the primary key 5674 and secondary keys 1675 and 3125 are forwarded to Meta-Directory Node B as per the data distribution structure indicated.

For simplicity, the rest of the examples in this section only show the primary range of values that are handled by a Meta-Directory node. Figure 23 illustrates the message

distribution when another business publishes its business information in the network. The business published in Figure 23 provides Web Services in the same category (i.e. software) as the business information published in Figure 22. In this case, there are two entries for the hash value 5674 as shown in Figure 23.

The service information can now be queried once it is published in the network. To illustrate how the services are deleted from the distributed Meta-Directory nodes, the following section shows how the delete requests are sent to the system and how the information is removed from the network.

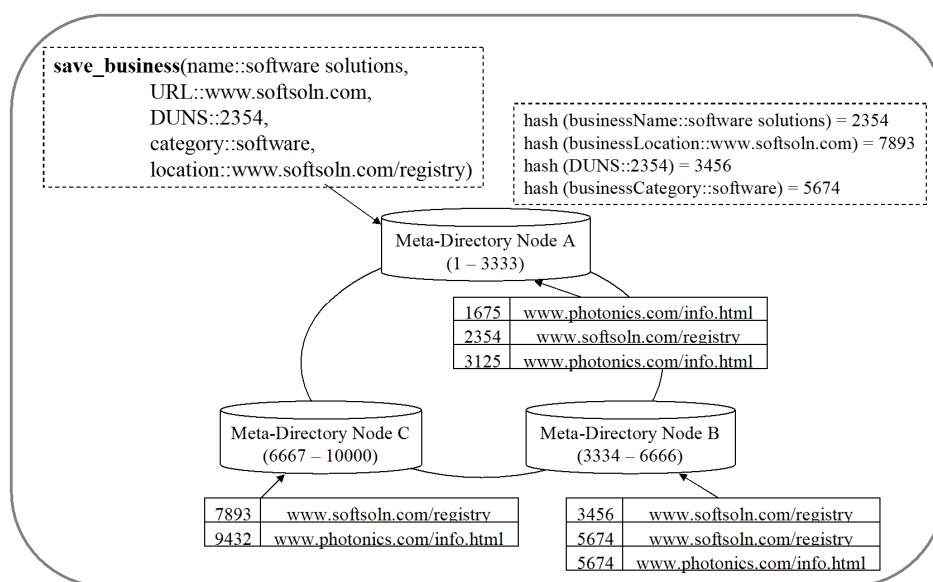


Figure 23 Service Publishing and Message Distribution

3.3.4 Deleting Service Information

When a service provider decides to remove information from the Meta-Directory system, the service provider will use one of the *delete* functions shown in the interface definition in Figure 21. If a service provider wants to delete business information from

the Meta-Directory system, they will send a *delete_business* message to one of the Meta-Directory nodes.

If the *delete_business* function in Figure 24 is forwarded to the Meta-Directory network in Figure 23, the attributes will be hashed by the Meta-Directory node receiving the query. The receiving node will then delete the key-value entries it is responsible for, in this case the entry with the key 2354 and value “www.softsoln.com/registry” will be deleted, and forward delete requests for the remaining keys to the nodes responsible for those entries. That is, a delete request for the keys 3456 and 5674 with corresponding values of “www.softsoln.com/registry” will be forwarded to Meta-Directory Node B and a delete request for the key 7893 and value “www.softsoln.com/registry” will be forwarded to Meta-Directory Node C. Once the key-value entries corresponding to the hashed keys are deleted from the network, the network will result in the hash table entries shown in Figure 24.

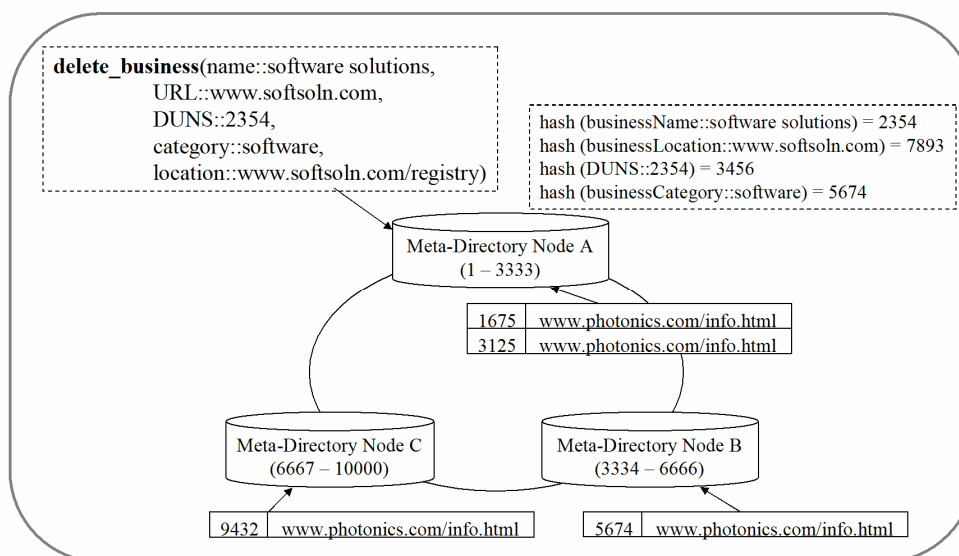


Figure 24 Deleting Service Information from Meta-Directory Nodes

3.3.5 Querying Service Information

Figure 25 illustrates a query message where a service requester is querying for businesses that provide services under the software category. In this example, there are two entries for the hashed value, thereby querying registries in both `www.softsoln.com/registry` and `www.photonics.com/info.html` locations. If the query message was `find_business(DUNS::2354, category::software)` instead, the two attributes are hashed independently producing the hash keys 3456 and 5674 respectively. For these hash values only `www.softsoln.com/registry` will be queried for the business information.

The next section introduces the configurable routing framework used in this thesis. This framework allows the system administrator to choose the overlay network used in connecting the Meta-Directory nodes during deployment.

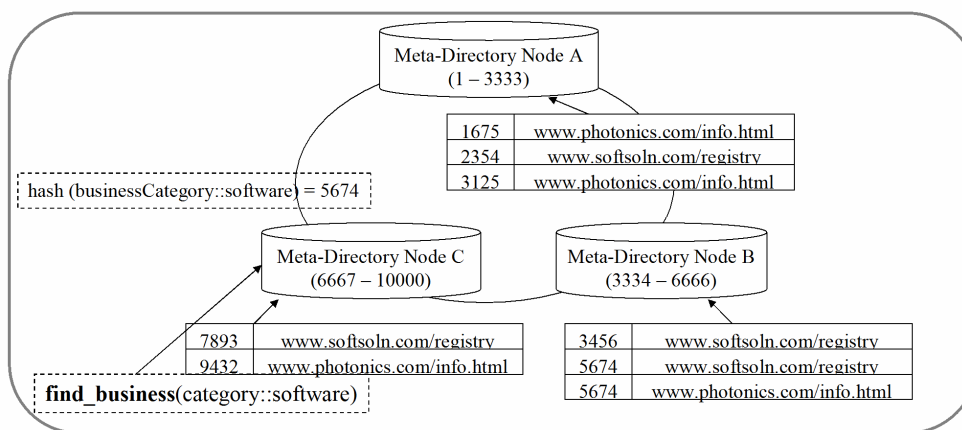


Figure 25 Service Discovery in Distributed Meta-Directory Nodes

3.4 Configurable Routing Framework

Because of the design of the system, such that it hides the details of the network deployment from the registries, the underlying communication framework between the Meta-Directory nodes can be configurable. Because of this property, the Meta-Directory

system allows the system administrator to choose the underlying network model between the distributed Meta-Directory nodes when the system is being deployed. This section looks at the network models that are supported and analyzes the performance of these models.

3.4.1 Network Models

This sub-section examines, analyses, and discusses the network models that are used for inter-connecting the Meta-Directory nodes. A theoretical performance analysis based on the number of hops per query, the total number of messages exchanged per query, and the total number of periodic messages exchanged with the number of nodes is performed.

In addition to the CHORD (DHT) protocol covered in Section 3.3.1, three other network models for Meta-Directory nodes in this thesis are discussed so as to compare the performance of structured as well as unstructured network models. The three additional models are:

- The Fully Connected Model
- The Fully Connected (DHT) Model
- The Super Peer Model

This analysis will help the system administrator in determining at start-up, and for a given expected load, which network will provide better performance. The network models can be divided into three different categories. In the first category, each Meta-Directory node is connected to every other Meta-Directory node in the network. The second category introduces a network model where each Meta-Directory node only knows of the location of a subset of Meta-Directory nodes in the network. In the final category, Super Peers and clusters of Meta-Directory nodes are introduced.

3.4.1.1 Fully Connected Model

In the first category we are going to look at a model where each Meta-Directory node in the network is directly connected to every other Meta-Directory node in the network. In the model in Figure 26, there is no structure to the data distribution among the Meta-Directory nodes. When a message is published, all the hashed entries are stored locally in the Meta-Directory node that received the publish request. Therefore, when a Meta-Directory node receives a query, it then broadcasts the request to all the other nodes in the network. This model is introduced because it minimizes response time since the query is broadcasted to all the nodes in the network and therefore the total number of hops required to find a response is always equal to one.

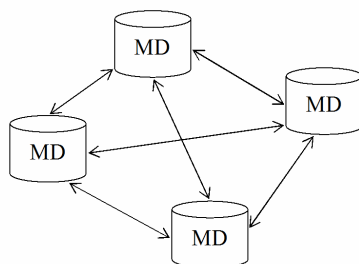


Figure 26 Fully Connected Model

The Fully Connected (FC) model is a **complete** graph $G = \{V, E\}$. Where the degree of each vertex in the graph = $N - 1$.

In order to understand the structure of the Fully Connected Model, the data distribution model in the hash tables needs to be illustrated. For the Fully Connected Model, the hash table data distribution model is referred to as the local distribution model because when data is published in a Meta-Directory node in this system, all the attributes are hashed and the resulting key-value pairs stored in the local hash table.

This is illustrated in Figure 27 where the two businesses are published in two different nodes; therefore the received business information is stored locally. Hashing is done so that in the future during the implementation of the runtime reconfiguration of the network models (with the transformation algorithms to be discussed in Chapter 6 the underlying data model does not have to be modified when the network is reconfigured between the CHORD (DHT) model and the Fully Connected model.

In the example in Figure 27, Meta-Directory Node A receives the `save_business` request from the “photonics” company. Meta-Directory Node A hashes the attributes as per the hashing convention discussed in Section 3.3.2, it then stores all the hashed keys in the local hash table with the hash value being the location of the service registry (`www.photonics.com/info.html`). “software solutions” on the other hand forwards its `save_business` request to the Meta-directory Node C, therefore all the hashed keys for the “software solutions” company are then stored locally in the Meta-Directory Node C.

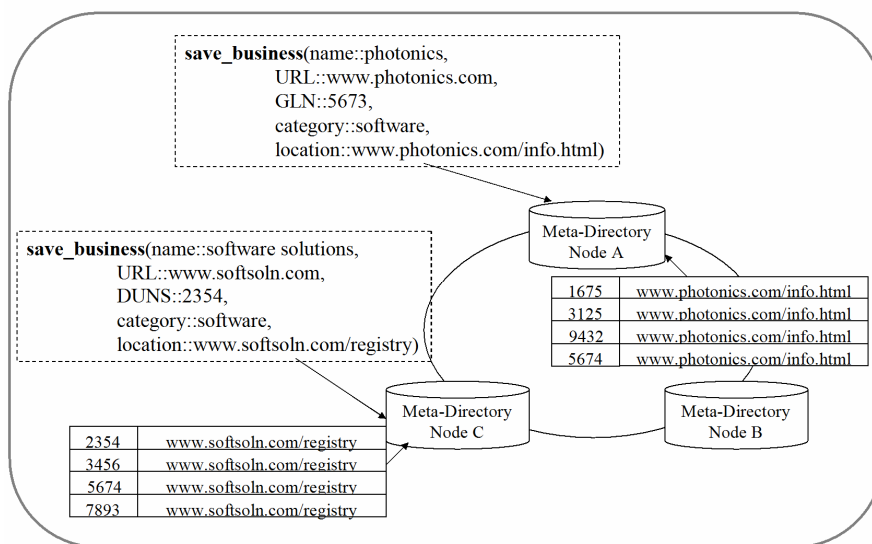


Figure 27 Local Distribution Model

When a service discovery request is received, if the receiving Meta-Directory node does not have the values for all the keys in the discovery request, the Meta-Directory node broadcasts the keys to the rest of the nodes in the network as there is no indication as to where the hashed key could be stored.

In this data distribution model, messages are not sent among nodes during a publish request as all the data is stored locally. The disadvantage of this approach is that the data may not be evenly distributed in the network as seen in Figure 27, where Meta-Directory Node B does not have any data since there was no business information published directly at that node.

The following section illustrates the Fully Connected (DHT) structure as well as the hash table data distribution in Fully Connected (DHT) networks.

3.4.1.2 Fully Connected (DHT) Model

In the Fully Connected (DHT) model, see Figure 28, the data distribution among the Meta-Directory nodes follows a structured distribution model such that each node only handles a subset of hash entries. In this case, when a Meta-Directory node receives a request, the Meta-Directory node only forwards the message to the appropriate node in charge of the hashed key. Therefore, when compared to the model in Section 3.4.1.1, this model does not flood the network during a discovery request but incurs a higher overhead during a publish request as the published entries have to be forwarded to the appropriate Meta-Directory nodes.

The Fully Connected (DHT) (FCDHT) model is a special case of the CHORD (DHT) graph in that it is a **complete** CHORD (DHT) graph where the degree of each vertex in the graph = $N - 1$.

Since in the Fully Connected (DHT) Model the key-value pairs are forwarded to the appropriate node, the algorithm used for the selection of which node the key-value pair is forwarded to needs to be described.

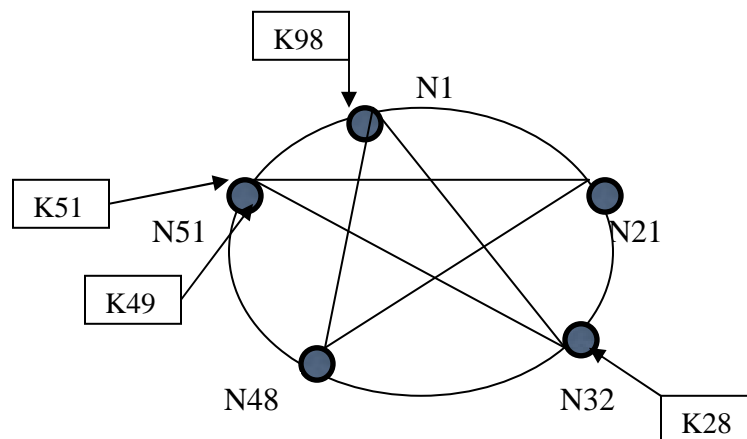


Figure 28 Fully Connected (DHT) Model

In Figure 29, the two businesses publish business information in the same Meta-Directory nodes as those in Figure 27. In this case, the entries are hashed in the Meta-Directory node that receives the request, but the Meta-Directory node receiving the request only saves the key-value pairs that it is responsible for in the local hash table and forwards the rest of the key-value pairs to the other Meta-Directory nodes in the network (as was discussed in Section 3.3.3). It should be noted that this distribution model can be implemented using the CHORD (DHT) forwarding algorithm by setting the size of the finger table greater or equal to the maximum number of nodes that will ever be in the system. This ensures that the graph is always complete when nodes join and leave the network.

In the example in Figure 29, hashing is done to produce hash values in the range of 1 to 10000. Therefore, to ensure that the data is evenly distributed in the network, Meta-

Directory Node A is responsible for the hash keys in the range of 1 to 3333, Meta-Directory Node B for hash keys in the range of 3334 to 6666 and Meta-Directory Node C for the keys in the range of 6667 to 100000.

Meta-Directory Node A receives the `save_business` request from the “photonics” company and hashes the attributes. Meta-Directory Node A then only stores the keys that fall in the range of 1 to 3333, in this case the keys 1675 and 3125 for the “photonics” corporation. The remaining hashed keys are forwarded to their respective nodes.

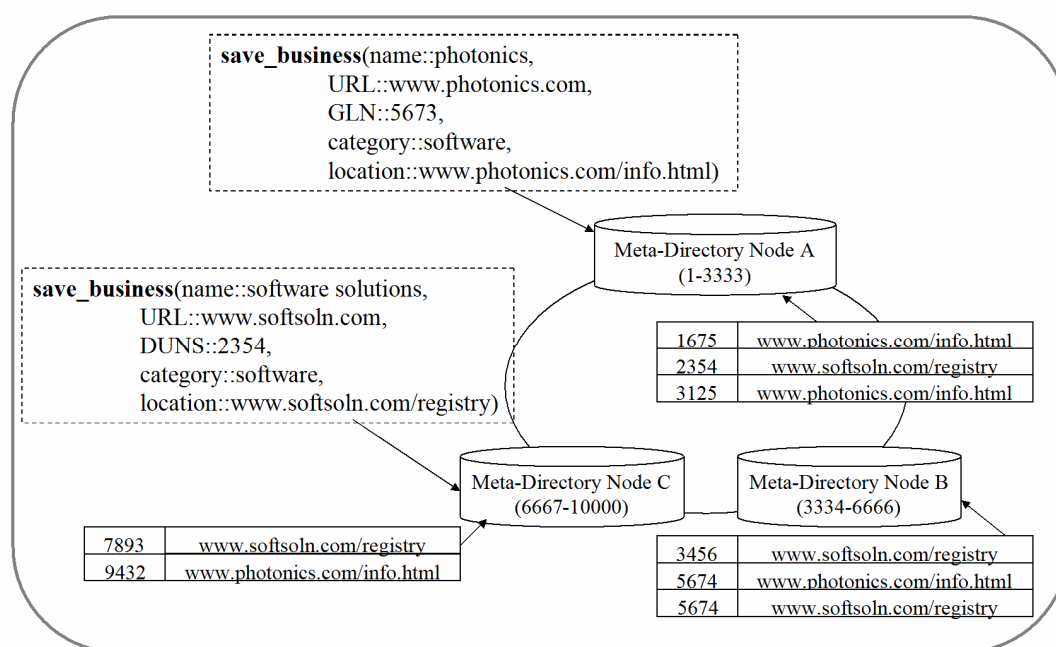


Figure 29 DHT Data Distribution Model

The advantage of using the DHT model is that the data is also replicated in the network such that the same entry can be found in at least two nodes at any time as illustrated in Section 3.3.3. This ensures that the system does not have a single point of failure and data is not lost if a Meta-Directory node goes offline. This distribution model

incurs an overhead during a publish request, but it ensures that the load is shared among the Meta-Directory nodes in the network.

3.4.1.3 Super Peer Model

Figure 30 illustrates a Super Peer network whereby groups of Meta-Directory nodes form clusters of collaborating nodes which communicate with the rest of the network through designated Super Peers. This network is hierarchical and the Meta-Directory nodes in the clusters form the Client Peers.

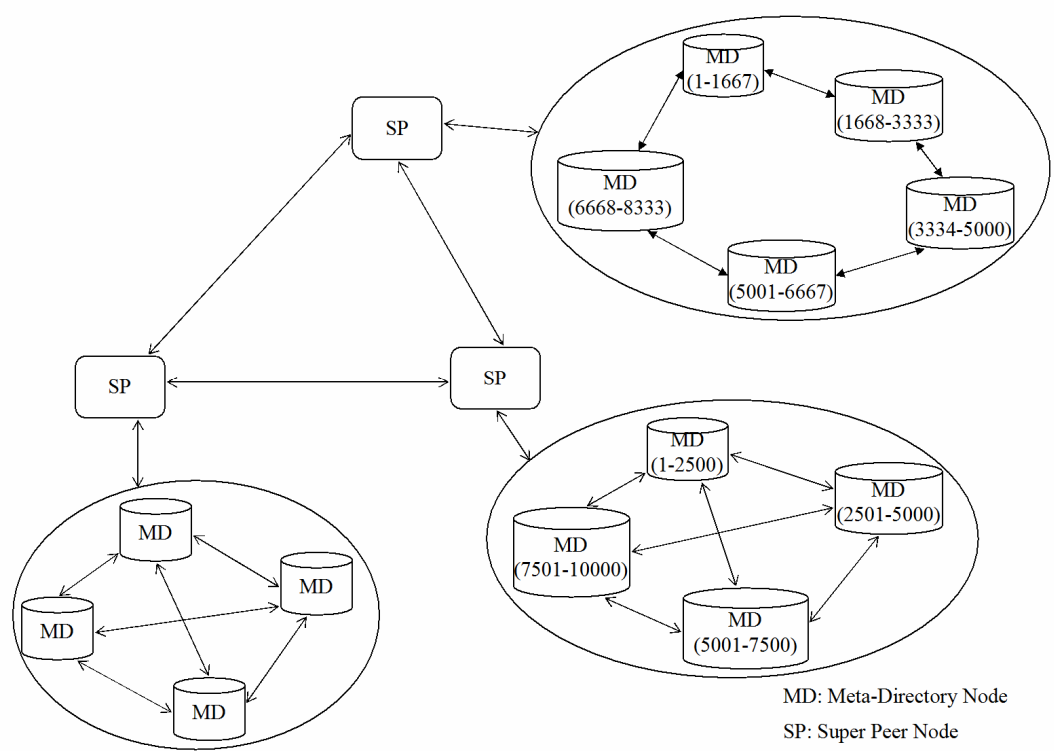


Figure 30 Super Peer Model

The Client Peers have the same responsibilities as the Meta-Directory nodes in the other categories and they can communicate using any of the previously discussed forwarding algorithms. The Super Peers have extra responsibilities as they have to

communicate with and manage the clusters as well as communicate with the other Super Peers in the network.

When a request is received by a Meta-Directory node, a search is first performed in the local hash table and then forwarded within the cluster depending on the forwarding algorithm used within that cluster. In this model, since the data stored in the clusters is independent of each other, after the search is first performed within the cluster the request is then forwarded to the Super Peer that broadcasts the request to the other Super Peers in the network. The rest of the Super Peers concurrently forward the request within their clusters and if a response is found, the response is forwarded back to the Meta-Directory node which sent the originating request.

The Super Peer network can be defined as follows:

- The Super Peer network consists of three sub-graphs: a Super Peer graph, a Cluster graph, and a graph that connects the Super Peer and the Cluster graphs
- A Super Peer graph $G = \{V, E\}$ is a FC graph where:

$$V = \{SP_1, SP_2, \dots, SP_{N_{SP}}\},$$
 and N_{SP} is the total number of super peer nodes

$$E = \{(SP_1, SP_2), (SP_1, SP_3), \dots, (SP_{N_{SP}-1}, SP_{N_{SP}})\}$$
- A cluster graph $G = \{V, E\}$ for each cluster which is either a FC, CHORD, or FCDHT graph
- A connection graph $G = \{V, E\}$ for each cluster which is a FC graph where:

$$V = \{SP_i, N_{ei}\},$$
 and SP_i is the Super Peer of cluster i and N_{ei} the node that is connected to the Super Peer responsible for cluster i where $i \in 1, \dots, N_c$ and N_c is the number of clusters

$$E = \{(SP_i, N_{ei})\}$$

The following sub-section presents a theoretical performance analysis of the Fully Connected, Fully Connected (DHT), CHORD (DHT), and the Super Peer model.

3.4.2 Performance Analysis of Network Models

A theoretical analysis was performed on the previously discussed network models and the results of this analysis are described in this section. For a number of different performance metrics, a worst case analysis was done to determine which model performs better based on the state of the network.

3.4.2.1 Total Number of Messages Exchanged

The first analysis was done in order to study the impact of the size of the network on the total number of messages exchanged among nodes per query message. With the Fully Connected model, each node is connected to all the nodes in the network and there is no indication as to where information is stored because the data is stored locally by the node receiving the publish request (see Section 3.4.1.1). Because of this model, when a request is received, if the node receiving the request does not have the required information, the node broadcasts the request to all the nodes in the network consisting of N nodes. Therefore the total number of query messages N_q sent is given by:

$$N_q = N - 1 \quad (3.1)$$

A response message is only sent by the node that has an answer to the query message. Hence, the total number of messages N_t exchanged in a fully connected network in the worst case is given by:

$$N_t = N_q + 1 \quad (3.2)$$

Using equation (3.1) and equation (3.2), we get:

$$N_t = N \quad (3.3)$$

This makes the total number of messages exchanged per query increase linearly with the size of the network (see Figure 31).

In the Fully Connected (DHT) model, that can be realized by using the CHORD (DHT) protocol where all the nodes in the network are neighbors of each other (see Figure 28), the data is stored based on the identification of the node. Therefore, when a query is received, if the node that received the query does not have the data, that node forwards the query to the node that is responsible for that hashed key (see discussion in Section 3.4.1.2). In this case only one request is sent in the network to the responsible node that sends a reply message. Therefore, the total number of messages exchanged per query message is given by:

$$N_t = 2 \quad (3.4)$$

and is therefore constant.

The structured forwarding model discussed in Section 3.3.1 can be fully realized using the CHORD (DHT) forwarding algorithm. In the worst case, the total number of query messages exchanged in a network of N nodes is given by [22]:

$$N_q = \log_2 N \quad (3.5)$$

A response message is only sent by the node having a response to the query message. Thus, the total number of messages exchanged per query in a CHORD (DHT) model is given by:

$$N_t = N_q + 1 \quad (3.6)$$

Using equation (3.5) and equation (3.6), we get the total number of messages:

$$N_t = \log_2 N + 1 \quad (3.7)$$

The total number of messages exchanged in the network increases logarithmically with the size of the network as can be seen in Figure 31.

As for the Super Peer model, it is characterized by having a maximum of N_C nodes per cluster and the nodes are evenly distributed in the network whenever a new node joins the network. For example, if the number of nodes increases from N_C to $N_C + 1$ nodes, the network will consist of two clusters with $\frac{N_C + 1}{2}$ nodes per cluster. With the current implementation, increasing the number of clusters in the network requires the system administrator to create a new Super Peer node. The algorithms to be used in the network reconfiguration to ensure that the number of nodes is evenly distributed between the clusters will be covered in Chapter 6. For the Super Peer model used in this evaluation $N_C = 5$ and the communication inside the clusters follows the CHORD (DHT) forwarding algorithm, while the Super Peer nodes communicate using the Fully Connected model. The plot in Figure 31 illustrates the worst case scenario during a query such that requests have to be forwarded to all the Super Peer nodes whenever a request is received. The total number of messages exchanged in a Super Peer network is equal to the total number of messages exchanged among the Super Peers and the total number of messages exchanged within the clusters.

Since the Super Peers communicate using the Fully Connected model, equation (3.1) is applied to a network having a total of N_{SP} Super Peers giving the total number of query messages exchanged among the Super Peers as:

$$\mathbf{N}_{qs} = \mathbf{N}_{SP} - \mathbf{1} \quad (3.8)$$

The total number of query messages exchanged within the clusters in a network of n clusters, is realized by applying equation (3.5) to all the clusters where each cluster i has a total of N_C nodes. Therefore, the total number of query messages exchanged within all the clusters is given by:

$$\mathbf{N}_{qc} = \sum_{i=1}^n \log_2(N_C) \quad (3.9)$$

A response is only returned by the cluster that has a response to the query message. This response message is first sent from the client peer to the Super Peer that then forwards the message to the requesting Super Peer that forwards the message directly to the requesting client node. Therefore, the total number of messages exchanged in a Super Peer network is given by:

$$\mathbf{N}_t = \mathbf{N}_{qs} + \mathbf{N}_{qc} + \mathbf{3} \quad (3.10)$$

Using equation (3.8), equation (3.9), and equation (3.10), we get:

$$\mathbf{N}_t = \mathbf{N}_{SP} + \sum_{i=1}^n \log_2(N_C) + \mathbf{2} \quad (3.11)$$

For example, Figure 31 shows that the number of messages increases from 3 to 7 when the number of nodes in the network changes from $N_C = 5$ to $N_C = 6$ for the Super Peer network. When there are 6 nodes, there will be two clusters with three nodes per cluster. Therefore, applying equation (3.11), the number of messages exchanged in the network is then given by:

$$\mathbf{N}_t = 2 + \log_2(3) + \log_2(3) + 2 \approx 7$$

In conclusion, in terms of the total number of messages exchanged per query, the Fully Connected (DHT) model performs better than all the other discussed networks as the total number of messages is always equal to 2. The Fully Connected model provides the worst performance as the total number of messages increases linearly to the size of the network. Therefore, there is a lot of network traffic when a query is received. The performances of the other two networks lie between these two extremes and they increase logarithmically with the size of the network as they are based on CHORD and in the CHORD algorithm the number of messages exchanged are determined by an $O(\log N)$ where N is the size of the network [22]. The CHORD (DHT) model performs better than the Super Peer model when the number of clusters increases as the Super Peer model incurs additional message exchange among the Super Peers which follow a Fully Connected protocol.

