A Pre-fetching and Caching System for Web Service Registries

by

Ming Huang

A thesis submitted to

the Faculty of Graduate Studies and Research

in partial fulfillment of

the requirement for the degree of

MASTER OF COMPUTER SCIENCE

School of Computer Science

at

Carleton University

Ottawa, Ontario

August, 2008

Your file Votre référence
ISBN: 978-0-494-44108-4
Our file Notre référence
ISBN: 978-0-494-44108-4

Abstract

In Service-Oriented Architectures (SOA), the Web Service registry system plays an indispensable and essential role. Web Service providers publish their Web Services to Web Service registry servers and Web Service requestors discover Web Services from the registry servers. Subsequently, Web Service requestors bind and obtain the desired Web Service from Web Service providers. Several existing problems in Web Service registry system make the discovery in Web Service registry servers impractical and inefficient. The topic of this research is focusing on improving the performance of Web Service discovery. This thesis devises an effective pre-fetching technique that can be integrated with existing caching techniques for reducing the latency associated with Web Service discovery. The proposed approach utilizes two core parts: a caching proxy located as close to the clients as possible and a pre-fetching proxy located as close to the Web Service registry servers as possible. Experimental results demonstrate that the proposed approach can significantly improve the performance of Web Service discovery.

Acknowledgements

I would like to express my most sincere thanks to my supervisors, Dr. Shikharesh Majumdar and Dr. Babak Esfandiari, for their invaluable wisdom, advice, guidance, encouragement, patience, and support during this thesis. I feel very fortunate to have had their supervision and extremely grateful for the learning experience they have given me. I would also like to thank Dr. Chung-Horng Lung, who took his valuable time to review and attend many of my presentations throughout my Master's program.

I extremely appreciate the Ontario Centres of Excellence and Alcatel-Lucent for providing operational funding for my research. I would also like to express my sincere thanks to my English editor, Vicky Bell, for her help in improving my thesis.

My special thanks go to my many teammates and my friends who have supported and encouraged me during my Master's program. I would especially like to thank Afiya Kassim, Yang Cao, Zhimin Song and Liang Dai.

Above all, exceptional and affectionate appreciation to my family: my wife, Pin Zeng; my younger sister, Ying Huang; my father, Maowu Huang; and my mother, Zhongjie Zhu. It was the love, the understanding, the encouragement, and the concrete and invaluable help from them that enabled me to accomplish this work. I feel indeed lucky to have such a wonderful family.

Table of Contents

ABSTRACT II
ACKNOWLEDGEMENTS III
TABLE OF CONTENTS IV
LIST OF TABLES VIII
LIST OF FIGURES IX
LIST OF EQUATIONS XII
GLOSSARY OF TERMS XIII
CHAPTER 1: INTRODUCTION 1
1.1 INTRODUCTION AND MOTIVATION 1
1.2 THESIS GOAL 2
1.3 THESIS CONTRIBUTIONS 2
1.4 ORGANIZATION OF THESIS 3
CHAPTER 2: BACKGROUND 5
2.1 INTRODUCTION 5
2.2 WEB SERVICES 5
2.3 WEB SERVICE COMPONENTS AND SERVICE-ORIENTED ARCHITECTURES (SOA) 7
2.3.1 Service-Oriented Architectures (SOA) *8*
2.3.2 UDDI *12*
2.3.3 SOAP *12*
2.3.4 WSDL *13*
2.4 WEB SERVICE DISCOVERY MECHANISMS AND REGISTRY SYSTEMS 14
2.4.1 UDDI Registry Systems Architecture *14*
2.4.2 UDDI Data Model *15*
2.4.3 How Does UDDI Work? *18*
2.4.4 UDDI Publication API *20*
2.4.5 UDDI Inquiry API *20*
2.5 SUMMARY 20
CHAPTER 3: RELATED WORKS AND PROPOSED APPROACH **22**
3.1 INTRODUCTION 22
3.2 CACHING IN TRADITIONAL WEB APPLICATIONS 23
3.2.1 What Data Should be Cached *23*
3.2.2 Where Caching Should Take Place *24*
3.3 CACHING IN WEB SERVICES 25
3.3.1 What Data Should be Cached *26*
3.3.2 Where Caching Should Take Place *29*
3.4 CACHING IN WEB SERVICE REGISTRY SYSTEMS 33
3.4.1 What Data Should Be Cached *33*

3.4.2 Where Caching Should Take Place 34
3.5 PROPOSED APPROACH FOR CACHING 36
3.5.1 What Data Should be Cached 36
3.5.2 Where Caching Should Take Place 37
3.6 PRE-FETCHING 38
3.6.1 In Traditional Web Applications 38
3.6.2 In Web Service and Web Service Registry Systems 41
3.6.3 Proposed Pre-fetching Approach for Integrating Caching 42
3.7 SUMMARY 43
CHAPTER 4:SYSTEM DESIGN **44**
4.1 INTRODUCTION 44
4.1.1 Problems with Existing Web Service Registry Systems 44
4.1.2 High Level Structure of the Proposed Caching and Pre-fetching System 45
4.1.3 Prototype for Integrating Caching and Pre-fetching in Web Service Registry Systems 46
4.2 CACHING ALGORITHM IN WEB SERVICE DISCOVERY SYSTEMS 47
4.2.1 What Data Should be Cached 48
4.2.2 Where the Cache Should be Located 49
4.2.3 Managing the Cached Data 51
4.3 THE PRE-FETCHING ALGORITHM IN WEB SERVICE DISCOVERY SYSTEMS 52
4.3.1 The Functionalities of the Pre-fetching Proxy 53
4.3.2 Where Should the Pre-fetches Take Place? 56
4.3.3 Building/Maintaining a Pre-fetching Graph 57
4.3.4 Pre-fetching Process 58
4.3.5 Example of Building, Maintaining and Pre-fetching in a Pre-fetching Graph 60
4.4 SUMMARY 61
CHAPTER 5:IMPLEMENTATION **64**
5.1 INTRODUCTION 64
5.2 JUDDI REGISTRY FRAMEWORK 64
5.2.1 Requirements for jUDDI Installation 64
5.2.2 jUDDI Implementation Details 66
5.2.3 Modified jUDDI for Implementing Caching Proxy 71
5.2.4 Modified jUDDI for Implementing the Pre-fetching Proxy 78
5.3 SUMMARY 79
CHAPTER 6:EXPERIMENTAL ENVIRONMENT AND MEASUREMENT PARAMETERS **81**
6.1 INTRODUCTION 81
6.2 THE PURPOSE OF THE EXPERIMENTS 81
6.3 RESOURCES AND ENVIRONMENTS REQUIRED 82
6.4 EXPERIMENTAL APPLICATIONS 83
6.5 MEASUREMENT PARAMETERS 87

6.5.1 Response Time ... *88*
6.5.2 Number of Web Service Inquiry Requests ... *92*
6.5.3 Cache Size and the Cache-Hit Ratio (H) ... *93*
6.5.4 Pre-fetch-Hit Ratio (H_p) ... *94*
6.5.5 Factors Impacting the Pre-fetch-Hit Ratio H_p and Cache-Hit Ratio H ... *98*
6.6 RELATIONSHIP AMONG PARAMETERS AND PERFORMANCE METRICS ... 100
6.7 SUMMARY ... 103
CHAPTER 7:EXPERIMENTAL RESULTS ... 105
7.1 INTRODUCTION ... 105
7.2 EVOLUTION OF CACHE-HIT RATIO (H) OVER TIME (T) ... 105
7.2.1 Purpose of the Experiment ... *105*
7.2.2 Experimental Results and Analysis ... *106*
7.2.2.1 The Effect of Varying the Threshold on Edge Weight ($W_{threshold}$) ... 106
7.2.2.2 The Effect of Varying the Depth of C-BFS (D) ... 111
7.2.2.3 The Effect of Varying the Time Interval Threshold ($T_{threshold}$) ... 114
7.2.2.4 The Effect of Varying the Application Overlap Rate (v) ... 118
7.2.2.5 Influence of Pre-fetching Rate (m) on Cache-Hit Ratio (H) ... 122
7.2.3 Summary ... *123*
7.3 EVOLUTION OF PRE-FETCHING-RATE (M) OVER TIME (T) ... 123
7.3.1 Purpose of the Experiment ... *123*
7.3.2 Experimental Results and Analysis ... *124*
7.3.2.1 The Effect of Varying the Threshold on Edge Weight ($W_{threshold}$) ... 124
7.3.2.2 The Effect of Varying the Application Overlap Rate (v) ... 125
7.3.3 Summary ... *126*
7.4 EVOLUTION OF RESPONSE TIME OVER TIME (T) ... 127
7.4.1 Purpose of the Experiment ... *127*
7.4.2 Experimental Results and Analysis ... *128*
7.4.2.1 Varying the Threshold on Edge Weight ($W_{threshold}$) ... 128
7.4.2.2 Investigation of EDT ... 131
7.4.2.3 Influence of Cache-Hit Ratio (H) on Response Time (r) ... 131
7.4.3 Summary ... *133*
7.5 INTERRELATIONSHIP BETWEEN PRE-FETCH-HIT RATIO (H_p) AND CACHE-HIT RATIO (H) ... 133
7.5.1 Purpose of the Experiment ... *133*
7.5.2 Experimental Results and Analysis ... *134*
7.5.2.1 Influence of UDDI Inquiry Requests over Time (t) on Pre-fetch Rate (m) ... 134
7.5.2.2 Influence of UDDI Inquiry Requests over Time (t) on the Number of Pre-fetch hits (h) ... 135
7.5.2.3 Relationship between Cache-Hit Ratio (H) and Pre-fetch-Hit Ratio (H_p) ... 136
7.5.3 Summary ... *138*
CHAPTER 8:CONCLUSIONS ... 139

8.1 SUMMARY OF RESEARCH 139
8.2 CONCLUSIONS 139
8.3 FUTURE WORK 141
BIBLIOGRAPHY 144
APPENDIX A 151
APPENDIX B 153
APPENDIX C 156
C.1 EVOLUTION OF PRE-FETCHING-RATE (M) OVER TIME (T) 156
C.1.1 The Effect of Varying the Depth of Conditional Breadth-First Search (D) 156
C.1.2 The Effect of Varying the Time Interval Threshold ($T_{threshold}$) 156
C.2 EVOLUTION OF RESPONSE TIME (R) OVER TIME (T) 157
C.2.1 The Effect of Varying the Depth of Conditional Breadth-First Search (D) 157
C.2.2 The Effect of Varying the Time Interval Threshold ($T_{threshold}$) 159
C.2.3 The Effect of Varying the application Overlap Rate (v) 160

List of Tables

Table 2-1 Required Inquiry API Methods in UDDI Specification [37] [39] 21

Table 3-1 Comparison of Three Caching Policies 25

Table 3-2 Cache Table [23] 28

Table 3-3 Cached UDDI Registration Data 35

Table 5-1 jUDDI Packages Involved in UDDI Discovery Systems 69

Table 5-2 jUDDI Packages Involved in UDDI Discovery Systems (Continue) 70

Table 6-1 Parameters – Impact of on Performance 101

Table 6-2 Metrics – Performance Measurement 101

Table 7-1 Cache-Hit Ratio (H) vs. UDDI Inquiry Requests Over Time (t) – Variations in $W_{threshold}$ (YCYP, $T_{threshold}$=10000ms, v=10%, D=2) 110

Table 7-2 Cache-Hit Ratio (H) vs. UDDI Inquiry Requests Over Time (t) – Variations in D (YCYP, $T_{threshold}$=10000ms, v=10%, $W_{threshold}$=2) 113

Table 7-3 Cache-Hit Ratios (H) vs. UDDI Inquiry Requests over Time (t) – Variations in $T_{threshold}$ (YCYP, v=10%, $W_{threshold}$=2, D=2) 116

Table 7-4 Cache-Hit Ratios (H) vs. UDDI Inquiry Requests Over Time (t) – Variations in v (YCYP, $T_{threshold}$=10000ms, $W_{threshold}$=2, D=2 & YCNP & NCNP) 121

Table 7-5 Response Time (r) vs. UDDI Inquiry Requests over Time (t) – Variations in $W_{threshold}$ (YCYP, $T_{threshold}$=10000ms, v=10%, D=2) 130

Table C-1 Response Time vs. UDDI Inquiry Requests over Time (t) – Variations in D (YCYP, $T_{threshold}$=10000ms, v=10%, $W_{threshold}$=2) 158

Table C-2 Response Time vs. UDDI Inquiry Requests over Time (t) – Variations in $T_{threshold}$ (YCYP, v=10%, D=2, $W_{threshold}$ =2) 160

Table C-3 Response Time vs. UDDI Inquiry Requests over Time (t) – Variations in v (YCYP, $T_{threshold}$=10000ms, D=2, $W_{threshold}$=2 & YCNP & NCNP) 161

List of Figures

Figure 2-1 Roles and Operations in Service-Oriented Architecture [35] 11

Figure 2-2 Layers of Technologies and Protocols Currently Used for Web Service [37] 13

Figure 2-3 Relationships among UDDI Data Structures 17

Figure 2-4 UDDI Core Data Structures [33] 17

Figure 2-5 How UDDI Works [8] 19

Figure 3-1 Caching SOAP Messages in Web Service 29

Figure 3-2 Client-side Proxy Cache Architecture [19] 31

Figure 3-3 Web Service Architecture (a) Without Distributed Caching and (b) With Distributed Caching [21] 32

Figure 3-4 Pre-fetching Strategies 39

Figure 3-5 Top 10 Pre-fetching [15] 40

Figure 4-1 Structure of Prototype for Integrating of Caching and Pre-fetching in Web Service Registry Systems 45

Figure 4-2 Cache UDDI entity objects as a "key-value" pair in JCS 50

Figure 4-3 UDDI Inquiry Request Hit in the Caching Proxy 50

Figure 4-4 UDDI Inquiry Request Not Hit in Caching Proxy 53

Figure 4-5 Graph-based Pre-fetching Process 56

Figure 4-6 Pseudo-code for Building/Maintaining a Pre-fetching Graph 58

Figure 4-7 Pre-fetching Graph – Updating, Maintaining and Pre-fetching Process 63

Figure 5-1 Implementation Structure 73

Figure 5-2 Sequence Diagram for Integration of Caching and Pre-fetching Proxy 75

Figure 5-3 XML Schema of ResponseList 76

Figure 6-1 Response Time (r) - Mean and Standard Deviation 85

Figure 6-2 Definition of EDT 90

Figure 6-3 Definition of Pre-fetch-Hit Ratio (Hp) – Before Caching 95

Figure 6-4 Definition of Pre-fetch-Hit Ratio (H_p) – After Caching..... 96

Figure 6-5 Relationship of Influence of Various Parameters on Metrics..... 102

Figure 7-1 Influence of UDDI Inquiry Requests Over Time (t) on Cache-Hit Ratio (H) – Variations in $W_{threshold}$ (YCYP, $T_{threshold}$=10000ms, v=10%, D=2)..... 107

Figure 7-2 Influence of Threshold on Edge Weight ($W_{threshold}$) on Cache-Hit Ratio (H).111

Figure 7-3 Influence of UDDI Inquiry Requests Over Time (t) on Cache-Hit Ratio (H) – Variations in D (YCYP, $T_{threshold}$=10000ms, v=10%, $W_{threshold}$=2).....112

Figure 7-4 Influence of Searching Depth of C-BFS (D) on Cache-Hit Ratio (H).....114

Figure 7-5 Influence of UDDI Inquiry Requests Over Time (t) on Cache-Hit Ratio (H) – Variations in $T_{threshold}$ (YCYP, v=10%, $W_{threshold}$=2, D=2) 115

Figure 7-6 Influence of Time Interval Threshold ($T_{threshold}$) on Cache-Hit Ratio (H).....117

Figure 7-7 Influence of UDDI Inquiry Requests Over Time (t) on Cache-Hit Ratio (H) – Variations in v (YCYP, $T_{threshold}$=10000ms, $W_{threshold}$=2, D=2 & YCNP & NCNP)..... 120

Figure 7-8 Influence of Overlap Rate of Applications (v) on Cache-Hit Ratio (H)..... 121

Figure 7-9 Relationship between Pre-fetching Rate and Cache-Hit Ratio 122

Figure 7-10 Evolution of Pre-fetching Rate (m) Over Time (t) as represented by the sequence of UDDI Inquiry Requests – Variations in $W_{threshold}$ (YCYP, $T_{threshold}$=10000ms, v=10%, D=2) 125

Figure 7-11 Evolution of Pre-fetching Rate (m) Over Time (t) as represented by the sequence of UDDI Inquiry Requests – Variations in v (YCYP, $T_{threshold}$=10000ms, $W_{threshold}$=2, D=2) 126

Figure 7-12 Influence of UDDI Inquiry Requests over Time (t) on Response Time (r) – Variations in $W_{threshold}$ (YCYP, $T_{threshold}$=10000ms, v=10%, D=2)..... 130

Figure 7-13 Influence of Web Service Discovery Application over Time (t) on EDT – (YCYP, $T_{threshold}$=10000ms, D=2, $W_{threshold}$=2, v=10% & YCNP, v=10% & NCNP)..... 132

Figure 7-14 Influence of Cache-Hit Ratio (H) on the Response Time (r) 132

Figure 7-15 Influence of UDDI Inquiry Requests over Time (t) on Pre-fetch Rate (m) – (YCYP, $W_{threshold}$=2, D=5, $T_{threshold}$=10000ms, v=10%)..... 135

Figure 7-16 Influence of UDDI Inquiry Requests over Time (t) on Number of Pre-fetch hits (h) – (YCYP, $W_{threshold}$=2, D=5, $T_{threshold}$=10000ms, v=10%) 136

Figure 7-17 Relationship between Cache-Hit Ratio (H) and Pre-fetch-Hit Ratio (H_p) – (YCYP, $W_{threshold}$=2, D=5, $T_{threshold}$=10000ms, v=10%)..... 137

Figure A-1 Structure of jUDDI (Part I)..... 151

Figure A-2 Structure of jUDDI (Part II) 152

Figure C-1 Evolution of Pre-fetching Rate (m) over Time (t) as Represented by the Sequence of UDDI Inquiry Requests (t) – Variations in D (YCYP, $T_{threshold}$=10000ms, v=10%, $W_{threshold}$=2)..... 156

Figure C-2 Evolution of Pre-fetching Rate (m) over Time (t) as Represented by UDDI Inquiry Requests – Variations in $T_{threshold}$ (YCYP, v=10%, $W_{threshold}$=2, D=2)..... 157

Figure C-3 Influence of UDDI Inquiry Requests over Time (t) on Response Time – Variations in D (YCYP, $T_{threshold}$=10000ms, v=10%, $W_{threshold}$=2)..... 158

Figure C-4 Influence of UDDI Inquiry Requests over Time (t) on Response Time – Variations in $T_{threshold}$ (YCYP, v=10%, D=2, $W_{threshold}$ =2)..... 159

Figure C-5 Influence of UDDI Inquiry Requests over Time (t) on Response Time – Variations in v (YCYP, $T_{threshold}$=10000ms, D=2, $W_{threshold}$=2 & YCNP & NCNP)..... 161

List of Equations

Equation 6-1 Value of A Performance Metric at Time *t* 84

Equation 6-2 Relationship between Effective Discovery Time (EDT) and Response Time (r) 88

Equation 6-3 Definition of Cache-Hit Ratio H 93

Equation 6-4 Definition of Pre-fetch-Hit Ratio H_p 97

Glossary of Terms

A	Number of Applications
API	Application Programming Interface
ARC	Adaptive Replacement Cache
ASF	Apache Software Foundation
B2B	Business-to-Business
B2C	Business-to-Consumer
BFS	Breadth-First Search
C-BFS	Conditional Breadth-First Search
CCM	Custom Cache Managers
D	Depth of the Conditional Breadth-First Search
DOM	Document Object Model
D-U-N-S	Data Universal Numbering System
EAI	Enterprise Application Integration
EAN	European Article Numbering Systems
EDT (ms)	Effective Discovery Time
FTP	File Transfer Protocol
h	Number of Pre-fetch hits
H (%)	Cache-Hit Ratio
H_p ***(%)***	Pre-fetch-Hit Ratio
HMLRU	Linked Hash Map LRU
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol over Secure Socket Layer
ISO	International Organization for Standardization
JCS	Java Caching System

JRE	Java SE Runtime Environment
LRU	Least Recently Used
m	Pre-fetching Rate
MRU	Most Recently Used
n	Number of Web Service Inquiry Requests in a Single Application
NAICS	North American Industry Classification System
NCNP	Experimental Model – No Caching and No Pre-fetching
OASIS	Organization for the Advancement of Structured Information Standards
r (ms)	Response Time of UDDI Inquiry Request
S_{req}	Size of Single UDDI Inquiry Request Object
S_{res}	Size of Single UDDI Response Object
t	Parameter Representing the Sequence of UDDI Inquiry Requests in Experiments
$T_{threshold}$ ***(ms)***	Time Interval Threshold
TTL	Time-To-Live
v (%)	Overlap Rate of Web Service Inquiry Requests between Applications
$W_{threshold}$	Threshold on Edge Weight
W3C	World Wide Web Consortium
YCNP	Experimental Model – Yes Caching but No Pre-fetching
YCYP	Experimental Model – Yes Caching and Yes Pre-fetching
SAX	Simple API for XML
SOA	Service-Oriented Architectures
SOAP	Simple Object Access Protocol
SSPLC	Semantic SOAP Protocol Level Cache
WSDL	Web Service Definition Language
UDDI	Universal Description, Discovery and Integration

UNSPSC	United Nations Standard Products and Services Code
URI	Uniform Resource Identifier
UUID	Universally Unique ID
XML	Extensible Markup Language

Chapter 1: Introduction

1.1 Introduction and Motivation

In Service-Oriented Architectures (SOA) based on Web Services, the Web Service discovery mechanism plays an indispensable role. The goal of Universal Description, Discovery and Integration (UDDI) [8] is to build registries that maintain the registered Web Service information published by Web Service providers.

However, Web Service registry replies can suffer from high latency, and as the number of registered Web Services continues to increase, the workload of Web Service registry servers will also greatly increase. These drawbacks of the current Web Service registry systems can make the discovery of Web Service based on a Web Service registry only impractical. Proper caching and pre-fetching techniques in Web Service registry systems should reduce the latency in discovering a specific Web Service, shrink the response time for searching a desired Web Service, and cut down the workload of the Web Service registry server. All of these benefits will improve the performance of Web Service registry systems. Caching improves system performance by storing responses to the discovery requests close to the requester so that subsequent discovery requests for the same Web Service can be satisfied from the cache. A pre-fetching technique, on the other hand, attempts to guess the information that will be required shortly after the current

discovery request, and fetches the information in advance to the cache so that average latency associated with Web Service discovery request can be reduced.

1.2 Thesis Goal

The goal of this thesis is to devise an approach based on caching and pre-fetching that can improve the performance of Web Service discovery in Web Service registry systems. The primary focus of this thesis is on pre-fetching technique. The proposed approach aims to resolve the drawbacks and gain all the benefits mentioned in Section 1.1. To achieve this research goal, alternative approaches are examined, potential benefits and limitations are evaluated, and the implementing algorithm, experimental tools and environment are designed. In addition, a prototype incorporating the above designs and concepts is developed and tested.

1.3 Thesis Contributions

In this thesis, a proposed pre-fetching technique for integrating with caching on Web Service registry has been presented to improve system performance and optimize the performance of Web Service discovery. A caching proxy is located as close to the client as possible and cache the Web Service inquiry request object and the response object associated with it. The pre-fetching proxy is located as close to the Web Service registry

server as possible and records the history of system behavior to build a relationship among Web Service inquiry requests. The history of system behavior is captured as a pre-fetching graph, which is dynamically built and updated in the pre-fetching proxy. Thereafter, the pre-fetching proxy can predict a cluster of additional Web Service inquiry requests based on the currently received original Web Service inquiry request and send the predicted requests to the Web Service registry server, together with the original request. The thesis focuses on the pre-fetching component and integrates it with a caching component provided by Apache Java Caching System (JCS) [41]. A prototype of the integrated system has been implemented and tested. A performance analysis of this approach based on measurements made on the prototype subjected to synthetic workload provides deep insights into system behavior and performance.

1.4 Organization of Thesis

The rest of the thesis is organized as follows: Chapter 2 provides a brief overview of the Web Service background relevant to this thesis. Chapter 3 reviews various existing caching and pre-fetching techniques for improving the performance of traditional Web applications, Web Service and Web Service registry systems, respectively. Chapter 4 presents the details of the pre-fetching algorithm and the proposed approach for integrating caching and pre-fetching in Web Service discovery systems, while Chapter 5 describes the implementation of the proposed approach. Chapter 6 discusses the

experimental procedures used for analyzing system performance, including the various resources and the environment required for successfully completing the experiments, and also briefly introduces the measurement parameters that can be used to analyze the experimental results. Results of the experiments and complete analysis of the impact of various parameters on the performance of Web Service registry systems are provided in Chapter 7. The last chapter of this thesis, Chapter 8, summarizes how the thesis meets the stated objectives and confirms the contributions claimed. A discussion of the limitations, potential enhancements to this research and directions for future research are also included in Chapter 8.

Chapter 2: Background

2.1 Introduction

Although the main thrust of this thesis is to study the improvement of the performance of Web Service discovery in Web Service registry systems, it will be helpful to review other related aspects of Web Services in order to understand the approach proposed in this thesis. Section 2.2 briefly introduces basic concepts underlying Web Service technology, while Section 2.3 discusses the three main components of Web Services. Section 2.4 provides an overview of the discovery mechanisms in Web Service registry systems, which is the main research area in this thesis.

2.2 Web Services

This section defines Web Services, describes the current standards and technologies associated with Web Services, and discusses the problems that can be solved using Web Services.

A Web Service is a platform-independent and implementation-independent software component that can be described using a service description language, such as Web Service Definition Language (WSDL); published to a registry of Web Services; discovered through a standard mechanism (at runtime or design time); invoked through a

declared Application Programming Interface (API), usually over a network; and composed with other services [33]. The World Wide Web Consortium (W3C) Web Service Architecture group [34] has defined a Web Service as a software system identified by a Uniform Resource Identifier (URI), whose public interfaces and bindings are defined and described using a standard, formal Extensible Markup Language (XML) notion, called its service description; the service description is stored in a specific document. Its definition—more specifically, the description document—can be discovered by other software systems. Because the service description provides all of the details necessary to interact with the service, including message formats (which detail the operations), transport protocols, and location, the requesting systems may then interact with the Web Service in a manner prescribed by its definition, using an XML-based message transfer protocol, such as Simple Object Access Protocol (SOAP), conveyed by general Internet protocols such as the Hypertext Transfer Protocol (HTTP), the Hypertext Transfer Protocol over Secure Socket Layer (HTTPS) and the File Transfer Protocol (FTP). Therefore, from a technical perspective, a Web Service is an interface that describes a collection of operations that are network accessible through standardized XML messaging to fulfill a set of specific tasks [33].

As defined above, a Web Service can be accessed through a published API that hides the implementation details of the Web Service so that it can be used independently of the hardware or software platform on which it is implemented, and independently of the

programming language in which it is written. This most important characteristic allows and enhances Web Service-based applications to be flexible, loosely coupled, inter-operable, component-oriented and cross-technology implementations can be decomposed and recomposed to reflect changes in the business [33]. Based on this important feature, Web Service technology is usually applied to two broad categories: Enterprise Application Integration (EAI) and Business-to-Business (B2B) partner integration over the Internet. It is different from the existing traditional Web applications, as it is an approach related to access and application integration rather than a specific implementation technology. A Web Service adapts to a typical B2B environment, while a traditional Web application adapts to a Business-to-Consumer (B2C) environment. In a B2B environment, an application interaction between businesses communicated over the Internet does not involve human intervention. On the other hand, in a B2C environment, an application interaction occurs between a human being, often using a Web browser, and applications provided by a business.

2.3 Web Service Components and Service-Oriented Architectures (SOA)

A pattern called Service-Oriented Architectures (SOA) arises whenever Web Services are applied to application integration. This section briefly introduces the concept of SOA and explains how the various Web Service technical standards, such as SOAP, WSDL, and

UDDI, and several roles in SOA, such as Web Service requestor, Web Service provider and Web Service registry, work together.

2.3.1 Service-Oriented Architectures (SOA)

A Service-Oriented Architecture (SOA) is a business-centric IT architectural approach that supports integrating a business as linked, repeatable business tasks, or services. SOA helps users build composite applications, which are applications that draw upon functionality from multiple sources within and beyond the enterprise to support horizontal business processes [35] [36]. Any Service-Oriented Architecture contains three core roles: Web Service requestor, Web Service provider, and Web Service registry.

- *Web Service requestor*: a single or collection of client applications that looks up Web Service description published to one or more Web Service registries and is responsible for using Web Service descriptions found to bind to or invoke a Web Service hosted by Web Service providers. In this thesis, the Web Service requestor can be considered the "client side" of a "Client-Server Architecture" between the Web Service requestor and the Web Service provider.
- *Web Service provider*: a creator and owner of an available Web Service. Its responsibilities include creating a Web Service description, publishing that service description to Web Service registries, and receiving Web Service invocation messages from one or more Web Service requestors. Generally the Web Service

provider can be considered the "server side" of a "Client-Server Architecture" between the Web Service requestor and the Web Service provider.

- *Web Service registry*: a Web Service registry uses one or more servers in which the Web Service provider publishes Web Service descriptions and allows Web Service requestors to search the collection of Web Service descriptions stored in the Web Service registry. It acts like a matchmaker between the Web Service requestor and the Web Service provider [35]. Once the Web Service registry has made the match, the Web Service requestor obtains the desired information, the rest of the interaction takes place directly between the Web Service requestor and the Web Service provider for the Web Service invocation.

Moreover, any Service-Oriented Architecture has three core operations, publishing, finding and binding, which define the contracts among the SOA's three core roles.

- *Publishing*: publishing is an operation of Web Service registration in the Web Service registry. When a Web Service provider publishes its Web Service description to a Web Service registry, the details of that Web Service are available to a community of Web Service requestors. The actual details of the publishing API depend on how the Web Service registry is implemented. The most popular Web Service registry implementations use the standard known as Universal Description, Discovery and Integration (UDDI) [35], which defines the specification of implementing the publishing operation.

- *Finding*: the finding operation is the logical dual of the publishing operation. With the finding operation, the Web Service requestor sets up a collection of search criteria, such as classification of business or type of service. The Web Service registry matches the finding criteria against its collection of published Web Service descriptions. The result of the finding operation is a list of Web Service descriptions that match the finding criteria that the Web Service requestor needs.
- *Binding*: the binding operation is an operation through which a Web Service requestor acquires the description as to where and how to invoke the Web Service, i.e. how to format the message in a protocol-specific manner or where the access point of the Web Service is. In other words, a binding specifies the protocol and data format to be used in transmitting messages. The Web Service requestor can invoke a Web Service and obtain the services from the Web Service provider once the binding is finished.

Figure 2-1 [35] illustrates the three core roles and three core operations in an SOA. Because the main thrust of this thesis is to study the improvement of the performance of Web Service discovery in Web Service registry systems, the research area focuses only on two of the core roles and one of the operations. The two core roles of interest in this thesis are the Web Service registry and the Web Service requestor. The operation of interest in this thesis is the finding operation that involves the Web Service registry and the Web Service requestor. Although the Web Service provider can generally be

considered the "server side" of a "Client-Server Architecture" between the Web Service requestor and the Web Service provider, in this thesis we only consider the Web Service registry as the "server side" of a "Client-Server Architecture" between the Web Service requestor and the Web Service registry. From now on, the term "Web Service discovery" includes operations between the Web Service requestor and the Web Service registry. The term "Web Service application" indicates systems that are responsible for all operations between the Web Service requestor and the Web Service provider.

So far, several core technology standards have been mentioned, such as SOAP, WSDL and UDDI. The sections following will briefly introduce these core technology standards in the Web Service field.

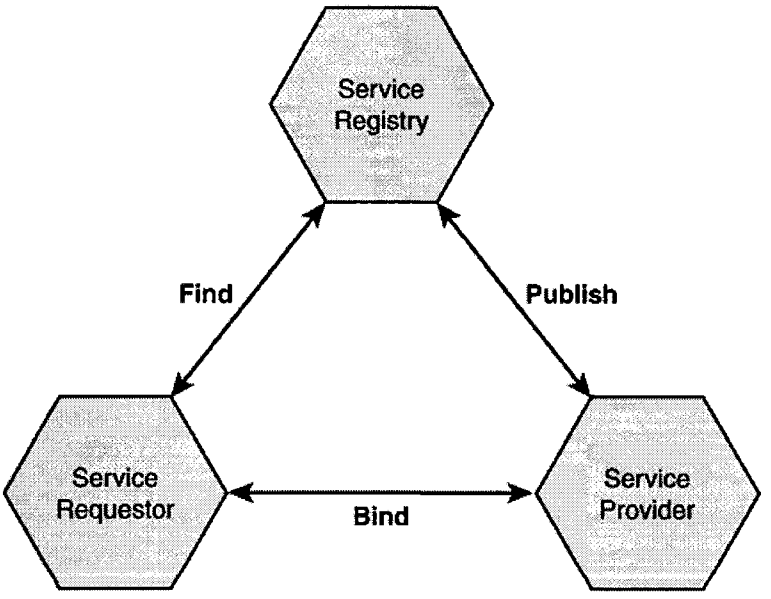

Figure 2-1 Roles and Operations in Service-Oriented Architecture [35]

2.3.2 UDDI

The UDDI specification defines messages, application programming interfaces and data structures for building distributed registries of Web Services and storing the business and technical information associated with these Web Services. It is a platform-independent, XML-based registry implementation standard for businesses advertising themselves on the Internet, enabling businesses to discover each other and defining how the services or software applications interact over the Internet. Because UDDI is an open industry initiative and sponsored by the Organization for the Advancement of Structured Information Standards (OASIS), and currently there are more than 310 companies involved in the UDDI project [28], there are many existing open source implementations, such as Apache jUDDI [32] and IBM UDDI4j [43]. Section 2.4 will introduce the Web Service discovery mechanism in Web Service registry systems, focusing in particular on the UDDI standard. The Apache jUDDI was chosen as the practical implementation of Web Service registry, since it was a very popular and powerful existing open source implementation when this study began. Section 5.2 will introduce the jUDDI framework.

2.3.3 SOAP

SOAP is a lightweight protocol for exchanging XML-based messages over decentralized and distributed environments, normally using HTTP/HTTPS. It consists of three parts:

- An envelope: defining a framework for describing what is in a message and how to process it;
- A set of encoding rules: expressing instances of application-defined data types; and
- A convention: representing remote procedure calls and responses.

SOAP forms the foundation layer of the Web Service stack, providing a basic messaging framework on which more abstract layers can build. Figure 2-2 [37] shows how to use SOAP in combination with UDDI, HTTP and HTTP Extension Framework in Web Service.

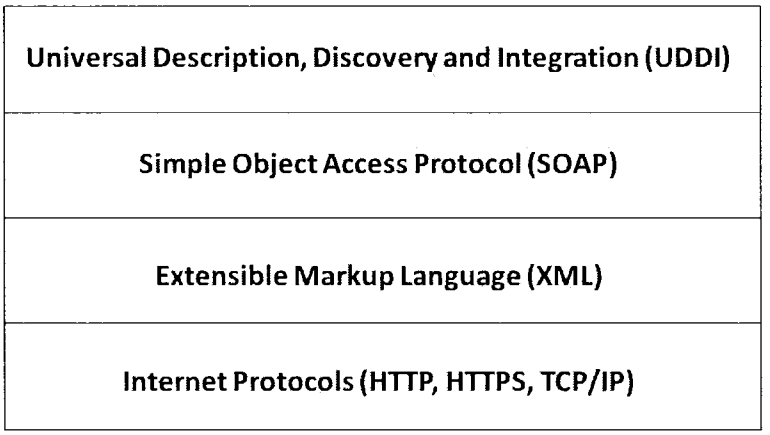

Figure 2-2 Layers of Technologies and Protocols Currently Used for Web Service [37]

2.3.4 WSDL

WSDL is an XML format that describes the interface definition for Web Services, details related to binding (network protocol and data encoding requirements), and the network location of the Web Service.

2.4 Web Service Discovery Mechanisms and Registry Systems

2.4.1 UDDI Registry Systems Architecture

As discussed earlier in this chapter, we choose the UDDI specification by OASIS as the practical specification of Web Service registry systems since it was the most popular specification when this research began.

UDDI is a web-based distributed directory that enables businesses to list themselves on the Internet and discover each other, similar to a traditional phone book's yellow and white pages. More specifically, UDDI uses Web Service and traditional Web protocol standards and technologies, such as HTTP, HTTPS, XML, and SOAP (see Figure 2-2) to define a Web Service discovery and publishing protocol. At its foundation, UDDI is a group of specifications that lets Web Service providers publish information about their Web Services and lets Web Service requestors search that information to find a Web Service and invoke it. This thesis focuses only on the discovery functionalities of UDDI that lets clients find Web Services; or, more accurately, a Web Service description that lets clients understand what the Web Service that they find do. Here one important basic concept has to be clarified. A client can only find the Web Service description, rather than the Web Service itself, in the UDDI Web Service registry, since a UDDI Web Service registry typically contains metadata for a Web Service embodied within a WSDL document [37]. A Web Service requestor can use the descriptions provided in the UDDI

Web Service registry to perform three types of searches [18] [6]:

- A *white-pages* search returns basic information such as name, address, contact, and unique identifiers about a company and its services.
- A *yellow-pages* topical search retrieves information according to industrial categorizations and taxonomies, such as the North American Industry Classification System (NAICS), the International Organization for Standardization (ISO), and the United Nations Standard Products and Services Code (UNSPSC) classification systems.
- A *green-pages* service search retrieves technical information about Web Services, as well as information describing how to execute these services.

2.4.2 UDDI Data Model

Under the covers, UDDI consists of an XML schema that defines UDDI's four core data structures—"Business Entity", "Business Service", "Binding Template", and "tModel" which can be stored within the Web Service registry [18], as well as a set of APIs that operate on those data structures. Registry entries and metadata are exposed as the following structures [37]:

- *Business Entity*: this structure encapsulates information describing one or more business services, including name, unique identifier, category, geographic location, and contact information.

- *Business Service*: this structure contains descriptive information about a group of related technical services, including the group name, description, and category information. A Business Service structure acts as a container for one or more Binding Templates structures.
- *Binding Template*: this structure contains information needed to invoke or bind to a specific business service. This information includes the service URI, routing and load-balancing facilities, and references to interface specifications contained in a corresponding tModel structure.
- *tModel*: this structure encapsulates information about interfaces and other technical concepts for a given service.

Figure 2-3 illustrates the relationship among the four core data structures. In addition, a description of the core data structures for a UDDI Web Service registry is presented in Figure 2-4.

As mentioned in Section 2.4.1, a Web Service requestor can use the descriptions provided in the UDDI Web Service registry to perform three types of searches: white-page search, yellow-page search and green-page search. White pages contain general contact information on a business itself, including a name, contact details, unique identifier numbers and the location of the business. The unique identifier numbers, like tax IDs, Data Universal Numbering System (D-U-N-S), or a European Article Numbering Systems (EAN) number, can be added to uniquely identify a business entity. White-page

search allows Web Service requestors to search for business entities according to general contact information stored in the Web Service registry. White page information is facilitated using the <*businessEntity*> element (see Figure 2-4).

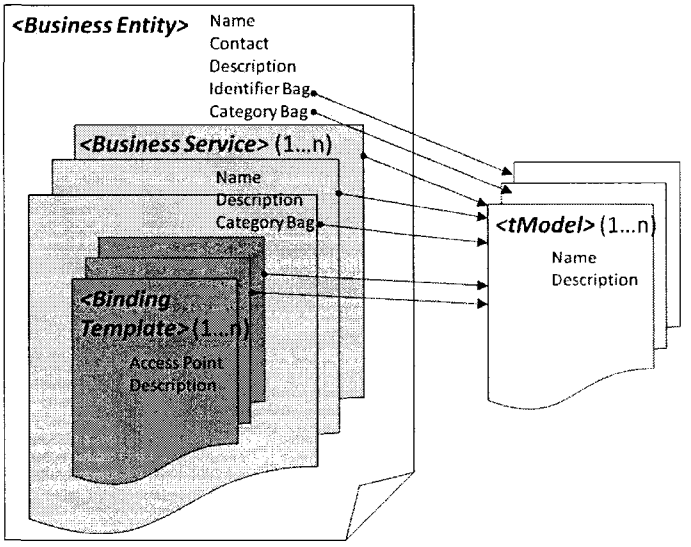

Figure 2-3 Relationships among UDDI Data Structures

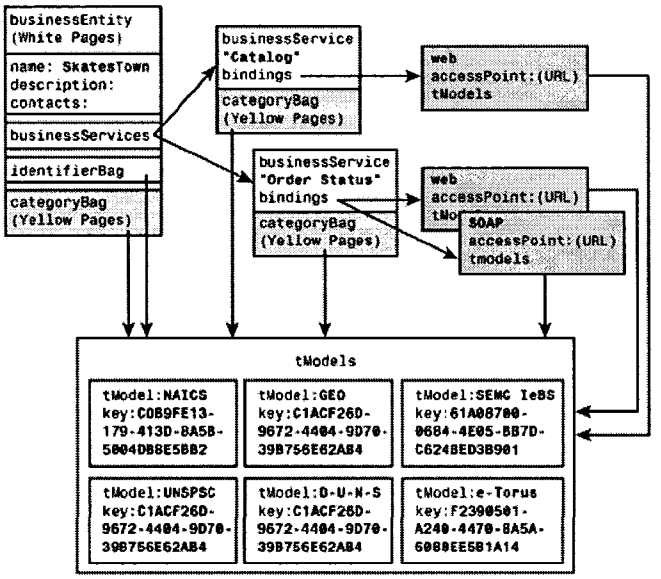

Figure 2-4 UDDI Core Data Structures [33]

Yellow pages contain categorized information about the types of services offered by a business entity. Categorization is done by assigning one or more taxonomies to a business

entity based on types of services provided by it. For example, the famous "Apple iTunes Stores" service might be categorized as an "Online Store" service, and at the same time be categorized as a "Music Store" service. Yellow page information is also associated with the <*businessEntity*> element. One famous Music taxonomy is the United Nations Standard Products and Services Code (UNSPSC) which provides an open, global multi-sector standard for efficient, accurate classification of products and services.