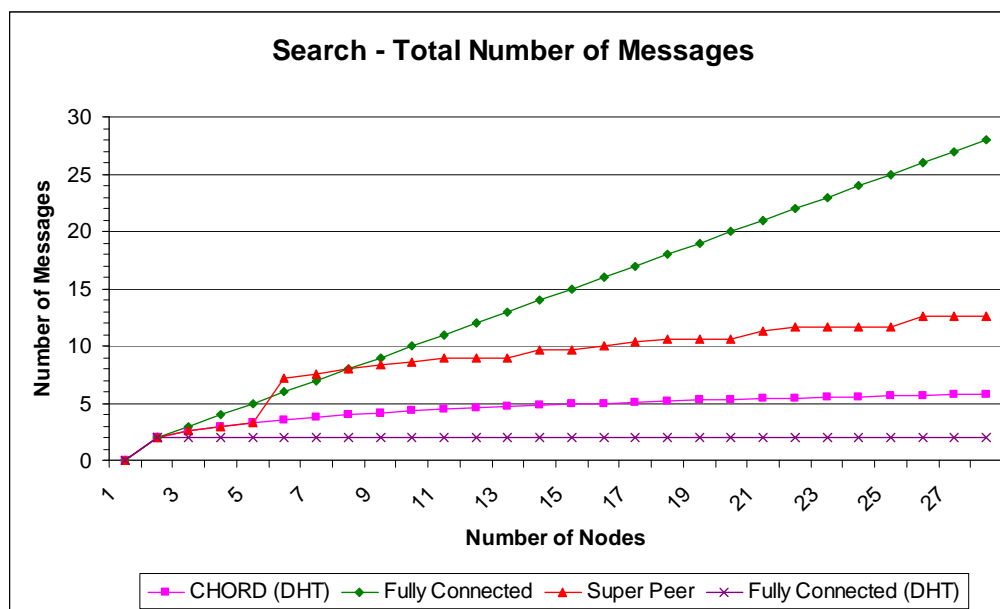


Figure 31 Effect of Network Size on the Total Number of Messages Exchanged per Query

3.4.2.2 Total Number of Hops

The second analysis was performed on the total number of hops a message takes in the network before a result is found, that is the total number of hops before the message reaches the node that can answer to the query. The total number of hops determines the network delay per query and it is defined as the number of nodes traversed by a query message before reaching the node that has the correct response. The variation in the maximum number of hops per query with the size of the network is presented and the analysis illustrates the worst case scenario where the node receiving the request from a client does not have a response and the request is forwarded within the rest of the nodes in the network. With the unstructured Fully Connected model, each node is directly connected to every other node in the network; therefore the total number of hops is given by:

$$N_t = 1 \quad (3.12)$$

Thus the total number of hops in a Fully Connected model is constant regardless of the size of the network (as seen in Figure 32).

In the Fully Connected (DHT) model, if the node that receives the query initially does not have an answer to the query, the node forwards that request to the node responsible for the hashed key. In this case, the total number of hops the query message takes within the network is given by:

$$N_t = 1 \quad (3.13)$$

The total number of hops in a Fully Connected (DHT) model is thus also a constant regardless of the size of the network (as seen in Figure 32).

With the structured forwarding model realized using CHORD, the total number of hops before a response is found in a network of N nodes is given by [22]:

$$\mathbf{N_t = \log_2 N} \quad \mathbf{(3.14)}$$

This is equal to the total number of messages exchanged in the network because the messages are not broadcast in the network but they are forwarded from one node to the next.

In the Super Peer model the total number of hops is the same as the CHORD (DHT) model when there is only one cluster. When there is more than one cluster in the network, the query message is first forwarded within the cluster that received the message. Since the clusters communicate using the CHORD (DHT) protocol, equation (3.14) is applied to the first cluster and the total number of hops in the first cluster having N_1 nodes is given by:

$$\mathbf{N_{h1} = \log_2 N_1} \quad \mathbf{(3.15)}$$

If a response is not found in the cluster that received the request, the query message is then broadcast by the Super Peer to the rest of the Super Peers in the network following the Fully Connected network model. Applying equation (3.12) to the Super Peers, the total number of hops in the Super Peers is given by:

$$\mathbf{N_{hs} = 1} \quad \mathbf{(3.16)}$$

The Super Peers concurrently forward the request within their clusters following the CHORD (DHT) protocol. Since the total number of hops refers to the number of nodes traversed in order to find the node that can answer the query, equation (3.14) is applied to

the cluster that has the response for the query making the total number of hops in the second cluster having N_2 nodes equal to:

$$N_{h2} = \log_2 N_2 \quad (3.17)$$

The total number of hops in a Super Peer network is thus given by:

$$N_t = N_{h1} + N_{hs} + N_{h2} \quad (3.18)$$

Using equation (3.15), equation (3.16), equation (3.17), and equation (3.18), the total number of hops in a Super Peer network is given by the following general equation:

$$N_t = 1 + \log_2 N_1 + \log_2 N_2 \quad (3.19)$$

For example, when the number of clusters increases to 2 (that is the size of the network increases from 5 to 6 nodes); the total number of hops increases from 2 to 4 as now two extra messages are exchanged between the two Super Peers. This causes a jump in the number of hops in the Super Peer network whenever a new cluster is created due to a new node joining the system. But as the size increases to 11 nodes, the number of hops reduces a little as now there are three clusters with two clusters having 4 nodes and one cluster having three nodes. Since the nodes are evenly distributed among the clusters, the maximum number of hops reduces. But when at least two clusters have 5 nodes per cluster, the maximum number of hops becomes constant at 6 hops (see Figure 32). Therefore, the total number of hops in a Super Peer network is equal to the total number of hops per cluster plus one extra hop for the message exchange among the Super Peers.

In terms of the total number of hops, the Fully Connected models show the best performance, as regardless of the size of the network the total number of hops per query message before a result is found is always equal to 1. With the CHORD (DHT)

architecture the total number of hops increases logarithmically. With the Super Peer model, the number of hops is higher than in a CHORD (DHT) network when the number of nodes is between 6 and 50. But when the number of nodes reaches a point such that there are two clusters in the Super Peer model each with 5 nodes, the maximum number of hops per query message stabilizes to 6 hops per message. Thus for larger networks the Super Peer model performs better than the CHORD (DHT) protocol in terms of the total number of hops.

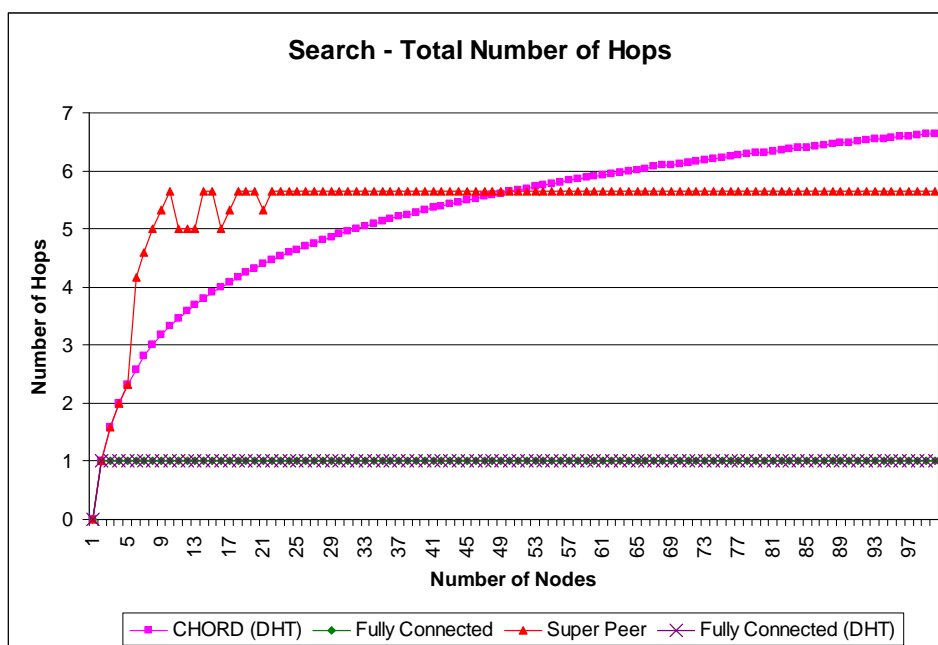


Figure 32 Effect of Network Size on the Total Number of Hops

3.4.2.3 Maintenance Overhead

The third analysis was performed on the overhead incurred in the maintenance of the network. In the Fully Connected protocol, the nodes are not organized in any manner and messages are only exchanged among the nodes when a node joins or leaves the system. When there is no change in the network configuration, there is no maintenance incurred

in this network. Therefore as can be seen in Figure 33, there are no periodic messages exchanged among the nodes in the Fully Connected model.

Both the Fully Connected (DHT) model and the CHORD (DHT) model are realized using the Distributed Hash Table architecture where all the nodes have a unique identification and keys are stored in the node that is responsible for them. With these architectures, there are overheads incurred for the maintenance of the system. With the CHORD (DHT) algorithm, maintenance is done periodically to ensure that the successors and predecessors of a node have not changed. For every node in a DHT network a total of three messages are exchanged periodically, the polling message from the predecessor to successor, the response message from the successor and the final notification from the predecessor to the successor [22]. Therefore, the total number of messages exchanged in a network of N nodes to ensure that the successor pointers are stable is given by:

$$N_s = (3 * N) \quad (3.20)$$

In addition to the messages exchanged to verify the successor, there is also maintenance overhead to ensure that the entries in the finger tables are stable. For maintenance of the finger table entries, in a network of N nodes, the number of periodic messages sent by each node for each finger table is given by [22]:

$$N_{ft} = \log_2 N \quad (3.21)$$

Therefore, applying equation (3.21) to all the finger table entries, the total number of messages periodically sent by each node having N_f finger table entries is given by:

$$N_{ft} = N_f * \log_2 N \quad (3.22)$$

Applying equation (3.22) to all the nodes in the network, the total number of periodic messages for maintenance of the finger table entries for a network of N nodes is given by:

$$N_{fN} = N * N_f * \log_2 N \quad (3.23)$$

The total number of periodic messages exchanged in a DHT network is equal to the sum of messages exchanged to stabilize the successor pointers and the number of messages exchanged to stabilize the finger table entries and is given by:

$$N_t = N_s + N_{fN} \quad (3.24)$$

Applying equation (3.20), equation (3.23), and equation (3.24), the total number of periodic messages exchanged in a DHT network is given by:

$$N_t = (3 * N) + (N * N_f * \log_2 N) \quad (3.25)$$

Figure 33 illustrates the Fully Connected (DHT) model where the size of the finger table is equal to $(N - 1)$ and the CHORD (DHT) model where the size of the finger table is equal to $\log_2 N$ since the nodes only know of their successors.

The Super Peer network used in this evaluation forwards messages within the clusters using the CHORD (DHT) model where the size of the finger table is equal to one. The total number of periodic messages exchanged in the Super Peer network is equal to the sum of all the periodic messages exchanged per cluster as there is no maintenance overhead among the Super Peers as they communicate using the Fully Connected model.

Since the clusters within the Super Peer network communicate using the CHORD (DHT) algorithm, equation (3.25) is applied to all the clusters. In a network with N nodes and n clusters, where each cluster i has a total of N_i nodes and N_{fi} finger table entries, the total number of periodic messages exchanged in the Super Peer network is then given by:

$$N_t = (3 * N) + \sum_{i=1}^n (N_i * N_{fi} * \log_2 N_i) \quad (3.26)$$

For the maintenance overhead, the Fully Connected architecture performs better than all the other architectures since there are no maintenance messages exchanged in the network. The Fully Connected (DHT) architecture on the other hand performs the worst with the number of messages increasing as a polynomial function of the size of the network. The Super Peer model performs slightly better than a network deployed using CHORD (DHT) alone as illustrated in Figure 33.

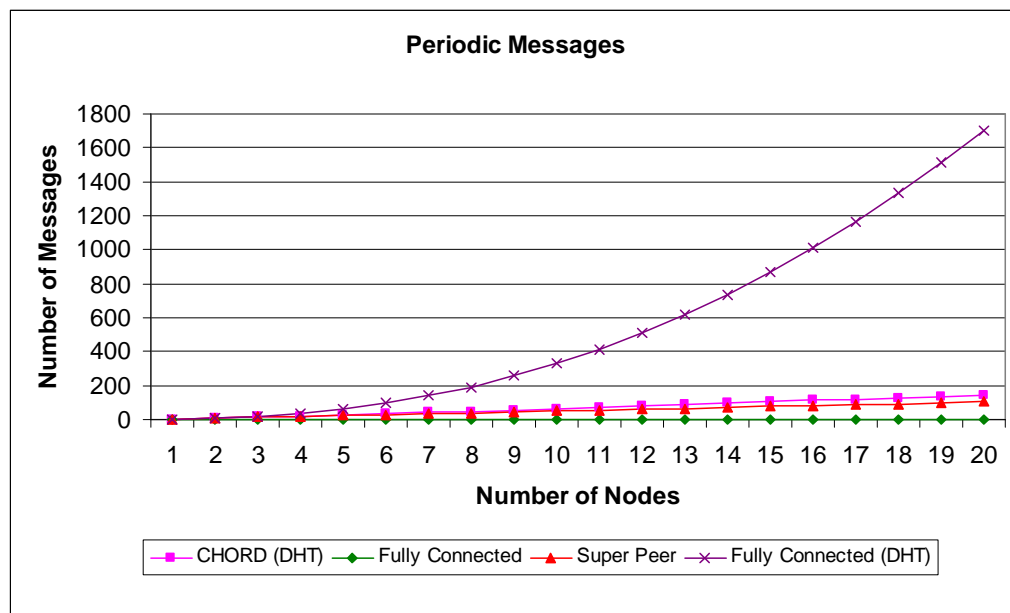


Figure 33 Effect of Network Size on the Periodic Messages Exchanged

Due to these results, it can be seen that there is no single network model that is the best for any given network size. For different performance metrics a specific model may be better than the remaining models. Therefore, a configurable system would be ideal for a distributed system that connects different enterprises with varying network sizes.

The following chapter discusses how the distributed Meta-Directory system was implemented. Chapter 4 also covers the implementation of the configurable routing framework whereby during system setup, the system allows the user to deploy the distributed network in any of the discussed network configurations.

CHAPTER 4 IMPLEMENTATION

The Meta-Directory system was implemented using Java as the programming language as well as a number of open source applications that were used as the foundation in the implementation.

This chapter outlines the Meta-Directory components and discusses the roles of each component along with the messages the component responds to. The following section discusses the components of the Meta-directory architecture in more detail and Section 4.2 discusses how the proposed Meta-Directory system was implemented by providing the tools used and the sequence diagrams.

4.1 Meta-Directory Architecture

This section covers the three components within the Meta-Directory architecture. These components, as shown in Figure 34, are the Service Broker, the Communication Overlay and the Hash Table. The Hash Table is where the key-value pairs corresponding to the registry and service information is stored. The actual information that is stored in the Hash Table was covered in detail in section 3.3.2.

The Communication Overlay provides the publish interface and query interface for the Service Provider and Service Requester. The Communication Overlay also forwards requests to the local Hash Table. The Communication Overlay is responsible for handling the network setup and message forwarding among the distributed Meta-Directory nodes in the network.

The Service Broker receives the Service Registry location and the query from the Communication Overlay and forwards the query to the appropriate Service Registry. The

Service Broker then forwards the response from the Service Registry to the Service Requester. Figure 35 gives an illustrative sequence diagram showing the message exchange among the components when a service is published and during a service discovery.

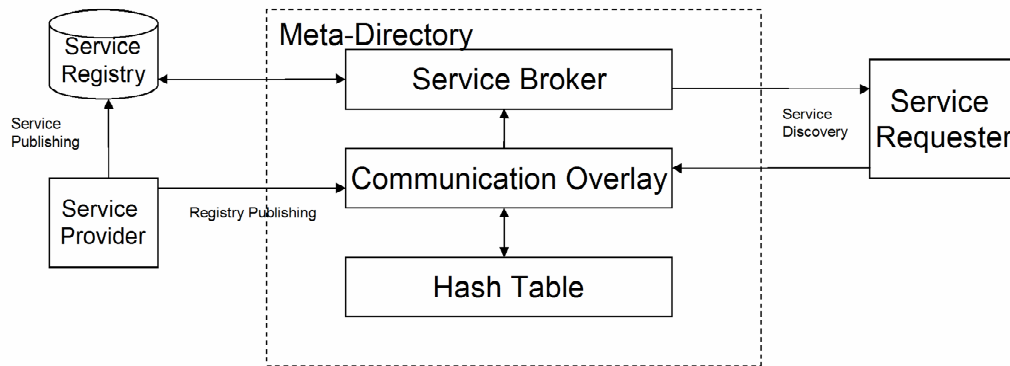


Figure 34 Meta-Directory Architecture

When a Service Provider wants to publish service information, the Service Provider first forwards the message to the Service Registry. Once the message is saved in the Service Registry, the Service Provider sends the message to the Meta-Directory System. This message is received by the Communication Overlay interface of the Meta-Directory node which then parses and hashes the attributes and forwards them to the Meta-Directory nodes responsible for that hash value for storage.

During a Service Discovery, the Service Requester sends the query to the Meta-Directory System through the Communication Overlay Interface. The interface parses and hashes the query and sends a search request to the Hash Table. The Hash Table returns the location of the Service Registry. The location of the Service Registry along with the query is then forwarded to the Service Broker. The Service Broker forwards the

query to the appropriate Service Registry and the response received from the Registry is then forwarded back to the Service Requester by the Service Broker.

The following sub-section covers the implementation of the Meta-Directory components and detailed sequence diagrams are then discussed.

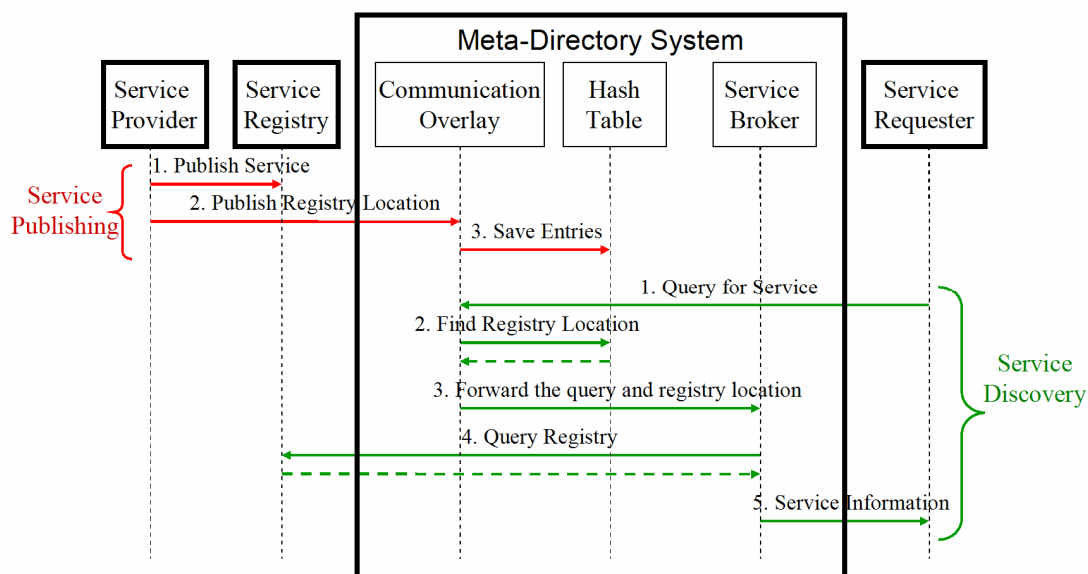


Figure 35 Service Publishing and Discovery in Meta-Directory Components

4.2 Meta-Directory System Implementation

This section explains how the configurable Meta-Directory system was implemented. For the implementation of the Meta-Directory system, an existing CHORD implementation was used to realize both the Fully Connected (DHT) and the CHORD (DHT) network models.

Open Chord [12] was used for the CHORD implementation. Open Chord is a stable Java implementation of the CHORD forwarding algorithm that was developed by the Distributed and Mobile Systems Group of Bamberg University. The Open Chord system allows the user to use the CHORD distributed hash table within any Java application. The

following properties of Open Chord provide advantages to the implementation of the Meta-Directory system.

- Open Chord provides easy to use interfaces for synchronous and asynchronous utilization of CHORD.
- The user can store any serializable Java object within the distributed hash table.
- The programmer can create custom keys to associate data with.
- Provides transparent maintenance of CHORD forwarding and routing tables.
- Facilitates configurable replication of entries within the distributed network. This is very useful because if a node dies or leaves without notice, the other nodes in the system will have the replicated data.
- Open Chord provides a remote communication protocol based on Java sockets.
- Open Chord also provides a local communication protocol that can be used to create networks within the same virtual machine for testing and presentation purposes.

For the CHORD (DHT) distribution model, Open Chord was deployed as is and no manipulation was done in the system for the forwarding algorithm. In order to emulate a structured Fully Connected (DHT) network, the Open Chord forwarding table was modified to include all the nodes in the network. The size of the forwarding table also has to be maintained such that it is always greater than the number of nodes in the network. This way, the forwarding table will always have $(N-1)$ nodes in the forwarding table where N is the total number of nodes.

Figure 36 illustrates the association between the two packages used in the Meta-Directory implementation. The *com.akassim.overlay* package contains the classes that

implement the fully connected, the Fully Connected (DHT), and the CHORD (DHT) forwarding algorithms for the Meta-Directory System. The *de.uniba.wiai.lspi.chord.service.impl* package implements the CHORD communication protocol using the Open Chord implementation. The *com.akassim.overlay* package achieves its functionality by using incorporating the functionality provided by the *de.uniba.wiai.lspi.chord.service.impl* package.

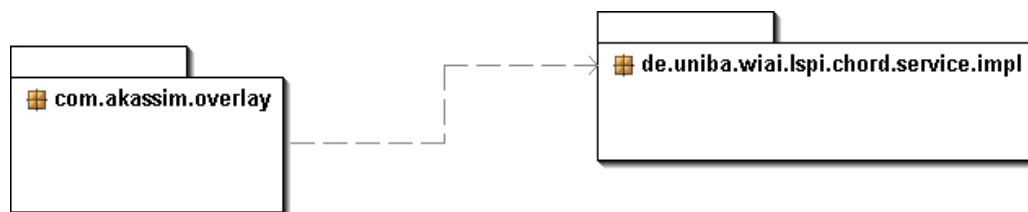


Figure 36 Meta-Directory Packages

The classes implemented in the *com.akassim.overlay* package are shown in the class diagram in Figure 37. The *Network* interface provides a unified interface for the message processing, network joining and network leaving functions of the *ChordRing*, *FullyConnected*, and *SuperPeer* classes. This is provided as the underlying communication model, and it is transparent to the user. The user will send the messages in the same format regardless of the underlying message distribution protocol.

The *ChordRing* class implements both the Fully Connected (DHT) distribution model and the CHORD (DHT) distribution model. The *ChordRing* class provides a *setSuccessorList* method so that the size of the forwarding table can be modified during runtime. This provides the ability of the network to evolve from a Fully Connected (DHT) model to a CHORD (DHT) system without the need of shutting down the system.

The *FullyConnected* class implements the fully connected unstructured distribution model. For this model, the hash table entries are saved locally by the receiving node as

discussed in section 3.4.1.1. This is because the Meta-Directory nodes are not classified to handle only a subset of hash table entries but whenever a publish request is received; the receiving Meta-directory node stores all the key value pairs in the local hash table.

The SuperPeer class implements the protocol for the Super Peer nodes. These Super Peer nodes are instantiated when there is more than one cluster in the network and the communication is per the Super Peer network as discussed in Section 3.4.1.3.

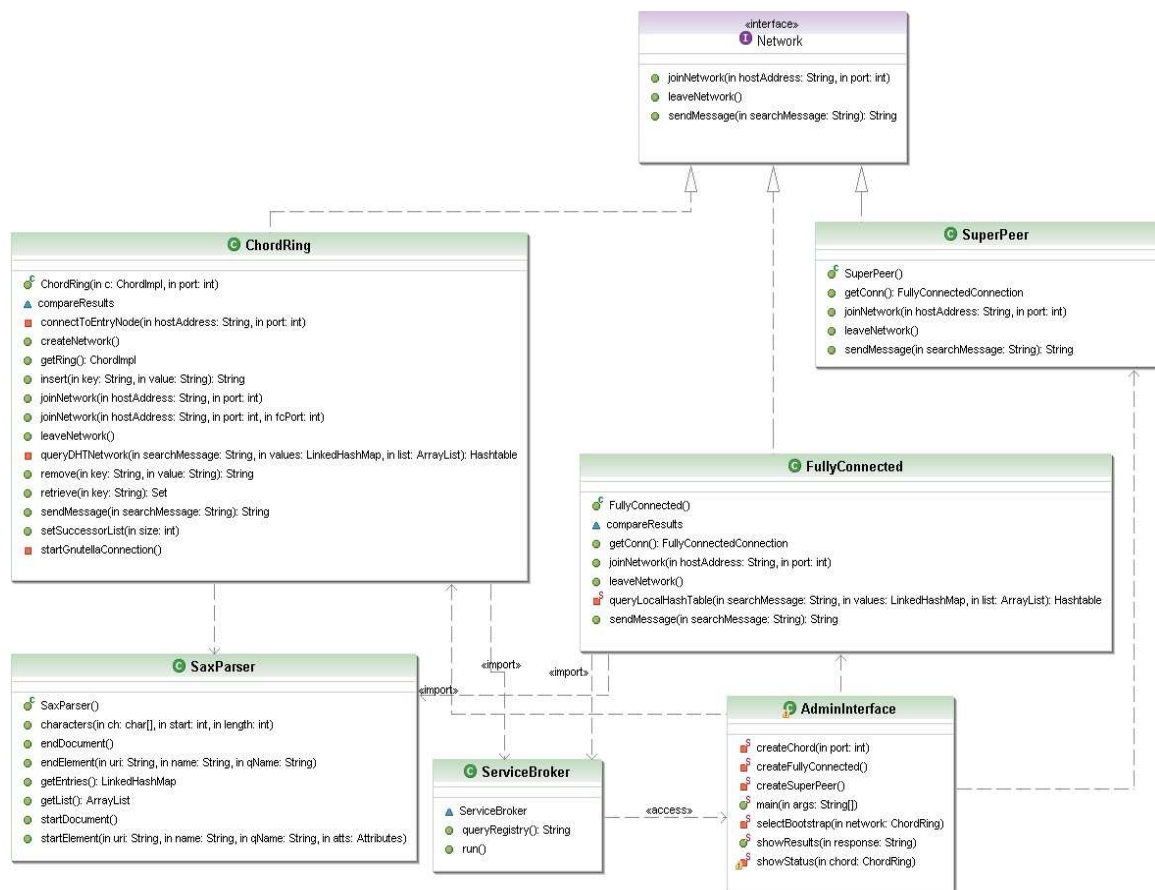


Figure 37 UML Class Diagram for the com.akassim.overlay Package

The ServiceBroker class implements the ServiceBroker component of the Meta-Directory architecture. As such, a ServiceBroker is instantiated whenever a registry location is found in the Meta-Directory System and the user's query has to be forwarded to the registry.

The SaxParser class implements a Simple API for XML (Sax) Parser that is a serial access parser API for XML [21]. This parser triggers the appropriate event whenever an element is found in the data. This class is used to parse the messages received as all communication is done using the XML format.

The AdminInterface class was implemented for testing purposes. This interface instantiates the different architectures as well as forwards requests to the architectures.

The following section discusses the sequence diagrams that realize the Meta-Directory use cases.

4.2.1 Meta-Directory Sequence Diagrams

This section illustrates the sequence diagrams that realize the Meta-Directory Use Cases. The first Use Case that we are going to look at is the “Select Topology” Use Case which is initiated by the System Administrator. This is the Use Case that allows the system administrator to select the network model during system deployment. This flexibility is provided so as to ensure that the network’s performance provides low latency as well as minimizes network bandwidth usage based on the initial state of the system.

Figure 38 shows the message interaction when a new Meta-directory node wants to join a Fully Connected network. A new instance of the FullyConnected class is created and whenever a new node has to be added as a neighbor, the joinNetwork method is invoked with the IP address and port number of the new node passed on as parameters.

When the system administrator wants to create a super peer node, the message interaction in Figure 39 is followed. A new instance of the SuperPeer class is created, and

the IP address and port number of the other super peers in the network as well as the entry node of the cluster the super peer is responsible for must be passed to the instance.

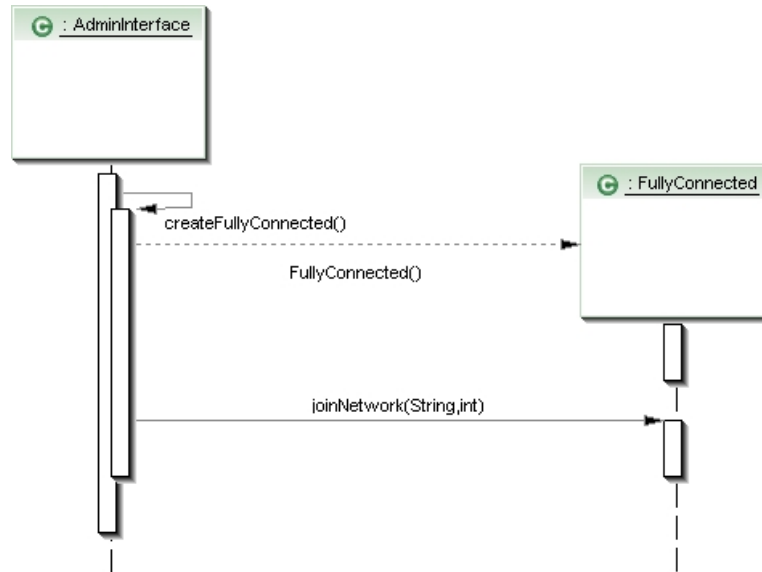


Figure 38 Setting up a Fully Connected Network

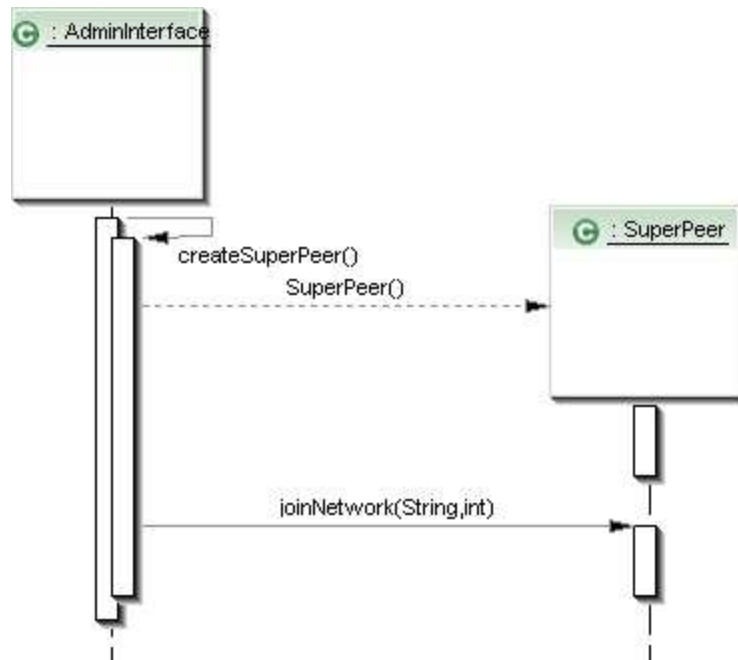


Figure 39 Creating a Super Peer Node

When a node joins or creates a DHT network, the message interactions in Figure 40 are invoked instead. In this case, a new instance of ChordRing is created first. If the node wants to create a new network, the createNetwork method is invoked on the ChordRing, otherwise a bootstrap node has to be selected from the existing network. The IP address and the port number of the bootstrap node is then passed to the ChordRing instance through the joinNetwork method.