Green pages contain detailed technical information about services offered by a business entity, like how to invoke the offered services, which unique identifier to use and which taxonomy is used to categorize business entity and the services offered by it. This information includes services location, the category to which this service belongs, and the specification for the services in the green pages. Green page information is facilitated by the <*businessService*>, <*bindingTemplate*> and <*tModel*> elements.

2.4.3 How Does UDDI Work?

Figure 2-5 [8] illustrates how the UDDI Web Service registry works with the Web Service requestor and the Web Service provider by using SOAP and WSDL protocols. A Web Service provider first publishes the Web Service description information in the UDDI Web Service registry, using WSDL. The Web Service requestor uses an inquiry API, provided by the UDDI Web Service registry, and searches for a description of the desired Web Service in it. The description of the desired Web Service will be delivered to

the Web Service requestor via the SOAP protocol. When the Web Service requestor gets the description of the Web Service, it connects to the Web Service provider over the SOAP protocol to invoke the Web Service. Finally, the Web Service is delivered via the SOAP protocol from its provider to the requestor. Because the purpose of this study is to improve the performance of Web Service discovery, only step 2 and step 3 in Figure 2-5 are involved in this study. To invoke specific Web Service using information from a UDDI registry, a caller typically follows these steps:

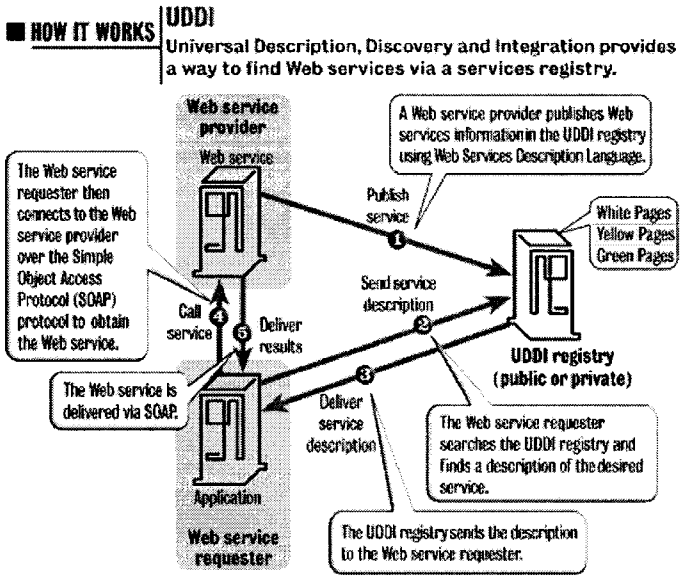

Figure 2-5 How UDDI Works [8]

- Locates the <*businessEntity*> information registered by the business providing the Web Service.
- Discovers additional details about the Web Service by accessing the <*businessService*> structure contained within the <*businessEntity*> structure. From there, the caller selects the appropriate <*bindingTemplate*> to use.

- Uses the technical information contained in the <*tModel*> corresponding to the selected <*bindingTemplate*> to build the client that will access the Web Service.

2.4.4 UDDI Publication API

The UDDI publication API consists of operations for creating, reading, updating, and deleting the information exposed by the UDDI structures discussed earlier. A caller can use these operations to register and/or modify any number of businesses or services. Because this thesis focuses only on the discovery of Web Services rather than the publication of Web Services, further details on the UDDI publication API are not included.

2.4.5 UDDI Inquiry API

The UDDI inquiry API consists of operations that enable one to browse a registry and to traverse a registry in order to obtain information about specific businesses and services. Table 2-1 lists the inquiry API methods that a UDDI Web Service registry must support [39].

2.5 Summary

A brief review of the background related to this thesis was described in this chapter. The review focused on related aspects of Web Service discovery in order to understand the

approach proposed in this thesis. First we reviewed the definition of Web Service and the basic architecture of SOA. Then, the main components of Web Service, UDDI, SOAP and WSDL were introduced.

Because we will propose an approach of integrating caching and pre-fetching to improve the performance of Web Service discovery in this thesis, a survey for various existing caching and pre-fetching techniques is presented in the following chapter.

Method	Description
find_binding	Used to locate bindings within or across one or more registered businessServices.
Find_business	Used to locate information about one or more businesses.
Find_relatedBusinesses	Used to locate information about businessEntity registrations that are related to a specific business entity whose key is passed in an inquiry.
Find_service	Used to locate specific services within registered business entities.
Find_tModel	Used to locate one or more tModel information structures.
Get_bindingDetail	Used to get bindingTemplate information suitable for making service requests.
Get_businessDetail	Used to get the businessEntity information for one or more businesses or organizations.
Get_businessDetailExt	Used to get extended businessEntity information.
Get_serviceDetail	Used to get full details for a given set of registered businessService data.
Find_binding	Used to locate bindings within or across one or more registered businessServices.

Table 2-1 Required Inquiry API Methods in UDDI Specification [37] [39]

Chapter 3: Related Works and Proposed Approach

3.1 Introduction

This chapter is a literature survey of various existing caching and pre-fetching techniques used for improving Web application performance. The chapter is organized as follows. Section 3.2 discusses work that has been done on caching in the traditional Web application realm. Section 3.3 extends the discussion to the Web Service domain while Section 3.4 extends it to the Web Service registry systems field. Section 3.5 presents the problems in existing Web Service registry systems and briefly introduces the proposed caching and pre-fetching mechanisms to improve current Web Service registry systems. Finally Section 3.6 discusses the previous work that has been done on the type of data that should be pre-fetched, and where the pre-fetching should take place to improve performance in the context of traditional Web applications, in the context of Web Service and the existing Web Service registry systems.

This chapter starts with a discussion of the existing caching and pre-fetching techniques used in traditional Web applications, which can be extended to the domain of Web Service. Because Web Service protocols build on existing traditional Web protocols, such as HTTP and HTTPS, it is advantageous to reuse as many existing traditional Web caching and pre-fetching techniques as possible in the context of Web Service registries. Thereafter, the focus will be on the specific caching and pre-fetching techniques used in

the context of Web Services. Finally, caching and pre-fetching will be discussed in the context of Web Service registries.

3.2 Caching in Traditional Web Applications

In traditional web applications, caching of web documents (e.g., HTML pages) or web components (e.g., images or audio, video files) is performed in order to reduce bandwidth usage, server load, and perceived delay for users. A web cache stores copies of documents or components passing through it; subsequent requests may be satisfied from the cache if certain conditions are met.

3.2.1 What Data Should be Cached

Holmedahl et al. [11] proposes cooperative caching of dynamic content on a distributed Web server. He suggests caching the results of requests for dynamic content executions and storing the cached data on the server side rather than the client side. His approach maintains caching information (meta-data) in a cache information table (in main memory) and stores cached data in a hard drive. The cache information table (meta-data) is shared among a cluster of cooperative workstations and it contains the basic information about the cached data, such as the requests for dynamic content, the results of the request execution, a reference to the cached data and the frequency of the occurrence of this request. Other papers have proposed hashing this information on the cached data to create

a unique identification for a request. Thereafter, this identification can be used to check whether the arriving request is a repeat or not.

3.2.2 Where Caching Should Take Place

Based on the locations in which web objects are cached, Tang et al. [25] classifies general web caching technology into three categories: (1) client's browser caching, (2) client-side proxy caching, and (3) server-side proxy caching. These three categories have advantages and disadvantages, which are listed in Table 3-1. The scenario column in Table 3-1 presents the situation in which a particular type of caching is effective.

Typical methods for improving performance are file caching with proxies. Research shows that for file (static Web page) fetches on general Web environments, the network is responsible for a significant portion of the latency of requests [11]. Web proxy caching should be very effective because it reduces the network bottleneck by keeping the cached copies of Web pages closer to the client, so that the response time is as short as possible. Purely static Web pages can easily be kept closer to the client. However, for some Web pages that include many dynamic web objects, processor utilization rather than network bandwidth is the bottleneck. That is, the processing time required for generating these dynamic components is responsible for a significant portion of the response time for a request. Holmedahl et al. [11] mentions that requests for dynamic content typically take at least an order of magnitude longer to complete than a file fetch. He suggests that

caching the results of requests for dynamic content execution on the server side rather than in client side would yield a considerable improvement in performance while the static content is cached closer to the client.

Category	Scenario	Location	Advantages	Disadvantages
Client's browser caching	User accesses the same object more than once in a short time	Client's local disk	Browser can fetch the Web object directly from the local disk to eliminate repeated network latency and to decrease Web server load;	Users are likely to access many sites, each for a short period of time, so that the hit ratios of per-user caches tend to be low
Client-side proxy caching	Many users access the same Web objects	Proxy near the client	1) Avoid repeated round-trip delays between the clients and the origin Web servers; 2) Several proxies can share their cache content with one another by enabling the cooperative caching architectures; 3) Each participating proxy can seek a remote cache hit from another participating proxy's cache so that the overall hit ratio can be further improved	Cache size needs to be much larger than a single client's browser cache
Server-side proxy caching	Web servers getting overloaded	Proxy near the servers	Distributes/routes the users' requests to the proper server-side proxies in order to reduce the Web server's load and to shorten the user-perceived response time	More load on communication networks in comparison to the two other categories

Table 3-1 Comparison of Three Caching Policies

3.3 Caching in Web Services

When client applications frequently access the same information from the Web Service provider, the server's workload increases and, consequently, application response time may deteriorate. Moreover, when hundreds of client applications access the Web Service

provider simultaneously, the performance of the Web Service can degrade severely. Therefore proper Web Service caching leads to the reduction of communication overhead and the workload of the Web Service providers.

Moreover, in Web Services, the requests and responses are both SOAP XML messages, and parsing XML messages has a very high overhead. If XML messages are directly stored as cached values on a cache, then the parsing should be performed again when the next hit occurs. This is a difference from traditional Web applications.

3.3.1 What Data Should be Cached

We know that requests to and from a Web Service are both XML messages on the wire. A request application object will first be serialized to a corresponding XML message, and then sent out. A response XML message from the server is parsed by an XML parser that creates a post-parsing representation from this message. If the XML parser is an XML Document Object Model (DOM) parser, a DOM tree object is created as the post-parsing representation from the XML message. If the parser is a Simple API for XML (SAX) parser, it reads the XML documents and notifies the deserializer of the SAX events sequentially. The deserializer constructs the application objects from the DOM tree object or SAX event sequence.

A typical Web Service cache creates a pair consisting of a "cache key" and "cache value," which will be stored as a cache entry in the cache table. Cache keys are created

from SOAP requests while the SOAP responses are stored as cache values. For a later identical SOAP request, if the same cache key is generated, the cache table is searched for the cache key, and the cached SOAP response value is returned to the requestor. Because the SOAP responses from Web Service are XML messages, the XML SOAP responses themselves can be directly stored as the cache value.

However, Takase and Tatsubori [24] indicate the parsing and deserialization of XML messages has a very high overhead. Therefore they propose an efficient Web Service response caching technique by using an effective data representation for storing messages in client side. There are three optimization methods to improve the performance of caches in client-side middleware: (1) caching post-parsing representations such as DOM objects or SAX events sequences as cache values instead of storing the XML message itself. The benefit of using this optimization is that XML parsing can be avoided again on the client-side machine when the cached XML data is retrieved from the cache; (2) caching application-specific objects as cache values that are exchanged between the client and the server applications; (3) caching only the reference value of the read-only object instead of caching the object itself. These optimizations reduce the overhead of XML processing or application object copying on client-side machines or client-side proxy machines.

Takase et al. [23] suggests a Web Service cache architecture based on XML. The architecture includes a cache table that manages cached entities. Each entry in the cache table includes a request and a response message and the corresponding hash values of the

request and response messages, as in Table 3-2 [23]. The proxy cache utilizes the request hash value to retrieve entries from the cache table more quickly. The response hash is used to check if the response message is identical to the one stored by the proxy cache.

Contents in cached entity	Note
Request URL	URL for a request
Response	XML document for a response
Request-hash	Hash value (20 bytes) for request
Response-hash	Hash value (20 bytes) for response

Table 3-2 Cache Table [23]

The proposed client-side proxy caching architecture by Ramasubramanian and Terry [19] acts as a simple tunnel for all HTTP messages (packets) that are not SOAP messages. However, the proxy cache will store some of the SOAP messages received in response to a SOAP request. If the requested Web Services are unavailable for some reason, as for the SOAP message request_01 shown in Figure 3-1, the cached SOAP response message, reponse_01, in the proxy cache will be returned to the client. If request_01 also changes the server data element, the copy of the data element in the cache will be changed and the SOAP message request_01 will be stored in a write-back queue to maintain the consistency of cached data (see Figure 3-1). When the expected Web Service is available again, the cached SOAP message request_01 will be sent to the destination (expected Web Service). Thereafter, the returned SOAP response message from the server will replace the old cached response_01 in the proxy cache associated with the SOAP

request_01 message. On the other hand, if the requested Web Service is available immediately, the SOAP message request_02 will not be sent directly to the destination. The Custom Cache Managers (CCM), which controls the behavior of the cache according to the annotations described in the WSDL document of each Web Service, will first search the response_02 in the cache and return it to the client from the cache. The request_02 will be sent to the destination only if CCM does not find the cached response_02 or it has expired.

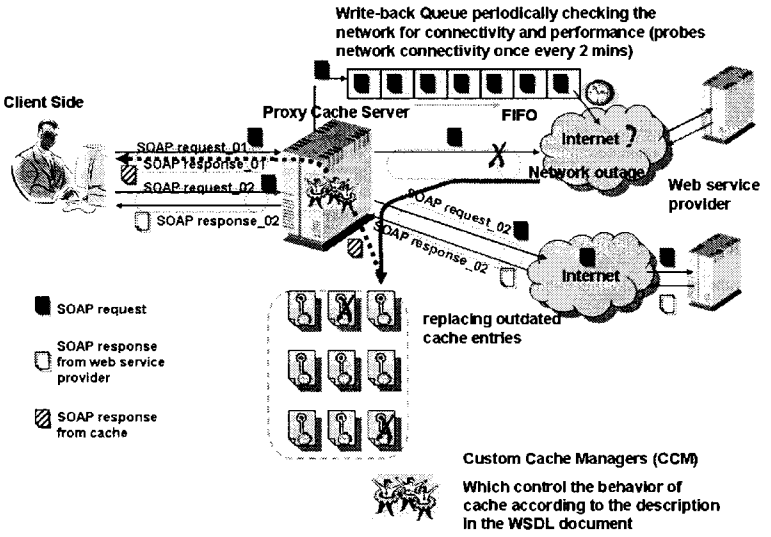

Figure 3-1 Caching SOAP Messages in Web Service

3.3.2 Where Caching Should Take Place

Azim and Hamid [3] describe how to create transparent, client-side caching for SOAP services using the "business delegate" and "cache management" design patterns. They propose that a client-side cache SOAP call will significantly improve the performance of Web Service, since a SOAP call's cost includes network latency and CPU cycles at the

SOAP server. However, the caching of a SOAP message in client local storage will have the same problem as a general Web cache. That is, the cache might become inconsistent with the original data source; changes made to the SOAP server after the data is recorded in the cache may not be seen by the client (read inconsistency), and changes made to the cache go un-reflected at the SOAP server (write inconsistency) [3].

Ramasubramanian and Terry [19] propose an architecture that provides a client-side proxy cache that mimics the behavior of a Web Service. The proposed client-side proxy cache architecture is completely transparent to both the client and server components of the Web Service. Figure 3-2 shows the architecture in which the proxy cache is near the client side. The Web Service cache resides in a HTTP proxy server on the client device. The proxy server recognizes a SOAP request from the presence of the *soap-action* header in the HTTP request message. All HTTP request messages from and to Web clients simply pass through this proxy server while the SOAP request messages are dealt with by the Custom Cache Managers (CCM). CCM controls the behavior of the cache according to the annotations described in the WSDL document of each Web Service. It deals with the received SOAP request messages in the two following ways: (1) Send the SOAP request messages to the destination (expected Web Service provider) if CCM cannot find the cached response message in the cache; or (2) Retrieve the cached response message if CCM can find the cached response message and send it to the client if the cached data has not expired. If it has expired, the SOAP request will be sent to the destination Web

Service. We have already summarized how the architecture proposed in [19] works in Section 3.3.1.

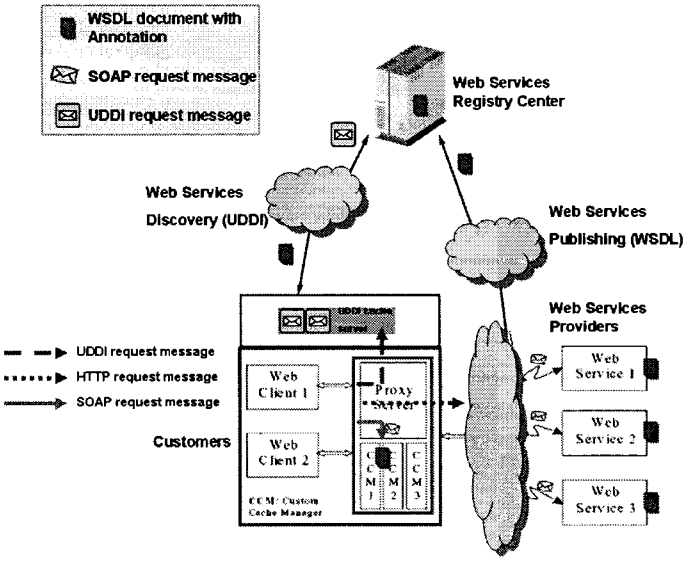

Figure 3-2 Client-side Proxy Cache Architecture [19]

Seltzsam et al. [21] propose another client-side caching technique called the semantic caching strategy for Web Service. They developed an XML-based declarative language to annotate WSDL documents with information about the caching-relevant semantics of requests and responses. Using this information, their semantic cache answers requests based on the responses of similar previously executed requests. They named the semantic caching approach Semantic SOAP Protocol Level Cache (SSPLC), which is closer to the client-end (see Figure 3-3). The improvement in performance is due to the semantic caching of responses from the Web Service SOAP request/response message exchange. Clients do not always need to access the origin Web Service providers directly anymore; on the contrary, clients only access the SSPLC and will quickly get the SOAP response

message, as long as the SOAP requests can be answered based on previously cached data, which is cached near the client side; the origin server hosting the Web Service is not involved. Therefore, the workload at the origin Web Service providers is reduced, as is bandwidth consumption, so that the response time is decreased. The main advantage and difference from the approach proposed by Ramasubramanian and Terry [19] is that SSPLC reuses the responses to prior SOAP requests to answer similar SOAP requests, not just exactly the same SOAP requests, whereas Ramasubramanian and Terry's approach [19] requires exactly the same SOAP requests for a cache hit to occur.

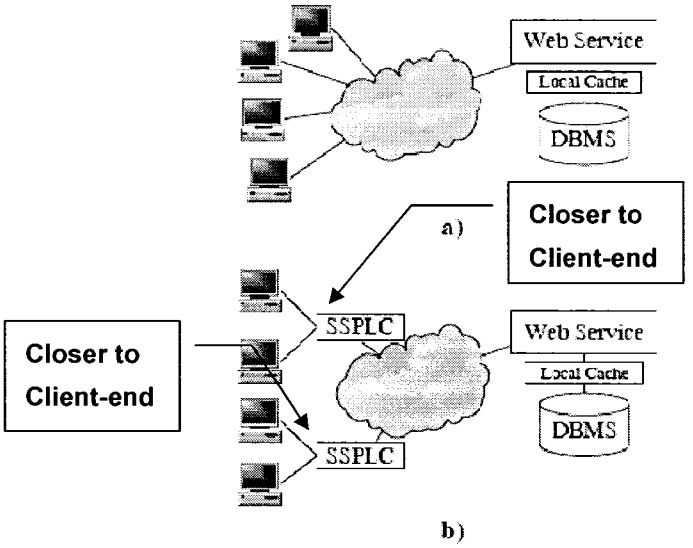

Figure 3-3 Web Service Architecture (a) Without Distributed Caching and (b) With Distributed Caching [21]

3.4 Caching in Web Service Registry Systems

3.4.1 What Data Should Be Cached

Kumar et al. [14] propose a model that provides a middleware registry cache to store a valid subset of the UDDI Web Service registry data with QoS specification without any modification to the existing UDDI standard. The basic objective is that only operational Web Services with QoS parameters will be cached into the proposed UDDI Web Service registry cache repository. As a result, the proposed UDDI registry cache contains only a valid subset of normal UDDI registry data so that it can easily be retrieved without the complete normal UDDI registry data having to be searched. The reason is that there are many incomplete or dead entries (registration data for Web Service) in standard UDDI registry centers, whereas the proposed UDDI registry cache maintains only valid UDDI Web Service registration data. In this case, the normal UDDI registration data for Web Service, such as the WSDL document of the Web Service, URIs of the Web Service and expiration date, as well as the QoS parameters of Web Service and validation information, need to be stored in the UDDI registry cache. The basic idea in the proposed UDDI registry cache architecture is to add a replica of valid information for the UDDI registries into a QoS registry Cache. This will reduce the number of records to be searched, and will provide the desired QoS in the retrieved Web Service.

A detailed discussion of UDDI caching is provided by Gert Sylvest [7]. The author

specifies what UDDI entity keys should be cached for future lookups and stored in client local storages. In order to understand Sylvest's idea, we need to introduce two definitions. Sylvest calls the concrete Web Service "Endpoints" and the abstract Web Service "Service." An endpoint such as "PayPal E-Invoice Service" indicates a concrete Web Service that must be an instance of a certain abstract Web Service, e.g. the "E-Invoice Service." The "Service," e.g. the "E-Invoice Service," is abstracted from a set of concrete endpoints that have the same or similar attributes. In order to locate a specific endpoint or search a certain kind of service where a consumer can send business messages, e.g. SOAP messages, a lookup may be performed in UDDI to locate the endpoint or service, based on the *EAN location number* used for looking up an endpoint and a service definition *tModel identifier* for discovery of a certain service, respectively. On a UDDI registry center, if a lookup were performed for every business message, the UDDI registry would incur a very heavy burden. Sylvest proposed that the client should cache the following UDDI registration data on their first lookup of an endpoint for subsequent lookups (see Table 3-3).

3.4.2 Where Caching Should Take Place

The papers discussed in this section were also discussed in the content of what data should be cached in Section 3.4.1. As mentioned in Section 3.4.1, Kumar et al. [14] propose a model that provides a middleware registry cache to store a valid subset of the

UDDI registration data with QoS specification without any modification to the existing UDDI standard. They propose that the QoS-based registry cache should be located close to the normal UDDI Web Service registry centers, because the UDDI registry cache is indeed a subset of the normal UDDI registry. Only the validated operational Web Service with QoS parameters will be stored in the UDDI registry cache. This will result in searching a fewer number of records that have already been validated. Searching fewer records in the registry cache is more efficient than searching the complete UDDI registry, which contains many non-operational services. A more detailed discussion of the paper was presented in Section 3.4.1.

Cached UDDI Registration data	Comment
bindingKey of the binding template representing the endpoint (concrete instance of Web Service).	The client first looks up the expected endpoint. If successful, the ***bindingKey*** is cached for subsequent discoveries. With the cached ***bindingKey***, an endpoint can be looked up by a simple call to the UDDI inquiry API operation ***get_bindingDetail*** rather than sending a request to the UDDI registry again.
tModelKey of tModel holding the instance identifier's custom attributes.	If the client needs to recheck the custom attributes, this can be done with a simple call to the UDDI inquiry API operation ***get_tModelDetail*** rather than sending a request to the UDDI registry.
endpointExpirationDate	When a client first looks up an endpoint, it should cache the ***webserviceExpirationDate***. In general, a client should default to the cached endpoint address as long as the endpoint does not fail on the network or the ***endpointExpirationDate*** has not passed.

Table 3-3 Cached UDDI Registration Data

Sylvest [7] suggests that the clients (Web Service Requestors) should cache the UDDI

registration data in their first lookup, because UDDI represents an indirection mechanism allowing clients to discover the expected Web Service. If the processing of discovery for each business message was always dependent on the UDDI Web Service registry, then a heavy burden would be put on it. Thus, Sylvest suggests that clients should cache the UDDI registration data in local storage. He also specifies what data should be cached in client-side storage to improve system performance.

3.5 Proposed Approach for Caching

3.5.1 What Data Should be Cached

The caching component proposed in this thesis caches the UDDI inquiry request object and UDDI response object associated to it rather than caching the UDDI entity keys alone. As described in Section 2.4.2, each type of UDDI entity is assigned a unique key by UDDI registry systems. However, the UDDI entity does not only contain a unique key but also many useful elements. For example, the *BusinessService* element is the root element for describing a logical business service, such as a stock quote service, or a securities buying service. It contains a unique service key that identifies the business service itself and a unique business key that identifies the business entity that provides this business service. In addition, the *BusinessService* element contains the *BindingTemplates* element that describes different implementation details and bindings

for the same logical service. The benefit of caching the entity key is that the client can simply get the desired *BusinessService* element by directly sending a UDDI inquiry request *get_serviceDetail* with a cached service key rather than sending a request *find_service* to the UDDI registry. Caching the entity key can reduce the amount of UDDI inquiry requests sent to the UDDI registry, as a result the workload of the UDDI registry and internet traffic will be abated. However the drawback of caching the entity key alone is that the client still needs to send a UDDI inquiry request, *get_serviceDetail* with unique key, to the UDDI registry. The approach proposed in the thesis suggests caching the UDDI entity object; that is, caching the *BusinessService* element itself rather than caching its key only. As a result, the UDDI inquiry request *find_service* and *get_serviceDetail* can only be sent to UDDI registry once. All the following UDDI inquiry requests *find_service* and *get_serviceDetail* can simply retrieve the cached *BusinessService* element from the location that cached the data storage. If the location of the cached data storage is very close to the client, then the proposed caching approach in Web Service registry systems will not only reduce the workload of the UDDI registry but also the internet traffic. This will be discussed in more detail in Section 4.2.1.

3.5.2 Where Caching Should Take Place

The proposed caching approach adopts some existing caching techniques from both traditional Web applications and Web Service applications for Web Service registry

systems. In the proposed approach, a caching proxy will be located as close to the client as possible and cache the Web Service inquiry request object and the response object associated with it. This should reduce the workload of the Web Service registry and response time for UDDI inquiry request. A more detailed discussion of the location of where caching takes place in the proposed caching approach for Web Service registry systems will be presented in Section 4.2.2.

3.6 Pre-fetching

3.6.1 In Traditional Web Applications

For general Web applications, a proxy cache is used to reduce response time. Appropriate cache management policies are required to achieve this objective. However, the additional cost (time and resource) of looking up the cached data in the proxy disk can counteract the improvement in performance. Abhari, Dandamudi and Majumdar [1] suggest a technique to improve storage management in Web proxy caches. Pre-fetching embedded components of a Web object from the server to a proxy is also investigated at a different level in [2]. In [1] they propose two techniques to improve the performance of storage management systems in a proxy cache. First, they suggest pre-fetching all the embedded components associated with the requested Web page from the proxy disk to main memory. This was shown to reduce hit latency by up to 50%, since all the related

files when accessed, were retrieved from memory rather than the proxy disk. Hit latency is the delay incurred when a hit document is fetched from a proxy disk to proxy main memory. Rousskov and Soloviev [20] report that the disk delay of a proxy is responsible for about 30% of the total hit response time. Research in [1] is directed at reducing the disk delay. Second, Abhari et al. [1] also propose storing all constituent embedded components belonging to a given Web document in contiguous proxy disk blocks, to reduce the disk accessing time (see Figure 3-4). This was observed to reduce hit latency by an additional 5% [1].

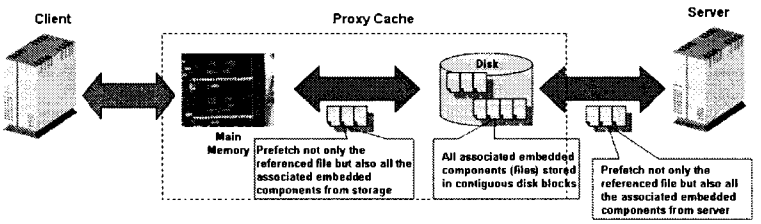

Figure 3-4 Pre-fetching Strategies

Several other pre-fetching policies have been researched and proposed for general Web caches. Gwertzman [9] proposes the server-initiated push-caching approach. Servers record their access histories in log files and get statistical data by analyzing the log files. He uses this statistical information to decide where to place caches of their data, as well as when and what data need to be pre-fetched.

Markatos and Chronaki [15] propose a Top 10 approach (see Figure 3-5). The approach requires cooperation between the client-end and the server-end. The server is responsible for analyzing the server access log files for pre-fetching from the Web server

and for preparing the Top 10 list. This Top 10 list contains the ten most frequently referenced Web pages. A pre-fetching agent in the client site is responsible for analyzing the client access log file and creating the pre-fetching profile for that client. The client profile is the list of servers for which pre-fetching should be activated. Thereafter, the system can make a decision about where and what data should be pre-fetched, according to the client profile and the server access statistics. A detailed discussion of the strategy is presented in [15].

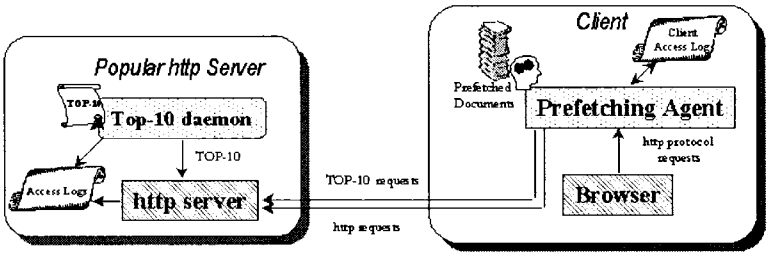

Figure 3-5 Top 10 Pre-fetching [15]

Bestavros [4] proposes a speculative mechanism to pre-fetch documents. A client's request for a document is serviced by sending not only the requested files but also all related documents. The server speculates which related documents (or pointers) will be requested by the client in the near future. This speculation is based on statistical information that the server maintains for each document it serves. The author defines a set of parameters that can be used to "fine-tune the level of speculation performed by the server" [4].

Most research focuses on either Web caching strategies or Web pre-fetching strategies. Tang, Chang and Chen [25] and Abhari et al. [2] realize that these two

techniques can complement each other when they are integrated, because Web caching exploits temporal locality, whereas Web pre-fetching utilizes the spatial locality of Web objects. However, Tang et al. points out that an effective design is required to successfully integrate these two techniques. For instance, if a Web server sends all possible pre-fetching information to the proxy without any control, the proxy will pre-fetch every implied object into its cache. A significant portion of the cache content will frequently be replaced because a proxy may concurrently serve many requests from multiple clients. As a result, the state of the cache content will become extremely unstable and the cache-hit ratio will rapidly decrease. On the other hand, if the pre-fetching rule is extremely strict, it will obviously cut down the benefit of Web pre-fetching.

3.6.2 In Web Service and Web Service Registry Systems

No literature about pre-fetching approaches in the Web Service application and Web Service discovery area was found. Valuable literature is available on existing traditional Web caching and pre-fetching approaches, which can be extended to the Web Service area. There is also a lot of work focusing on using caching and pre-fetching approaches to improve the performance of the Web Service invocation serving a requestor. However, only a very small amount of literature deals with improving the performance of Web Service discovery. This thesis will focus on this domain. We will present the problems in

current Web Service registry systems and introduce the proposed caching and pre-fetching strategies in the Web Service discovery area in Chapter 4.

3.6.3 Proposed Pre-fetching Approach for Integrating Caching

Although Web caching is the acknowledged primary approach for improving the performance of the Web, the crucial bottleneck of using Web caching alone is the low cache hit ratio. Some studies have shown that the maximum hit ratio achievable by Web caching algorithms is just in range of 40% to 50% [1] [15]. One effective approach to break through the bottleneck is by pre-fetching related files into the cache with the data being cached.

In this thesis, a graph-based pre-fetching technique that can be integrated with a caching technique on Web Service registry has been devised to improve system performance. A caching proxy is located as close to the client as possible and caches the Web Service inquiry request object and the response object associated with it. The pre-fetching proxy is located as close to the Web Service registry server as possible and records the history of system behavior to build relationships among Web Service inquiry requests. The relationships are captured in a pre-fetching graph, which is dynamically built and updated in the pre-fetching proxy. Thereafter, the pre-fetching proxy predicts a cluster of additional Web Service inquiry requests based on the currently received original Web Service inquiry request and sends the predicted requests to the Web Service

registry server, together with the original request. We will present the details in Chapter 4.

3.7 Summary

In this chapter, we reviewed existing caching and pre-fetching techniques for improving the performance of traditional Web applications, Web Service and Web Service registry systems, respectively. Because the Web Service inquiry requests and associated responses are all SOAP messages (see Figure 2-2), reusing as much of the existing caching and pre-fetching techniques in traditional Web application and Web Service fields as possible in the context of Web Service registry systems to improve the performance of Web Service discovery is attractive. This is the primary reason that this chapter focused on reviewing existing Web page and Web Service caching and pre-fetching approaches that can be extended to Web Service registry systems.

Chapter 4: System Design

4.1 Introduction

4.1.1 Problems with Existing Web Service Registry Systems

In addition to the usual problems encountered in general Web applications, such as latency in serving Web requests and the impact of traffic on network bandwidth, there are several additional problems exclusive to the current Web Service registry systems. An increasing number of registered Web Services can greatly increase the workload of a Web Service registry server. Moreover, the high cost of XML parsing and schema validation will have a significant impact on the performance of Web Service discovery. All these issues associated with the current Web Service registry systems can result in long delay in Web Service discovery. Proper caching and pre-fetching techniques in Web Service registry systems can reduce the latency of discovery of a specific and desired registered Web Service, and cut down the workload of a Web Service registry server.

Section 4.1.2 briefly introduces the high level structure of the proposed caching and pre-fetching approach in Web Service registry systems. Both the caching and pre-fetching approaches can improve the performance of Web Service discovery.

4.1.2 High Level Structure of the Proposed Caching and Pre-fetching System

The proposed system includes two components: a caching proxy and a pre-fetching proxy (see Figure 4-1).The proposed caching approach adopts some existing caching techniques from both traditional Web applications and Web Service applications for Web Service registry systems. Because the Web Service inquiry requests and responses during Web Service discovery consist of SOAP messages (see Figure 2-2), reusing the existing traditional Web caching techniques as possible in the context of Web Service registry systems caching is attractive. A caching proxy, shown in Figure 4-1, will be located as close to the client as possible and will cache the Web Service inquiry request object and the response object associated with it.

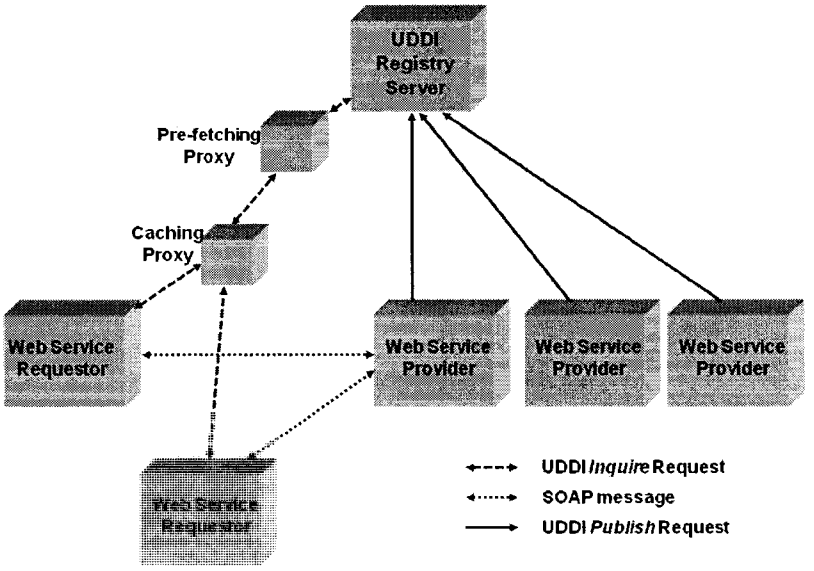

Figure 4-1 Structure of Prototype for Integrating of Caching and Pre-fetching in Web Service Registry Systems

The proposed pre-fetching proxy, shown in Figure 4-1, will be located as close to the

server (Web Service registry server) as possible and records the history of system behavior, in order to build relationships among the Web Service inquiry requests. Thereafter, the pre-fetching proxy can try to guess the subsequent inquiry requests that will be generated by the client and pre-fetch a cluster of such possible subsequent Web Service inquiry requests associated with the original Web Service inquiry request and send all of them to the Web Service registry server. In addition, the original Web Service inquiry request will also be sent to the Web Service registry. A detailed description of the system designs for the caching and pre-fetching components are presented in Section 4.2 and Section 4.3 respectively.

4.1.3 Prototype for Integrating Caching and Pre-fetching in Web Service Registry Systems

As mentioned above, in Section 3.6.3, although Web caching is a well-known primary approach for improving the performance of any Web application, the crucial bottleneck with using Web caching alone is the low cache hit ratio. Because a Web Service application is a kind of Web application, it is likely that Web Services will be subjected to similar problems. One effective approach to breaking through the bottleneck is pre-fetching related files with the cached data into the cache. Integration of pre-fetching techniques with caching in Web Service discovery systems may be used to further increase the cache hit rate by anticipating and pre-fetching future client requests.

Figure 4-1 indicates the client-proxy-server framework structure of this proposed approach. This illustration is a more practical implementation of Figure 2-1; therefore, it also contains three core components: the *Web Service requestors*, *Web Service providers* and *Web Service registry*. Moreover, the proposed approach includes two extra components, the *caching proxy* and *pre-fetching proxy*, which were introduced in Sections 4.1.2. A more detailed explanation with regard to caching and the pre-fetching proxy will be given in Sections 4.2 and 4.3, respectively. We have implemented the Web Service registry systems by using OASIS's UDDI v2.0 specification in this research. From now on, the term "UDDI registry" or "UDDI server" will indicate "Web Service registry systems" or "Web Service registry server," respectively. The term "UDDI" will indicate OASIS's UDDI v2.0 specification in particular.

4.2 Caching Algorithm in Web Service Discovery Systems

Although the proposed caching approach in this thesis reuses traditional Web cache techniques in the context of Web Service discovery systems, discussion is needed on the details of the caching algorithm in Web Service discovery systems. As with other Web caching algorithms, three core issues remain important: what data should be cached, where the caches should be located and how the cached data should be managed. Section 4.2.1 focuses on what data should be cached in Web Service discovery systems, Section 4.2.2 focuses on where the caching should occur, while Section 4.2.3 provides a more

detailed discussion concerning the management of cached data in Web Service discovery systems.

4.2.1 What Data Should be Cached

As briefly discussed in Section 3.5.1, the proposed caching algorithm should cache the UDDI inquiry request object and the UDDI response object associated with it, rather than caching the UDDI entity keys alone. As previously described in Section 2.4.2, each type of UDDI entity is assigned a unique key by the UDDI registry server. However, the UDDI entity not only contains a unique key, but also several useful elements. Although Sylvest [7] suggests UDDI entity keys should be cached for future lookups and stored in the local client storages, the drawback of caching the entity key alone is that the client still needs to send a UDDI inquiry request, e.g. *get_serviceDetail* with a cached *BusinessService* key, to the UDDI registry server. The only benefit of Sylvest's suggestion should be that it saves on the time for finding the satisfactory entity since the entity key has been cached in the local client storage.

The caching system proposed in this thesis will cache all the UDDI entity objects involved in the entire discovery process, including the UDDI inquiry request entity object and the response entity object associated with it. The UDDI inquiry request entity object is cached as a "key," while the UDDI response entity object associated with the UDDI inquiry request entity object is cached as a "value" in the object cache systems. We

implement the cache systems in Web Service discovery systems using an Apache open-source object cache project—Java Caching System (JCS) [41]. A "key-value" pair of objects, or "elements," can be put into JCS and referenced via a "key," much like a hash table.

Figure 4-2 illustrates the structure mentioned earlier. In this illustration, *find_service* is a UDDI inquiry request entity object with several attributes, e.g. authInfo, name and categoryBay. The *serviceList* is the UDDI response entity object associated with the *find_service* entity. Eventually the *find_service* and *serviceList* are cached as a "key-value" pair in JCS. Thereafter, when the same UDDI inquiry request (key) is received by the caching proxy, the cached UDDI response (value) associated with the UDDI inquiry request (key) can be quickly retrieved via the key from the caching storage. If the UDDI inquiry request (key) received by the caching proxy is not found in JCS, the caching proxy simply forwards it to the pre-fetching proxy (see Figure 4-3) which will be discussed in detail in Section 4.3.1.

4.2.2 Where the Cache Should be Located

As briefly discussed in Section 3.5.2, in the proposed approach, a caching proxy will be located as close to the client as possible and cache the Web Service inquiry request object and the response object associated with it; see Figure 4-1. In this way, network latency will be minimized. The improvement in the performance of Web Service discovery is

provided by the shortening of the response time and reducing the workload of the Web Service registry server.