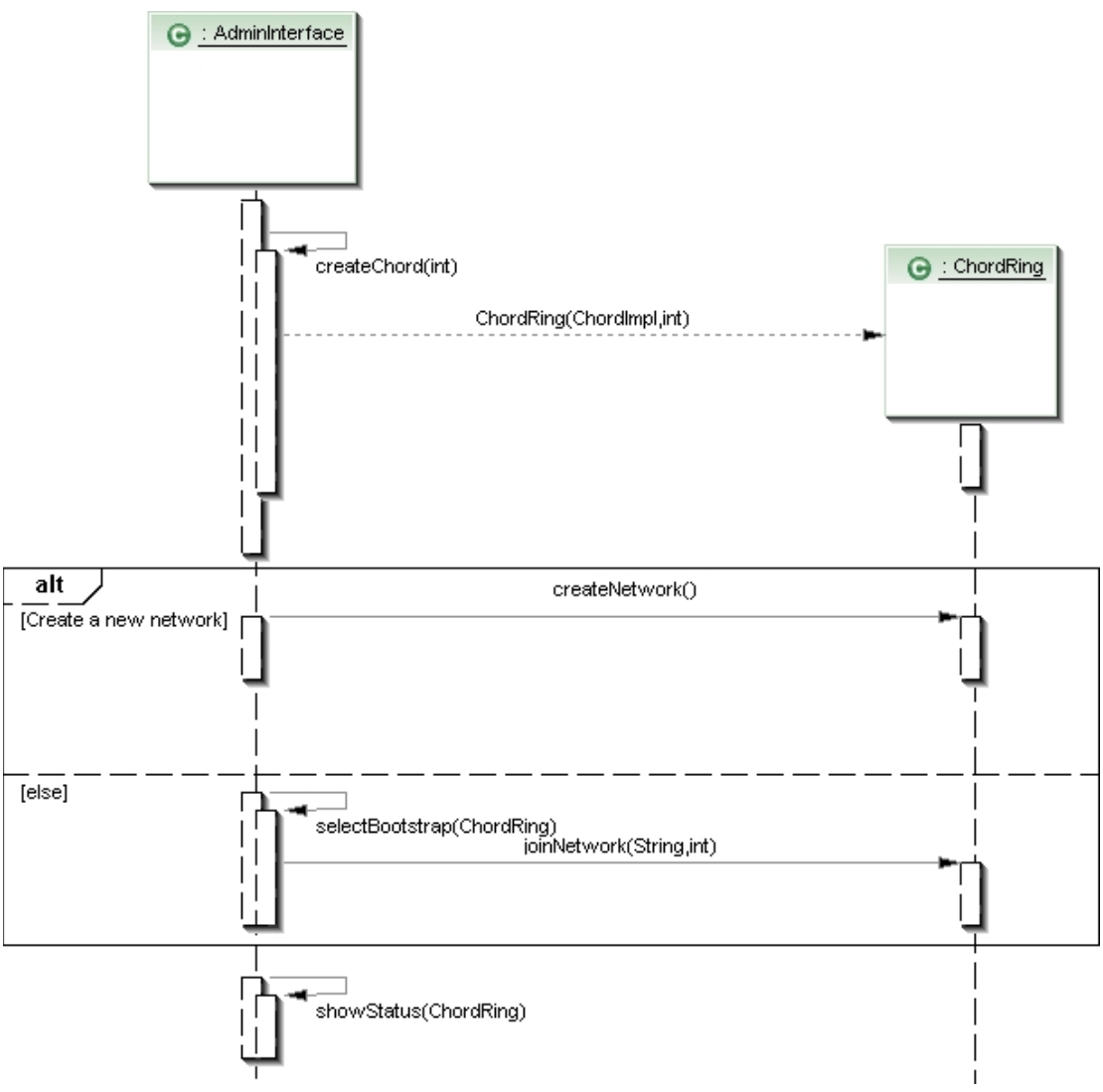


Figure 40 Joining or Creating a DHT Network

During runtime, the network can be varied between the Fully Connected (DHT) and the CHORD (DHT) model. This property is made possible through the ChordRing interface as illustrated in Figure 41. The setSuccessorList method is invoked on any one of the ChordRing instances and the internal routing architecture of the Meta-Directory node is modified to the selected architecture. The Meta-Directory node then invokes the setSuccessorList method on the rest of the nodes in the network.

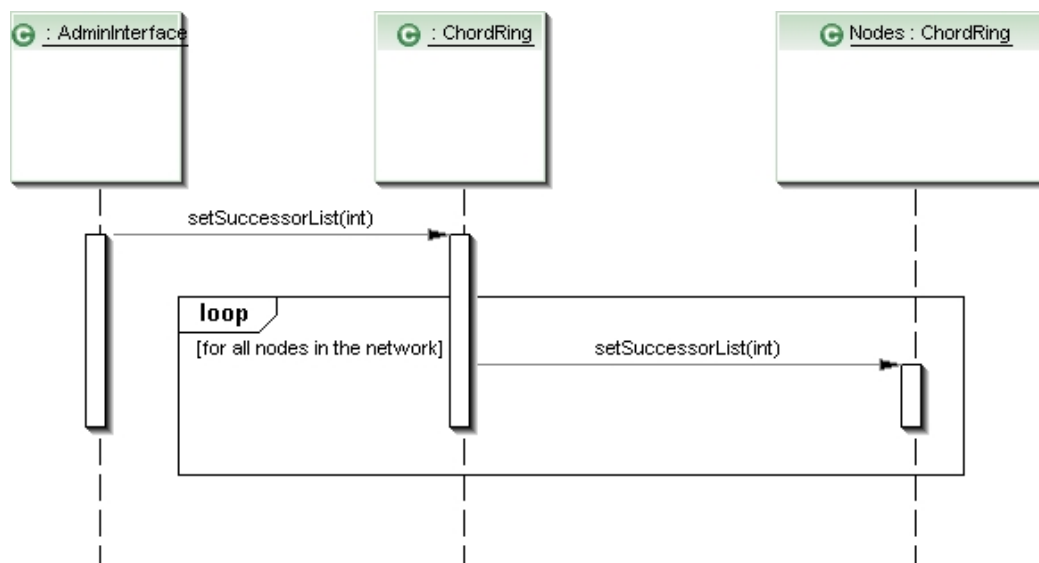


Figure 41 Adapting the Network between Fully Connected (DHT) and CHORD (DHT)

Figure 42 shows the message exchange when a service is published in a fully connected network. The sendMessage method is invoked with the XML query as the String parameter. The FullyConnected instance instantiates a SaxParser and forwards the XML query to the parser. Once the message is parsed and the attributes identified through the getEntries method, the key-value pairs are then forwarded and saved in the local hash table using the addAll interface. The FullyConnected instance then invokes the

showResults method of the AdminInterface that would show if the publish request was successful.

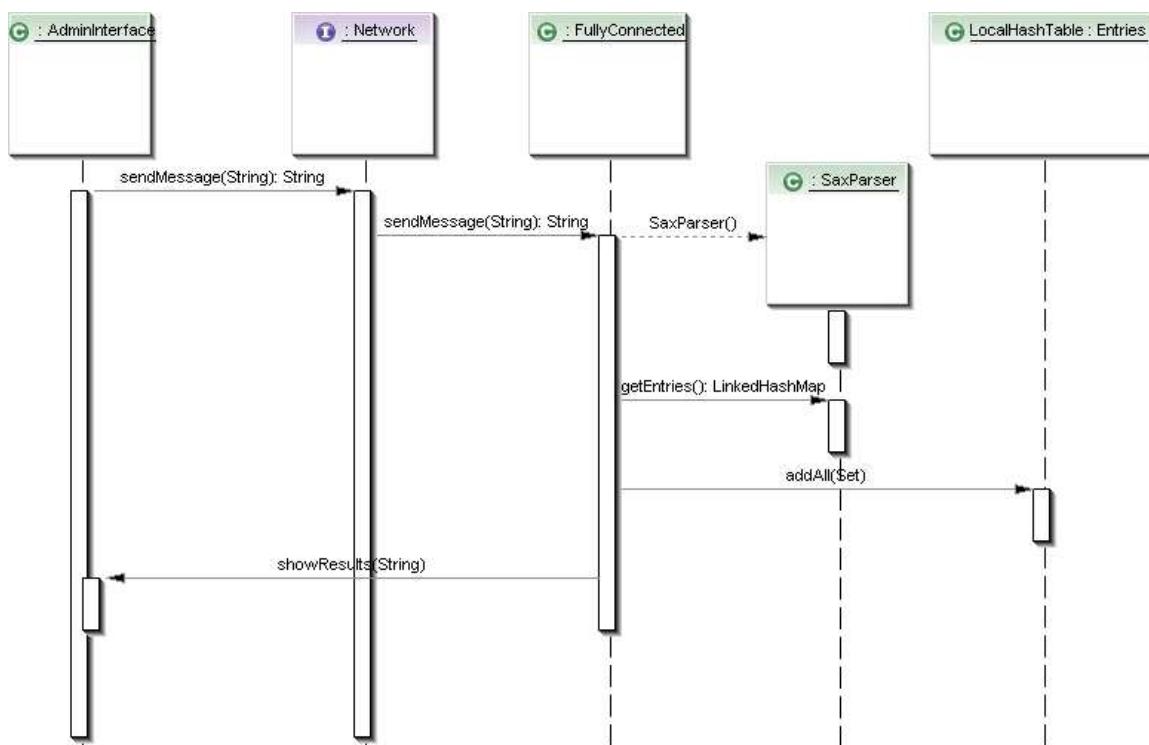


Figure 42 Publishing a Service in a Fully Connected Network Sequence Diagram

When a service is published in a DHT network, the message is forwarded from the Network interface to the ChordRing instance as illustrated in Figure 43. The ChordRing instantiates a SaxParser and forwards the XML message to the parser. Once the message is parsed, the ChordRing instance saves the attributes in the DHT network. The Open Chord package handles the data distribution and forwarding algorithms of the system in this case. Therefore, the attributes are distributed through the DHT network based on their hashed key values. Once the data is handled, the ChordRing instance invokes the showResults method of the AdminInterface indicating the outcome of the request. That is, if the publish request was successful or not.

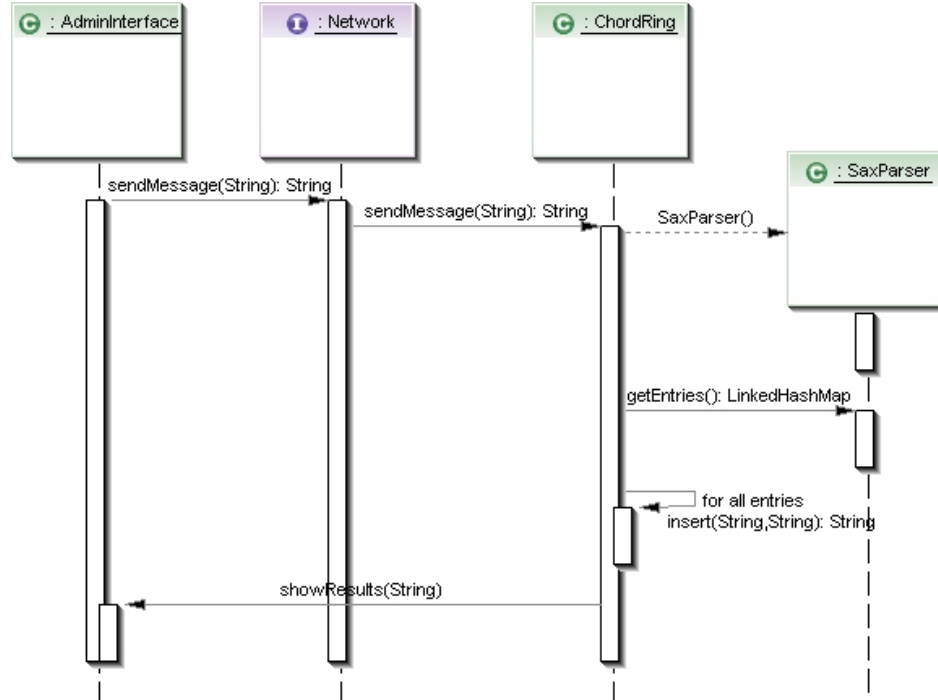


Figure 43 Publishing a Service in a DHT Network Sequence Diagram

Figure 44 illustrates the sequence diagram for the delete service use case in the fully connected network. This sequence diagram is similar to Figure 42, except when querying the hash table, instead of publishing the values, they are deleted from the local hash table. It should be noted that this architecture forces the Service Provider to only send the delete request to the same Meta-Directory node that the publish request was sent to. The showResults method would then indicate if the delete service request was successful.

The flow of messages when a service is deleted from a DHT network is similar to the interactions when a service is published. This is shown in Figure 45. The only difference in this case is that once the attributes are hashed, a remove entries request is forwarded in the network for all the hashed keys. The main difference between the delete function in the DHT network and that of the Fully Connected network is that the Service Requestor

can send the delete service request to any of the Meta-Directory nodes in the DHT network.

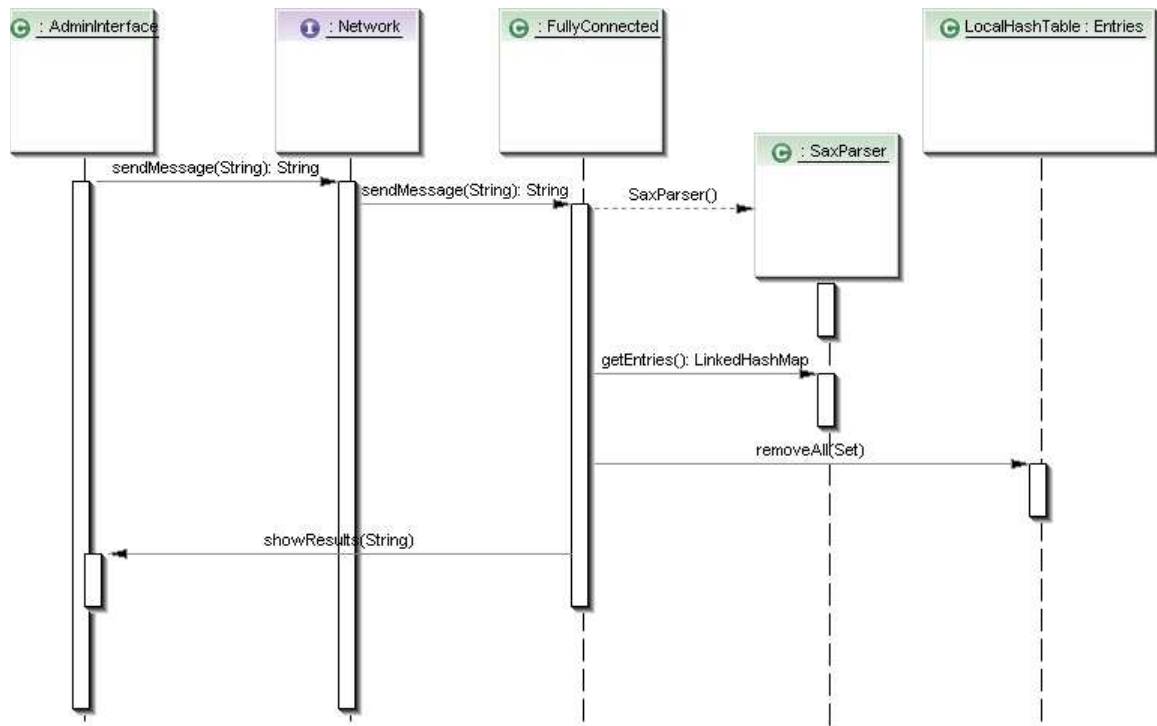


Figure 44 Delete Service in a Fully Connected Network Sequence Diagram

The main feature of the Meta-Directory system is the service discovery use case. In a fully connected network, Figure 46, the message is first passed to the SaxParser which parses the XML message. Once the attributes are received, they are first forwarded to the local hash table for retrieval of the corresponding values. Since the key-value entries can be stored anywhere in the network, the message is broadcast to all the nodes in the Meta-Directory network. The initializing Meta-Directory node then waits for responses from the other Meta-Directory nodes.

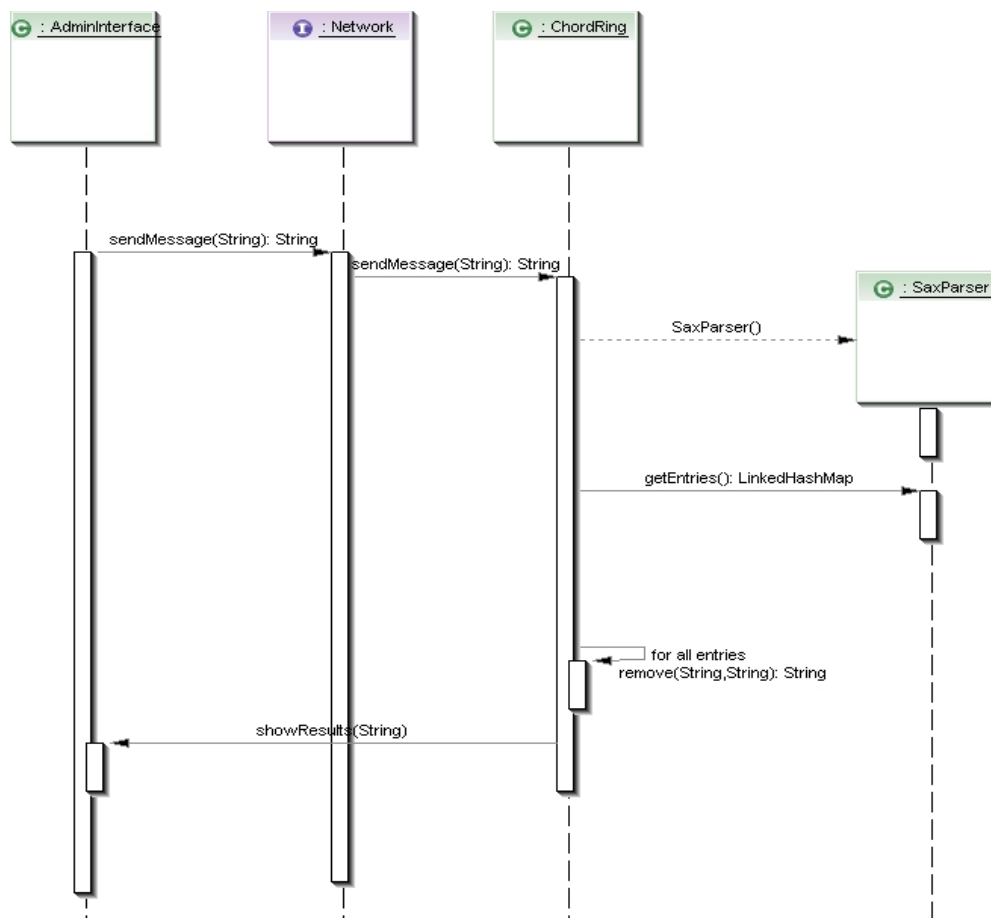


Figure 45 Delete Service in a DHT Network Sequence Diagram

Once the responses are received, the Meta-Directory node then invokes the `compareResults` method. With this method, the Meta-Directory node ensures that the query is only forwarded to the appropriate Service Registries. For example, if a user sends a service discovery request where they are looking for a company called “Network Solutions” that offers services in the “movie” category. These two attributes are hashed independently and forwarded to the network. The results will include all the Service Registries that have information on all the services offered by “Network Solutions” as well as all the companies that offer services under the “movie” category. Therefore, the Meta-Directory node then compares the results and only queries the Service Registry(s)

that provide service information in the “movie” category for “Network Solutions”. The Meta-Directory node then instantiates a ServiceBroker giving the ServiceBroker instance the query message along with a list of the Service Registries. The ServiceBroker then queries all the Service Registries in the list and returns the response to the user.

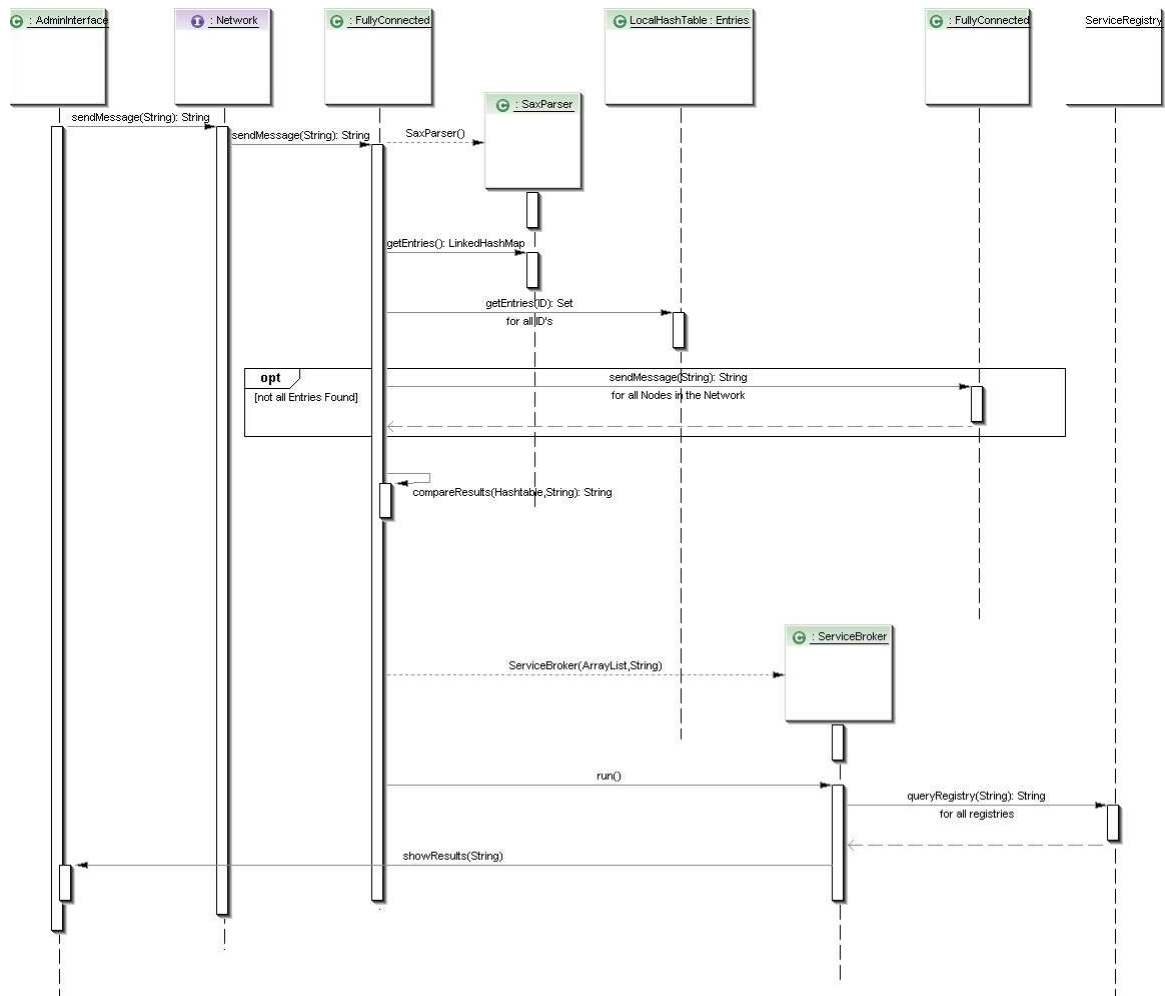


Figure 46 Service Discovery in a Fully Connected Network Sequence Diagram

In the case of a DHT based network, once the attributes are parsed and hashed, the values corresponding to the hashed keys are retrieved from the network using the Open Chord interface as shown in Figure 47. The results are then compared using the compareResults function as was done in the fully connected network discussed in the

previous paragraph. The results are then forwarded to an instance of the ServiceBroker which queries all the registries retrieved with the service discovery request. The response from the Service Registries is then forwarded back to the user.

A Meta-Directory system implementation was thus realized and the following section evaluates the prototype implemented by performing functional tests on the system as well as analyses the scalability of the system through performance tests done using PlanetLab.

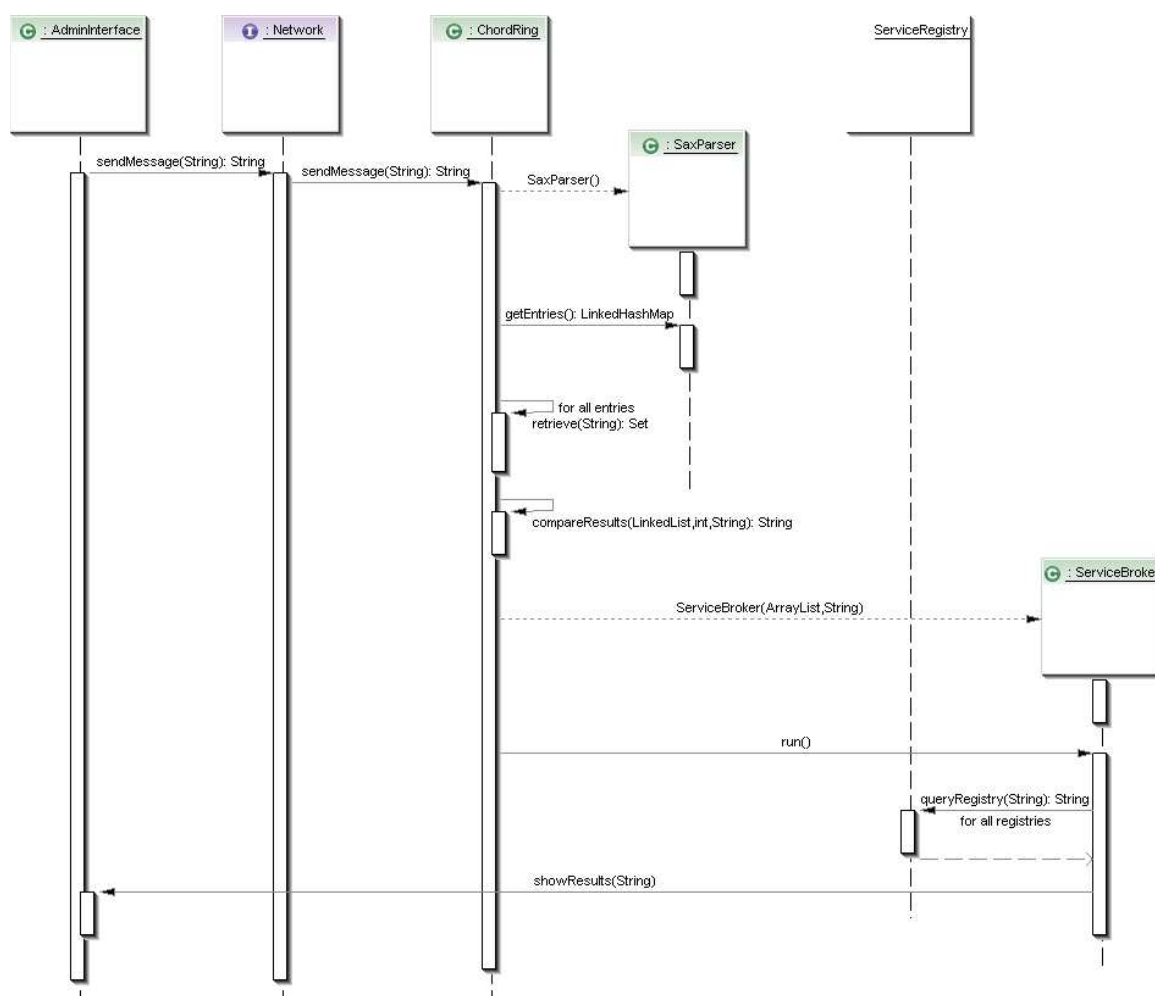


Figure 47 Service Discovery in a DHT Network Sequence Diagram

CHAPTER 5 PROTOTYPE TESTING

A Meta-Directory prototype was designed and implemented as per the design and implementation methodologies discussed in the previous chapters. During implementation, unit tests were performed on each function to ensure that the system performs as per the use cases described. In addition to the unit tests, evaluations were performed so as to analyze the Meta-Directory's performance in terms of the functional requirements. The functional testing was done to ensure that the prototype conforms to the Meta-Directory's specification as per the use cases discussed. Performance analysis was also done on the Meta-Directory prototype to illustrate the scalability of the system. This chapter presents and discusses the results performed on the system.

5.1 Functional Testing

This sub-section looks at the functional testing performed on the Meta-Directory prototype implemented. In order to view the results of the Meta-Directory node, an administrative interface was developed that can be used to probe the Meta-Directory nodes as well as display the results from the Meta-directory nodes.

Currently the Meta-Directory prototype allows static configuration of the network model during deployment. When the administrative interface is started, the user has an option of deciding which network (see Figure 48), among the Fully Connected, Fully Connected (DHT), CHORD (DHT), and the Super Peer networks the Meta-Directory node is going to use in the overlay layer when communicating with other Meta-Directory nodes.