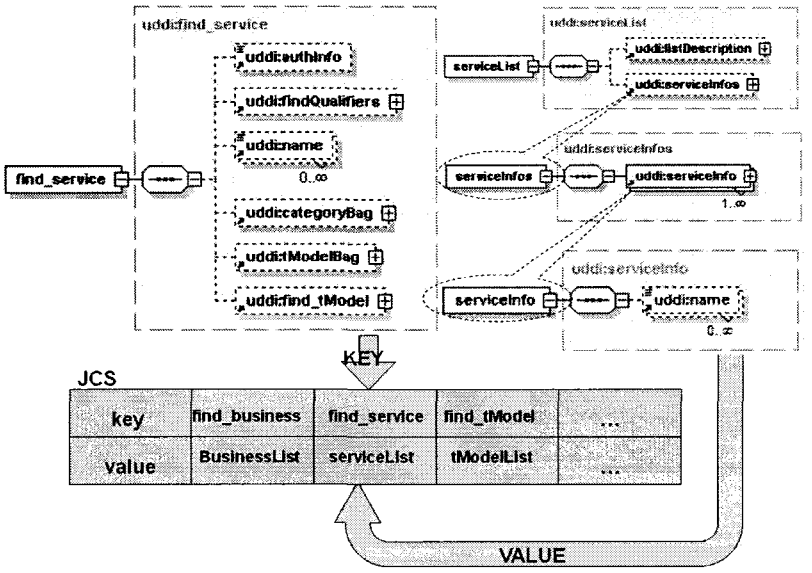

Figure 4-2 Cache UDDI entity objects as a "key-value" pair in JCS

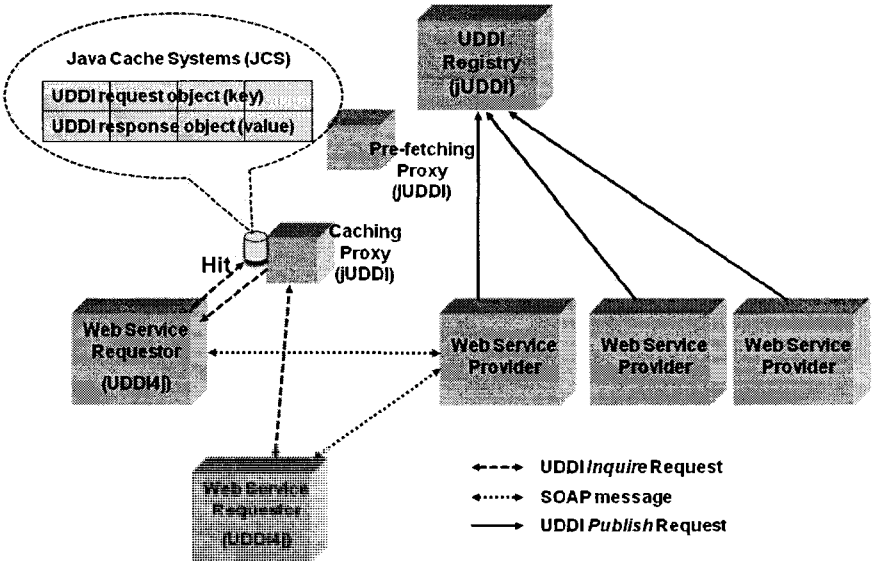

Figure 4-3 UDDI Inquiry Request Hit in the Caching Proxy

4.2.3 Managing the Cached Data

As mentioned earlier, the proposed cache approach is implemented in Web Service discovery systems using an Apache open-source object cache project—Java Caching System (JCS) [41].

JCS provides numerous additional features, e.g. memory management, data expiration (idle time and max life) and disk overflow (and defragmentation). In this thesis, we focus on the configuration of the memory management and disk management components of JCS. Currently, JCS provides four memory management options: (1) Least Recently Used (LRU) Memory Cache, (2) Linked Hash Map LRU (HMLRU) Memory Cache, (3) Most Recently Used (MRU) Memory Cache, and (4) Adaptive Replacement Cache (ARC) Memory Cache. The LRU Memory Cache is the memory management strategy that Apache currently recommends for JCS [41]. The LRU Memory Cache removes the least recently used items when the cache is full, restricting the number of items that can be stored in memory. JCS enforces configurable parameters such as Time-to-Live (TTL), the maximum number of objects in memory and the maximum idle time. Expired items can be completely cleaned from memory by the shrinker thread; otherwise, they will be removed at the next retrieval attempt or when capacity is reached. If a disk cache is configured for JCS, the items will be spooled to the disk when the memory capacity is reached. JCS provides several disk swap options; for example,

indexed disk and Oracle Berkeley DB Java Edition. The Indexed Disk Cache is the recommended disk cache managing means [41]. It maintains the cached data on the disk and the keys in memory for minimizing lookup times.

4.3 The Pre-fetching Algorithm in Web Service Discovery Systems

As mentioned in Section 3.6.2, although there is existing research focusing on discussion on caching and pre-fetching approaches to improve the performance of Web Service invocations, very few studies deal with improving the performance of Web Service discovery. Although we are aware that Web Service discovery performance can be improved using the proposed caching algorithm described in Section 4.2, the benefit of the caches may be limited. To further reduce retrieval latency, the workload of the Web Service registry server and the response time for the Web Service inquiry request, pre-fetching becomes an attractive solution. This section introduces a graph-based pre-fetching strategy for Web Service discovery. The proposed graph-based pre-fetching proxy tries to anticipate the subsequent Web Service inquiry requests that will be generated by the client and pre-fetches a cluster of such possible subsequent Web Service inquiry requests associated with the original Web Service inquiry request. These additional pre-fetch requests will be sent to the Web Service registry server, together with the original Web Service inquiry request. We will briefly present the functionalities of the

proposed pre-fetching proxy in Section 4.3.1, and show where the pre-fetching proxy should be placed in Section 4.3.2. The core of this section deals with how the pre-fetching proxy works, and this will be presented in Section 4.3.3 and Section 4.3.4.

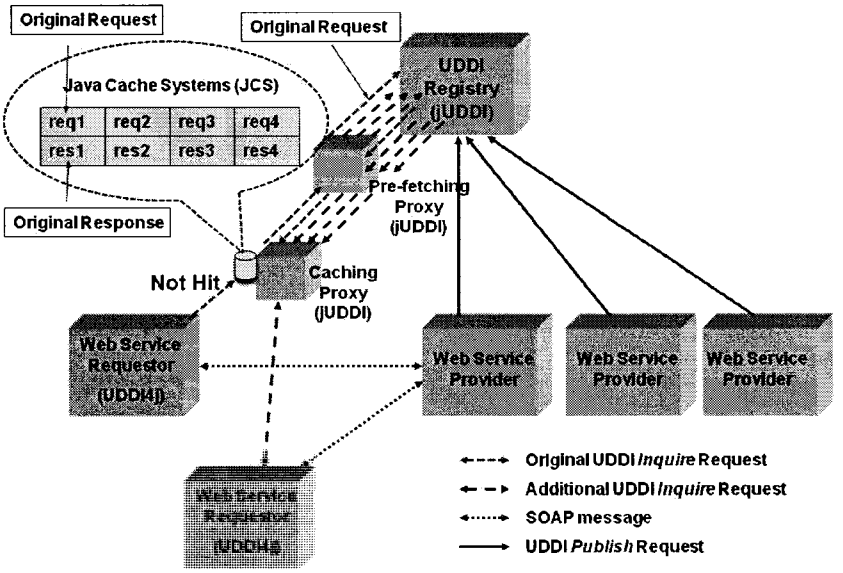

Figure 4-4 UDDI Inquiry Request Not Hit in Caching Proxy

4.3.1 The Functionalities of the Pre-fetching Proxy

We briefly introduced the prototype for the integration of caching and pre-fetching in Web Service discovery in Section 4.1.3. When the caching proxy receives a Web Service inquiry request object (key), it first checks whether the key already exists in the caching proxy. If this received Web Service inquiry request object (key) is found in the cache, it retrieves the cached Web Service response object (value) via the key from the caching storages (see Figure 4-3). Otherwise, the caching proxy simply forwards the Web Service inquiry request it has just received to the pre-fetching proxy (see Figure 4-4).

The pre-fetching proxy has two main functions: (1) to build and keep a pre-fetching graph up to date, according to the Web Service inquiry requests received in succession; (2) to generate a cluster of extra pre-fetch Web Service inquiry requests according to the original Web Service inquiry request currently received. The pre-fetched inquiry requests are generated using the pre-fetching graph. In this section, we briefly discuss the function of building and maintaining a pre-fetching graph; the function of pre-fetching Web Service inquiry requests will be discussed in Section 4.3.4.

The core component of the pre-fetching proxy, the pre-fetching graph, is a directed graph $G = (V, E)$, which records the dependencies between the consecutive Web Service inquiry requests received by the pre-fetching proxy.

A node $v \in V$ in G represents a Web Service inquiry request received. A node is uniquely identified by the name of the Web Service inquiry request object and the hash value of all the parameters of Web Service inquiry request. For example, node $n1 \in V$ in G represents a UDDI inquiry request *uddiInquiryRequestName1* with parameters *arg1, arg2, arg3...arg7*, while $n2 \in V$ in G represents UDDI inquiry request *uddiInquiryRequestName2* with parameters $arg1'$, $arg2'$, $arg3'...arg7'$. So node $n1$ is identical to node $n2$ if and only if *uddiInquiryRequestName1* equals *uddiInquiryRequestName2* and the value of hashing *arg1, arg2, arg3...arg7* equals the value of hashing $arg1'$, $arg2'$, $arg3'...arg7'$. The existence of a node in the pre-fetching graph G indicates that a Web Service inquiry request represented by this node has been

previously received by the pre-fetching proxy; therefore, an identical node is not re-added to pre-fetching graph G when the same inquiry request is presented again to the registry system.

A directed edge $(a, b) \in E$ in G represents the dependency between two consecutive received Web Service inquiry requests by the pre-fetching proxy. The direction of directed edge $(a, b) \in E$ indicates that the Web Service inquiry request $b \in V$ is received by the pre-fetching proxy immediately following the Web Service inquiry request $a \in V$. An edge $(a, b) \in E$ can be added to the pre-fetching graph G if and only if the time interval between the pre-fetching proxy receiving the two consecutive Web Service inquiry requests a and b is less than or equal to a pre-defined time interval threshold. Related inquiry requests resulting from a single business process are expected to be generated in quick succession. The time interval threshold should be chosen based on the inter-arrival time for such related inquiry requests. The value should be such that we capture the relationships among these related inquiry requests in the pre-fetching graph and exclude the unrelated requests that may be generated by a different business process for example. Thus, time interval threshold is a tuning parameter that should be based on the characteristics of the system workload. Moreover, each edge has a weight representing the total number of occurrences of the event that the Web Service inquiry request $b \in V$ has been received by pre-fetching proxy immediately following the Web Service inquiry request $a \in V$ and the time interval is less than or equal to a pre-defined

time interval threshold.

The left side of Figure 4-5 shows a simple example of a pre-fetching graph that is discussed next (the right side of the figure is discussed in Section 4.3.4). The pre-fetching graph "records" the history of system behaviors; for example, UDDI inquiry request *find_business(args)* is immediately followed by UDDI inquiry request *find_binding(args)* 15 times. The UDDI inquiry request *get_business(args)* is immediately followed by the UDDI inquiry request *find_binding(args)* once only.

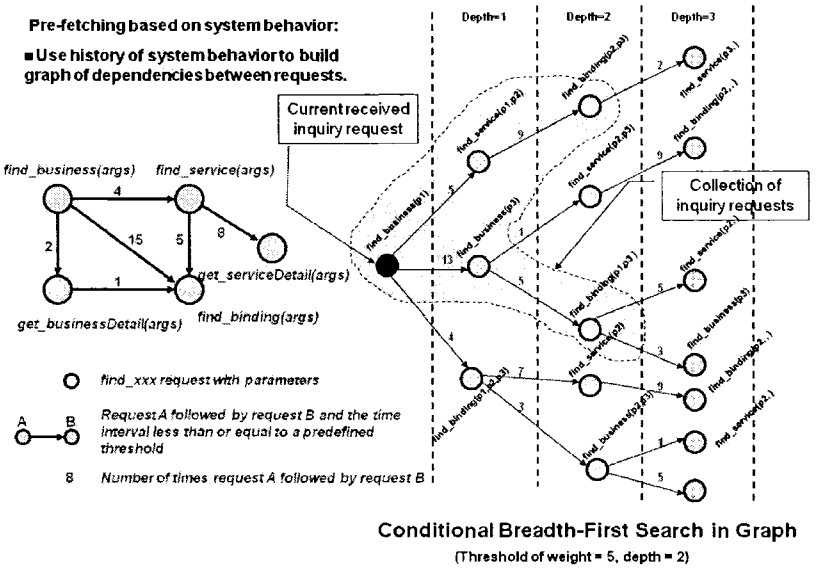

Figure 4-5 Graph-based Pre-fetching Process

4.3.2 Where Should the Pre-fetches Take Place?

The proposed pre-fetching proxy, shown in Figure 4-1, will be located as close to the Web Service registry server as possible. This is because the pre-fetching proxy makes use of the history of behavior of various Web Service requestors to build the graph of

dependencies between Web Service inquiry requests; the more Web Service inquiry requests received, the more accurate this makes anticipating a cluster of possible subsequent Web Service inquiry requests associated with the original Web Service inquiry request. A pre-fetching proxy that is close to the Web Service registry server, and thus receiving requests from many different clients, will have a greater chance of receiving more Web Service inquiry requests and making one client benefit from the combined history of all clients' behaviors.

4.3.3 Building/Maintaining a Pre-fetching Graph

An important function of the pre-fetching proxy is to build and maintain a pre-fetching graph. The pre-fetching proxy identifies whether a node represents a currently received Web Service inquiry request that already exists in pre-fetching graph G. If the node is not found, the node is simply added to the pre-fetching graph. Subsequently, the pre-fetching proxy checks the time interval between the currently received Web Service inquiry request and the previously received one. A directed edge *(previous_received_request, current_received_request)* with weight 1 will be added to G if the time interval is less than or equal to a pre-defined time interval threshold. If the node already exists in G, then it checks whether the directed edge *(previous_received_request, current_received_request)* is already in graph G; If the edge already exists in graph G and the time interval is less than or equal to the time interval threshold, then it increases the

weight of the edge. If the edge does not exist in graph G and the time interval is less than or equal to the time interval threshold, then it adds a new edge with weight 1 to graph G. Otherwise the edge will be ignored. Figure 4-6 illustrates the pseudo-code for building/maintaining a pre-fetching graph.

4.3.4 Pre-fetching Process

In Section 4.3.1, the pre-fetching proxy was described as having two main functions: (1) building and keeping a pre-fetching graph up to date, according to Web Service inquiry requests received in succession; and (2) pre-fetching a cluster of extra Web Service inquiry requests, based on the currently received original Web Service inquiry request and the pre-fetching graph. In this section, we focus on the second function of the pre-fetching proxy: the pre-fetching operation proper.

```
Pre-condition: Receive previous UDDI inquiry request → create node p;

Receive current UDDI inquiry request → create node n;

IF n does not exist in G
        Add n into G;
        IF time interval between receiving p and n <= time threshold
                Add directed edge (p, n) with weight 1 into G;
ELSE
        IF edge (p, n) does not exist in G
                IF time interval between receiving p and n <= time threshold
                        Add directed edge (p, n) with weight 1 into G;
        ELSE
                IF time interval between receiving p and n <= time threshold
                        Add one to weight of directed edge (p, n);

Post-condition: Pre-fetching process based on pre-fetching graph G;
```

Figure 4-6 Pseudo-code for Building/Maintaining a Pre-fetching Graph

The core of the proposed pre-fetching algorithm is a Conditional Breadth-First Search (C-BFS) algorithm based on the pre-fetching graph. C-BFS is a modified Breadth-First Search (BFS) with additional conditions. In graph theory, a BFS is a graph search algorithm that begins at the root node and explores all the immediate neighboring nodes. Then, for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal node or all nodes in the graph have been explored [44]. The proposed C-BFS is explained:

- C-BFS is based on a pre-fetching graph G.
- The root node $r \in V$ of C-BFS is the node that represents the Web Service inquiry request object currently received by the pre-fetching proxy.
- C-BFS begins at the end of the updating/maintaining process (see Figure 4-6) at the node corresponding to the Web Service inquiry request currently received, until all nodes that satisfy the additional breadth-first searching conditions have been explored. The output of C-BFS is a list of these satisfactory nodes.
- The additional breadth-first searching conditions include a pre-defined depth of C-BFS search D, and a pre-defined threshold on directed edge weight $W_{threshold}$, that are both properties of pre-fetching graph G. The C-BFS only explores the neighboring nodes whose distance from the root node of search is less than or equal to D, and a breadth-first search goes further if and only if the weight of the adjacent directed edge is greater than or equal to $W_{threshold}$.

To demonstrate the operation of the proposed C-BFS pre-fetching algorithm, an example is shown on the right side of Figure 4-5. We assume there is a pre-fetching graph and the current received UDDI inquiry request is *find_business(p1)*, shown as a solid black node. We also assume the predefined depth of C-BFS search is 2 and the threshold on directed edge weight is 5. The C-BFS results in a list of satisfactory nodes:

[*find_business(p1), find_service(p1,p2), find_business(p3),find_binding(p1,p3), find_binding(p2,p3)*]

The searching does not include the directed edge (find_business(p1), find_binding(p1,p2,p3)), as the weight of this edge is only 4, which is less than the pre-defined threshold on edge weight, so this edge will be ignored even though the search does not yet reach the depth of the C-BFS search. Therefore, the node find_binding(p1,p2,p3) will be not included in the result list of satisfactory nodes.

4.3.5 Example of Building, Maintaining and Pre-fetching in a Pre-fetching Graph

In this section, we present a simple "walk-through" sample experiment to show how the pre-fetching proxy builds and maintains the pre-fetching graph. We also show how the pre-fetching proxy anticipates using the pre-fetching graph, a cluster of possible subsequent UDDI inquiry requests. The sample experiment is done using the experimental conditions (1) depth of conditional Breadth-First Search D=3; and (2) threshold on directed edge weight, $W_{threshold}$ =3.

The executing UDDI inquiry request sequence is shown in Figure 4-7, where UDDI inquiry requests n_1, n_2, n_3, n_4 and n_5 represent *FindBusiness(261529116)*, *GetBusinessDetail(-910656319)*,*FindService(1290036844)*,*GetServiceDetail(-1933008719)*, and *FindBinding(783380029)*, respectively. The integer number associated with the name of the UDDI inquiry request is the hash value of all the parameters within the UDDI inquiry request.

Two pre-fetching operations take place when the above inquiry request sequence occurs on the system. Figure 4-7 shows the pre-fetching graph being built, maintained and performing the pre-fetching process based on received UDDI inquiry requests. The dark node indicates the node associated with the currently received UDDI inquiry request by the pre-fetching proxy. The solid lines indicate all previously added edges to the pre-fetching graph while the dashed line indicates the directed edge from the previously received inquiry request to the currently received inquiry request. The thick dashed lines in step 18 and 20 indicate the anticipated additional UDDI inquiry requests based on the current received original UDDI inquiry request. The first pre-fetching operation occurs in step 18 and the second in step 20. The node numbers appearing within boxes in the UDDI inquiry request sequence shown in Figure 4-7 trigger the pre-fetching operations.

4.4 Summary

This presented details on the proposed approach that integrates caching and pre-fetching

in Web Service discovery systems. We first presented more details on the proposed caching approach in Section 4.2. The proposed caching algorithm reuses a number of the existing traditional Web cache techniques in the context of Web Service discovery systems. Like most of the research on caching algorithms, we are interested in three core issues: what data should be cached, where the caches should be located, and how the cached data should be managed. Section 4.2.1 concluded that the data cached in Web Service discovery systems should include the pair of UDDI inquiry request entity object and associated response entity objects, and Section 4.2.2 concluded that caching should occur near the client, while Section 4.2.3 provided more detailed discussion concerning how to manage the cached data in Web Service discovery systems.

Section 4.3 introduced a graph-based pre-fetching strategy in the Web Service discovery area to further reduce the retrieval latency and the workload of the Web Service registry server. The proposed graph-based pre-fetching proxy anticipates future Web Service inquiry requests and pre-fetches these requests. These pre-fetched extra Web Service inquiry requests will be sent to the Web Service registry server, together with the original Web Service inquiry request. Section 4.3.1 briefly presented the functionalities of the proposed pre-fetching proxy, while Section 4.3.2 concluded that the pre-fetching proxy should be placed near the registry server. Section 4.3.4 presented a discussion on how the pre-fetching proxy works.

This chapter focused on the concepts underlying the proposed approach. The next

chapter will discuss how to implement the proposed approach and Chapter 6 and 7 will focus on experiments conducted on a prototype and the experimental results respectively.

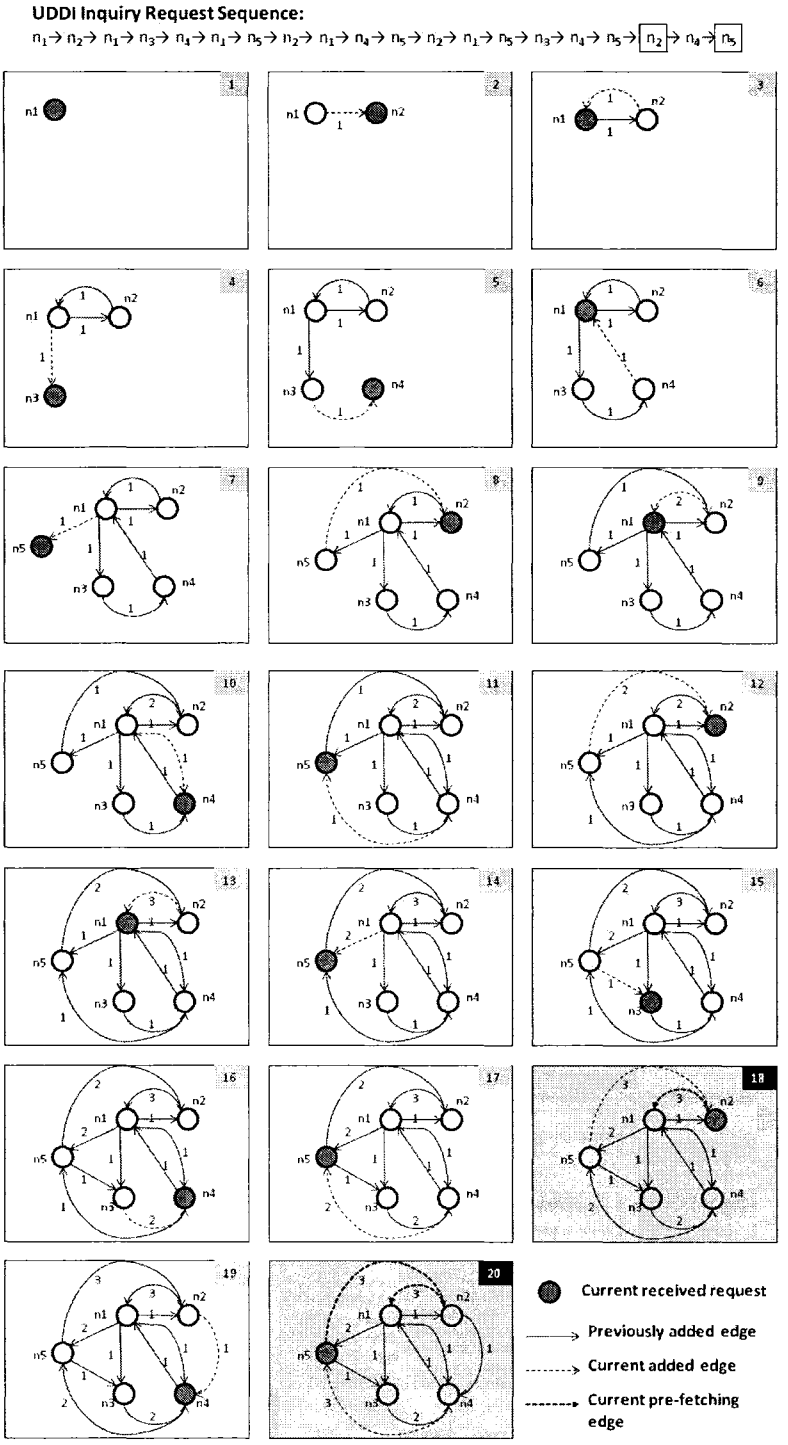

Figure 4-7 Pre-fetching Graph – Updating, Maintaining and Pre-fetching Process

Chapter 5: Implementation

5.1 Introduction

There are many existing open-source-based implementations of Web Service registry systems, such as Apache Software Foundation jUDDI [32] and IBM UDDI4j [43]. The Apache Software Foundation jUDDI was chosen for this thesis, since it was one the most popular open-source-based implementation of UDDI specification standard when this study began. Although jUDDI has a client proxy of its own that provides the client APIs of UDDI inquiry requests, the IBM UDDI4j was adopted as client implementation in this research because we are familiar with UDD4j client APIs for interacting with a UDDI registry server. Section 5.2 briefly introduces the jUDDI framework. The chapter consists of Section 5.2.1 and Section 5.2.2 which present the original jUDDI, Section 5.2.3 which discusses the modified jUDDI created for implementing the caching proxy and Section 5.2.4 which discusses the modified jUDDI created for implementing the pre-fetching proxy.

5.2 jUDDI Registry Framework

5.2.1 Requirements for jUDDI Installation

This section discusses a particular framework, jUDDI, which is used to represent the

structures of a UDDI registry system and how it can be used to implement a Web Service registry. As we mentioned earlier, jUDDI is an open-source pure Java implementation that complies with version 2.0 of the UDDI specification for Web Service. Being a typical Java Web application, jUDDI can be deployed to any application server or servlet engine that supports version 2.1 or later of the servlet API. It also requires Java SE Runtime Environment (JRE) version 1.3 or later [32]. The Apache HTTP Server version 2.2.3 [17] and Tomcat version 6.0.2 [45] were selected as the HTTP server and application server/servlet engine, since jUDDI and Tomcat are both open-source projects of the Apache Software Foundation. It also requires an external datastore in which to store and manage the persistent registry data. Typically, the external datastore is a relational database management system, such as MySQL, Oracle or DB2.

To set up our own UDDI registry for this research, we installed and configured the following implementation environments:

- Java 2 SDK: we used Sun's Java 2 SDK SE, version 1.6.0_01;
- Web server and/or servlet container: we used Apache Tomcat, version 6.0.2;
- Web server: we used Apache HTTP Server, version 2.2.3-win32;
- SOAP processing framework: we used Apache Axis, version 1.4;
- Data storage mechanism: we used the MySQL relational database, version 5.0.27;
- UDDI registry framework: we used the registry framework jUDDI, version 0.9;
- Caching system: Java Caching Systems (JCS), version 1.3.

To properly configure and deploy jUDDI in the environment described above,

understanding its architecture is important. A brief discussion of the jUDDI architecture is presented next.

5.2.2 jUDDI Implementation Details

To invoke a UDDI function, jUDDI employs the services of three configurable sub-components or modules that handle persistence (the DataStore), authentication (the Authenticator) and the generation of a Universally Unique ID (UUID) (the UUIDGen) [32]. jUDDI is bundled and pre-configured to use the default implementations of each of these modules to help the registry get up and running quickly. The persistence module (jUDDI DataStore) is a place to store registry data. Authentication (jUDDI Authenticator) is a mechanism that authenticates a Web Service publisher to publish Web Services and manage its Web Services stored in jUDDI. Authentication is a two-step process: the first step confirms that the ID/password combination provided by the user (Web Service publisher) in a *get_authToken* request is valid. The second step confirms that the publisher has been defined for jUDDI. A publisher is said to be defined when a row identifying the publisher exists in the *PUBLISHER* table of the jUDDI DataStore. The generation of UUID (performed by UUIDGen) is a mechanism for assigning a unique identification number to each UDDI entity, such as *BusinessEntity*, *BusinessService*, *BindingTemplate* and *tModel*. UDDI specification version 3.0.2 [39] indicates that each of these entities is to be uniquely identified by a UUID when these elements are

published. Additionally, jUDDI uses the UUID generator to create AuthTokens.

We have provided a brief description of jUDDI. Further description of the details of jUDDI and its default implementations are beyond the scope of this thesis, as we are only interested in the discovery mechanism of the jUDDI registry system. Therefore in our work, we ignore the persistence and authentication issue which are relevant only when publishing a Web Service in the jUDDI registry or managing an existing Web Service that has been published and stored in the jUDDI registry.

The UDDI specification version 3.0.2 [39] describes two types of core request APIs: *inquiry request* APIs and *publishing request* APIs. The inquiry requests APIs are a set of public non-authenticated operations that can be used in what is considered browsing or searching mode, and their purpose is similar to that of database search operations, which can extract information from the jUDDI registry and return several responses based on a set of search criteria [33]. The publishing requests are not of interest in our research, as the Web Service discovery process only employs inquiry requests. Therefore, in the rest of this section, we focus only on the jUDDI implementation of inquiry request APIs and jUDDI responses associated with inquiry requests.

jUDDI consists of a core UDDI request processor that un-marshals incoming UDDI requests, invoking the appropriate UDDI function and marshalling UDDI responses associated with these incoming UDDI requests. Marshalling and un-marshalling are the processes of converting XML data to/from Java objects [32]. In this thesis, Java objects

specifically indicate the three types of jUDDI registry objects related to inquiry requests: entity objects, request objects and response objects. Below, we show an example class for each kind of jUDDI registry object:

- Entity Object: *org.apache.juddi.datatype.business.BusinessEntity*;
- Request Object: *org.apache.juddi.datatype.request.FindBusiness*;
- Response Object: *org.apache.juddi.datatype.response.BusinessList;*

Section 5.2.3 and Section 5.2.4 will discuss more details on the modification in jUDDI for implementing the proposed approach that integrates caching and pre-fetching to improve the performance of Web Service discovery introduced in Chapter 4. In this section, we briefly introduce the jUDDI structure, for a basic understanding of how jUDDI inquiry works.

Not all packages of jUDDI are relevant in the context of this research because we are only interested in the packages that are required in the discovery scenario in UDDI registry systems. Therefore this section covers only discussions on the packages, listed in Table 5-1. Figure A-1 and Figure A-2 in Appendix A illustrate the main structure of jUDDI.

The entry point of jUDDI is class *org.apache.juddi.registry.AbstractService*, which extends from *javax.servlet.http.HttpServlet*. jUDDI uses Apache Axis to handle SOAP messaging. Axis defines a transparent transport framework that allows different transports to be used. In this thesis, we only use the HTTP protocol, so any servlet derived from the

javax.servlet.http.HttpServlet class is a proper class for handling UDDI inquiry requests. The *AbstractService* class uses the services of the *RegistryEngine* class to do the actual inquiry request processing. After receiving an inquiry request, the *RegistryEngine* class converts the XML-based UDDI inquiry request to Java objects (un-marshalling), invokes the appropriate Java objects, and converts Java objects to an XML-based response (marshalling). A detailed description is presented in the pseudo-code of *org.apache.juddi.registry.AbstractService* class of jUDDI registry server shown in next page.

Package/Interface/Class	Description
org.apache.juddi.datatype	All classes implementing common UDDI Registry Objects, e.g. *BusinessKey* and *KeyedReference*;
org.apache.juddi.datatype.binding	All classes implementing Objects concerning *Binding Template*, e.g. *BindingTemplate* and *AccessPoint*;
org.apache.juddi.datatype.business	All classes implementing Objects concerning *Business Entity*, e.g. *BusinessEntity*;
org.apache.juddi.datatype.request	All classes implementing Objects concerning inquiry request, e.g. *FindBusiness*, *FindService*, *FindBinding* and *FindTModel*;
org.apache.juddi.datatype.response	All classes implementing Objects concerning response associated to inquiry request, e.g. *BusinessDetail*, *ServiceDetail*, *TmodelDetail* and *BindingDetail*;
org.apache.juddi.datatype.service	All classes implementing Objects concerning *Business Service*, e.g. *BusinessService*;
org.apache.juddi.datatype.tmodel	All classes implementing Objects concerning *tModel* e.g. *TModel*;
org.apache.juddi.function	All classes implementing the actual logic for each UDDI invocation message, this package defines the function classes – one for each logical UDDI invocation, e.g. *FindBusinessFunction* and *FindTModelFunction*;

Table 5-1 jUDDI Packages Involved in UDDI Discovery Systems

Package/Interface/Class	Description
org.apache.juddi.handler	All classes implementing the marshalling and un-marshalling processes for each data type and function class. Un-marshals incoming UDDI requests, invoking the appropriate UDDI function and marshalling UDDI responses associated to these incoming UDDI requests. The core class in this package is *HandlerMaker* that holds a static HashMap linking the string representation of operations to instance of the appropriate maker class (e.g. map *BusinessDetail* to *BusinessDetailHandler*). Returns a reference to an instance of a maker object. HandlerMaker follows the Singleton pattern. Use getInstance instead of the 'new' operator to get an instance of this class.
org.apache.juddi.proxy	Represents a version 2.0 UDDI registry and implements all services as specified in the v2.0 specification. The core class in this package is *RegistryProxy* which extended from *AbstractRegistry*.
org.apache.juddi.registry	jUDDI uses Apache Axis to handle SOAP messaging. Axis defines a transparent transport framework that allows different transport to be used. In this thesis we only use HTTP protocol so any servlet derived from the *javax.servlet.http.HttpServlet* class is a proper class for handling UDDI inquiry requests. The core classes in this package are *AbstractService* which extended from *javax.servlet.http.HttpServlet* and *RegistryEngine* which extended from *AbstractRegistry*.
org.apache.juddi. AbstractRegistry	Abstract class implemented interface *Iregistry* which represents a version 2.0 UDDI registry and implements all services as specified in the v2.0 specification.
org.apache.juddi. Iregistry	Interface which represents a version 2.0 UDDI registry and implements all services as specified in the v2.0 specification.

Table 5-2 jUDDI Packages Involved in UDDI Discovery Systems (Continue)

A pseudo-code of *org.apache.juddi.registry.AbstractService* class in jUDDI registry server is shown in next page. The core consists of three processes: un-marshal the raw XML from the SOAP request message to get the UDDI inquiry request object then execute the requested UDDI function to obtain the appropriate UDDI response object, and finally, marshal the UDDI response object to XML format and append the results to the SOAP response message. The three parentheses highlight the three core processes

respectively. We will show how to modify the original *AbstractService* class to implement the caching proxy and pre-fetching proxy in the next two sections.

Pseudo-code of "AbstractService" in jUDDI Registry Server

```
1 Receive current UDDI inquiry request in HttpServletRequest;
2 Parse the XML found in HttpServletRequest into SOAP request message;
3 Extract the UDDI request from the SOAP body of the SOAP request message;
4 IF UDDI inquiry request does not exist within the SOAP request message
5     Throw a FatalErrorException and indicate a Client-side (WS Requestor) error;
6 ELSE
7     Grab the local name of the UDDI inquiry request element from the
      UDDI inquiry request;
8     Lookup the appropriate XML handler using the local name;
9     Unmarshal the raw XML into the appropriate UDDI inquiry request object
      using appropriate handler;
10    Grab a reference to the shared UDDI RegistryEngine instance
      and execute the requested UDDI function to obtain the UDDI response object;
11        RegistryEngine registry = RegistryServlet.getRegistry();
12        uddiResponseObj = registry.execute(uddiRequestObj);
13    Lookup the appropriate response handler which will be used to
      marshal the UDDI response object into the appropriate XML format;
14    Attach the results to the SOAP body of the SOAP response message;
```

Unmarshal Process: XML→UDDI Request Object

Execute

Marshal Process: UDDI Response Object→ XML

5.2.3 Modified jUDDI for Implementing Caching Proxy

Chapter 4 discussed the proposed approach that integrates caching and pre-fetching to improve the performance of Web Service discovery systems. Figure 4-1 showed a client-proxy-server framework structure for this proposed approach. The integrated system includes two extra components, the *caching proxy* and *pre-fetching proxy.*

This section and the next section will present details on how to implement the two

additional components using jUDDI. Figure 5-1 illustrates the implementation structure of the system that integrates the caching proxy and the pre-fetching proxy in Web Service discovery systems. Figure 5-2 illustrates the sequence diagram for integration of caching and pre-fetching proxy in Web Service discovery systems. The caching proxy and pre-fetching proxy are both implemented using jUDDI. We disabled some of the original functions and added some new ones. The proxies do not require an external data store in which to store and manage the persistent registry data because neither the caching proxy nor the pre-fetching proxy have inquiring and publishing functions. They both deal with only the UDDI inquiry request they have currently received, and decide what the next action should be. If the current received UDDI request is not a UDDI inquiry request, such as a UDDI publish request or admin request, then they throw a *FatalErrorException* and indicate that this is a Client-side (Web Service requestor) error. So, we have not included the jUDDI components that handle publishing and administrative functions in prototype implementation. The new functionalities that were created for the prototype are discussed next.

The caching proxy: it has the responsibility of deciding whether the UDDI inquiry request has been received and cached previously. If so, it retrieves the cached UDDI response associated with the UDDI inquiry request from the local caching storage. Otherwise, it forwards the UDDI inquiry request to the pre-fetching proxy and waits for the UDDI response object from the pre-fetching proxy. The caching proxy implementation uses the

Apache JCS library [41]. The JCS configuration used in the research is presented in Appendix B.

The pre-fetching proxy: it is responsible for deciding how to anticipate a cluster of possible subsequent UDDI inquiry requests associated with the original UDDI inquiry request currently received by the pre-fetching proxy. The pre-fetching proxy then sends these anticipated possible subsequent UDDI inquiry requests, together with the original UDDI inquiry request, to the jUDDI registry server.

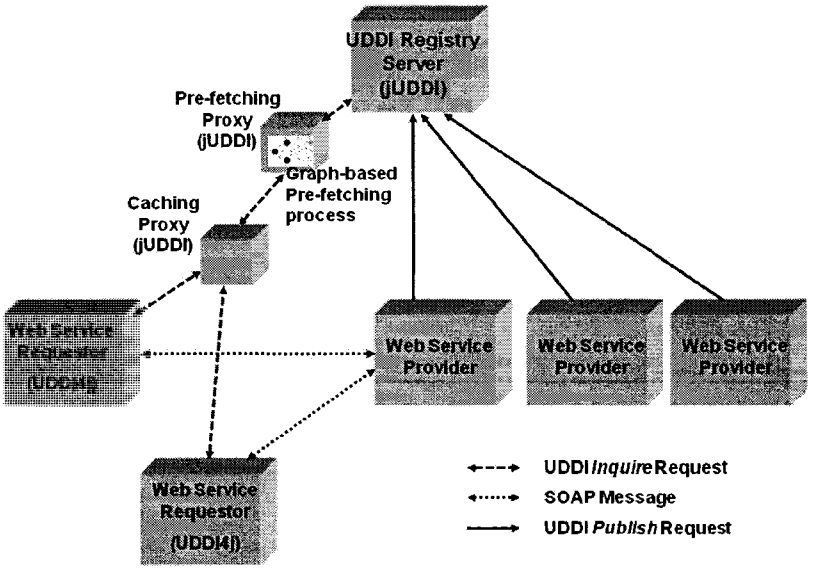

Figure 5-1 Implementation Structure

The pseudo code of *org.apache.juddi.registry.AbstractService* class in the caching proxy is shown before the end of this section. The pseudo-code between line 1 and line 9 is exactly the same as the pseudo code in the jUDDI registry server because this corresponds to an un-marshalling process for parsing the raw XML to the UDDI inquiry request object. Thereafter, the caching proxy needs to decide whether the currently

received UDDI inquiry request has been received and cached previously. If so, it directly retrieves the cached UDDI response object (value) using the UDDI inquiry request (key) from the local caching storage and marshals the UDDI response object to XML and appends it to the SOAP response message. Otherwise, the caching proxy forwards the UDDI inquiry request to the pre-fetching proxy and waits for the UDDI response object from it. In this case, the UDDI inquiry request currently received by the caching proxy, and forwarded to the pre-fetching proxy, is called as the *original* UDDI inquiry request. The primary difference with the implementation of the *AbstractService* class in the caching proxy is that it grabs a reference to the UDDI *RegistryProxy* instance rather than the shared *RegistryEngine* instance and forwards the original UDDI inquiry request to the target, for example, the pre-fetching proxy.

There are two types of responses to the original UDDI inquiry request: the *original* UDDI response, e.g. *BusinessList* or *ServiceList*, and the *compound* UDDI response object, e.g. *ResponseList* and *ResponseBundle*. The caching proxy will get the original UDDI response if the pre-fetching proxy does not anticipate any possible subsequent UDDI inquiry requests associated with the original UDDI inquiry request. After getting the original UDDI response from the pre-fetching proxy, the caching proxy simply caches the key-value pair of the original UDDI inquiry request object (key) and the original UDDI response object (value) into the local caching storage, as illustrated in Figure 4-2. Thereafter, it marshals the original UDDI response object to XML and appends it to the

SOAP response message. The caching proxy will get the compound UDDI response if the pre-fetching proxy does anticipate one or more possible subsequent UDDI inquiry requests associated with the original UDDI inquiry request.