It should be noted that when a Fully Connected node is created on a machine, the system automatically selects the port number 6346 as the default port that the Fully Connected node is listening on. On the other hand, when creating a DHT network the user has to specify the port the Meta-Directory node will be listening on as shown in Figure 48. Also, more than one node can be created on the same machine by specifying a different port number, for the Fully Connected node the port number automatically increments to the next available port.

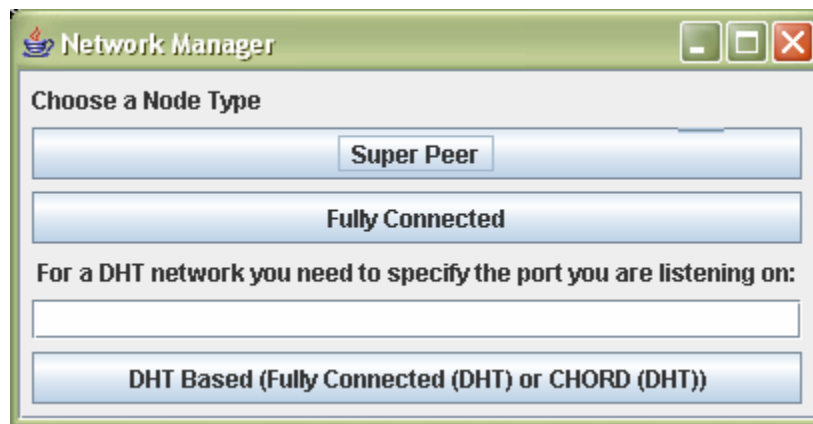


Figure 48 Network Setup

5.1.1 CHORD (DHT) Network

A CHORD (DHT) network is realized by creating a network graph as per the discussion on CHORD (DHT) networks in Section 3.3.1. This section illustrates how a CHORD (DHT) network can be created as well as shows how information is published and searched in the network.

5.1.1.1 Network Setup

When the user decides to start a DHT node, the user has the option of either creating a new DHT network or joining an already existing DHT network as illustrated in Figure

49. If the user decides to join a pre-existing network, the IP address, the DHT port, and FC port of the bootstrap node in the pre-existing network has to be provided as shown in Figure 50. The FC port has to be provided to facilitate the communication between clusters if the network consists of multiple clusters. If a new DHT network is created, the user automatically gets a DHT administrative screen as shown in Figure 51.



Figure 49 Creating a DHT Node

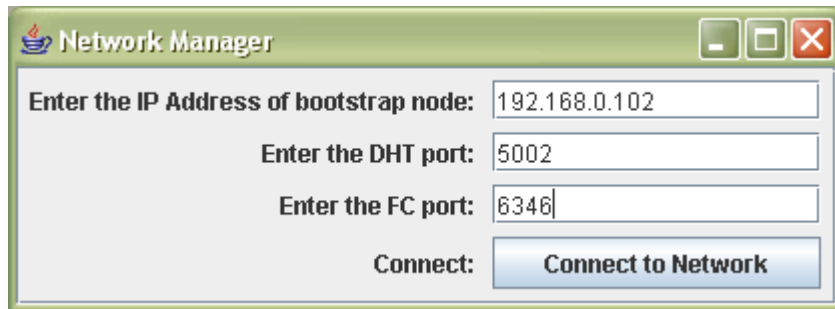
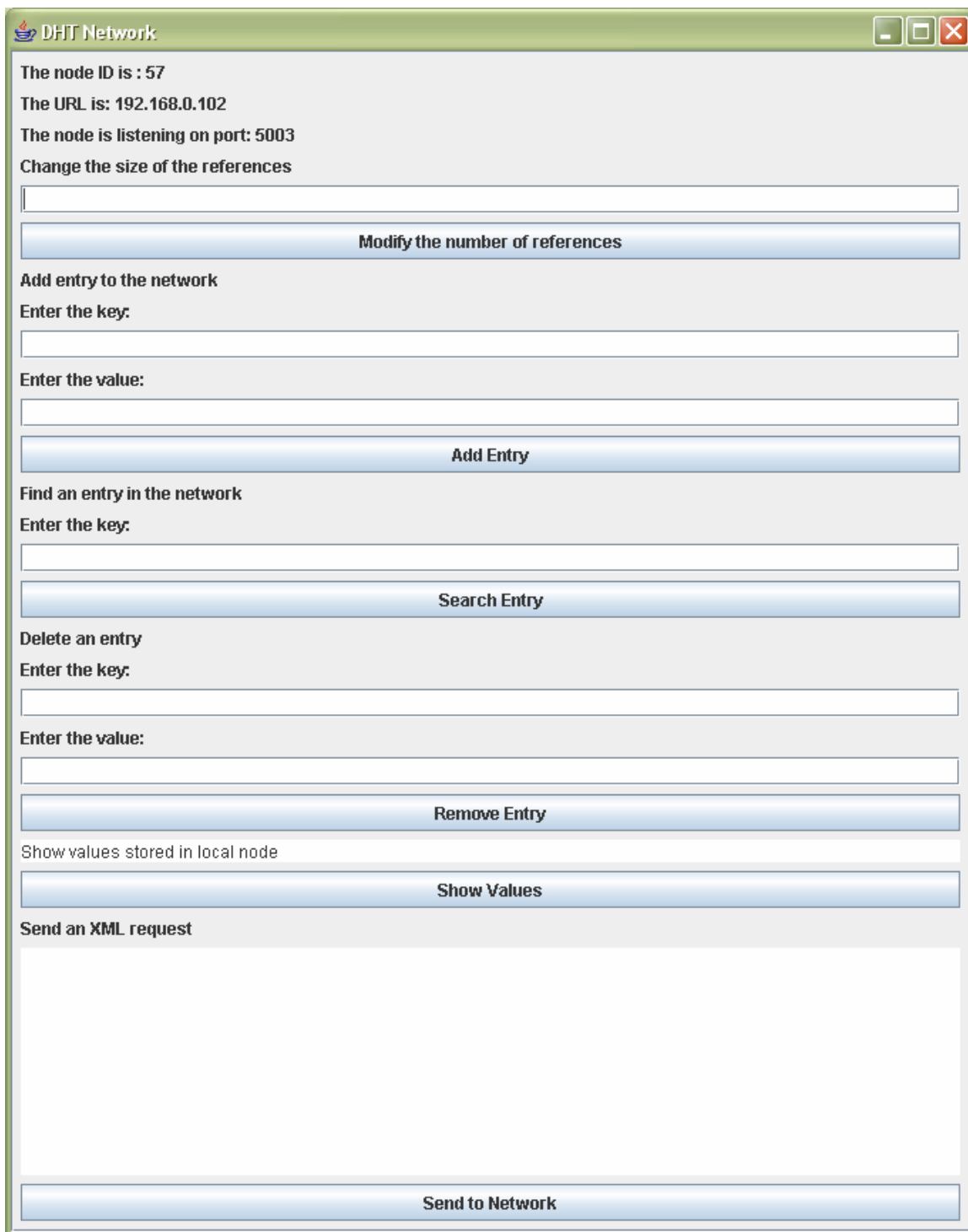


Figure 50 Connecting to a DHT Network

The administrative screen in Figure 51 illustrates the node ID in the DHT network, as well as the IP address and the port that the node is listening on. With this screen, the Meta-Directory nodes can be tested by manually inserting, searching, and deleting entries from the hash tables or by passing an XML request through the text screen provided.

During runtime, the user can also modify the DHT network from a Fully Connected (DHT) to a CHORD (DHT) network by modifying the number of references the DHT

node has in its finger table. The next section shows how information is published in a CHORD (DHT) network.



The screenshot shows a web-based administrative console for a DHT Network. The window title is "DHT Network". The interface is organized into several sections, each with a heading and a corresponding button:

- The node ID is : 57**
The URL is: 192.168.0.102
The node is listening on port: 5003
- Change the size of the references**
An empty text input field.
Modify the number of references (button)
- Add entry to the network**
Enter the key:
An empty text input field.
Enter the value:
An empty text input field.
Add Entry (button)
- Find an entry in the network**
Enter the key:
An empty text input field.
Search Entry (button)
- Delete an entry**
Enter the key:
An empty text input field.
Enter the value:
An empty text input field.
Remove Entry (button)
- Show values stored in local node**
Show Values (button)
- Send an XML request**
A large empty text area.
Send to Network (button)

Figure 51 DHT Administrative Console

5.1.1.2 Publishing Service Information

For the functional testing performed in this section, all the requests were sent using XML messages as shown in Figure 52. In order to evaluate the system, business and service information must first be published in the network. Figure 52 shows one of the XML messages sent through the network to publish a business service.

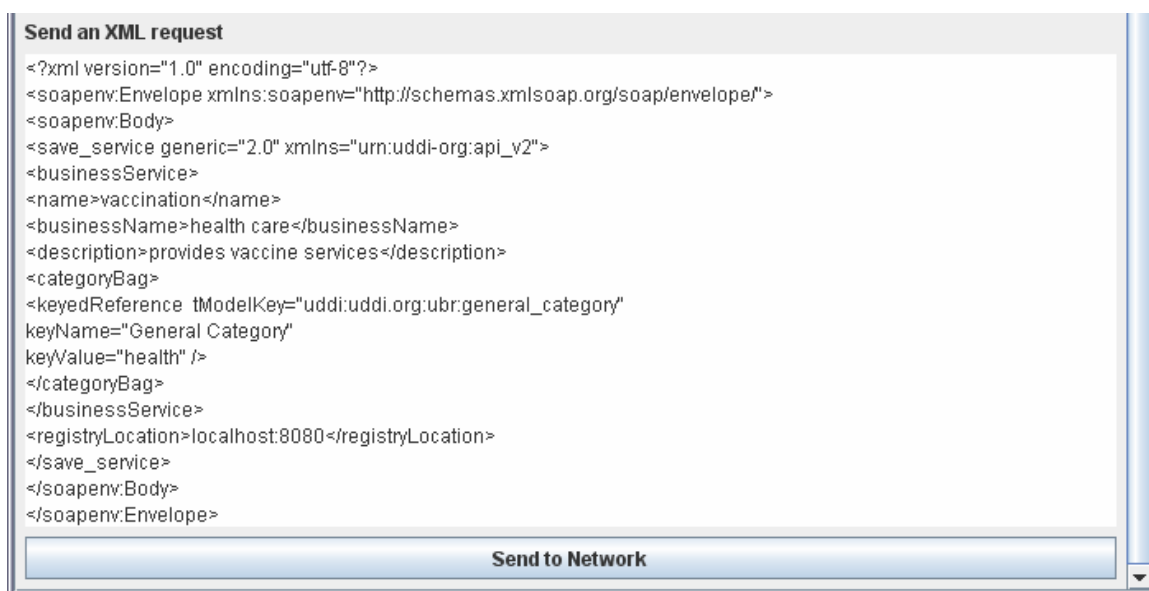


Figure 52 Publishing a Service on the Admin Console

Once business and service information was saved in the network, search messages were passed to the network through the Admin Console to determine if the prototype was functioning properly as per the use cases. These queries and system responses are discussed in the following section.

5.1.1.3 Querying Service Information

A find service request was then sent through the network with “vaccination” as the attribute for the service name. In this case the network returns two locations (as seen in

Figure 53) that have “vaccination” as the service name. As per the specifications of the Meta-Directory system, the system also queries the locations found and returns the results from the registries. Figure 54 shows the results returned from the system.



Figure 53 Registry Locations

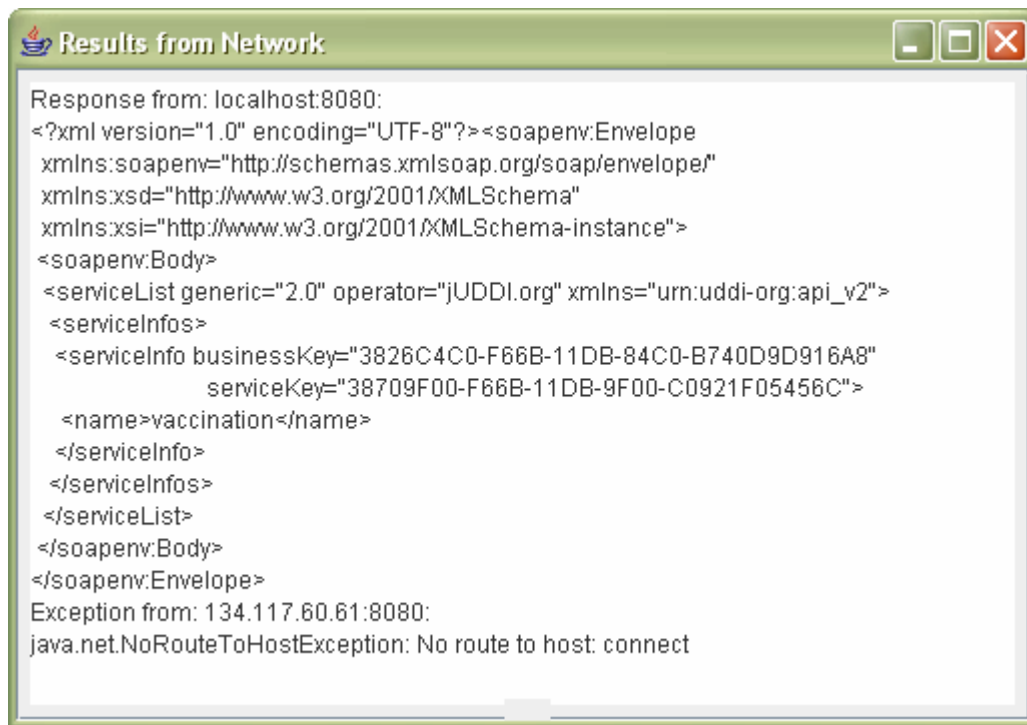


Figure 54 Response from the Registries Queried

The location at “134.117.60.61:8080” did not have a Service Registry running hence an exception was returned to the user. These results show that the system handles exceptions from unavailable Service Registries.

When the query request is narrowed, such that the query message illustrates both the service name as “vaccination” and the business name as “health care”, the system in this case will only query one registry as shown in Figure 55 and Figure 56.

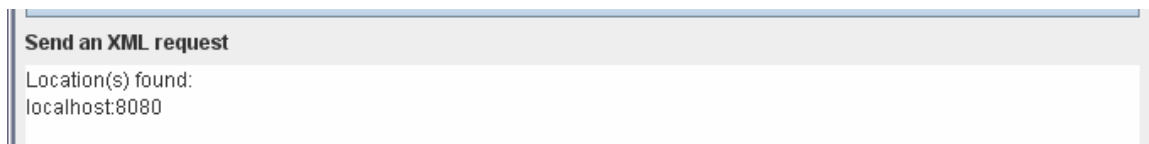


Figure 55 Registry Location Returned by Narrowing the Scope of the Query

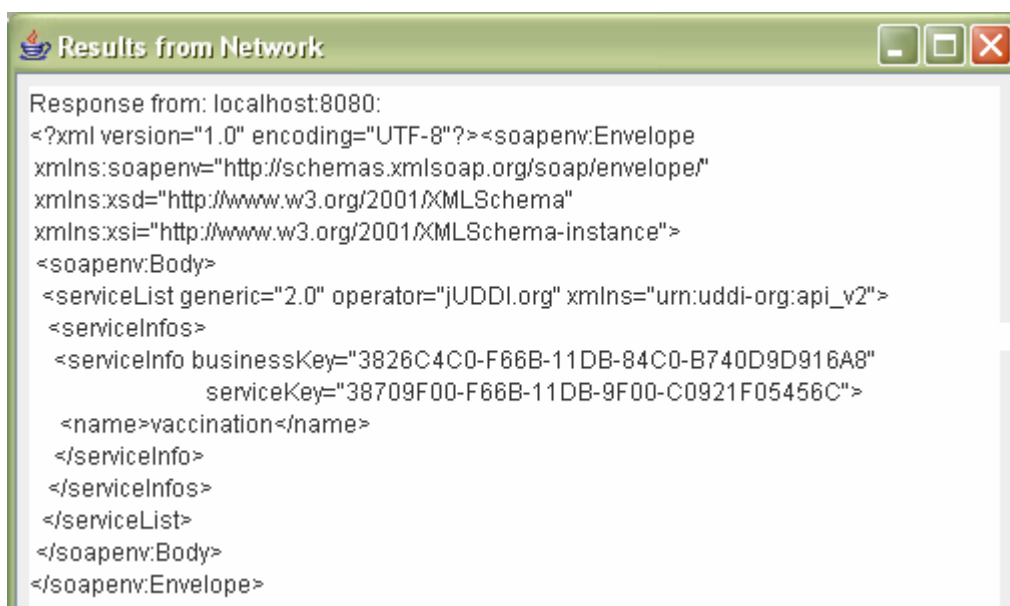


Figure 56 Response from the Registry

5.1.2 Fully Connected Network

A Fully Connected network is realized by creating a network graph as per the discussion on Fully Connected networks in Section 3.4.1.1. This section illustrates how a Fully Connected network can be created as well as shows how information is published and searched in the network.

5.1.2.1 Network Setup

When the user decides to deploy the network using the Fully Connected protocol, a node communicating with the Fully Connected network is thus deployed and the screen in Figure 57 is then shown. The view shows the IP address of the node as well as the port the node is listening on. The user can also use this interface to connect to additional Meta-Directory nodes using the Fully Connected protocol by providing the IP address as well as the port number to connect to.

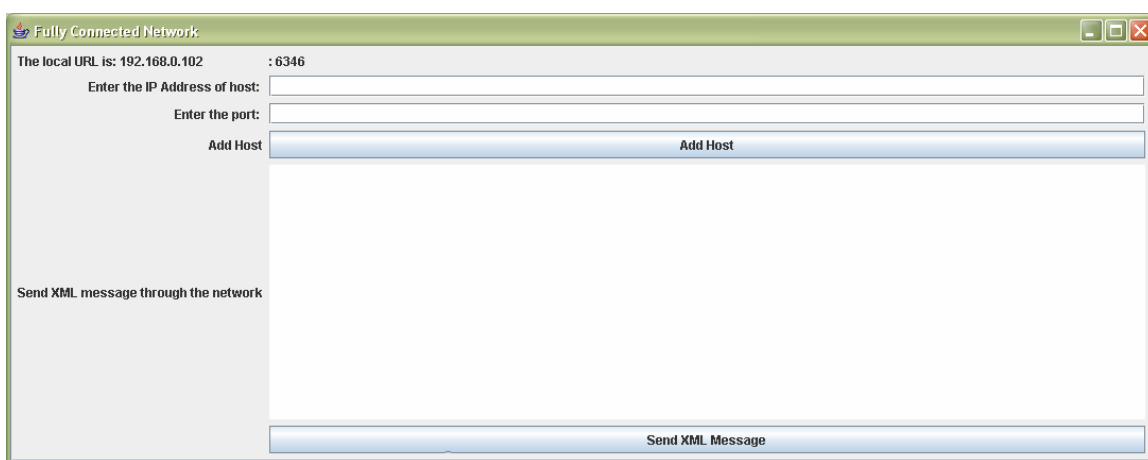


Figure 57 Fully Connected Administrative Console

For testing the functionality of a Fully Connected node, three nodes were created and connected to each other as per the Fully Connected protocol in Section 3.4.1.1. These nodes and the ports they were listening on are illustrated in Figure 58.

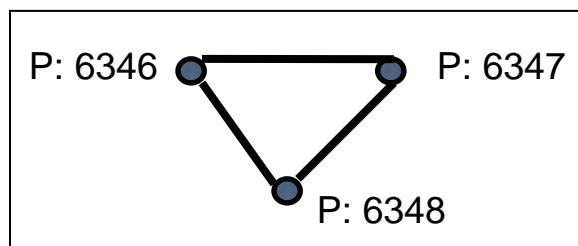


Figure 58 Fully Connected Network Instance

5.1.2.2 Publishing Service Information

When a fully connected node receives a publish request, the node hashes all the attributes received and stores the key-value pairs in the local hash table as per the Fully Connected protocol discussed in Section 3.4.1.1. Figure 59 shows the service information that is published in the fully connected node that is listening on port 6346. If the service information is published successfully, the corresponding key-value pairs are displayed as shown in Figure 60.

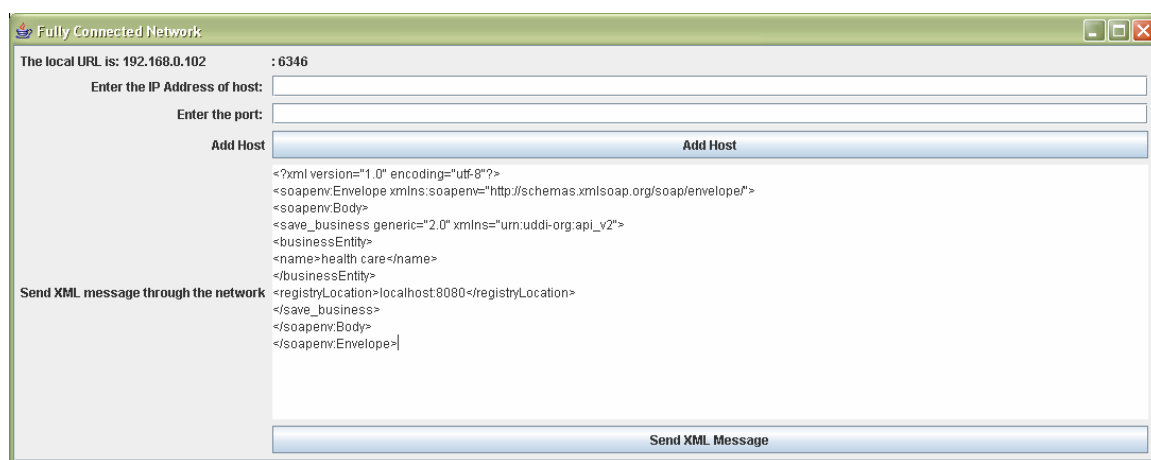


Figure 59 Service Publishing in a Fully Connected Network

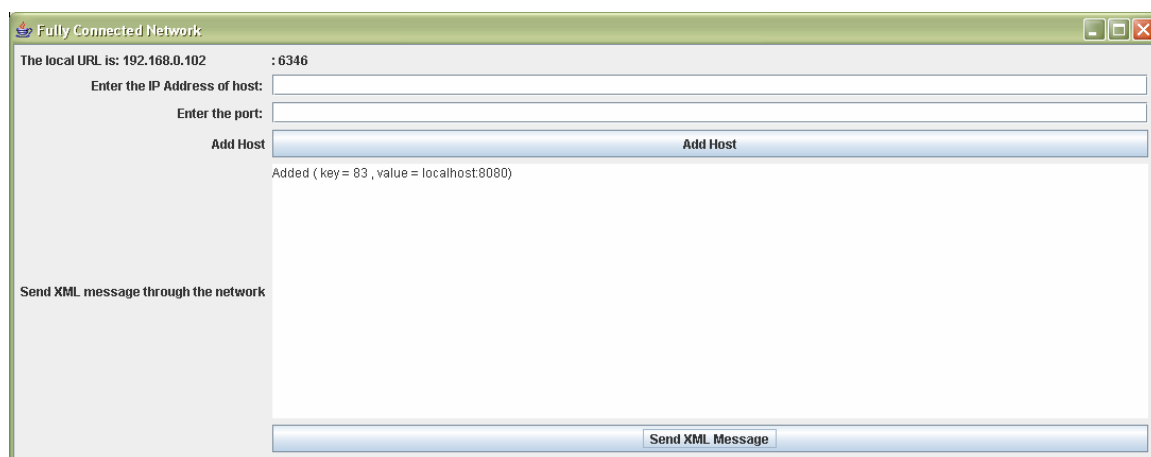


Figure 60 Successful Service Publishing in a Fully Connected Network

5.1.2.3 Querying Service Information

To illustrate the communication and the hash values distribution of a Fully Connected network, a query was issued for the service stored in the node listening on port 6346 by the node listening on port 6348 as shown in Figure 61. As per the Fully Connected network specifications, the request is forwarded to all the nodes in the network. In this instance (see Figure 58) the request is forwarded to the nodes listening on ports 6347 and 6346 as illustrated in Figure 62 and Figure 63.

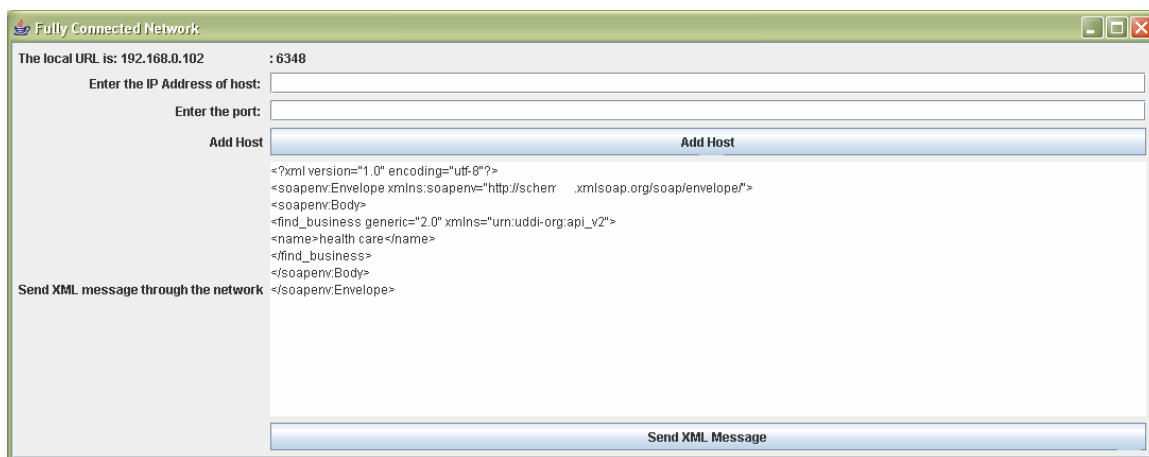


Figure 61 Query Issued by the Node Listening on Port 6348

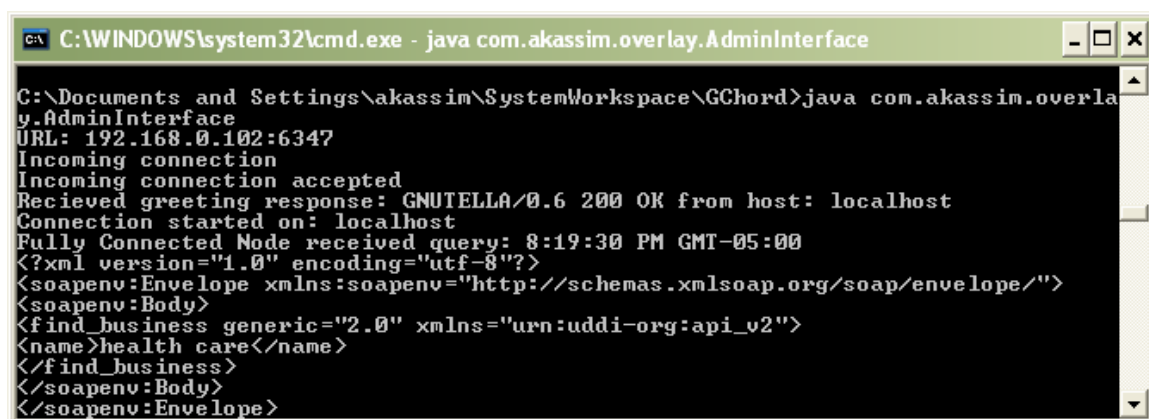
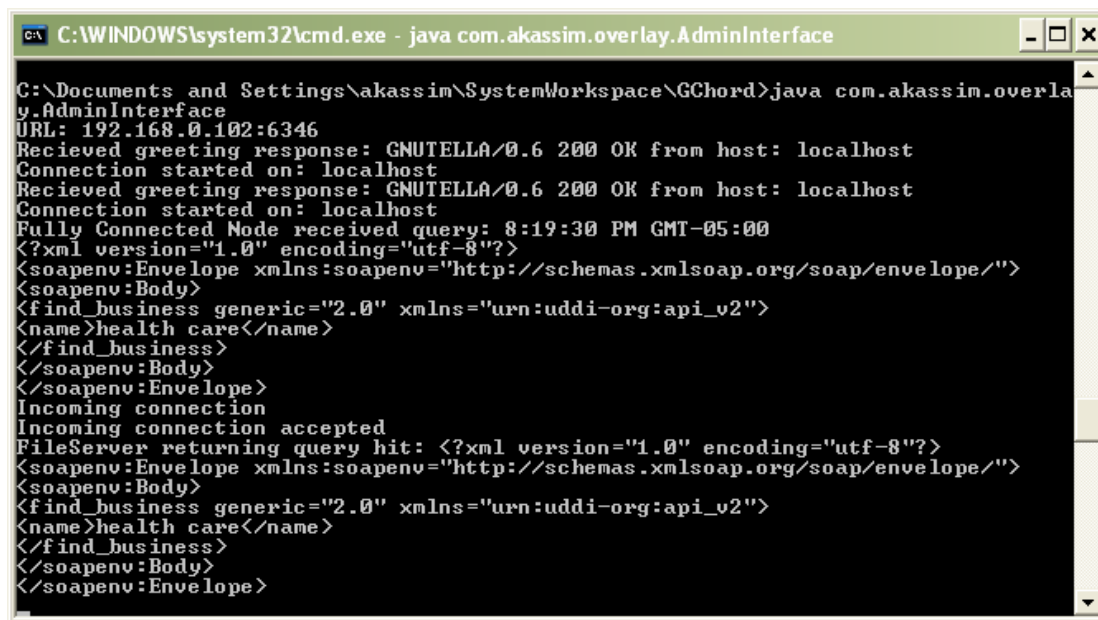


Figure 62 Request Received by Node Listening at Port 6347



```

C:\WINDOWS\system32\cmd.exe - java com.akassim.overlay.AdminInterface
C:\Documents and Settings\akassim\SystemWorkspace\GChord>java com.akassim.overlay.AdminInterface
URL: 192.168.0.102:6346
Received greeting response: GNUTELLA/0.6 200 OK from host: localhost
Connection started on: localhost
Received greeting response: GNUTELLA/0.6 200 OK from host: localhost
Connection started on: localhost
Fully Connected Node received query: 8:19:30 PM GMT-05:00
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
<name>health care</name>
</find_business>
</soapenv:Body>
</soapenv:Envelope>
Incoming connection
Incoming connection accepted
FileServer returning query hit: <?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
<name>health care</name>
</find_business>
</soapenv:Body>
</soapenv:Envelope>

```

Figure 63 Request Received by Node Listening on Port 6346

The nodes receiving the forwarded query then search their local hash tables and a response is only sent to the requesting node if a registry location is found. In this case the node listening on port 6346 returns the response to the requesting node that was listening on port 6348 as shown in Figure 63. The response received by the requesting node is shown in Figure 64 and it contains the IP address and port of the node that responded along with the location of the service registry. The query shown in Figure 61 is forwarded to the registry location and the response from the service registry is also shown in Figure 64. The response from the service registry includes the names of the services that the business offers as well as the business and service keys so that the requesting node can use these keys to query for more information directly from the service registry.

```

C:\WINDOWS\system32\cmd.exe - java com.akassim.overlay.AdminInterface
C:\Documents and Settings\akassim\SystemWorkspace\GChord>java com.akassim.overlay.AdminInterface
URL: 192.168.0.102:6348
Incoming connection
Incoming connection accepted
Incoming connection
Incoming connection accepted
<[]=1>
****Node received reply: 8:19:36 PM GMT-05:00
Port: 6346
IP Address:192.168.0.102
ID: GUID: [c0][a8][0][66][c0][a8][0][66][c0][a8][0][66][c0][a8][0][66]
Received response:
Response from: localhost:8080:
<?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <businessList generic="2.0" operator="jUDDI.org" xmlns="urn:uddi-org:api_v2">
      <businessInfos>
        <businessInfo businessKey="3826C4C0-F66B-11DB-84C0-B740D9D916A8">
          <name>health care</name>
          <description>provides health services</description>
          <serviceInfos>
            <serviceInfo businessKey="3826C4C0-F66B-11DB-84C0-B740D9D916A8" serviceKey="38709F00-F66B-11DB-9F00-C0921F05456C">
              <name>vaccination</name>
            </serviceInfo>
          </serviceInfos>
        </businessInfo>
      </businessInfos>
    </businessList>
  </soapenv:Body>
</soapenv:Envelope>

****END Search session received reply****

```

Figure 64 Response Received by Requesting Node Listening on Port 6348

5.1.3 Super Peer Network

A Super Peer network is realized by creating a network graph as per the discussion on Super Peer networks in Section 3.4.1.3. This section illustrates how a Super Peer network can be created as well as shows how information is published and searched in a Super Peer network.