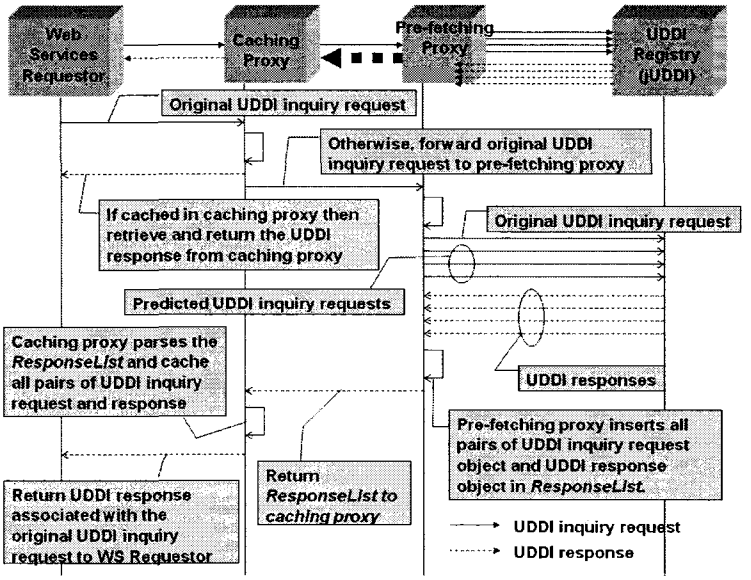

Figure 5-2 Sequence Diagram for Integration of Caching and Pre-fetching Proxy

Figure 5-3 illustrates the structure of the compound UDDI response, *ResponseList*. Each *ResponseList* consists of one or more *ResponseBundle* elements. Only one *ResponseBundle* element has the attribute "original," while the rest of the *ResponseBundle* elements have the attribute "extra." The *ResponseBundle* element with attribute "original" wraps a pair of the "original" UDDI inquiry request object and the UDDI response object associated with it. Similarly, each *ResponseBundle* element with attribute "extra" wraps a pair of the "additional" UDDI inquiry request object and the UDDI response object associated with it. The "additional" UDDI inquiry request object is one possible subsequent UDDI inquiry request associated with the "original" UDDI

inquiry request which is anticipated by the pre-fetching proxy based on a pre-fetching graph.

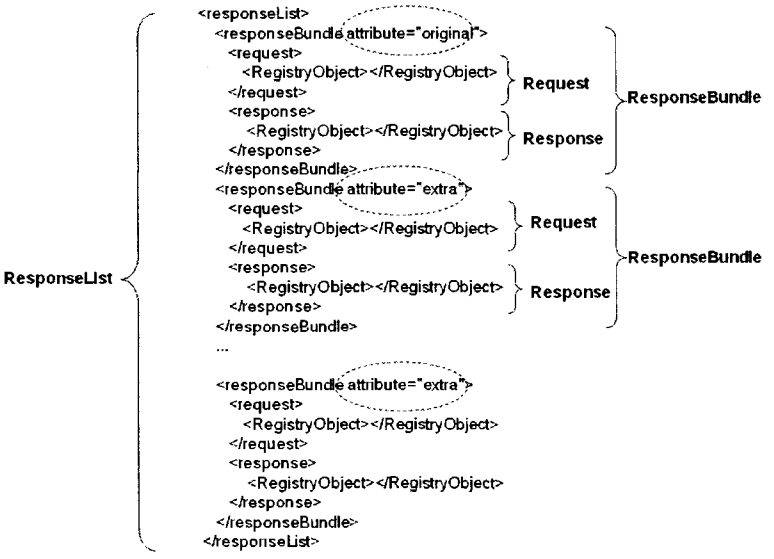

Figure 5-3 XML Schema of ResponseList

After getting the compound UDDI response from the pre-fetching proxy, the caching proxy will go through each pair composed of the UDDI inquiry request object and the UDDI response object and cache them as a "key-value" pair in the local caching storage. Moreover, if and only if the UDDI inquiry request is an original UDDI inquiry request, then it marshals the UDDI response object associated with the original UDDI inquiry request to XML and appends it to the SOAP response message sent to the client.

Pseudo-code of "AbstractService" in Caching Proxy

```
1 Receive current UDDI inquiry request in HttpServletRequest from Client;
2 Parse the XML found in HttpServletRequest into SOAP request message;
3 Extract the UDDI request from the SOAP body of the SOAP request message;
4 IF UDDI inquiry request does not exist within the SOAP request message
5       Throw a FatalErrorException and indicate a Client-side (WS Requestor) error;
6 ELSE
7     Grab the local name of the UDDI inquiry request element from the
      UDDI inquiry request;
8     Lookup the appropriate XML handler using the local name;
9     Unmarshal the raw XML into the appropriate UDDI inquiry request object (original)
      using appropriate handler;
10    IF UDDI inquiry request object (key) has been cached previously and not expired
11          Retrieve the cached UDDI response object (value) from local storage;
12          Lookup the appropriate response handler which will be used to
            marshal the UDDI response object into the appropriate XML format;
13          Attach the results to the SOAP body of the SOAP response message;
14    ELSE
15          Grab a reference to the UDDI RegistryProxy instance
            and forward the UDDI inquiry request to Pre-fetching Proxy;
16          RegistryProxy cacheProxy = new RegistryProxy();
17          uddiResponseObj = cacheProxy.execute(uddiRequestObj);
18          IF uddiResponseObj is a single normal UDDI response object
19                Cache this UDDI request-response object pairs;
20                Lookup the appropriate response handler which will be used to
                  marshal the UDDI response object into the appropriate XML format;
21                Attach the results to the SOAP body of the SOAP response message;
22          ELSE
23                FOR EACH element in bundle of UDDI request-response object pairs
24                      get reference to UDDI request object;
25                      IF UDDI inquiry request object (key) has not been cached previously or has expired
26                            Cache this UDDI request-response object pairs;
27                      IF this UDDI request object is original UDDI inquiry request object
28                            Lookup the appropriate response handler which will be used to
                              marshal the UDDI response object into the appropriate XML format;
29                            Attach the results to the SOAP body of the SOAP response message;
```

Unmarshal Process XML→UDDI Request Object

Cached

Not pre-fetching

Pre-fetching

Not Cached

5.2.4 Modified jUDDI for Implementing the Pre-fetching Proxy

As in case of the caching proxy, the pre-fetching proxy is implemented using jUDDI. The caching proxy will forward the original UDDI inquiry request to the pre-fetching proxy and wait for the original or compound UDDI response object from the pre-fetching proxy. The pre-fetching proxy is responsible for deciding how to anticipate a cluster of possible subsequent UDDI inquiry requests associated with the original UDDI inquiry request currently received by the pre-fetching proxy. Then the pre-fetching proxy sends these anticipated inquiring requests together with the original UDDI inquiry request to the jUDDI registry server.

The pseudo-code of *org.apache.juddi.registry.AbstractService* class in the pre-fetching proxy is shown at the end of the section. The code between line 1 and line 9 is exactly the same as the pseudo-code in the jUDDI registry server because this is an un-marshalling process to parse the raw XML to the UDDI inquiry request object. The original UDDI inquiry request object is used to update and maintain the pre-fetching graph. A pre-fetching process is also executed based on the original UDDI inquiry request and the pre-fetching graph. There are two possible results of the execution of a pre-fetching process: either the result contains only the original UDDI inquiry request, or it contains the original UDDI inquiry request together with additional UDDI inquiry requests corresponding to the pre-fetching operation. In the first case, the pre-fetching

proxy simply returns the single original UDDI response to the caching proxy when it gets it from the jUDDI registry server; in the second case, the pre-fetching proxy has to wrap all pairs, consisting of a UDDI inquiry request object and UDDI response object associated with it, to the *ResponseList* one by one, then return the *ResponseList* to the caching proxy.

5.3 Summary

This chapter discussed the implementation of the proposed approach to integrate caching and pre-fetching in Web Service discovery systems. We first discussed the implementation of the jUDDI registry framework in Section 5.2.2. This provided the necessary framework for discussing the modification of the jUDDI registry framework performed to implement the proposed approach in Web Service discovery systems. Section 5.2.3 and 5.2.4 show the implementation of the proposed approach presented in Chapter 4.

In the next chapter we will discuss the experimental environment and the measurement parameters that are used to analyze the experimental results quantitatively. The analysis of the performance of Web Service registry systems will be discussed in Chapter 7.

Pseudo-code of "AbstractService" in Pre-fetching Proxy

```
1 Receive current UDDI inquiry request in HttpServletRequest from Caching Proxy;
2 Parse the XML found in HttpServletRequest into SOAP request message;
3 Extract the UDDI request from the SOAP body of the SOAP request message;
4 IF UDDI inquiry request does not exist within the SOAP request message
5       Throw a FatalErrorException and indicate a Client-side (WS Requestor) error;
6 ELSE
7       Grab the local name of the UDDI inquiry request element from the
        UDDI inquiry request;
8       Lookup the appropriate XML handler using the local name;
9       Unmarshal the raw XML into the appropriate UDDI inquiry request object (original)
        using appropriate handler;
10      Execute the pre-fetching process based on the original UDDI inquiry request:
              (1) Update and maintain the pre-fetching graph;
              (2) Perform pre-fetching;
11      Grab a reference to the UDDI RegistryProxy instance;
12      RegistryProxy prefetchProxy = new RegistryProxy();
13      IF result of executing pre-fetching process only contains original UDDI inquiry request
14            Forward the original UDDI inquiry request to jUDDI registry server;
15            uddiResponseObj = prefetchProxy.execute(uddiRequestObj);
16            Lookup the appropriate response handler which will be used to
              marshal the UDDI response object into the appropriate XML format;
17            Attach the results to the SOAP body of the SOAP response message;
18      ELSE IF result of executing pre-fetching process only contains several UDDI inquiry requests
19            FOR EACH UDDI inquiry request
20                  Forward UDDI inquiry request to jUDDI registry server;
21                  uddiResponseObj = prefetchProxy.execute(uddiRequestObj);
22                  Lookup the appropriate response handler which will be used to
                    marshal the UDDI response object into the appropriate XML format;
23                  Append the XML data to ResponseList;
24            Attach the ResponseList to the SOAP body of the SOAP response message;
```

Unmarshal Process: XML→UDDI Request Object

Update and Maintain Pre-fetching graph

Not Pre-fetching

Pre-fetching

Chapter 6: Experimental Environment and Measurement Parameters

6.1 Introduction

We will discuss the experimental procedures in this chapter. The various resources and environment required to successfully complete the experiments are discussed in Section 6.3 while Section 6.4 briefly describes the experimental applications used for performance analysis. Section 6.5 briefly introduces the measurement parameters that can be used to quantitatively analyze the experimental results. The impact of various parameters on the performance metrics of Web Service registry systems will be discussed in Section 6.6. Section 6.7 summarizes this chapter.

6.2 The Purpose of the Experiments

Essentially, the experiments in this research have two purposes. The first purpose is to test whether the additional components and features in the proposed approach that integrates the caching proxy and the pre-fetching proxy in Web Service registry systems are successfully implemented. These components and features include:

- Caching proxy: managing cached UDDI objects; handling the UDDI inquiry request object received; handling the UDDI response object associated with

UDDI inquiry request object; and

- Pre-fetching proxy: maintaining the pre-fetching graph; executing the pre-fetching process; forwarding the UDDI inquiry requests to the UDDI registry server and handling the UDDI response associated with the UDDI inquiry request.

The second purpose is to analyze the performance of the proposed approach. The improvements should be reflected in the hit ratio measured in the caching proxy and the response time of the UDDI inquiry request measured at the client end.

6.3 Resources and Environments Required

We introduced software to set up our own UDDI registry system for implementing the approach proposed in Section 5.2.1. In this section, we briefly list all hardware and operating systems installed and configured for the implementation environment.

Section 4.1.3 illustrated the prototype structure for integrating the caching proxy and the pre-fetching proxy in Web Service registry systems. We set up and ran each component on a separate workstation. Implementing the proposed approach requires a workstation running jUDDI modified to implement the caching proxy; a workstation running jUDDI modified to implement the pre-fetching proxy; a workstation running the original jUDDI as a UDDI registry server; and finally, a workstation running UDDI4j as the client. The features of the client workstation and UDDI registry server are listed below:

- Microsoft Windows XP Media Center Edition, Version 2002, Service Pack 2
- Intel® Core™2 CPU 6300 @ 1.86GHz
- 1.99 GB of RAM
- Intel® 82562V 10/100 Network Connection

The features of the caching proxy and the pre-fetching proxy workstations are listed below:

- Microsoft Windows XP Professional, Version 2002, Service Pack 2
- Intel® Pentium® 4 CPU 3.00GHz
- 1.99 GB of RAM
- Intel® 82562V 10/100 Network Connection

The memory required to accommodate the pre-fetching graph is expected to be much smaller in comparison to the size of the memory used in the caching proxy. This is because the nodes and edges of the graph that need to be stored consume only a small amount of memory. Moreover, the maximum number of nodes is limited by the total number of UDDI inquiry messages types and the total number of services in the registry. Thus the memory available in the pre-fetching proxy is enough to accommodate the graph generated during the experiments.

6.4 Experimental Applications

Because we cannot obtain the real data of Web Service discovery registry so it is hard to

test our proposed approach by using real data. Therefore a performance analysis of the implemented integrated approach based on measurements made on the prototype subjected to synthetic workload provides deep insights into the integration system's behavior and performance.

We create five different Web Service discovery applications which contain business logic to look for desired Web Services. Each application consists of 30 UDDI inquiry requests to implement particular business logic. Each experimental run comprises of running these applications in a particular sequence, so there are 150 UDDI inquiry requests in each run. Each application is independent of each other but part of their business logic overlap. We create a full permutation of these 5 applications to simulate an environment that avoids the tying of the experimental results to a particular sequence of these applications. Therefore the total number of experiments run is the factorial of 5 that is equal to 120.

A number of performance metrics described later in this thesis are measured as a function of Time (t). We define the parameter t for representing the sequence of UDDI inquiry requests in our experiments. For each experimental run t is set to 1 when the first UDDI inquiry request occurs and is incremented for the occurrence of every subsequent UDDI inquiry request. The value of a performance metric X at time t is given by,

$$X = \frac{\sum_{i=1}^{120} X_i(t)}{120} \qquad \textit{Equation 6-1}$$

where $X_i(t)$ is the value of X observed at time t in the i^{th} experimental run.

We also calculated the standard deviation for each measurement. Early on, in each run, the difference in the message sequences and in the arrival order has a big impact on cache hits and pre-fetches, there is a lot of variation. But after this transient phase, the overall messages received by the caching and pre-fetching proxies tend to become more similar, and the standard deviation decreases as a result (see Figure 6-1 for example). Because of the high standard deviation during the transient phase, the differences between measurements and trends are not statistically significant, so we will limit our conclusions to the remaining phase.

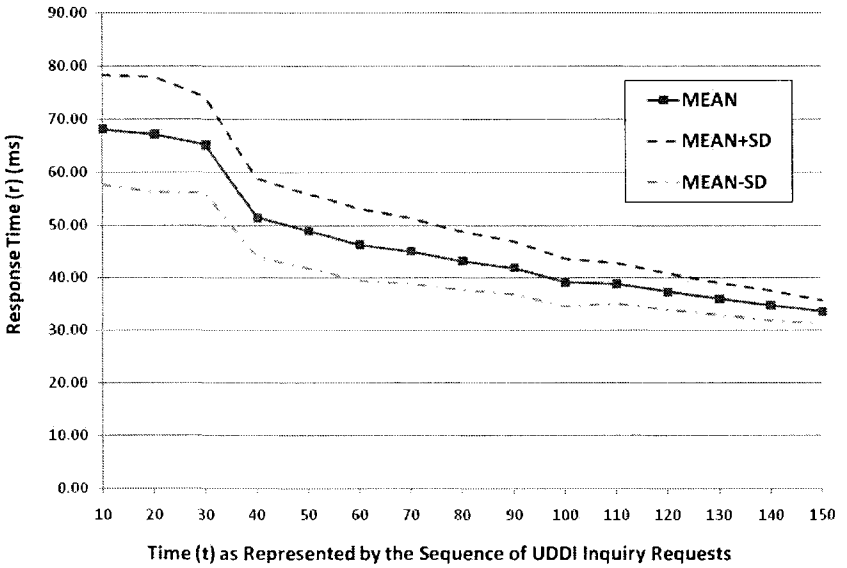

Figure 6-1 Response Time (r) - Mean and Standard Deviation
(YCYP, v=10%, D=2, $T_{threshold}$=10000ms, $W_{threshold}$=2)

A brief description of the applications is provided. A company in New York is producing and selling skateboards and snowboards. It wants to look for some parts suppliers who can provide the main components of skateboards and snowboards, such as

boards and wheels. In addition, for purpose of reducing the cost, this company wants these eligible parts suppliers provide on-line services, for example on-line products catalog showing all available parts, on-line price checking, on-line orders, on-line order status checking and on-line payment and invoice services. Moreover, this company wants eligible parts suppliers to provide HTTP-based and/or SOAP-based access point for all their available services.

Another travel agent wants to offer to people the ability to book complete vacation packages: plane/train/bus tickets, hotel reservations, car rental, excursions, etc. Travel related Service providers (airlines, bus companies, hotel chains, car rental companies etc.) provide Web Services to query their offerings and perform reservations. Credit card companies also provide services to guarantee payments made by consumers. Due to the loosely coupled-nature of Web Services, the travel agent does not need to have a priori agreements with service providers or credit card companies. This allows the travel agent to have access to more services, offering more combinations of options to its customers, the credit card companies to offer their services broadly and therefore make their customers happy, and the service providers to offer their services broadly and easily and therefore generating more business for themselves. Both of these two applications contain common UDDI inquiry requests looking for Web Service providers that offer various secure payments and invoices.

6.5 Measurement Parameters

This section details the performance metrics used for the quantitative analysis of system performance. In order to clearly understand the definitions below, four core concepts—"original UDDI inquiry request," "additional UDDI inquiry request," "original UDDI response," and "compound UDDI responses" are defined prior to the rest.

- ***Original UDDI Inquiry Request*** is an inquiry statement in a client application, such as the inquiry request "*find_business (parameters)*". It is only initiated by client applications. In the caching proxy, it is either used as a "key" to retrieve the cached UDDI response from local storage or simply forwarded to the pre-fetching proxy. In the pre-fetching proxy, it is used as a "root" node to execute a C-BFS pre-fetching process.
- ***Original UDDI Response*** is a UDDI response associated with the original UDDI inquiry request. The original UDDI response is either returned from the UDDI registry server or retrieved from the caching proxy.
- ***Additional UDDI Inquiry Request*** is a UDDI inquiry request that is not an original UDDI inquiry request initiated by client applications, but is an additional request generated by the pre-fetching proxy for the UDDI registry server.
- ***Compound UDDI Response*** is a special UDDI response consisting of the pair made up of the original UDDI inquiry request and the original UDDI response

associated with it, and the pairs made up of the additional UDDI inquiry requests and UDDI responses associated with them, as shown in Figure 5-3.

6.5.1 Response Time

As we know, the process of discovering the desired Web Services consists of a sequence of UDDI inquiry requests. We use the term *Effective Discovery Time* (*EDT*) to indicate the response time of Web Services discovery overall, while the term *Response Time* is used to indicate the response time *r* for a single UDDI inquiry request. The relationship between EDT and Response Time can be described as follows:

$$EDT = \sum_{i=1}^{n} r_i \qquad \textit{Equation 6-2}$$

where n is the total number of original UDDI inquiry requests in one Web Service application. Note that the execution time of the code segment between two successive UDDI inquiry requests is negligibly small. Thus, the major contributions to EDT come from the delays associated with the inquiry requests.

- ***Effective Discovery Time (EDT)*** is defined as the time that elapses between the start of the application and the receipt of the response to the last UDDI inquiry request of the application running on the client workstation. Essentially, EDT consists of a group of response times of single UDDI inquiry requests. In a typical scenario, a client application may be looking for a business entry first, so the first

UDDI inquiry request may be *find_business*. Usually the last step in the search activity, either the *find_binding* operation or the *get_bindingDetail* is used to find binding templates within a business service. Thereafter the client application can get the desired access point that informs the client application where to bind the Web Service that has been found. Figure 6-2 illustrates the measurement of EDT. The first UDDI inquiry API request in the illustration is *find_business (parameters)*, while the last UDDI inquiry API request is *get_bindingDetail (bindingKey),* where the *bindingKey* is the unique identifier for the *bindingTemplate*. The key is assigned by the UDDI registry server and cannot be edited or modified when a new binding is registered, associated with a Web Service by the provider. This key is then used in subsequent queries and updates for binding. In a certain sense, the *bindingTemplate* structure is the ultimate goal of Web Service discovery as it allows a Web Service requestor to get information about businesses, their descriptions, contact information, categorization and taxonomy information, and the kind of Web Services they offer. After a decision is made by a requestor regarding the use of a particular Web Service from a particular provider, the requestor needs to acquire the necessary technical information for the service to actually be invoked from the *bindingTemplate* structure.

We have considered four different cases for the analysis of caching and pre-fetching.

Case One: refers to a standard Web Service discovery process that does not use either the caching proxy or the pre-fetching proxy. The UDDI inquiry request will be sent to a Web Service registry server via the caching proxy and the pre-fetching proxy, with the caching and pre-fetching operations disabled. The UDDI response data will be sent back to the client application via the caching proxy and the pre-fetching proxy as well. This case will be referred to as "No Cache No Pre-fetch (NCNP)" in Chapter 7.