5.1.3.1 Network Setup

To create a Super Peer network, super peer nodes must be created to manage the clusters of Meta-Directory nodes. A Super Peer node is created by selecting the Super Peer button using the network setup administrative console in Figure 48. The screen

would then change to that shown in Figure 65 indicating that the node is now a Super Peer node. The console shows the IP address as well as the port the super peer node is listening on. With the administrative console, the user can connect to other nodes by inputting the IP address of other super peer nodes in the network as well as the IP address of the entry node of the respective cluster.

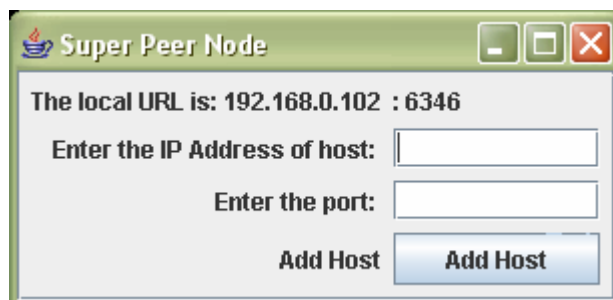


Figure 65 Super Peer Administrative Console

Two clusters communicating using CHORD (DHT) were created using the network setup instructions in Section 5.1.1.1, each with three nodes as shown in Figure 66. Two super peer nodes listening on ports 6346 and 6347 were also created and each super peer node was connected to the entry node of the cluster as well as to each other.

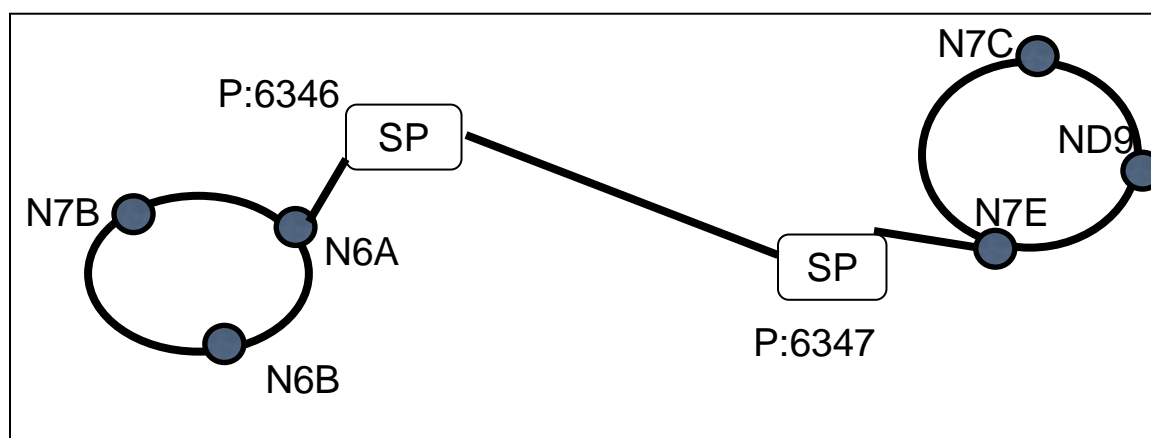


Figure 66 Super Peer Network Instance

5.1.3.2 Publishing Service Information

For this Super Peer network, services are published in the clusters in the same way as services are published in a CHORD (DHT) network as described in Section 5.1.1.2. To test the Super Peer network, the service information was published in node ND9 as shown in Figure 67. Once the service was published, the results in Figure 68 were displayed to show that the service information was saved successfully. Service information can be published in the rest of the nodes in a similar manner.

The following section discusses and illustrates how services are queried in a Super Peer network.

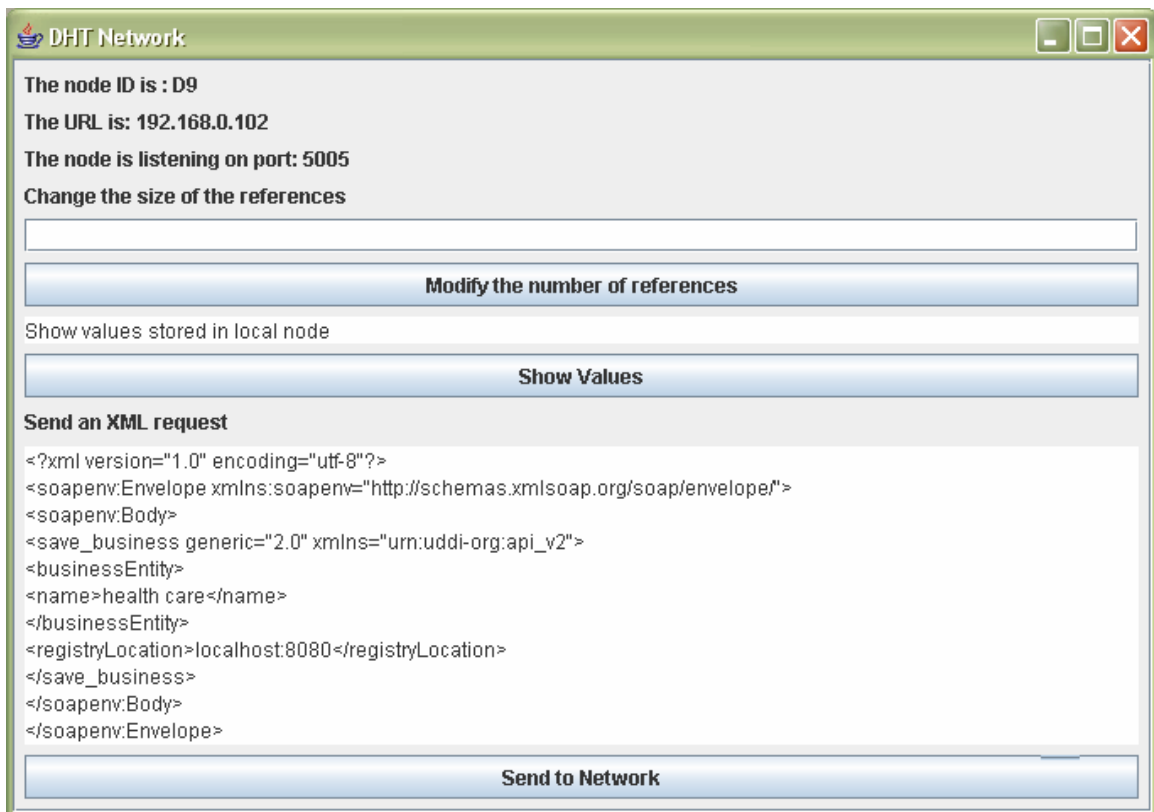


Figure 67 Service Publishing in SP Cluster

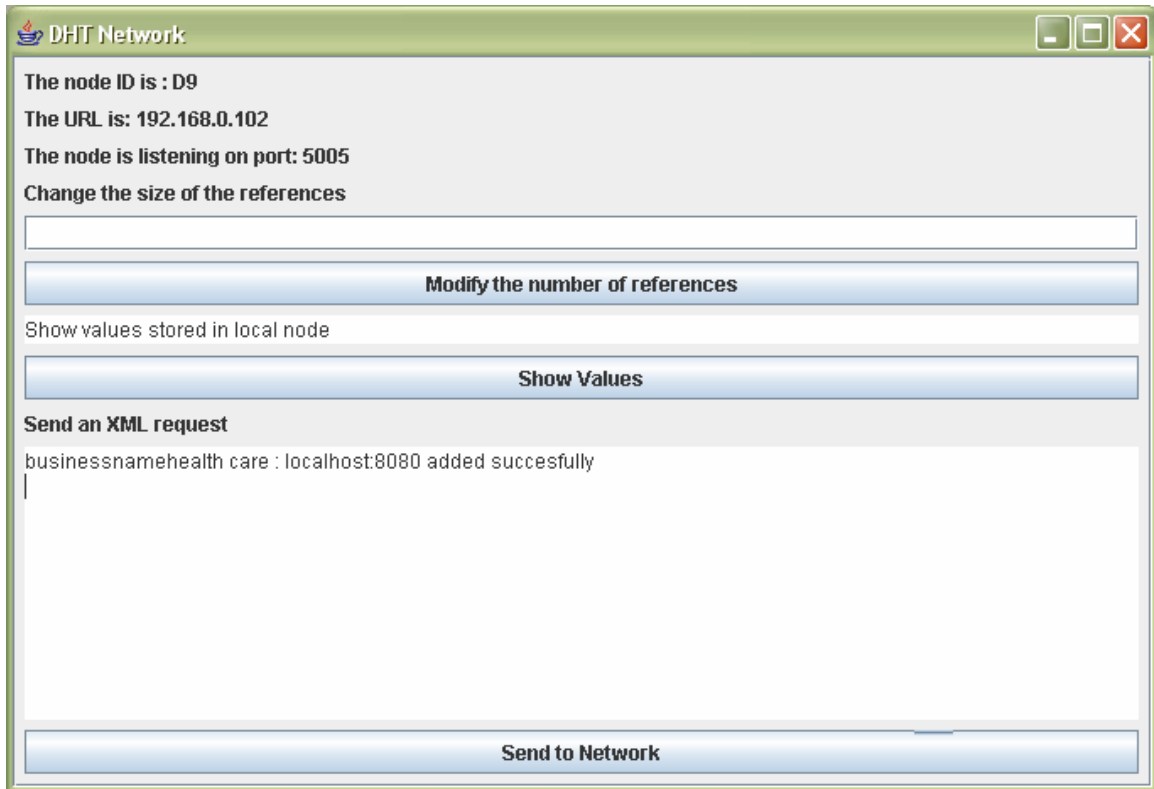


Figure 68 Successful Service Publishing in SP Cluster

5.1.3.3 Querying Service Information

Service discovery in a Super Peer network was tested by introducing a query on node N7B (see Figure 66). This service discovery request tests the network model in the Super Peer network to ensure that queries are forwarded within clusters if the cluster that received the query does not have a response. In this test case, the query is looking for a business with a business name “health care”. As we saw in the previous section, this business was saved in node ND9 to test inter-cluster communication; the query was initiated at node N7B (see communication graph in Figure 66).

Node N7B searched within its cluster and since there is no service saved in that cluster with the business name “health care”, node N7B forwarded the request (see Figure 69) to the entry node which is node N6A in this case.



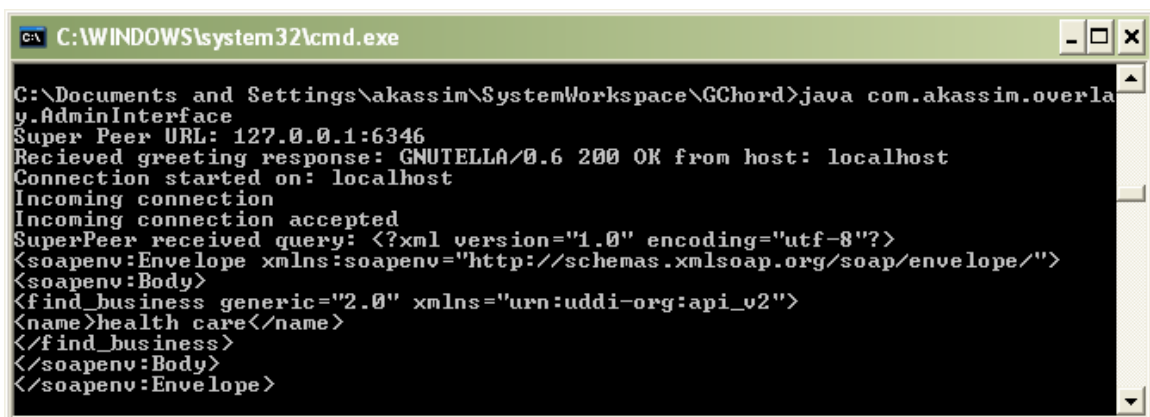
```

C:\WINDOWS\system32\cmd.exe
Recieved greeting response: GNUTELLA/0.6 200 OK from host: localhost
Connection started on: localhost
DHT network joined with node ID 6A
The URL is: ocssocket://127.0.0.1:5002/
Node received query: <?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
<name>health care</name>
</find_business>
</soapenv:Body>
</soapenv:Envelope>

```

Figure 69 Request Received by Entry Node

The entry node N6A forwarded the request to the super peer node it is connected to. According to Figure 66, this is the super peer node that is listening at port 6346. The query message is received by the super peer node as shown in Figure 70 and then forwarded to the super peer node at port 6347 (see Figure 71). The super peer node at port 6347 passed the request to the entry node of the second cluster. In this case, recall communication graph in Figure 66, node N7E is the entry node of the second cluster.

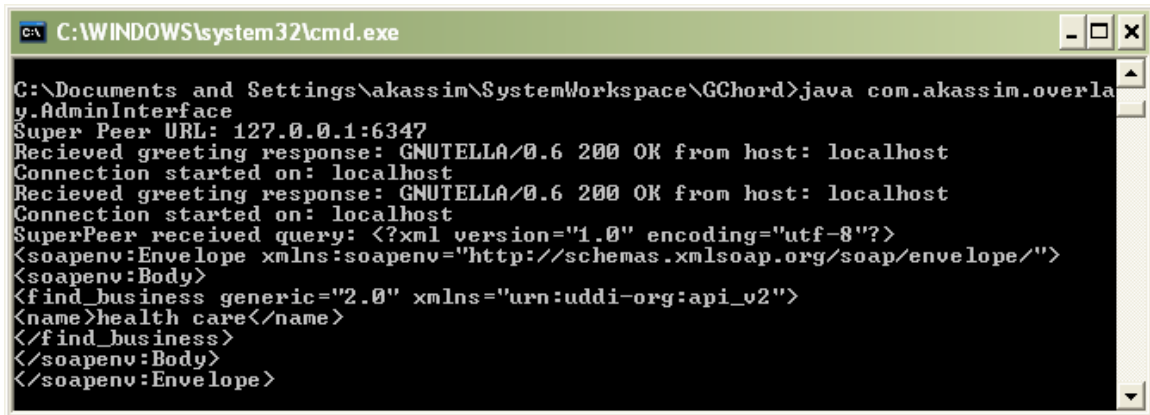


```

C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\akassin\SystemWorkspace\GChord>java com.akassin.overla
y.AdminInterface
Super Peer URL: 127.0.0.1:6346
Recieved greeting response: GNUTELLA/0.6 200 OK from host: localhost
Connection started on: localhost
Incoming connection
Incoming connection accepted
SuperPeer received query: <?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
<name>health care</name>
</find_business>
</soapenv:Body>
</soapenv:Envelope>

```

Figure 70 Request Received by Super Peer Node at Port 6346



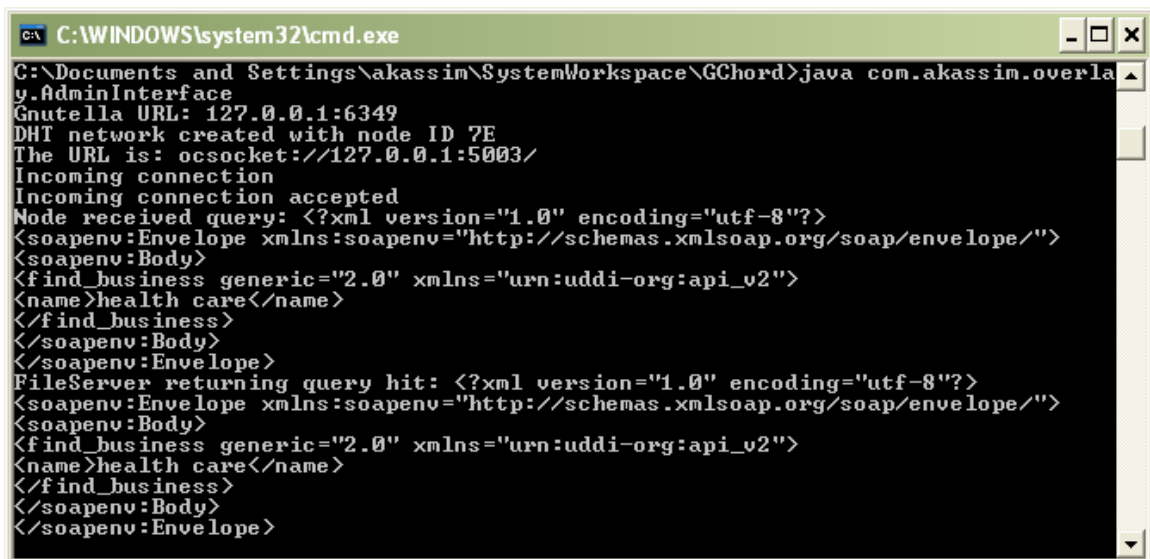
```

C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\akassin\SystemWorkspace\GChord>java com.akassin.overla
y.AdminInterface
Super Peer URL: 127.0.0.1:6347
Received greeting response: GNUTELLA/0.6 200 OK from host: localhost
Connection started on: localhost
Received greeting response: GNUTELLA/0.6 200 OK from host: localhost
Connection started on: localhost
SuperPeer received query: <?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
<name>health care</name>
</find_business>
</soapenv:Body>
</soapenv:Envelope>

```

Figure 71 Request Received by Super Peer Node at Port 6347

Node N7E then received the request and queried the second CHORD (DHT) cluster as shown in Figure 72. Once the response was found in the second cluster, node N7E returned the query response to the node where the originating request came from. In this case, the response is forwarded directly to node N7B (see Figure 73). The response has the business information node N7B was querying for along with the location of the registry that was queried as well as the business keys and service keys used in the registry so that node N7B can use these keys to get more information directly from the registry.



```

C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\akassin\SystemWorkspace\GChord>java com.akassin.overla
y.AdminInterface
Gnutella URL: 127.0.0.1:6349
DHT network created with node ID 7E
The URL is: ocsocet://127.0.0.1:5003/
Incoming connection
Incoming connection accepted
Node received query: <?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
<name>health care</name>
</find_business>
</soapenv:Body>
</soapenv:Envelope>
FileServer returning query hit: <?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/">
<soapenv:Body>
<find_business generic="2.0" xmlns="urn:uddi-org:api_v2">
<name>health care</name>
</find_business>
</soapenv:Body>
</soapenv:Envelope>

```

Figure 72 Request Received by the Entry Node N7E

```

C:\WINDOWS\system32\cmd.exe
C:\Documents and Settings\akassin\SystemWorkspace\GChord>java com.akassin.overla
y.AdminInterface
Gnutella URL: 127.0.0.1:6348
DHT network created with node ID 7B
The URL is: ocsocet://127.0.0.1:5000/
Incoming connection
Incoming connection accepted
Incoming connection
Incoming connection accepted
****Node Received Reply****
Port: 6349
IP Address:127.0.0.1
Received response:
Response from: localhost:8080:
<?xml version="1.0" encoding="UTF-8"?><soapenv:Envelope xmlns:soapenv="http://sc
hemas.xmlsoap.org/soap/envelope/" xmlns:xsd="http://www.w3.org/2001/XMLSchema" x
mlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <soapenv:Body>
    <businessList generic="2.0" operator="jUDDI.org" xmlns="urn:uddi-org:api_v2">
      <businessInfos>
        <businessInfo businessKey="3826C4C0-F66B-11DB-84C0-B740D9D916A8">
          <name>health care</name>
          <description>provides health services</description>
          <serviceInfos>
            <serviceInfo businessKey="3826C4C0-F66B-11DB-84C0-B740D9D916A8" serviceKey
="38709F00-F66B-11DB-9F00-C0921F05456C">
              <name>vaccination</name>
            </serviceInfo>
          </serviceInfos>
        </businessInfo>
      </businessInfos>
    </businessList>
  </soapenv:Body>
</soapenv:Envelope>

```

Figure 73 Response Received by Node N7B

Section 5.1 illustrated how the Meta-Directory system can be initialized as well as how information is published and queried for each network model. As it can be seen that the administrative console is very user friendly, it is easy to set up a network of Meta-Directory nodes. The following section shows how real time scalability tests were performed on the distributed Meta-Directory system.

5.2 Performance Analysis

One of the main contributions of this thesis is a scalable Meta-Directory system that gives rise to good query response times. In order to illustrate the scalability of the system, performance analysis is performed on the prototype. This section presents the

performance analysis approach, the test plan for the performance analysis, and the results of the experiments that were performed.

5.2.1 Test Plan for the Performance Analysis Experiments

Before performing the performance analysis experiments, a test plan that outlines the parameters to be varied and the metrics to be collected had to be developed. Table 4 lists the performance metrics used to illustrate the scalability of the Meta-Directory system. The response time for a query is the difference in time between a Meta-Directory node receiving a query request from a service requestor and the Meta-Directory node returning the response to the requestor. The average bandwidth used illustrates the average network bandwidth used when the network is handling a query request. The average number of hops is the number of Meta-Directory nodes a query message traverses on average before a response is found, while the average number of messages exchanged shows the number of messages exchanged in the Meta-Directory system on average in order to find a response to a query. The routing table size illustrates the memory overhead in maintaining the Meta-Directory system.

Table 4 Performance Metrics

Experiment Metrics
RT: Average response time for a query
B: Average bandwidth used per query
NH: Average number of hops per query
NM: Average number of messages exchanged per query
SRT: Size of the routing table

A number of parameters are varied to test the system. Table 5 lists the parameters used in testing the scalability of the Meta-Directory system. The number of Meta-Directory nodes is varied in order to understand the impact of the network size on performance. The number of service registries illustrates the effect of the distributed nature of the service registries. The number of services per registry determines the memory load on the distributed Meta-Directory nodes as this determines the number of entries in the Meta-Directory network. The number of key-value entries in the Meta-Directory nodes indicates the amount of memory consumed by the Meta-directory nodes.

Table 5 Performance Parameters

Experimental Parameters
Number of Meta-Directory nodes
Number of service registries
The number of services per registry
The number of key-value entries in the Meta-Directory nodes
The type of query
The number of attributes per query
The query arrival rate
The type of distribution for the query arrival rate

There are three types of queries that can be sent to a Meta-Directory system. These are the *find_business*, *find_service* and *find_tModel* queries as shown in the interface definitions in Figure 21. The number of attributes used in a query has a direct effect on

performance metrics such as the query response time. This is because each attribute may be stored in a different Meta-directory node and a query would result in messages forwarded to the same number of nodes as the number of attributes. The query arrival rate and the inter-arrival time distribution, such as Exponential, determine the load on the Meta-Directory nodes. The following section presents the different approaches that were discussed to be applied in testing the scalability of the Meta-Directory system.

5.2.2 Performance Analysis Approaches

A number of approaches were proposed to analyze the performance of the Meta-Directory system proposed in this thesis. The first approach that was proposed is based on running the prototype on an isolated network so as to measure the performance. This provides control over all the network parameters. This approach was not feasible as we were not able to get an isolated network of at least 400 machines so as to test the scalability of the system. Since a large isolated network was not at our disposal, running a simulator based on the Meta-Directory system was proposed as a simulation would also allow control over the network parameters.

It was not possible to develop a simulator for the system due to the limitation on time. After further research we discovered that there is a program called Java in Simulation Time (JiST) which is a high-performance discrete event simulation engine that runs over a standard Java virtual machine [2].

JiST provides an environment where the simulation code does not have to be written in a simulation specific language such as the discrete event network simulator [24]. JiST also provides control over all the network parameters as it creates a simulation of the actual program. Unfortunately JiST was only compatible with Java version 1.4 and the

Meta-Directory system relies on an implementation of CHORD that was built using version 1.5 of Java.

During the course of our research, we ran across a network known as PlanetLab [14] which is a global research network that provides members access to a distributed network of computers. These computers are provided worldwide from academic institutions and research labs where people run tests on distributed storage, network mapping, peer-to-peer systems, distributed hash tables, and query processing [14]. We were able to get access to the PlanetLab network and run our prototype on the distributed network. The following section gives the overview of the tests run on PlanetLab, provides and discusses the results.

5.2.3 Performance Analysis using PlanetLab

Tests were performed on the PlanetLab network to analyze the performance of the Meta-Directory prototype on a distributed network. The average response time as the number of Meta-Directory nodes increases in the system is analyzed in this section. Performance analysis was also done to see the effect of the number of attributes used in a query on the average response time. Even though PlanetLab offers a distributed test bed, users do not have control over the network. Parameters used in the experiments cannot be isolated and as such there is a limitation to the experiments that can be performed on the PlanetLab network. An overview of the experiments, followed by a discussion on the network parameters and the experimental results are presented in the following subsections.

5.2.3.1 Experimental Overview

PlanetLab is a community that provides an open platform for developing, deploying, testing and accessing planetary scale services [14]. PlanetLab was initiated by Princeton University and its main aim is to provide a worldwide network of computers that can be used for research in planetary scale services. A number of research institutions and universities all around the world are members of PlanetLab. The following section provides an overview of the parameters used in setting up the PlanetLab experiments.

5.2.3.2 Experimental Parameters

In order to evaluate the performance of the Meta-Directory prototype using PlanetLab, a number of parameters had to be set in order to initialize the system. These parameters are set so as to configure the network in order to generate the results that are going to be analyzed in this section.

Table 6 lists the parameters that were kept constant throughout the experiments. Each node randomly generates queries from a pool of available services, with the time between queries modeled as an exponential distribution with the mean equal to the *lookup mean* value. The *number of registries* parameter indicates the total number of registries in the network. This value was set to 3. It took over 10 hours to upload the binaries needed to set up a UDDI registry on a PlanetLab node. The average number of businesses per Meta-Directory node was set to 10 so that the average number of key-value entries per Meta-Directory node would be 40. This is because each business information entry would require the creation of four key-value entries, one each for the business name, the discoveryURL, the category and the identifier as shown in the Meta-Directory interface presented in Figure 21. An average of 40 key-value entries per Meta-

Directory node provides a low probability of having all the requested key-value pairs in the same Meta-Directory node. This property forces the Meta-Directory node receiving the request to contact remote Meta-Directory nodes for the values.

Table 6 Fixed Experimental Parameters

Parameter	Value
Lookup mean	300 000 ms
Number of registries	3
Number of business per Meta-Directory node	10
Number of key-value entries per Meta-Directory node	40
Run time	6 HRS
Sleep time before generating the initial request	500 000 ms
The type of query	find_business

The *run time* of the experiment is the run length for each experiment that was done on the PlanetLab network. This value was set to 6 hours and each experiment was repeated 10 times. For each experiment we collected at least 5000 results. Before a Meta-Directory node started generating requests, a sleep time was set to 500,000 ms to provide enough time for all the Meta-Directory nodes to join the network and the key-value entries to be distributed to the responsible nodes. In other words, the sleep time ensures that queries are generated after the network is stable.

In this experiment, the query generated was of type *find_business*. This type of query was used because it is the only one whereby the number of attributes per query can be equal to 4 which is the maximum number of attributes.

In addition to the parameters shown in Table 6, the actual business information saved in the Meta-Directory nodes had to be initialized. Initially, we tried to retrieve UDDI data from publicly available service registries that could be reused in our experiments. After further research, it was discovered that currently there is no publicly available UDDI registry and the last one, which was provided by xmethods [28], was suspended back in 2005. Therefore, all the business information had to be created so that every business entry was unique. For the experiments described in this section, each business entry was unique and as such each query request would result in the Meta-Directory system to query only one service registry.

Experiments were performed to evaluate the effect of the size of the network and the number of attributes on the average response time. The following sub-sections discuss the results of these experiments.

5.2.3.3 Experimental Results

This section presents and discusses the scalability of the Meta-Directory prototype when the prototype was evaluated on the PlanetLab test bed. The prototype that was deployed was based on the CHORD (DHT) network model as this is a likely choice for large networks as shown in the theoretical analysis in Section 3.4.2. Two types of experiments were run: one in which the number of Meta-Directory nodes in the system was varied and the other in which the number of attributes per query was varied. The results of the experiments are presented next.

5.2.3.3.1 Effect of the Size of the Network

This section analyzes the average response time incurred per query message and this is the average time taken when a Meta-Directory node receives a query request to the time the response is received from the service registry by the Meta-Directory node. Therefore this is the total time taken to search for the values, which is the registry location, in the Meta-Directory system and the time taken to query the service registry and receive a response from the service registry.

The sequence diagram in Figure 74 shows the messages that contribute to the query response time measured in these experiments. When a Meta-directory node receives a query request, the node first hashes all the attributes in the request in order to get the *keys*. For each *key*, the Meta-Directory node then forwards a *get_Value* request to the Meta-Directory node responsible for that key. Once the Meta-Directory node gets all the *values* which indicate the registry locations for the *keys*, the node then compares the results and only queries the registry that has information on all the attributes by sending a *find_business* message to it.

In testing the effect of the network size on the average response time, the number of attributes used in all the queries generated was equal to 4. The number was set to 4 so as to get the maximum response time of the system since 4 is the maximum number of attributes possible in the Meta-Directory interface.

Initially PlanetLab promised the existence of over 800 nodes in the network. But after joining the network, we were only able to get about 300 stable nodes at a time. This limited our experiments to a maximum of 250 nodes as most of the nodes would become unavailable during the experiments.

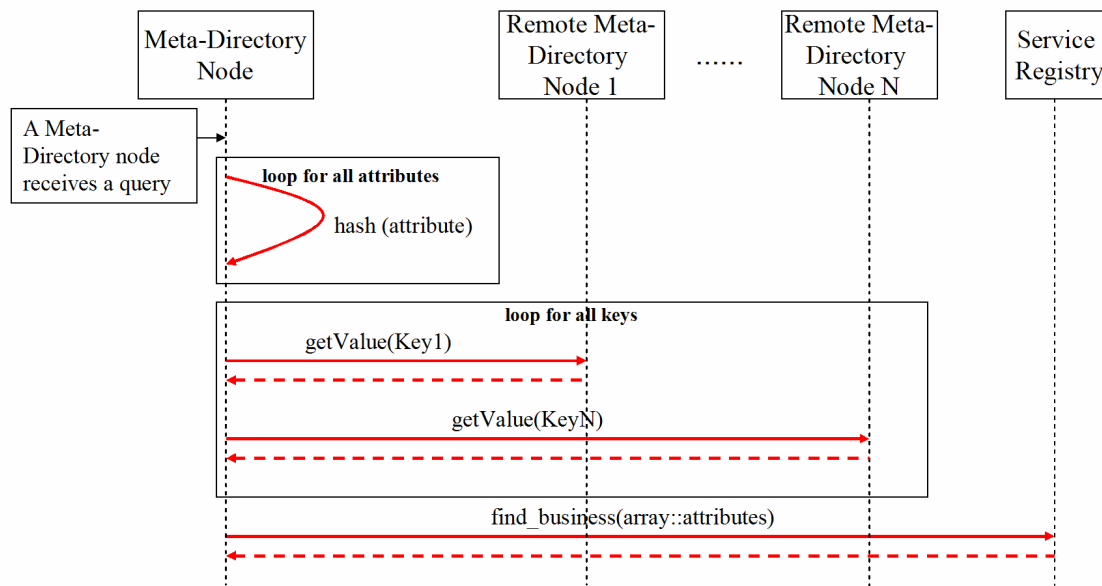


Figure 74 Sequence Diagram for PlanetLab Experiments

PlanetLab provided a very unstable network since the parameters cannot be controlled during the experiment and a number of other users are using the system at the same time. Hence the results were variable and had a number of outliers and huge standard deviations in the ranges of $5 * 10^6$. The outliers had to be removed which reduced the standard deviation of the experiments to around $3 * 10^3$. The graphs given in this thesis show the trend of the results with the reduced standard deviation.

In Figure 75 it can be seen that the query response time increases logarithmically with the size of the network which is consistent with the theoretical evaluation that was presented in Section 3.4.2.

The maximum response time is the average maximum response time recorded for all the query responses. During the experiments, it was also observed that the maximum response time also increases logarithmically with the network size as seen in Figure 75. It was observed that the maximum response time was around double the average response

time. This maximum response time for a network of 250 nodes is approximately 24 seconds (see Figure 75) in the PlanetLab network which is heavily loaded by applications of other users in the system. If the experiments were performed in an isolated network, the response times are expected to be lower.

The small slope of the query response time curve in Figure 75 indicates that the Meta-Directory system proposed in this thesis is scalable with respect to the number of nodes in the system.

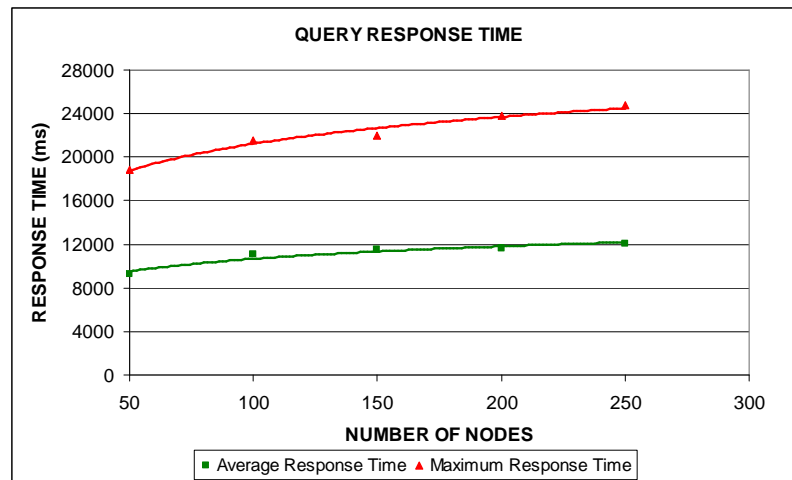


Figure 75 Effect of the Size of the Network on the Query Response Time

5.2.3.3.2 Effect of the Number of Attributes Used in a Query

This section investigates the effect the number of attributes used in a query on the average response time of the Meta-Directory system. From the discussion on the hash table structure presented in Section 3.3.2 and from the Meta-Directory interface functions described in Figure 21, it can be seen that the *find_business*, *find_service*, and *find_tModel* functions allow the client to use one or more attributes in the query. This flexibility led us to investigate the effects of the number of attributes used in a query on the average response time.

To observe these effects, experiments were performed on a network of 50 machines using the PlanetLab test bed. The messages that contribute to the response time are the same as those shown in the sequence diagram in Figure 74. The PlanetLab experiments showed that the average response time increases linearly with the number of attributes as shown in Figure 76, and the maximum response time also increases linearly with the number of attributes.

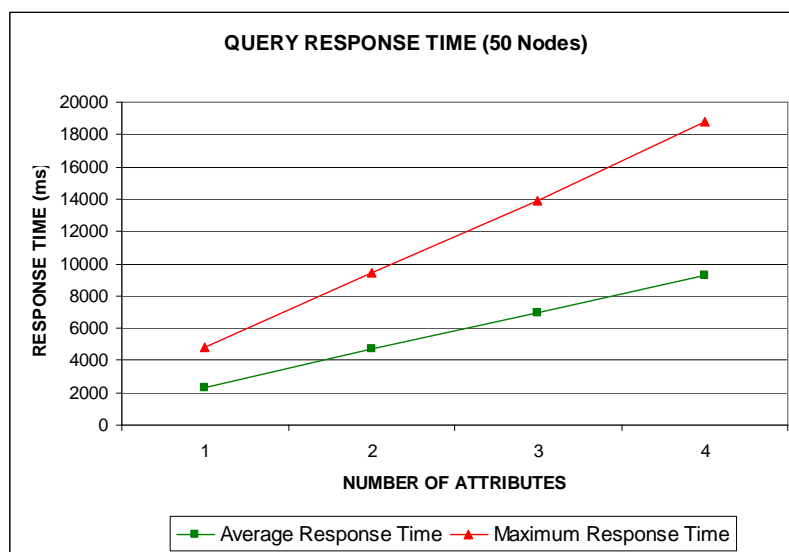


Figure 76 Effect of the Number of Attributes on the Query Response Time

Even though the PlanetLab environment provides a widely distributed network which is ideal for the testing of distributed systems, it presents an extremely challenging environment for our Meta-Directory system as it is heavily loaded by applications of other users and very volatile as it is not guaranteed that all the nodes will be available during the course of the experiment. It should be noted that the PlanetLab results are for a pessimistic scenario because there are other applications by other users which are running at the same time. Since we have no control over the environment of the PlanetLab system, we were not able to measure the total number of messages exchanged, nor the

bandwidth consumed by the query messages. Due to these limitations, simulations were instead performed on the underlying protocol, i.e. CHORD, using a simulator that was able to provide the desired results. This simulator can also run simulations for experiments for more than 250 nodes which was a limitation with the PlanetLab environment.

Further analysis of the Fully Connected (DHT) and CHORD (DHT) based on simulation is presented in the next section. Simulation was only performed on the DHT models as the simulator was specific to DHT models [8]. The simulation based analysis provides the results for a higher number of nodes as well as provides additional metrics, such as bandwidth used, which could not be captured in the prototype analysis presented in this section.

5.2.4 Performance Analysis using P2PSim

Simulations were performed to analyze the Meta-Directory's configurable routing framework and how it helps in terms of the network delay, path length and the bandwidth used per query. The memory overhead incurred in running the network is also analyzed in this section. Simulations were performed so as to ensure that the system is scalable to large network sizes which could not be achieved using instances of the prototype in a small number of machines. Simulations were performed on the CHORD (DHT) algorithm and the Fully Connected (DHT) algorithm.

Simulation was also done so as to evaluate a network that is close to a real distributed deployment where the nodes are not active all the time in the network. The main difference between this evaluation and the theoretical evaluation in Section 3.4.2 is that in the simulation model query messages are sent by multiple nodes

concurrently in the network following an exponential distribution that is discussed in Section 5.2.4.2. The theoretical analysis did not consider multiple clients active concurrently in the system. The number of bytes consumed by the routing tables is also a factor that could not be determined by the theoretical evaluation. The average number of bytes used per query could also not be determined in the theoretical evaluation but is captured in the simulation results.

An overview of the simulation, followed by a discussion on the simulation parameters and the simulation results are presented in the following sub-sections.

5.2.4.1 Simulation Overview

In order to evaluate the effectiveness of the configurable routing architecture, a peer-to-peer simulator called P2PSim [8] developed at the Parallel and Distributed Operating Systems Group at M.I.T. was used. The P2PSim simulator is a multi-threaded discrete event simulator that is written in C++ and runs on UNIX operating systems. The simulator already provides three different implementations of the CHORD (DHT) protocol, an instance called Chord that only allows successors, an instance called ChordFinger that allows successors and a successor list and finally an instance called ChordFingerPNS that allows successors, a successor list and a proximity neighbor selection list. The proximity neighbor selection list stores neighbors based on the estimated latency among the nodes. The most stable release of the P2PSim simulator is based on the ChordFingerPNS instance. Therefore, the results in this section were generated using the ChordFingerPNS instance as is for the CHORD (DHT) protocol and with additional parameters for the Fully Connected (DHT) protocol and the configurable

network. The parameters that were introduced to simulate the protocols are explained and discussed in the next section.

5.2.4.2 Simulation Parameters

Simulation parameters are the values used to configure the simulator in order to generate the results that are going to be analyzed in this section. Table 7 illustrates the parameters that were kept constant throughout the simulation for the CHORD (DHT) and Fully Connected (DHT) protocols. The *lifemean*, *deathmean* and *lookupmean* values are randomly generated with an exponential distribution with the indicated mean values. The *lifemean* parameter indicates the average time a node is alive in the simulation while the *deathmean* parameter is the average time a node is not available in the system. These parameters allow the simulator to model a network where the nodes are not always available. Since the *exittime* which is the total simulation time was set to 21600000 ms as recommended by [8], the *lifemean* was set to 3600000 ms and the *deathmean* to 100 ms so that the nodes are alive in the system for an average of 2159950 ms which is 99.998% of the simulation time.

Each node sends search requests for a randomly generated key, with the time between search requests modeled as an exponential distribution with the mean equal to the *lookupmean* value. As was suggested in [8], the *stattime* that indicates the time during the simulation when the collection of performance statistics should start was set to 10800000 ms. *pnstimer* indicates how often the finger table entries should be stabilized and *basictimer* indicates how often the nodes should stabilize the successor and predecessor (as per the CHORD specification in [22]). These routing table entries need to be stabilized as the nodes in the network are not active all the time hence the entries

could be pointing to a node that is not available. These timers were both set to 9000 ms to ensure that the routing tables were consistent with the network state. With the *recurs* parameter set to one, it indicates that the lookups should be performed recursively, that is a new request is sent through the network even if the response from the previous request was not received yet. This was done to ensure that the simulation closely models a real network.

The *maxlookuptime* determines when to stop sending a lookup retry if a response is not correct. A value of 0 indicates that no lookup retries should be performed in this simulation so as to minimize the effect of the network traffic for resent messages to the overall network traffic. With the *initstate* parameter set to 1 it indicates that when the simulation starts, all the nodes should start at a stable state such that the network is already set up and the finger tables are already populated. The average *round trip time* indicates the network delay between two nodes that are directly connected. This is the sum of the average time elapsed for a request message to reach the remote node and that for the response message to return to the requesting node.

For the simulations, a *base* parameter had to be introduced in the simulation. The *base* parameter indicates the number of entries each DHT node should have in its finger table such that the state of the finger table is equal to $(B-1) \cdot \log_B N$ where B is the *base* and N is the size of the network. For a CHORD (DHT) network the *base* was set to 2 (as per the CHORD [22] specification) for all the simulations while for the Fully Connected (DHT) network the *base* was set equal to the size of the network.

Simulations were performed using the P2PSim simulator for the size of the network ranging from 50 nodes to 1400 nodes. Simulations could only be performed to a

maximum of 1400 nodes due to the limitations of the simulator in simulating the Fully Connected (DHT) protocol. The simulations took one week to complete and were run on three linux machines with Intel Pentium 4 processors, with 2.6GHz of processing speed, 58.4GB of hard drive and 512MB of RAM.

Table 7 Simulation Parameters and Values

Parameter	Value
lifemean	3600000 ms
deathmean	100 ms
lookupmean	600000 ms
exittime	21600000 ms
stattime	10800000 ms
pnstimer	9000 ms
basictimer	9000 ms
recurs	1
maxlookuptime	0
initstate	1
round trip time	30 ms

The simulation results were collected for the network size varying from 50 to 1400 nodes and the network delay, path length, bandwidth used per query, and the memory overhead results are illustrated and discussed in the following section.

5.2.4.3 Simulation Results

This section evaluates and discusses the performance of a Fully Connected (DHT) protocol and the CHORD (DHT) protocol. The results for the memory overhead

incurred in the network, the response delay, the path length for a query message and the amount of bandwidth used per query message are shown and discussed.

5.2.4.3.1 Memory Overhead

This section analyses the memory consumed by the routing tables in the two network models. According to the results in Figure 77, it is clearly shown that the size of the routing table increases linearly with the size of the network in a Fully Connected (DHT) model while the size of the routing tables in the CHORD (DHT) network is stable to the network size.

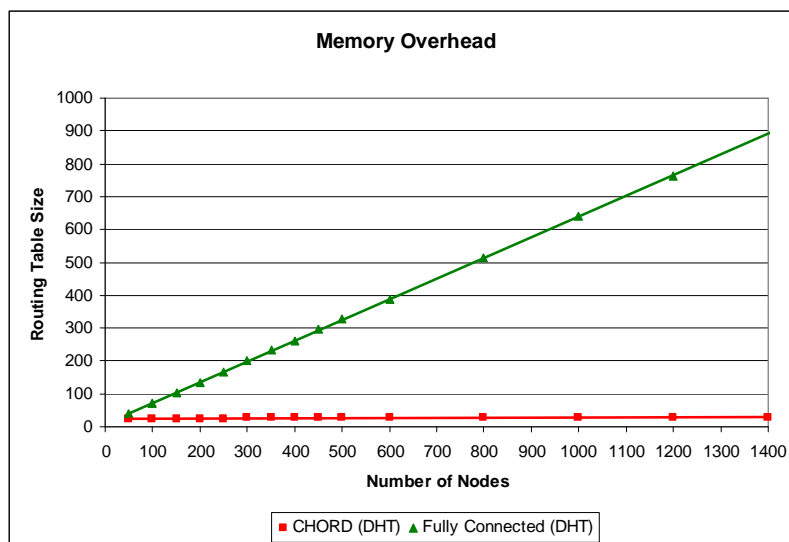


Figure 77 Effect of the Size of Network on the Memory Overhead

Therefore, the memory overhead incurred due to the routing tables of the Fully Connected (DHT) protocol is $O(N)$ (where N is the size of the network) while that of the CHORD (DHT) protocol is $O(1)$ as shown in Figure 77. The following section illustrates the results of the network delay analysis.

5.2.4.3.2 Network Delay

The network delay in this simulation is in terms of the time taken from a request being sent from a node to the time the response is received by the node. Recall that the simulation was performed on the nodes with an average round trip time of 30 ms (see Table 7). Figure 78 shows the average network delay in ms per lookup in the CHORD (DHT) protocol and the Fully Connected (DHT) protocol. The network delay for the Fully Connected (DHT) protocol remains constant at around 35 ms with increasing network size while that of the CHORD (DHT) protocol increases logarithmically with network size. These results are consistent with the theoretical results acquired from the evaluation in Section 3.4.2.

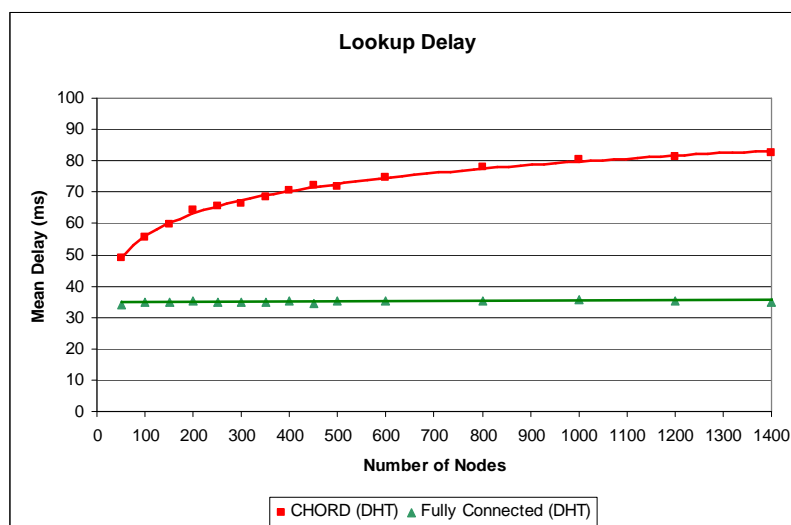


Figure 78 Effect of the Size of the Network on the Lookup Delay

With the results in Figure 78, it can be seen that the Fully Connected (DHT) protocol minimizes network delay per query as was indicated by the theoretical evaluation done in Section 3.4.2. It should also be noted that the protocol not only minimizes delay but it also ensures that the average delay remains constant with varying network size. The rate of increase of the delay in the CHORD (DHT) network when the size of the network was

between 50 and 500 nodes was high, but when the network size was at around 600 nodes, the rate of increase of the network delay in the CHORD (DHT) network was beginning to stabilize. The following sub-section shows the results of the path length analysis on the two algorithms which illustrates the average number of hops incurred per query.

5.2.4.3.3 Path Length

The path length in this thesis is defined as the average number of hops required to get the response back to the requesting node. The results from the P2PSim simulator, where the size of the network was varied from 50 – 1400 nodes, are illustrated by the graph in Figure 79. The results from the simulator for the average number of hops in the CHORD (DHT) network and the Fully Connected (DHT) network are consistent with the theoretical calculations that were done in Section 3.4.2.

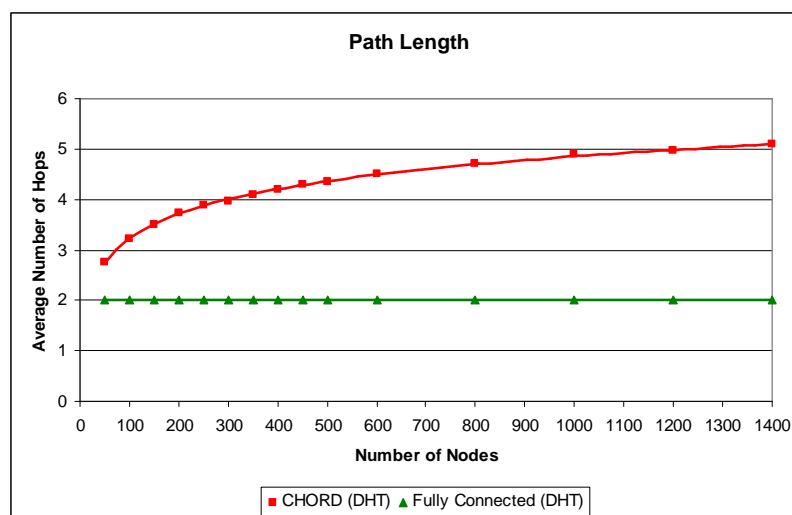


Figure 79 Effect of the Size of the Network on the Path Length

The average number of hops for the CHORD (DHT) algorithm increases logarithmically with the network size while the number of hops for the Fully Connected (DHT) algorithm remains stable at 2 hops per request. These results show that the Fully

Connected (DHT) algorithm also minimizes the average number of hops a message traverses per query messages.

Another parameter that was also compared was the network bandwidth used per query message. This illustrates the overall traffic in the network during a request and is discussed in detail in the following section.

5.2.4.3.4 Network Bandwidth

The network bandwidth depends on the average number of bytes exchanged in the network per query message. This includes the number of bytes used when forwarding the request through the network to the appropriate node and the number of bytes used to forward the response back to the requesting node.

The average number of bytes per query in the Fully Connected (DHT) model is approximately 100 bytes for different network sizes as shown in Figure 80. The average number of bytes used per query message in a CHORD (DHT) network on the other hand increases logarithmically with the network size.

In a CHORD (DHT) environment, when a node has a query, it first exchanges protocol specific messages so as to find the node that has the appropriate key. After the node is found, the query is then forwarded to that particular node. In a CHORD (DHT) network, the number of these protocol specific messages increases logarithmically with network size. Thus, the average number of bytes per query message in Figure 80 increases logarithmically for a CHORD (DHT) network. In a Fully Connected (DHT) network, no messages are exchanged to find the appropriate node and thus the average number of bytes per query message is approximately 100 for the different number of nodes.

In Figure 80, we can see that in a network of 1400 nodes, the difference in bytes exchanged per query message between the Fully Connected (DHT) and CHORD (DHT) is around 100 bytes. When the total number of queries are taken into consideration, the difference in the total number of bytes will also increase and contribute to an increase in the network bandwidth.

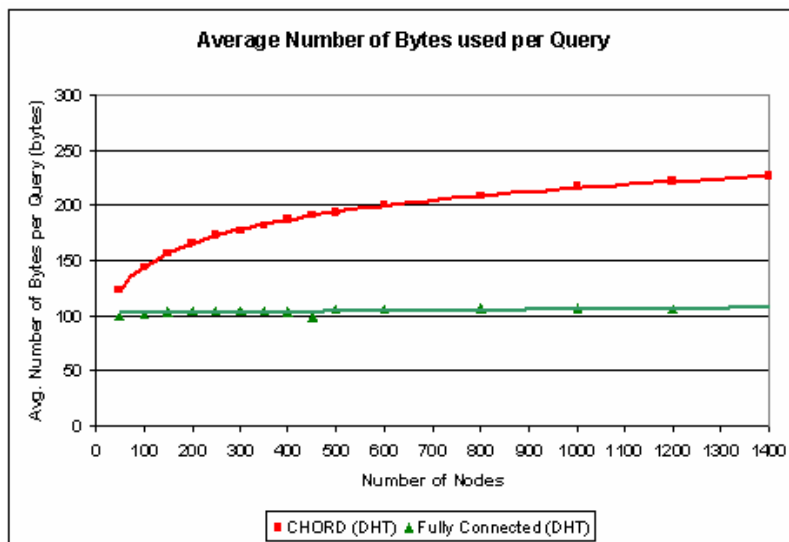


Figure 80 Effect of the Size of the Network on the Bandwidth Used per Query

Finally, the adaptable framework with the corresponding transformation algorithms is presented and analyzed in the following chapter through their complexity and invariants. These algorithms can be implemented so that runtime network transformation can be performed.

CHAPTER 6 ADAPTABLE ROUTING

FRAMEWORK

This chapter introduces the adaptable routing framework that would take the configurable framework one step further by allowing the system configuration between the Meta-Directory nodes to be changed after the system has been deployed.

The algorithms for the adaptable framework are useful because if the system administrator wants to minimize the network traffic incurred by the periodic messages exchanged so as to increase the available network bandwidth, the network can be changed from the Fully Connected (DHT) to the CHORD (DHT) network when the size of the network is such that the network traffic created by the total number of messages for the Fully Connected (DHT) model is too high for the network (see Figure 33). On the other hand, in order to minimize query delay, the network can be changed from the CHORD (DHT) to the Super Peer network when the number of nodes is such that the total number of hops per query message for the Super Peer network is less those for a CHORD (DHT) network (see Figure 32).

This framework will make it possible to evolve from a Fully Connected model to the Super Peer model and vice versa. Adaptability will also be supported among the Fully Connected (DHT), CHORD (DHT), and the Super Peer networks. This flexibility will allow the Meta-Directory nodes to provide good performance for a specific system configuration and state.

A framework for an adaptable configuration could be designed in the future. In this chapter, we will provide a preliminary study of the required algorithms to provide such a

framework. Once implemented, this framework would allow a network administrator to change the routing used among the Meta-Directory nodes based on the state of the system such as number of Meta-Directory nodes and bandwidth available in the network.

This adaptable framework would be built upon the four network models discussed in Section 3.4.1. Once again, the four network models are:

- The CHORD (DHT) model
- The Fully Connected model (FC)
- The Fully Connected (DHT) model (FCDHT)
- The Super Peer model which can either be a:
 - Fully Connected Super Peer model (SPFC) where the clusters communicate using the FC model
 - The Fully Connected (DHT) Super Peer model (SPFCDHT) where the clusters communicate using the FCDHT model
 - The CHORD (DHT) Super Peer model (SPCHORD) where the clusters communicate using the CHORD (DHT) model

The performance metrics were calculated with varying network sizes, and from these results, it can be seen that when the network is small, one dedicated Meta-Directory node can be selected to act as a centralized repository where all the registry information will be stored. When the network size increases to a level that the one Meta-Directory node cannot handle the load, a network of Meta-Directory nodes can be created where related businesses would share the storage space which will be managed by super peers. These super peer nodes will then be peers of each other and would communicate directly. As the size of the network increases further, these super peers can create a P2P communication

overlay protocol such as CHORD (DHT) for communication. This will further reduce the communication overhead among the super peers for very large networks. During runtime, the framework would allow the system administrator to change the network model used among the four discussed models. The arrows in Figure 81 illustrate the network changes that would be supported by the algorithms introduced in the following section.

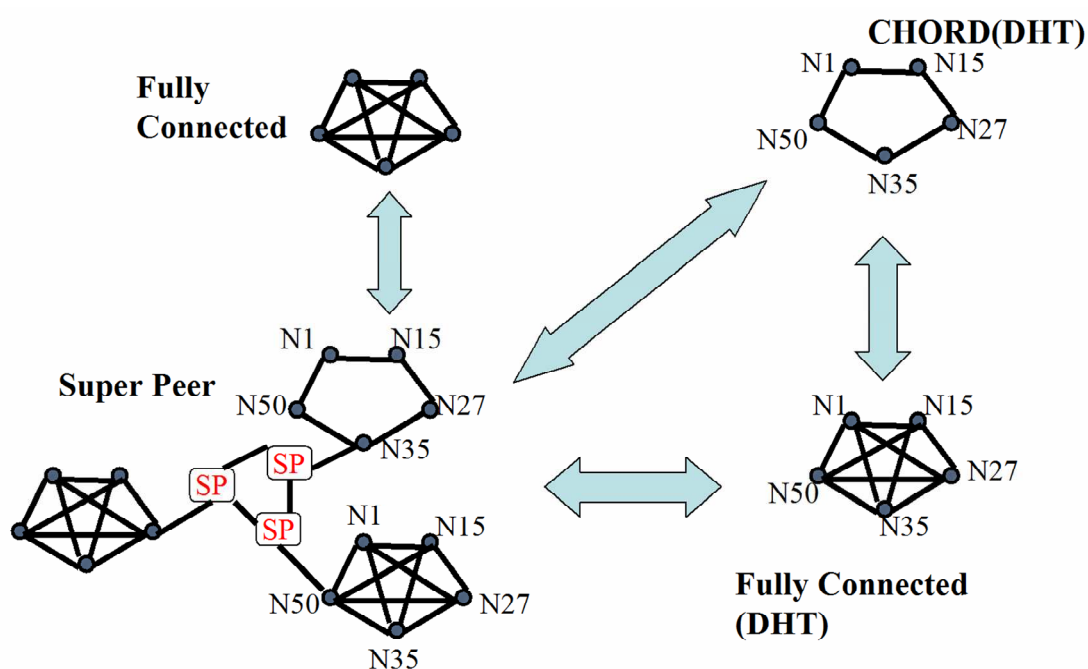


Figure 81 Runtime Re-Configuration

6.1 Adaptable Routing Algorithms

The following pseudo code illustrates the algorithms that should be used when the network needs to be modified. The algorithms in bold are sub-routines and they are discussed in detail further in this section. It should be noted again that these algorithms have not been implemented but they provide a basis for the adaptable framework which can be supported by the Meta-Directory system in the future.

The following assumptions are used in realizing the adaptable routing algorithms:

- i. We assume that a system administrator monitors the network in terms of query delay, memory, and network bandwidth usage. Then she/he decides when the system configuration should be changed.
- ii. There is a monitoring system that can keep track of the current state of the system in terms of the underlying communication graph, the number of nodes, the type of each node (super peer or client node), and the location of each node.
- iii. The nodes in the network understand the configuration commands and are able to react to such commands.
- iv. It is acceptable for the system to be unavailable temporarily during a network configuration change.
- v. The nodes can maintain their states prior to being placed under the maintenance mode and they can resume the queries that were not complete once the system is back online.
- vi. The nodes can stop processing publish and query requests when they are in maintenance mode but they can process configuration change commands.
- vii. At any point in time, each node is reachable from every other node in the network

The actions in Figure 82 have to be performed by the system administrator. Before the underlying configuration is modified, the system has to be placed in maintenance mode, then the network can be modified, and then the system is removed from maintenance. The rest of the sections in this chapter illustrate the algorithms that should be followed when the configuration for the Meta-Directory nodes needs to be modified during runtime.

```

setMaintenance (true, nodes);

changeNetwork (fromNetwork, toNetwork);

setMaintenance (false, nodes);

```

Figure 82 Actions Performed by a System Administrator when the Network is Modified

6.1.1 Changing the Maintenance State of the System

The protocol described in Figure 83 is followed when the system is set or removed from the maintenance mode. This protocol can only be applied when the assumptions listed in Section 6.1 hold.

```

void setMaintenance (boolean input, List nodes) {
    if (input) {
        for all (nodes) {
            Set state of node to "maintenance";
            Stop accepting any requests from service providers and
            requestors;
            Save state of requests that were not complete;
        }
    } else {
        for all (nodes) {
            Set state of node to "active";
            Complete requests that were queued;
            Start accepting requests;
        }
    }
}

```

Figure 83 Toggling the State of the System

6.1.2 Transforming the Network from an SPFC to an FC Instance

Figure 84 illustrates the protocol followed when the network is changed from an SPFC to an FC network. The algorithm is a general algorithm that can be applied to any number of clusters.

```

boolean changeNetwork (SPFC , FC) {
    List clusters = getClusters(SPFC);

    List superpeers = SPFC.getSuperPeers( );
        // returns a reference to all the Super Peer nodes

    for (i = 1; i < clusters.size( ); i++ ) {

        Cluster currentCluster = clusters.getEntry(i);

        List nodes = currentCluster.getNodes( );
            // returns a reference to all the nodes in the cluster

        for (j = 0; j < nodes.size( ); j++) {

            currentRT = nodes.getEntry(j).getRoutingTable( );
                // returns a reference to the node's routing table

            for (k = (i - 1); k >= 0; k--) {
                currentRT.addRTEEntries
                    (clusters.getEntry(k).getNodes());
                    // add all the nodes in the clusters with a
                    // lower index than the current node's index
                    // in the array to the current cluster
            }
        }
    }

    for (i = 0; i < superpeers.size( ); i++) {
        SuperPeer currentSP = superpeers.getEntry(i);
        currentSP.destroySP( ); // Destroy the Super Peer node
    }
    return true;
}

```

Figure 84 Changing an SPFC Network to an FC Network

The following example illustrates the steps followed and the resulting network when the algorithm in Figure 84 is applied to an SPFC instance that has two clusters. Each step is illustrated in Figure 85 when it is applied to the SPFC instance.

1. Select one cluster to be the Primary Cluster (PC) and the other cluster will be the Secondary Cluster (SC)
2. For each node in the SC
 - a. Add all the nodes in the PC as neighbors

3. Destroy the Super Peer nodes

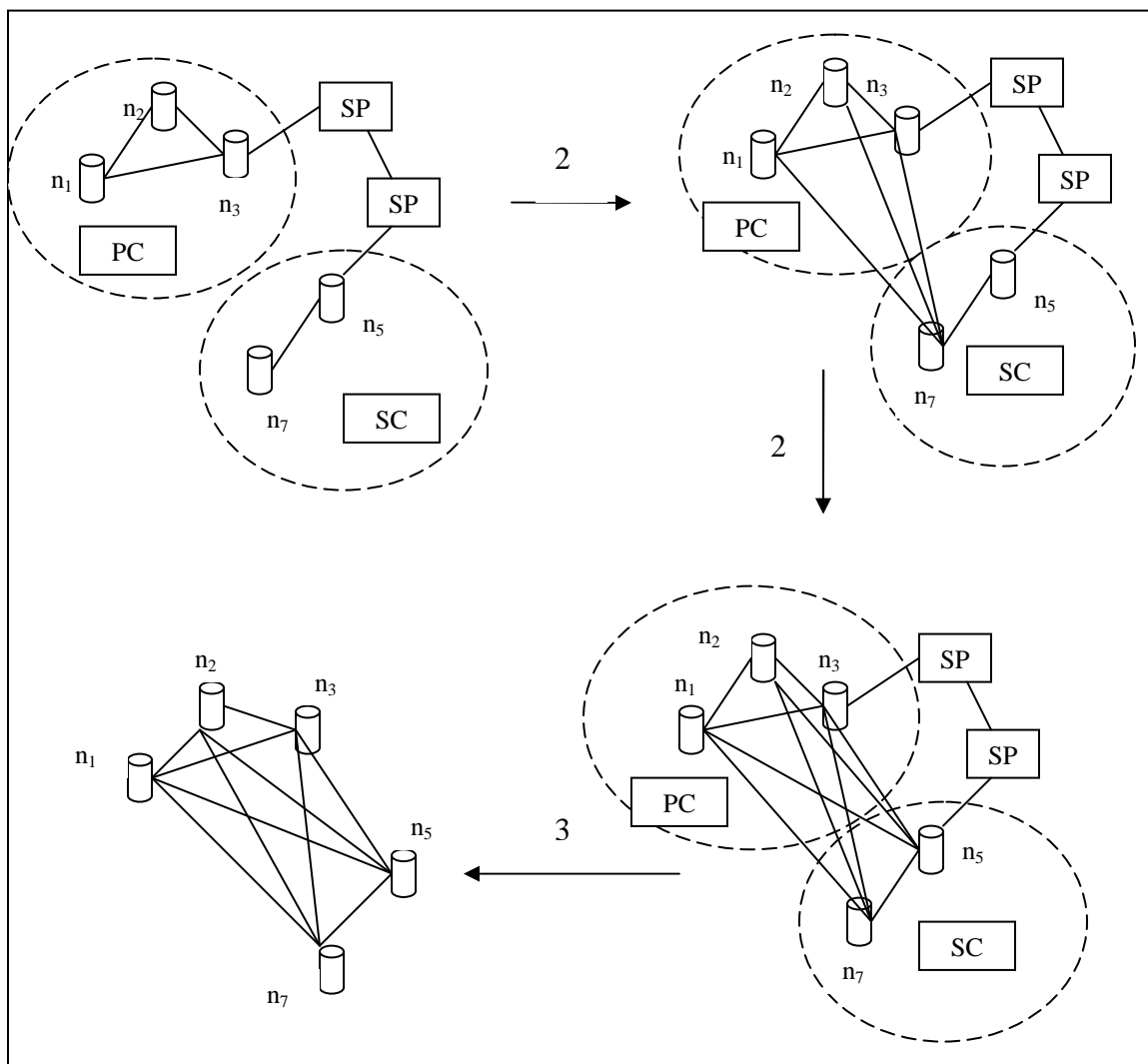


Figure 85 Example of Transformation from SPFC to FC

6.1.3 Transforming the Network from an SPCHORD to a CHORD (DHT)

Instance

Figure 86 illustrates the protocol followed when the network is changed from an SPCHORD to a CHORD (DHT) network instance. The algorithm is a general algorithm that can be applied to any number of clusters.

```

boolean changeNetwork (SPCHORD, CHORD) {
    List clusters = getClusters(SPCHORD);
    List superpeers = SPCHORD.getSuperPeers ( );

    Cluster primaryCluster = clusters.getEntry(0);

    SuperPeer primarySP = primaryCluster.getSuperPeer ( );
        // returns a reference to the Super Peer node

    for (i = 1; i < clusters.size( ); i++) {

        currentCluster = clusters.getEntry(i);
        entryNode = currentCluster.getEntryNode( );
        currentSP = currentCluster.getSuperPeer ( );
        keyValueEntries = entryNode.retrieveAllEntries( );
            // retrieves all the key-value registry entries in the
            // cluster
        currentSP.publishToCluster (primarySP, keyValueEntries);

        entryNode.deleteEntries(keyValueEntries);

        List nodes = currentCluster.getNodes( );

        for (j = 0; j < nodes.size( ); j++) {
            currentNode = nodes.getEntry(j);
            currentNode.leaveCluster(currentCluster);
            currentNode.joinCluster(primaryCluster);
        }
    }

    for (i = 0; i < superpeers.size( ); i++) {
        currentSP = superpeers.getEntry(i);
        currentSP.destroySP( );
    }
    return true;
}

```

Figure 86 Changing an SPCHORD Network to a CHORD (DHT) Network

The following example illustrates the steps followed and the resulting network when the algorithm in Figure 86 is applied to an SPCHORD instance that has two clusters. Each step that causes the change in the network graph is illustrated in Figure 87 when it is applied to the SPCHORD instance. Only the network graph changes are illustrated and changes in the key-value entries are not shown in Figure 87.

1. Select the primary cluster (PC) that all the nodes will be joining to create one cluster in the end, the other cluster will be the secondary cluster (SC)

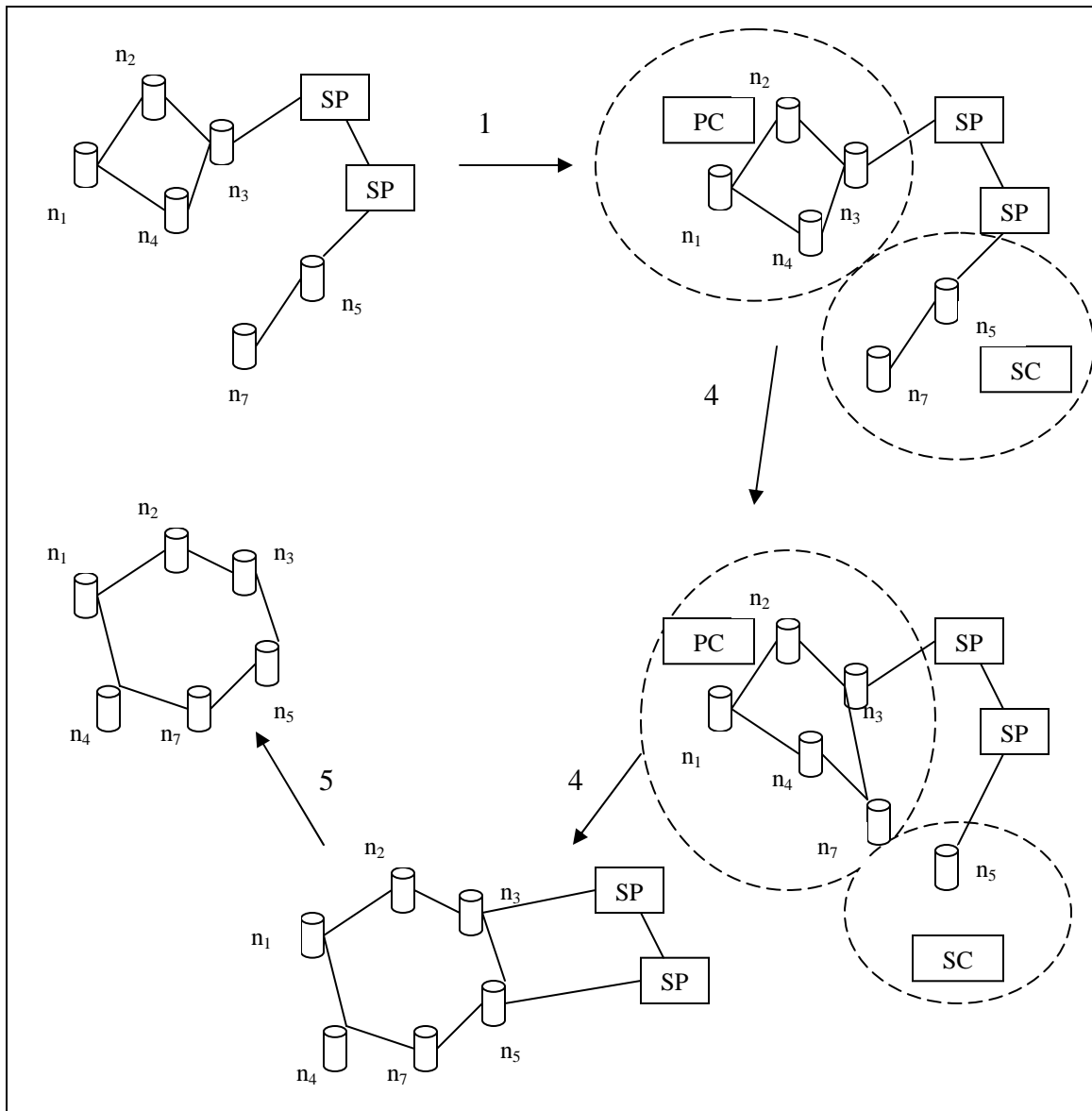


Figure 87 Example of Transformation from SPCHORD to CHORD (DHT)

2. The SC's entry node retrieves all the key-value entries in the SC and sends a publish request to the PC through its SP
3. The Meta-Directory node in the PC that is connected to the SP saves the key-value pairs it receives into the PC
4. For each node in the SC
 - a. leave the SC and join the PC as a new node (i.e. the nodes have no entries)

5. Destroy the Super Peer nodes

6.1.4 Transforming the Network from an FC to an SPFC Instance

Figure 88 illustrates the protocol followed when the network is changed from an FC network to an SPFC network where the network is split into m clusters and the number of nodes per cluster is balanced within the clusters.

The following example shown in Figure 89 illustrates the steps followed and the resulting network when the algorithm in Figure 88 is applied to an FC instance to create an SPFC instance with two clusters.

1. Create two Super Peer nodes
2. Select which nodes belong to the PC and which nodes belong to the SC
3. For the SP responsible for the PC
 - a. Pass the IP address of the entry node of the PC
4. For the SP responsible for the SC
 - a. Pass the IP address of the entry node of the SC
 - b. Pass the IP address of the SP responsible for the PC
5. For the nodes that should be in the PC
 - a. Drop all connections to the nodes that are supposed to be in the SC

```

boolean changeNetwork (FC, SPFC, m) {

    List superpeers, entryNodes, routingTables;

    List nodes = FC.getNodes( );

    for (i = 0; i < m; i++) {
        superpeers.addEntry(createSP( )); // create m Super Peers
    }

    for (i = 0; i < m; i++) {
        entryNodes.addEntry(nodes.getEntry(round(nodes.size( )/m) * i));
    }

    for (i = 0; i < m; i++) {
routingTables.addEntry(superpeers.getEntry(i).getRoutingTable( ));
        // get a reference to the Super Peer routing tables
    }

    for (i = 0; i < m; i++) {
        List nodesToSP;
        nodesToSP.addEntry(entryNodes.getEntry(i));
        // add entry node to list

        if (i != 0) {
            for (j = i - 1; j >= 0; j--) {
                nodesToSP.addEntry(superpeers.getEntry(j));
                // add the Super Peers whose indices in the list are
                //less than the current Super Peer
            }
        }
        routingTables.getEntry(i).addRTEntries(nodesToSP);
        // populate SP's RT
    }

    for (i = 0; i < m - 1; i++) {
        for (j = round(nodes.size( )/m) * i ;
            j < round(nodes.size( )/m) * (i+1); j++) {

            currentNode = nodes.getEntry(j);
            currentRT = currentNode.getRoutingTable( );

            for (k = round(nodes.size( )/m) * (i+1);
                k < nodes.size( ); k++)
                currentRT.removeRTEntry(nodes.getEntry(k));
            }
        }
    }
    return true;
}

```

Figure 88 Changing an FC Instance to an SPFC Instance

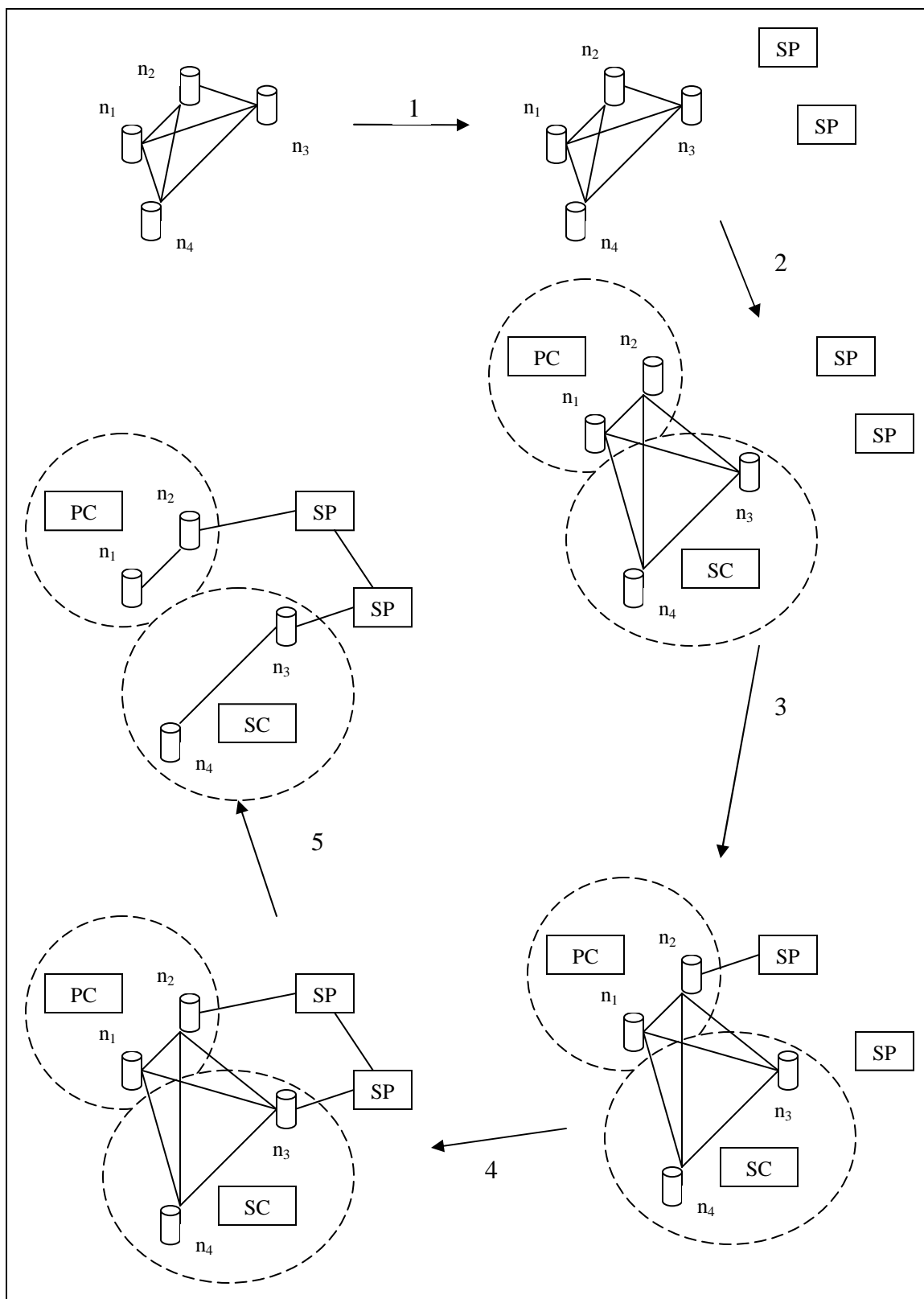


Figure 89 Example of Transformation from FC to SPFC

6.1.5 Transforming the Network from a CHORD (DHT) to an SPCHORD

Instance

Figure 90 illustrates the algorithm followed when the network is changed from a CHORD (DHT) network to an SPCHORD network where the network is split into m clusters and the number of nodes per cluster is balanced within the clusters.

The following example illustrates the steps followed and the resulting network when the algorithm in Figure 90 is applied to a CHORD (DHT) instance to create an SPCHORD instance with two clusters. Figure 91 illustrates the changes in the network graph when the algorithm is applied but does not show the key-value pair messages forwarded within the network.

1. Create two Super Peer nodes
2. Select which nodes belong to which cluster, one primary cluster (PC) and a secondary cluster (SC)
3. For the SP responsible for the PC
 - a. Pass the IP address of the entry node of the PC
4. For the SP responsible for the SC
 - a. Pass the IP address of the entry node of the SC
 - b. Pass the IP address of the SP responsible for the PC
5. The node that is supposed to be in the SC and is connected to the SP responsible for the SC (entry node), leaves the PC and creates a SC
6. For each node belonging to the SC
 - a. leaves the PC and joins the SC
7. To even out the key-value entries in the nodes:

```

boolean changeNetwork (CHORD, SPCHORD, m) {
    List nodes = CHORD.getNodes( );
    for (i = 0; i < m; i++) {
        superpeers.addEntry(createSP( )); // create m Super Peers
    }
    for (i = 0; i < m; i++) { // get reference to entry nodes
        entryNodes.addEntry(nodes.getEntry(round(nodes.size( )/m) * i));
    }
    for (i = 0; i < m; i++) {
        routingTables.addEntry(superpeers.getEntry(i).getRoutingTable( ));
    }
    for (i = 0; i < m; i++) {
        List nodesToSP;
        nodesToSP.addEntry(entryNodes.getEntry(i));
        // add entry node to list
        if (i != 0) {
            for (j = i - 1; j >= 0; j--) {
                nodesToSP.addEntry(superpeers.getEntry(j));
                // add the Super Peers whose indices in the list are
                //less than the current Super Peer
            }
        }
        routingTables.getEntry(i).addRTEntries(nodesToSP);
        // populate SP's RT
    }
    for (i = 1; i < m; i++) {
        currentEntryNode = entryNodes.getEntry(i);
        currentEntryNode.leaveCluster(CHORD);
        currentCluster = currentEntryNode.createCluster( );
        for (j = 1+[round(nodes.size( )/m) * i] ;
            j < round(nodes.size( )/m) * (i+1); j++) {
            currentNode = nodes.getEntry(j);
            currentNode.leaveCluster(CHORD);
            currentNode.joinCluster(currentCluster);
        }
    }
    ListkeyValueEntries = entryNodes.getEntry(0).retrieveAllEntries( );

    for (i = 0; i < m - 1; i++) {
        // balance the key-value pairs per cluster
        for (j = round(keyValueEntries.size( )/m) * i;
            j < round(keyValueEntries.size( )/m) * (i+1); j++) {
            entriesToForward.addEntry(keyValueEntries.getEntry(j));
        }
        superpeers.getEntry(0).publishToCluster
            (superpeers.getEntry(i+1), entriesToForward);
        entryNodes.getEntry(0).deleteEntries(entriesToForward);
    }
    return true;
}

```

Figure 90 Changing a CHORD (DHT) Instance to an SPCHORD Instance

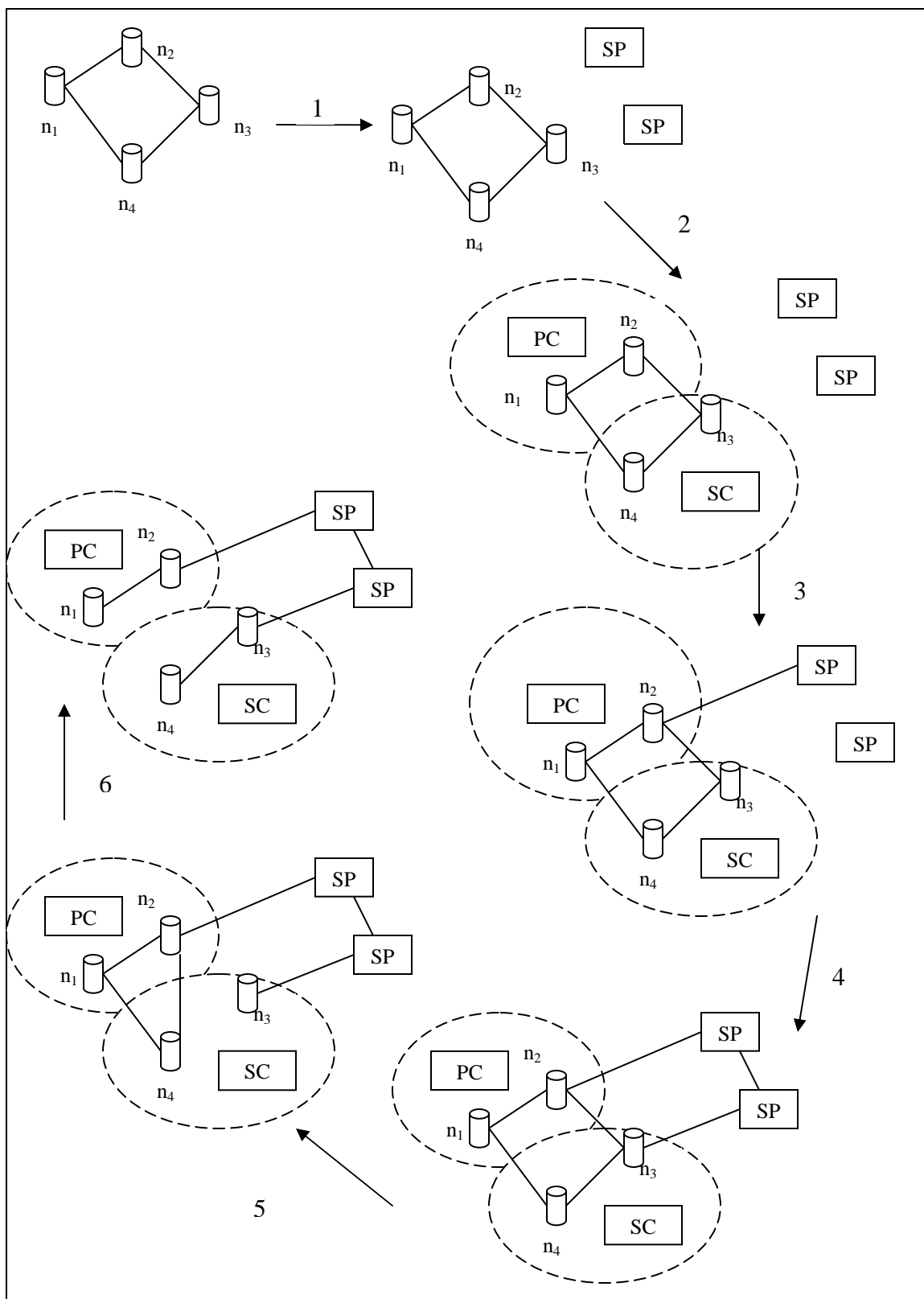


Figure 91 Example of Transformation from CHORD (DHT) to SPCHORD

- a. The Meta-Directory node in the PC that is connected to the SP responsible for the PC, retrieves half of the key-value pairs in the PC and sends a publish request for the retrieved entries to the SC through its SP
- b. The SP connected to the PC passes the publish request to the SP connected to the SC
- c. The SP connected to the SC passes the publish request to the Meta-Directory node it is connected to
- d. The Meta-Directory node receiving the request saves the key-value pairs it receives from the publish request in the SC
- e. The entry node to the PC deletes the forwarded entries from the PC

6.1.6 Transforming the Network from a CHORD (DHT) to an FCDHT

Figure 92 illustrates the algorithm followed when the network is changed from a CHORD (DHT) network to a FCDHT network. The routing table size is modified so that it accommodates all the nodes in the network to form a complete network graph.

```

boolean changeNetwork (CHORD, FCDHT) {
    List nodes = CHORD.getNodes( );
    for (i = 0; i < nodes.size( ); i++) {
        nodes.getEntry(i).setRoutingTableSize(nodes.size( ) - 1);
        // When this is applied in CHORD, the network changes its
        // finger table entries to reflect the size
    }
    return true;
}

```

Figure 92 Changing a CHORD (DHT) Instance to an FCDHT Instance

6.1.7 Transforming the Network from an FCDHT to a CHORD (DHT)

Figure 93 illustrates the algorithm followed when the network is changed from a FCDHT network to a CHORD (DHT) network. The routing table size is set to conform to the CHORD protocol [22].

```

boolean changeNetwork (FCDHT, CHORD) {
    List nodes = FCDHT.getNodes( );
    for (i = 0; i < nodes.size( ); i++) {
        nodes.getEntry(i).setRoutingTableSize(log2nodes.size( ));
    }
    return true;
}

```

Figure 93 Changing an FCDHT Instance to a CHORD (DHT) Instance

6.1.8 Algorithms Specific to the SP Nodes

Figure 94 shows the algorithm followed by the Super Peer network to get a reference to all the clusters in the network and Figure 95 illustrates the protocol used to forward a set of key-value pair entries from one Super Peer cluster to another.

```

Cluster getClusters (Network SPNetwork) {
    List superpeers = SPNetwork.getSuperPeers( );
    // gets a reference to all the SP nodes

    List clusters;
    for (i = 0; i < superpeers.size( ); i++) {
        clusters.addEntry(superpeers.getEntry(i).getCluster( ));
        // returns a reference to a cluster
    }
    return clusters;
}

```

Figure 94 Get the Reference of all the Clusters in a Super Peer Network


```

void publishToCluster (SuperPeer SP, List keyValueEntries) {
    SP.publishEntries(keyValueEntries);
    // the message is passed through the network from the receiving
    // Super Peer to the Super Peer SP.
}

void publishEntries (List keyValueEntries) {
    Cluster localCluster = getCluster( );
    Node entryNode = localCluster.getEntryNode( );

    for ( i = 0; i < keyValueEntries.size( ); i++) {
        entryNode.publishEntry (keyValueEntries.getEntry(i));
        //save the key-value pair in the cluster
    }
}

```

Figure 95 Publish Key-Value Pairs to a Cluster through the Super Peer Node

6.1.9 Algorithm Specific to the Entry Nodes

The algorithm in Figure 96 is applied to an entry node so as to delete all the key-value pair entries from the cluster.

```

void deleteEntries (List keyValueEntries) {

    for ( i = 0; i < keyValueEntries.size( ); i++) {

        removeEntry(keyValueEntries.getEntry(i));
        // delete the key-value pair from the cluster

    }

}

```

Figure 96 Delete Key-Value Pairs from a Node

6.1.10 Algorithm used by the RT

The algorithm in Figure 97 is used by the Fully Connected network when a node is added as a neighbor in the routing table. The node where this algorithm is invoked sends

a connection request message to the remote node and once the connection is accepted, the two nodes then have a direct point to point connection.

The following section analyzes the complexity of the algorithms as well as identifies the invariants for each transformation.

```
void addRTEntries (List locations) {
    for ( i = 0; i < locations.size( ); i++) {
        addIP(locations.getEntry(i));
    }
}
```

Figure 97 Create a Point-To-Point Connection with Remote Nodes in List

6.2 Analysis of Transformation Algorithms

Analysis was performed for the transformation algorithms in terms of the time complexity as well as the invariants for each transformation. The complexity is expressed in big O notation. Table 8 shows the complexity of each algorithm, and discussions on how the complexities were deduced can be found in Appendix A. Table 9 shows the invariants for each transformation where:

- N_T is the total number of nodes
- N_{SP} is the total number of Super Peer nodes
- N_C is the total number of clusters
- N_{ci} is the total number of nodes in a cluster
- N_{KV} is the total number of key-value registry entries
- N_{RT} is the total number of routing table entries for each node which is equal to the degree of the node

- N_E is the graph's size which is the total number of edges or connections in the network

The following chapter summarizes the thesis by first analyzing the models supported by the Meta-Directory system proposed in this thesis. A discussion is provided illustrating the properties as well as the advantages of the Meta-Directory system. Limitations and future directions of the research are also discussed in Chapter 7.

Table 8 Time complexity of Algorithms

Algorithm	Complexity
setMaintenance (boolean input, List nodes)	$O(N_T)$
addRTEntries (List locations)	$O(N_{RT})$
deleteEntries (List keyValueEntries)	$O(N_{KV})$
publishToCluster (SuperPeer SP, List keyValueEntries)	$O(N_{KV})$
getClusters (Network SPNetwork)	$O(N_{SP})$
changeNetwork (SPFC , FC)	$O(N_C^2 * N_T^2)$
changeNetwork (SPCHORD, CHORD)	$O(N_C * N_T * \text{Log } N_T) + O(N_C * N_{KV})$
changeNetwork (FC , SPFC)	$O(N_C * N_T^2)$
changeNetwork (CHORD , SPCHORD)	$O(N_C^2) + O(N_C * N_T * \text{Log } N_T) + O(N_C * N_{KV})$
changeNetwork (CHORD , FCDHT)	$O(N_T)$
changeNetwork (FCDHT , CHORD)	$O(N_T)$

Table 9 List of Invariants for every Network Transformation

Network Transformation Instance	Invariants	
	Pre-Conditions	Post-Conditions
From an SPFC to an FC instance	i. $N_T = \sum_{i=1}^{N_c} N_{ci} + N_C$ ii. $N_{RT} = N_{ci} - 1$ iii. $N_E = \sum_{i=1}^{i=N_c} \frac{N_{ci}(N_{ci} - 1)}{2} + \frac{N_C(N_C - 1)}{2} + N_C$	i. $N_T = \sum_{i=1}^{N_c} N_{ci}$ ii. $N_{RT} = N_T - 1$ iii. $N_E = \frac{N_T(N_T - 1)}{2}$
	iv. The total number of key-value registry entries per node is constant v. The total number of key-value registry entries in the network is constant	
From an SPCHORD to a CHORD instance	i. $N_T = \sum_{i=1}^{N_c} N_{ci} + N_C$ ii. $N_{RT} = \log_2 N_{ci}$ iii. $N_E = \sum_{i=1}^{N_c} N_{ci}(\log_2 N_{ci} - 0.5) + \frac{N_C(N_C - 1)}{2} + N_C$	i. $N_T = \sum_{i=1}^{N_c} N_{ci}$ ii. $N_{RT} = \log_2 N_T$ iii. $N_E = N_T(\log_2 N_T - 0.5)$
	iv. The total number of key-value registry entries in the network is constant	
From an FC to an SPFC instance	i. $N_T = \sum_{i=1}^{N_c} N_{ci} \geq N_C$ ii. $N_{RT} = N_T - 1$ iii. $N_E = \frac{N_T(N_T - 1)}{2}$	i. $N_T = \sum_{i=1}^{N_c} N_{ci} + N_C$ ii. $N_{RT} = N_{ci} - 1$ iii. $N_E = \sum_{i=1}^{i=N_c} \frac{N_{ci}(N_{ci} - 1)}{2} + \frac{N_C(N_C - 1)}{2} + N_C$
	iv. The total number of key-value registry entries per node is constant v. The total number of key-value registry entries in the network is constant	

From a CHORD to an SPCHORD instance	i. $N_T = \sum_{i=1}^{N_c} N_{ci} \geq N_C$ ii. $N_{RT} = \log_2 N_T$ iii. $N_E = N_T(\log_2 N_T - 0.5)$	i. $N_T = \sum_{i=1}^{N_c} N_{ci} + N_C$ ii. $N_{RT} = \log_2 N_{ci}$ iii. $N_E = \sum_{i=1}^{N_c} N_{ci}(\log_2 N_{ci} - 0.5) + \frac{N_C(N_C - 1)}{2} + N_C$
	iv. The total number of key-value registry entries in the network is constant	
From a CHORD to an FCDHT instance	i. $N_T = \sum_{i=1}^{N_c} N_{ci}$ ii. $N_{RT} = \log_2 N_T$ iii. $N_E = N_T(\log_2 N_T - 0.5)$	i. $N_T = \sum_{i=1}^{N_c} N_{ci}$ ii. $N_{RT} = N_T - 1$ iii. $N_E = \frac{N_T(N_T - 1)}{2}$
	iv. The total number of key-value registry entries per node is constant v. The total number of key-value registry entries in the network is constant	
From an FCDHT to a CHORD instance	i. $N_T = \sum_{i=1}^{N_c} N_{ci}$ ii. $N_{RT} = N_T - 1$ iii. $N_E = \frac{N_T(N_T - 1)}{2}$	i. $N_T = \sum_{i=1}^{N_c} N_{ci}$ ii. $N_{RT} = \log_2 N_T$ iii. $N_E = N_T(\log_2 N_T - 0.5)$
	iv. The total number of key-value registry entries per node is constant v. The total number of key-value registry entries in the network is constant	

CHAPTER 7 CONCLUSIONS

This thesis introduced a system that supports large scale service discovery in distributed Web Service registries. The Meta-Directory system allows the service requesters to send one request to a Meta-Directory node which will forward the request to the relevant registries without the service requester having any prior knowledge to the location of these registries. The distributed nature of the Meta-Directory system adopted in this thesis is transparent to the clients who are provided with a single interface.

This design decision also allows us to introduce a configurable framework for the underlying network configuration for the Meta-Directory nodes. Concluding remarks regarding the Meta-Directory system, discussions on the limitations of the current prototype and future directions of this research are covered in this chapter.

7.1 Summary

The Meta-Directory system that was implemented provides a configurable system in which the network model used for interconnecting the Meta-Directory nodes can be chosen from Fully Connected, Fully Connected (DHT), CHORD (DHT) and the Super Peer models. A summary of these supported networks is provided in this section.

7.1.1 Fully Connected Network

The Fully Connected network provides a configuration whereby there is no specific structure as to where the hashed values are stored in the network and each node is connected to every other node in the network. When a message is published in the

network, the Meta-Directory node receiving the publish request hashes the attributes and stores all the hashed keys in the local hash table.

The advantage to this approach is that there is no maintenance overhead on the Meta-Directory nodes regarding the hashed keys. The nodes only need to have the location of all the other nodes in the network in their routing tables. The main disadvantage to this approach is that a query incurs a high bandwidth overhead because if the node receiving the query does not have the key being searched, the node has to broadcast the query message to all the other nodes in the network. Therefore, this model is not scalable but is suitable for small networks as it does not consume network bandwidth for maintenance of the network.

7.1.2 Fully Connected (DHT) Network

The Fully Connected (DHT) network is similar to the previously discussed Fully Connected network in Section 7.1.1. The main difference between these two networks is that the Fully Connected (DHT) network is structured such that each node is only responsible for a subset of hashed keys. With this model, when a publish request is received, the Meta-Directory node receiving the request hashes the attributes and then only stores the hashed keys it is responsible for in the local hash table. The remaining keys are then sent to the Meta-Directory nodes responsible for them.

The main advantage to this approach compared the Fully Connected model (Section 7.1.1), is that during a query, if a key is not found in the local hash table, the query is forwarded directly to the node responsible for the hashed key. Therefore, only one request message is sent to the network and the rest of the nodes are not flooded with the request message therefore reducing the network bandwidth used per query.

The main disadvantage of this model is when the size of the network is large. This can cause a significant maintenance overhead in the network as periodic messages are exchanged among all the nodes in the network (Section 3.4.2) to maintain the correct state of the network.

7.1.3 CHORD (DHT) Network

The CHORD (DHT) network is realized as per the CHORD specifications in [22]. This is similar to the Fully Connected (DHT) model as each node is only responsible for a range of hash values. The difference between the two models is that in the CHORD (DHT) network, each node only knows of a subset of nodes in the network which are known as the neighbors of the node.

Since each node only knows of a subset of nodes in the network, if a request is forwarded in the network, the request has to be routed through the node's neighbors in order to reach the node that is responsible for that hashed key. This causes a disadvantage as a higher number of messages to be sent through the network per query compared to the Fully Connected (DHT) model. This network however introduces less maintenance overhead as each node only needs to manage the routing entries for a subset of nodes in the network.

7.1.4 Super Peer Network

The Super Peer network introduces clusters of Meta-Directory nodes and Super Peers which are responsible for the maintenance of the clusters as well as having a direct communication to the rest of the Super Peers in the network.

In large networks, the Super Peer model provides a constant delay per query message as shown by the total number of hops analysis in Figure 32. This makes the Super Peer model scalable as the clusters can be maintained at a network size suitable for the state of the system and the network delay will be minimized regardless of the number of nodes in the system when compared to the CHORD (DHT) network. The Super Peers only need to know of the state of their clusters and maintain direct communication with the other Super Peers in the network.

7.2 Discussion

We were able to design and implement a distributed Meta-Directory system that also gives the user the ability of selecting the network configuration for the distributed Meta-Directory nodes. The framework introduced in this thesis addresses availability, scalability, ease of management, and flexibility. These attributes are discussed and summarized in this sub-section.

7.2.1 Availability

The distributed Meta-Directory architecture ensures that the operation of the system is not interrupted if any one of the nodes is offline. This is achieved by replicating the data stored in the hash tables such that each hash table's keys overlap with the keys handled by the predecessor and successor. Therefore, the system does not have a single point of failure and is always available.

7.2.2 Scalability

Scalability is achieved in the Meta-Directory system as all of the networks offered by the system are highly scalable. A Super Peer network performs very well in large

networks as it introduces clusters of nodes with each node having a super peer that propagates queries within the cluster as well as within the super peers.

7.2.3 Ease of Management

The Meta-Directory system is easy to manage as all the underlying query propagation, network configuration changes and communication are handled by the Meta-Directory system. The system administrator only needs to set up the system by specifying the preferred network configuration and everything else is handled by the underlying system. The overall management of the Meta-Directory nodes is thus seamless.

7.2.4 Transparency

The system is transparent as it does not modify any of the underlying architecture of the service registries. The communication between the distributed Meta-Directory nodes is not tied to the architecture of the service registries as the Meta-Directory nodes are decoupled from the service registries.

The system is also flexible as the network can be selected from any of the four supported networks during the deployment of the system. This flexibility ensures that the system provides good performance in terms of network delay per query, bandwidth used per query, the path length per query and the memory overhead of the routing tables based on the state of the system during deployment.

The advantages of the distributed Meta-Directory system can be briefly summarized as follows:

- The Meta-Directory system allows the service provider to publish service information in any of the local registries and forward the registry information to any of the Meta-Directory nodes in the system. Since the Meta-Directory nodes are not tied to any local registry, as was done in [1], this decouples the registries from the Meta-directory nodes so that the local registries are always available for service publishing and discovery.
- The Meta-Directory system does not change the underlying communication model of any of the already existing Web Service components, which is the communication between the service providers, service requesters, and service registries.
- The Meta-Directory architecture enhances system performance by providing a system administrator the ability of choosing an appropriate interconnection model among Fully Connected, Fully Connected (DHT), CHORD (DHT), and Super Peer networks during deployment.
- Since the Meta-Directory nodes are an added layer on top of the existing registries, the architecture accommodates existing local business registries of different enterprises hence creating a network of registries.
- The load is shared among the distributed Meta-Directory nodes.
- The data in the Meta-Directory nodes is replicated in such a way that a key-value pair can be found in two Meta-Directory nodes at any point in time. This ensures that the system does not provide a single point of failure.
- Depending on the system adopted, the service requester only needs to know the location of at least one registry or one Meta-Directory node and the query will be

propagated within the network by the underlying communication overlay layer. The distributed nature of the system is transparent to the user.

- The query does not have to be propagated to all the distributed registry nodes since information about the services that are stored in the service registries are saved in the Meta-Directory nodes. Therefore, only the service registries that have the information requested will be queried.

7.3 Limitations

In the Meta-Directory system proposed, there is one limitation to the results presented in this thesis:

- In order to fully investigate the performance of the Meta-Directory system implemented, further experiments need to be performed by varying more parameters. Such experiments could not be performed due to time limitation. These experiments should be performed using the PlanetLab test bed or by creating a simulator so that control over all the system parameters such as network traffic can be achieved.

7.4 Future Work

This thesis also provided algorithms that can be applied when the underlying network configuration needs to be changed after the system has already been deployed. Further analysis and implementation of these algorithms are needed, so that the system administrator could indicate the underlying configuration the system should be changed to and the system would change the underlying network model accordingly.

With the algorithms provided, runtime topology change between the Fully Connected model to any of the DHT models is not possible after the system is deployed. Design and implementation of these algorithms need to be investigated.

As future research in this topic, an implementation of a dynamic runtime network change algorithm needs to be investigated. This framework will allow the system administrator to set parameters such as the size of the network and the bandwidth that will be used as thresholds for determining if the current network between the Meta-Directory nodes should be changed. The system will check the current system state and compare it with the values set by the system administrator and perform the network changes automatically without requiring further input from the administrator.

Another area that should be looked at as future research is the implementation of replicated Meta-Directory nodes when a network is deployed with only one Meta-Directory node. This is important to ensure that the system is always available and it does not incur a single point of failure when there is only one centralized Meta-Directory node. This redundant node will keep a replication of all the entries in the primary Meta-Directory node.

Further design and implementation is required on the administrative console so that the administrator can view the overall state of the system. With this view, the user can determine the best configuration given the state of the system and the system requirements.

REFERENCES

- [1] S. Banerjee, S. Basu, S. Garg, S. Garg, S. Lee, P. Mullan, P. Sharma, “Scalable Grid Service Discovery based on UDDI”, in *Proceedings of the 3rd International Workshop on Middleware For Grid Computing*, Grenoble, France, December 2005, pp.1-6.
- [2] R. Barr, “JiST / SWANS – Java in Simulation Time / Scalable Wireless Ad Hoc Network Simulator”, 2005, available from <http://jist.ece.cornell.edu/>. (accessed November, 2007).
- [3] D. Barry, “Web Services and Service-Oriented Architectures”, 2005, available from <http://www.service-architecture.com>.(accessed July, 2007).
- [4] L. Clement, A. Hately, C. V. Riegen, T. Rogers (eds.), “UDDI Specification Technical Committee Draft: Version 3.0.2”, 2004, available from http://uddi.org/pubs/uddi_v3.htm. (accessed June, 2006).
- [5] J. Colgrave, K. Januszewski, “Using WSDL in a UDDI Registry: Version 2.0.2”, 2004, available from <http://www.oasis-open.org/committees/uddi-spec/doc/tn/uddi-spec-tc-tn-wsdl-v2.htm>. (accessed June, 2006).
- [6] Z. Du, J. Huai, Y. Liu, “Ad-UDDI: An Active and Distributed Service Registry”, in *Proceedings of the 6th International Workshop on Technologies for E-Services*, Trondheim, Norway, September 2005, pp. 58-71.
- [7] S. Dustdar, M. Treiber, “A View Based Analysis on Web Service Registries”, *Distributed and Parallel Databases Journal*, Vol. 18, No. 2, September 2005, pp. 147-171.

- [8] T. Gill, F. Kaashoek, J. Li, R. Morris, J. Stribling, “P2PSim: A Simulator for Peer-to-Peer Protocols”, 2005, available from <http://pdos.csail.mit.edu/p2psim/index.html>. (accessed July, 2007).
- [9] “Gnutella Protocol Definition”, 2003, available from <http://rfc-gnutella.sourceforge.net/>. (accessed June, 2007).
- [10] W. Hoschek, “The Web Service Discovery Architecture”, in *Proceedings of the International IEEE/ACM Supercomputing Conference*, Baltimore, USA, November 2002, pp. 1-15.
- [11] F. B. Kashani, C. Chen, C. Shahabi, “WSPDS: Web Services Peer-to-Peer Discovery Service”, in *Proceedings of the International Conference on Distributed System Computing*, Las Vegas, USA, June 2004, pp. 733-743.
- [12] “Open Chord Specification”, 2007, available from <http://open-chord.sourceforge.net>. (accessed June, 2007).
- [13] M.P. Papazoglou, B.J. Kramer, J. Yang, “Leveraging Web-Services and Peer-to-Peer Networks”, in *Proceedings of the 15th International Conference on Advanced Information Systems Engineering*, Klagenfurt, Austria, June 2003, pp. 485-501.
- [14] “PLANETLAB – An open platform for developing, deploying, and accessing planetary-scale services”, 2007, available from <http://www.planet-lab.org/>. (accessed November, 2007).
- [15] Y. Qiao, L. Serghi, G. Papandreou, S. Majumdar, K. Parker, “Web Services Extranet Registry”, *Technical Report*, Alcatel-Lucent and Carleton University, Ottawa, ON, 2005.

- [16] O. D. Sahin, C. E. Gerede, D. Agrawal, A. E. Abbadi, O. Ibarra, J. Su, "SPiDer: P2P-Based Web service Discovery", in *Proceedings of the 3rd International Conference in Service-Oriented Computing*, Amsterdam, The Netherlands, December 2005, pp. 157-169.
- [17] B. Sapkota, D. Roman, D. Fensel, "Distributed Web Service Discovery Architecture", in *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Distributed System and Web Applications and Services*, Gadeloupe, French Caribbean, February 2006, pp. 136-142.
- [18] B. Sapkota, L. Vasiliu, I. Toma, D. Roman, C. Bussler, "Peer-to-Peer Technology Usage in Web Service Discovery and Matchmaking", in *Proceedings of the 6th International Conference on Web Information Systems and Engineering*, New York, USA, November 2005, pp. 418-425.
- [19] M. Schlosser, M. Sintek, S. Decker, W. Nejdl, "A Scalable and Ontology-Based P2P Infrastructure for Semantic Web Services", in *Proceedings of the Second International Conference on Peer-to-Peer Computing*, Washington, USA, September 2002, pp. 104-111.
- [20] C. Schmidt, M. Parashar, "A Peer-to-Peer Approach to Web Service Discovery," *World Wide Web Journal*, Vol. 7, No. 2, June 2004, pp. 211-229.
- [21] "Simple API for XML", 2007, available from http://en.wikipedia.org/wiki/Simple_API_for_XML. (accessed August, 2007).
- [22] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, H. Balakrishnan, "Chord: A scalable peer-to-peer lookup services for distributed system applications", in *Proceedings of*

- the Special Interest Group on Data Communication*, San Diego, USA, August 2001, pp. 149-160.
- [23] C. Sun, Y. Lin, and B. Kemme, "Comparison of UDDI Registry Replication Strategies," in *Proceedings of the International Conference on Web Services (ICWS'04)*, San Diego, USA, July 2004, pp. 218-225.
- [24] "The Network Simulator – ns-2", 2007, available from <http://www.isi.edu/nsnam/ns/>. (accessed November, 2007).
- [25] I. Toma, B. Sapkota, J. Scicluna, J. M. Gomez, D. Roman, D. Fensel, "A P2P Discovery mechanism for Web Service Execution Environment", in *Proceedings of the 2nd International WSMO Implementation Workshop*, Innsbruck, Austria, June 2005, pp. 134-144.
- [26] K. Verma, K. Sivashanmugam, A. Seth, A. Patil, "METEOR-S: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services", *Journal of Information Technology and Management*, Vol. 6, No. 1, January 2005, pp. 17-39.
- [27] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, Vol. 35, No. 2, February 1997, pp. 46-55.
- [28] "XMethods," 2005, available from <http://www.xmethods.net>. (accessed November, 2007).

APPENDIX A: DERIVATION OF TRANSFORMATION COMPLEXITIES

setMaintenance (boolean input, List nodes):

Figure 83 describes the protocol followed when the system's maintenance mode is changed. For each node in the system, the state is toggled between "maintenance" and "active" therefore the complexity for a total of N_T nodes is $O(N_T)$.

addRTEntries (List locations):

The algorithm in Figure 97 is applied to all the *locations* passed to the function. Since it is applied to all the locations N_{RT} , for each routing table the complexity is $O(N_{RT})$.

deleteEntries (List KeyValueEntries):

Figure 96 is the algorithm used to delete all the entries that are passed through *KeyValueEntries* from the node. This is applied to all the key-value entries N_{KV} to provide a complexity of $O(N_{KV})$.

publishToCluster (SuperPeer SP, List keyValueEntries):

From the algorithm described in Figure 95, the Super Peer publishes all the key-value entries N_{KV} passed, to the entry node. Therefore the complexity of this algorithm is $O(N_{KV})$.

getClusters (Network SPNetwork):

Figure 94 shows the algorithm followed when a network wants to get a reference to all the clusters in the network. In this algorithm, the network queries all the Super Peers N_{SP} to provide a complexity of $O(N_{SP})$.

changeNetwork (SPFC , FC):

When the network is changed from an SPFC instance to an FC instance, the algorithm in Figure 84 is applied. First the algorithm gets a list of clusters by using the *getClusters* algorithm. Then the algorithm applies three nested loops. The first loop is applied over all the clusters, the second applied over all the nodes in the cluster, and the third loop applied over all the clusters. Inside the third loop, a function call is made to the *addRTEntries* algorithm. After the nested loops, there is also another loop applied to all the Super Peers.

Overall, in a network of N_{SP} Super Peers, N_C clusters, N_{ci} nodes in a cluster, and N_{RT} routing table entries, the complexity of this algorithm is then:

$$O(N_{SP}) + [O(N_C) * O(N_{ci}) * O(N_C) * O(N_{RT})] + O(N_{SP})$$

Since N_{SP} is equal to N_C , the complexity reduces to:

$$O(N_C^2) * O(N_{ci}) * O(N_{RT})$$

N_{ci} can also be expressed as $N_T - \sum N_{ci}$ and N_{RT} can also be expressed as $N_T - N_{ci}$

therefore the complexity can be expressed as:

$$O(N_C^2) * O(N_T - \sum N_{ci}) * O(N_T - N_{ci})$$

Since N_T is greater than $\sum N_{ci}$ and N_{ci} , the overall complexity of the algorithm reduces to:

$$O(N_C^2 * N_T^2).$$

changeNetwork (SPCHORD, CHORD):

When the network is changed from an SPCHORD instance to a CHORD instance, the algorithm in Figure 86 is applied. First the algorithm gets a list of clusters by using the *getClusters* algorithm. Then the algorithm applies two nested loops. The first loop is applied over all the clusters, and the second is applied over all the nodes in the cluster. Inside the first loop, two function calls are made, one to the *publishToCluster* algorithm and the second to the *deleteEntries* algorithm. Inside the second loop that is applied to all the nodes in the cluster, two consecutive calls are made to the nodes, one to leave and the other to join a cluster. After the nested loops, there is also another loop applied to all the Super Peers.

Overall, in a network of N_{SP} Super Peers, N_C clusters, N_{KV} key-value entries, N_{ci} nodes in a cluster, and N_T total nodes, the complexity of this algorithm is then:

$$O(N_{SP}) + O(N_C) * [O(N_{KV}) + O(N_{KV}) + O(N_{ci}) * O(\text{Log } N_T)] + O(N_{SP})$$

Since N_{SP} is equal to N_C , the complexity reduces to:

$$O(N_C) * [O(N_{KV}) + O(N_{ci}) * O(\text{Log } N_T)]$$

N_{ci} can also be expressed as $N_T - \sum N_{ci}$ therefore the complexity can be expressed as:

$$O(N_C) * [O(N_{KV}) + O(N_T - \sum N_{ci}) * O(\text{Log } N_T)]$$

Since N_T is greater than N_{ci} , the overall complexity of the algorithm reduces to:

$$O(N_C) * [O(N_{KV}) + O(N_T) * O(\text{Log } N_T)]$$

This overall complexity can be expressed as:

$$O(N_C * N_T * \text{Log } N_T) + O(N_C * N_{KV})$$

changeNetwork (FC , SPFC):

When the network is changed from an FC instance to an SPFC instance, the algorithm in Figure 88 is applied. First the algorithm has three consecutive loops over the number of clusters which is a parameter passed during the call. Then the algorithm applies two nested loops. The first loop is applied over all the clusters and the second is also applied over all the clusters. Inside the first loop, a function call is made to the *addRTEentries* algorithm. Then the algorithm is followed by three nested loops. The first loop applied over all the clusters, the second applied to all the nodes in the cluster, and the third applied over all the remaining nodes in the network minus the nodes in the current cluster.

Overall, in a network of N_C clusters, N_{RT} routing table entries, N_{ci} nodes in a cluster, and N_T total nodes, the complexity of this algorithm is then:

$$O(N_C) + O(N_C) + O(N_C) + O(N_C) * [O(N_C) + O(N_{RT})] + O(N_C) * [O(N_{ci}) * O(N_T - N_{ci})]$$

N_{ci} can also be expressed as $N_T - \sum N_{ci}$ therefore the complexity can be expressed as:

$$O(N_C^2) + O(N_C * N_{RT}) + O(N_C) * [O(N_T - \sum N_{ci}) * O(N_T - N_{ci})]$$

Since N_T is greater than N_{ci} , the overall complexity of the algorithm reduces to:

$$O(N_C^2) + O(N_C * N_{RT}) + O(N_C) * O(N_T^2)$$

Since N_T is greater than N_C and N_{RT} , the overall complexity of the algorithm reduces to:

$$O(N_C * N_T^2).$$

changeNetwork (CHORD , SPCHORD):

When the network is changed from an FC instance to an SPFC instance, the algorithm in Figure 90 is applied. First the algorithm has three consecutive loops over the

number of clusters which is a parameter passed during the call. Then the algorithm applies two nested loops. The first loop is applied over all the clusters and the second is also applied over all the clusters. Inside the first loop, a function call is made to the *addRTEntries* algorithm. Then the algorithm is followed by two nested loops. The first loop is applied over all the clusters and the second loop over all the nodes in the cluster. In the second loop, two consecutive calls are made to the nodes, one to leave and the other to join a cluster. Then the algorithm applies another loop over all the clusters, whereby inside this loop another loop is applied over all the key-value entries in that cluster followed by a call to the *publishToCluster* algorithm and a call to the *deleteEntries* algorithm.

Overall, in a network of N_C clusters, N_{RT} routing table entries, N_{ci} nodes in a cluster, N_T total nodes, N_{KVi} key-value entries in a cluster, and N_{KV} total key-value entries in the network, the complexity of this algorithm is then:

$$O(N_C) + O(N_C) + O(N_C) + O(N_C) * [O(N_C) + O(N_{RT})] + [O(N_C) * O(N_{ci}) * O(\text{Log } N_T)] + O(N_C) * [O(N_{KVi}) + O(N_{KV}) + O(N_{KV})]$$

N_{ci} can also be expressed as $N_T - \sum N_{ci}$ therefore the complexity can be expressed as:

$$O(N_C^2) + O(N_C * N_{RT}) + [O(N_C) * O(N_T - \sum N_{ci}) * O(\text{Log } N_T)] + O(N_C) * [O(N_{KVi}) + O(N_{KV})]$$

Since N_T is greater than N_{ci} and N_{RT} , and N_{KV} is greater than N_{KVi} the overall complexity of the algorithm reduces to:

$$O(N_C^2) + O(N_C * N_T * \text{Log } N_T) + O(N_C * N_{KV}).$$

changeNetwork (CHORD , FCDHT):

When the network is changed from a CHORD instance to an FCDHT instance, the algorithm in Figure 92 is applied. In this algorithm, for each node in the system the routing table size is modified. Therefore, the complexity of this algorithm for a network of N_T total nodes is: $O(N_T)$.

changeNetwork (FCDHT , CHORD):

When the network is changed from an FCDHT instance to a CHORD instance, the algorithm in Figure 93 is applied. In this algorithm, for each node in the system the routing table size is modified. Therefore, the complexity of this algorithm for a network of N_T total nodes is: $O(N_T)$.