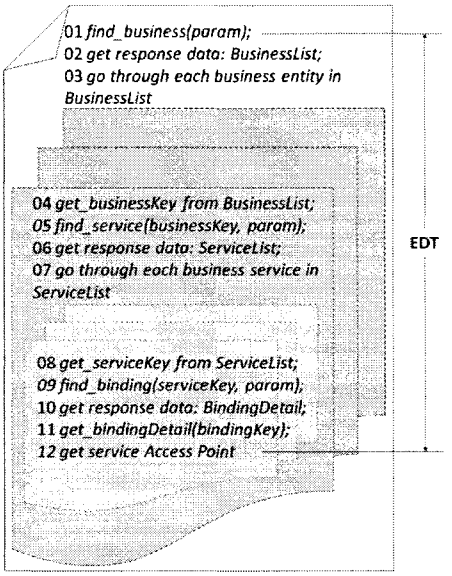

Figure 6-2 Definition of EDT

Case Two: refers to a Web Service discovery process that uses caching but no pre-fetching. The UDDI response data can simply be retrieved from the caching proxy and returned to the client application if both the UDDI inquiry request object and UDDI response object associated with it have been cached and have not expired. Otherwise, if the UDDI inquiry request object has not been cached or the associated response has expired, the caching proxy simply forwards the currently received UDDI inquiry request

to the Web Service registry server. Thereafter, the caching proxy will cache the pair of the UDDI inquiry request and the response data associated with it in local storage. This case will be referred to as "Yes Cache No Pre-fetch (YCNP)" in Chapter 7.

Case Three: refers to a Web Service discovery process that is optimized with a pre-fetching proxy but has no caching proxy. But this test case is not applicable. Because the purpose of pre-fetching is to predict the subsequent UDDI inquiry requests as accurately as possible and cache these anticipated UDDI inquiry requests in a caching proxy to increase the Cache-Hit Ratio. The pre-fetching proxy cannot improve the performance without the existence of a caching proxy.

Case Four: refers to a Web Service discovery process that uses both a caching proxy and a pre-fetching proxy. The caching proxy will check whether the UDDI inquiry request has been cached previously. The UDDI response data can simply be retrieved from the caching proxy and returned to the client application if both the UDDI inquiry request object and the UDDI response object associated with it have been cached and have not expired. Otherwise, if the request and response object have not been cached or have expired, the caching proxy simply forwards the original UDDI inquiry request to the pre-fetching proxy. The pre-fetching proxy will execute the pre-fetching algorithm to cluster a collection of additional UDDI inquiry requests based on the currently received original UDDI inquiry request. Thereafter, the pre-fetching proxy will send the collection of UDDI inquiry requests, including the original UDDI inquiry request and the additional

UDDI inquiry requests, to the Web Service registry server. The Web Service registry server will execute the collection of inquiry requests and return the response data to the caching proxy via the pre-fetching proxy. The response data associated with the original inquiry message will be cached in the caching proxy and returned to the client, whereas the response data associated with the additional inquiry messages will only be cached in the caching proxy and not returned to the client immediately. This case will be referred to as the "Yes Cache Yes Pre-fetch (YCYP)" in Chapter 7.

6.5.2 Number of Web Service Inquiry Requests

- ***Number of Applications (A)*** indicates the total number of applications with a business logic that includes one or more UDDI inquiry requests for certain business purposes. For example, a vacation services query includes inquiries for plane and inquiries for hotel reservations.
- ***Number of Web Service Inquiry Requests in One Application (n)*** indicates the number of original UDDI inquiry requests used in one Web Service discovery application.
- ***Overlap Rate of Web Service Inquiry Requests between Applications (v)*** indicates the percentage of UDDI inquiry requests in common. An example is presented in Section 6.4.
- ***Pre-fetching Rate (m)*** pre-fetching rate at time t is the number of pre-fetched

additional UDDI inquiry requests associated with the original UDDI inquiry request that was received by the pre-fetching proxy at *t*.

6.5.3 Cache Size and the Cache-Hit Ratio *(H)*

The cache size is the amount of space available in the caching proxy for storing the key-value pairs of the UDDI inquiry request and the UDDI response associated with it. In our experiments, we do not measure the influence of cache size in the caching proxy on the performance as the size of UDDI inquiry request object and associated response object are both small. The available memory size for JCS is 1.00 GB in the caching proxy and it is enough to accommodate all the requests and response object pairs.

- ***Cache-Hit Ratio (H)*** at time *t* is the ratio of the total number of cache-hits that have occurred from time 0 to time *t* to the total number of original UDDI inquiry requests received in the caching proxy during this same time interval. Thus, if each application generates *n* original UDDI inquiry requests, the Cache-Hit Ratio *H* can be expressed as follows:

$$H = \frac{Total\ number\ of\ cache\ hits}{Total\ number\ of\ original\ inquiry\ requests\ received\ in\ caching\ proxy} = \frac{Total\ number\ of\ cache\ hits}{A \times n} \qquad \textit{Equation 6-3}$$

Cache-Hit Ratio H is one of the most important performance metrics, and a higher value of *H* indicates that more original UDDI inquiry requests will encounter a low

latency, as an UDDI inquiry request giving rise to a cache hit will get the desired UDDI response directly from the local storage of the caching proxy rather than the remote UDDI registry server. Moreover, a higher value of H implies a lower workload for the remote UDDI registry server and less traffic through the networks.

6.5.4 Pre-fetch-Hit Ratio (H_p)

In this thesis, we are interested in another important performance metrics besides the Cache-Hit Ratio: the Pre-fetch-Hit Ratio. This performance metrics reflects the precision with which the proposed pre-fetching proxy anticipates, using the pre-fetching graph, a cluster of possible subsequent UDDI inquiry requests.

As described in Section 4.3.1, the pre-fetching proxy anticipates a cluster of extra UDDI inquiry requests, according to the currently received original UDDI inquiry request, based on a pre-fetching graph (see Figure 4-4).

Several extra UDDI inquiry requests may be generated by the pre-fetching proxy, depending on three critical parameter values of C-BFS: the Time Interval Threshold ($T_{threshold}$), the depth of Conditional Breadth First Search (D) and the Threshold on Edge Weight ($W_{threshold}$). An extra UDDI inquiry request in this anticipated collection associated with the original currently received UDDI inquiry request may or may not be generated by an application in the future. **If the subsequent original UDDI inquiry request generated before the next cache-missed original UDDI inquiry request which will**

trigger the next pre-fetch, is in the collection of anticipated UDDI inquiry requests, we call this subsequent original UDDI inquiry request a *pre-fetch hit;* otherwise we call it a *pre-fetch miss* (see Figure 6-3 and Figure 6-4).

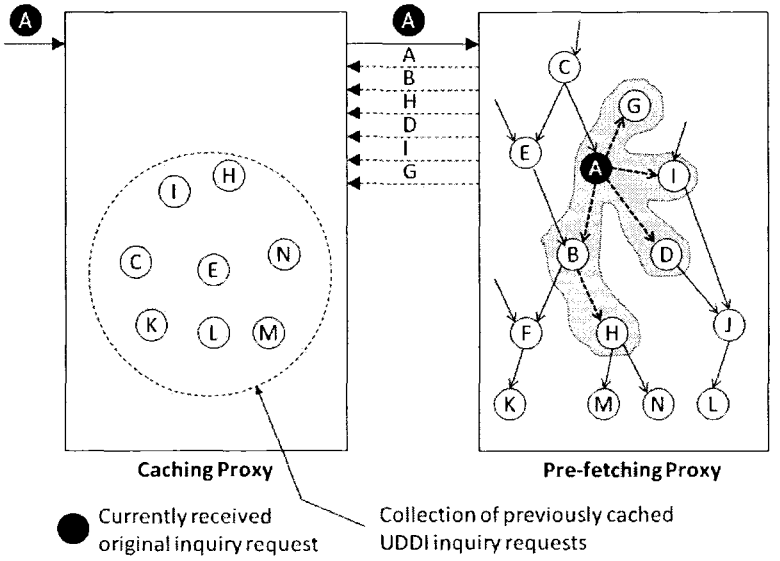

Figure 6-3 Definition of Pre-fetch-Hit Ratio (Hp) – Before Caching

We give an example to show the difference between the Cache-Hit Ratio and the Pre-fetch-Hit Ratio. As shown in Figure 6-3, we suppose that the original UDDI inquiry request "A" has never been cached in the caching proxy or it has expired, so it should be forwarded to the pre-fetching proxy and trigger a C-BFS process using the pre-fetching graph. Several extra UDDI inquiry requests may be pre-fetched in the pre-fetching proxy, e.g. UDDI inquiry requests "B", "D", "H", "I" and "G". All extra UDDI inquiry requests in the anticipated collection and "A" are returned and cached in this caching proxy, as shown in Figure 6-4.

There are two sets in the caching proxy as shown in Figure 6-4. The small set refers

to the anticipated collection, which contains the pre-fetched extra UDDI inquiry requests associated with the latest received original UDDI inquiry request "A". The big set is the collection containing all previously cached UDDI inquiry requests in the caching proxy. There is an intersection of two sets, which contains two UDDI inquiry requests, "H" and "I". A UDDI inquiry request in the intersection of two sets means the inquiry request has been cached before; in addition, it is pre-fetched in association with the currently received original UDDI inquiry request. In this case, the two cached UDDI inquiry requests will be updated.

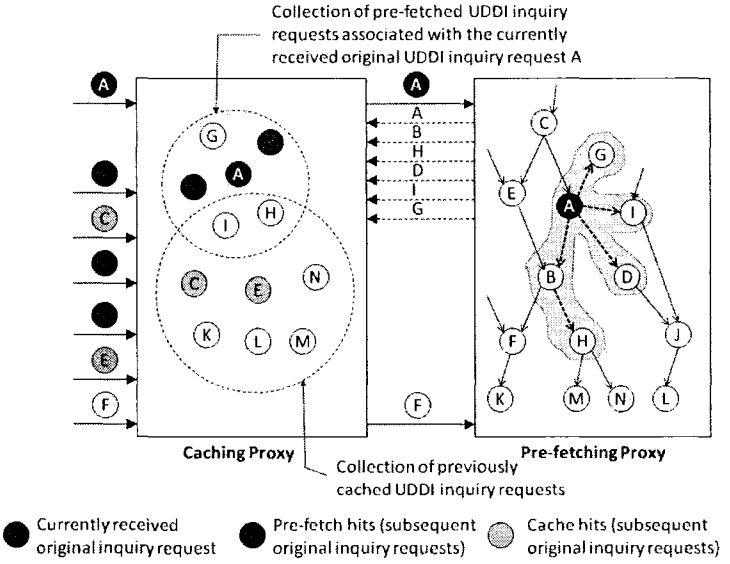

Figure 6-4 Definition of Pre-fetch-Hit Ratio (H_p) – After Caching

We suppose the caching proxy receives the subsequent original UDDI inquiry requests "B→C→B→D→E→F" after receiving "A", as shown in Figure 6-4. All the subsequent original UDDI inquiry requests, except "F", contribute once to increase the Cache-Hit Ratio. It is worth that the original UDDI inquiry request "B" contributes twice

to the Cache-Hit Ratio. However, only the original UDDI inquiry requests, "B" and "D", contribute to the Pre-fetch-Hit Ratio, according to its definition in Equation 6-4. Also since "F" is not in the cache, it will trigger the next pre-fetch.

Pre-fetch-Hit Ratio is an indicator of the performance of the pre-fetching technique. The definition of the Pre-fetch-Hit Ratio is as follows:

Pre-fetch-Hit Ratio (H_p) is the ratio of the total number of *pre-fetch hits* to the total number of anticipated additional UDDI inquiry requests associated with the latest received original UDDI inquiry request:

$$H_p = \frac{h}{m} \qquad \textit{Equation 6-4}$$

The number of pre-fetch hits (h) at time t is the total number of pre-fetch hits of distinct original UDDI inquiry requests between two consecutive cache-missed original UDDI inquiry requests. We can therefore calculate the Pre-fetch-Hit Ratio for the example of Figure 6-3 and Figure 6-4. In this case, the number of pre-fetch hits h is 2 while the Pre-fetching Rate m is 5. Therefore, the Pre-fetch-Hit Ratio H_p is 2/5. Note that the original UDDI inquiry request "B" contributes only once to the Pre-fetch-Hit Ratio, although it gave rise to cache hits twice. The Pre-fetch-Hit Ratio is different from the Cache-Hit Ratio, defined in Section 6.5.3, in a significant way: the numerator in the definition of the Cache-Hit Ratio is the cumulative number of cache hits, regardless of the distinctness of the original UDDI inquiry requests, whereas the numerator in the

definition of the Pre-fetch-Hit Ratio (see Equation 6-4) is the total number of pre-fetch hits for distinct original UDDI inquiry requests. The same original UDDI inquiry requests that are used more than once contribute only once to the total number of pre-fetch hits (see for example, original UDDI inquiry request "B" in Figure 6-4). Because the pre-fetching system is trying to guess the correct sequence of UDDI inquiry requests, so a repeat of the same UDDI inquiry request, while it benefits from the pre-fetching system, does not increase the pre-fetch hits count.

6.5.5 Factors Impacting the Pre-fetch-Hit Ratio H_p and Cache-Hit Ratio H

As described in Section 4.3, in order to understand the impact of the pre-fetching mechanism on the performance of Web Service discovery systems, we need to analyze the three key parameters that can have a high impact on Pre-fetching Rate m, therefore an influence on the Pre-fetching-Hit Ratio H_p and the Cache-Hit Ratio H.

- ***Depth of Conditional Breadth-First Search (D)***: A higher value of this index indicates a deeper breadth-first search in the pre-fetching graph. The deeper the search in the pre-fetching graph, the more eligible UDDI inquiry request nodes will be pre-fetched and sent to the UDDI registry server associated with the original inquiry request (see Section 4.3.4). Hence, the depth of C-BFS D has a significant impact on the Pre-fetching Rate m.

- ***Threshold on Directed Edge Weight ($W_{threshold}$)***: The directed edge in the pre-fetching graph is eligible if and only if the weight of the edge is equal to or greater than this predefined minimum value of weight. The UDDI inquiry request associated with the end node of this eligible directed edge will be the eligible additional pre-fetched UDDI inquiry request. A higher value for this index may lead to a fewer eligible directed edges being found; therefore, the fewer eligible additional UDDI inquiry requests associated with the end node of the eligible directed edges will be pre-fetched and then sent to the UDDI registry server. Hence, the threshold of the weight of directed edge $W_{threshold}$ has a significant impact on the Pre-fetching Rate m.
- ***Time Interval Threshold ($T_{threshold}$)***: The two consecutive original UDDI inquiry requests may be sent by the same client or different clients. The pre-fetching proxy will always keep track of the previously received original UDDI inquiry request. The time interval will be counted after the currently received original UDDI inquiry request arrives. The directed edge from previously received one to currently received one will be eligible if and only if the time interval is equal to or less than $T_{threshold}$. Therefore, the threshold of time interval $T_{threshold}$ has an impact on the pre-fetching graph itself (see Section 4.3.3). In addition, because it is difficult to obtain real data of Web Service registry so we have to create the artificial Web Service discovery applications. Therefore we assume that the

model of inquiry requests arrival rate to caching proxy is a uniform model.

6.6 Relationship among Parameters and Performance Metrics

Table 6-1 shows the list of measurement parameters that were defined in Section 6.5. Table 6-2 shows the performance metrics used in performance analysis of the system. Parameters D, $W_{threshold}$, $T_{threshold}$ and performance metrics EDT, m, h, H_p are specifically defined for this work while the others are well known in caching research. We have assumed both the parameters *Average Size of single UDDI Inquiry Request* S_{req} and *Average Size of single UDDI Inquiry Response* S_{res} to be constants. A UDDI inquiry request object contains only one or more querying criteria, while the UDDI response object contains one or more querying results. However, there is an optional attribute, "*maxRows*", in the UDDI inquiry request, whose value is used to instruct the UDDI registry server to limit the number of results returned. In this experiment, we assume that the attribute "*maxRows*" of all the UDDI inquiry requests is held at the same value so that all UDDI response objects will contain the same number of returned results.

The experiments measure and analyze the interactions among Effective Discovery Time EDT, Cache-Hit Ratio H, Pre-fetch-Hit Ratio H_p, Pre-fetching Rate m, Number of Applications A, Number of Web Service Inquiry Requests in Single Application n and Overlap Rate of Web Service Inquiry Requests between Applications v. Moreover, it is necessary to measure and analyze the impact of Depth of C-BFS D, Threshold on

Directed Edge Weight $W_{threshold}$ and Time Interval Threshold $T_{threshold}$ on the Pre-fetching Rate m, the Effective Discovery Time and Cache-Hit Ratio H. The interaction among various parameters and performance metrics is captured in Figure 6-5.

Parameters	Symbol
Number of Applications	A
Number of Web Service Inquiry Requests in Single Application	n
Overlap Rate of Web Service Inquiry Requests between Applications	v
Average Size of Single UDDI Inquiry Request Object	S_{req}
Average Size of Single UDDI Response Object	S_{res}
Parameter Representing the Sequence of UDDI Inquiry Requests in Experiments	t
Depth of C-BFS	D
Threshold on Edge Weight	$W_{threshold}$
Time Interval Threshold (ms)	$T_{threshold}$

Table 6-1 Parameters – Impact of on Performance

Metrics	Symbol	Equation
Cache-Hit Ratio (%)	H	$H = \frac{\#\ of\ cache\ hits}{A \times n}$
Pre-fetching Rate	m	-
Number of Pre-fetch hits	h	-
Pre-fetch-Hit Ratio (%)	H_p	$H_p = \frac{h}{m}$
Effective Discovery Time (ms)	EDT	$EDT = \sum_{i=1}^{n} r_i$
Response Time (ms)	r	-

Table 6-2 Metrics – Performance Measurement

Figure 6-5 shows two target metrics used in this research: (1) Response Time r used

to evaluate the benefit gained from the proposed pre-fetching technique for integrating with caching on Web Service registry; (2) the Pre-fetching-Hit Ratio that reflects the pre-fetching accuracy that is related to the cost associated with the proposed approach. The Response Time can be directly influenced by Cache-Hit Ratio while the Pre-fetching-Hit Ratio can be directly affected by middleware metrics Pre-fetching Rate. Higher Cache-Hit Ratio results in more original UDDI inquiry requests finding responses in the caching proxy that can shorten the Response Time. According to the definition of Pre-fetching-Hit Ratio in Equation 6-4, a higher Pre-fetching Rate tends to reduce the Pre-fetching-Hit Ratio.

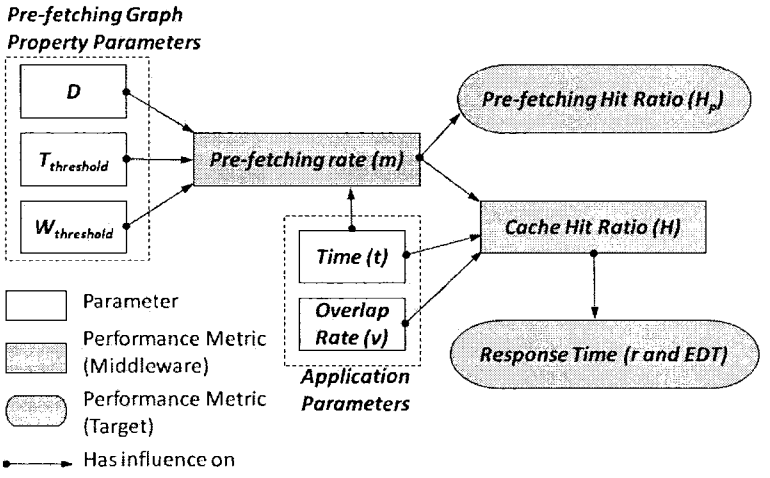

Figure 6-5 Relationship of Influence of Various Parameters on Metrics

The Cache-Hit Ratio and Pre-fetching Rate are both middleware performance metrics that are affected by varying parameters associated with the pre-fetching graph and by the other middleware metrics. For example, Pre-fetching Rate is directly affected by varying the pre-fetching graph property parameters D, $W_{threshold}$, $T_{threshold}$ and time t while it has

influences on Cache-Hit Ratio and Pre-fetching-Hit Ratio. A higher Pre-fetching Rate results in more additional UDDI inquiry requests being pre-fetched from the pre-fetching proxy and cached in the caching proxy, leading to a higher the Cache-Hit Ratio. Also the Pre-fetching Rate changes with time t. This is because that the number of edges, the weight of edges and the number of nodes in pre-fetching graph all increase over time. For any given set of parameters D, $W_{threshold}$ and $T_{threshold}$, the Pre-fetching Rate can increase over time as more and more edges are eligible over time, therefore more additional UDDI inquiry requests associated with the end node of the eligible edges can be pre-fetched.

In addition to m, v affects Cache-Hit Ratio. The application overlap rate influences the Cache-Hit Ratio over time t because a higher overlap rate means more common UDDI inquiry requests among Web Service applications, Therefore, there is a greater chance that a subsequent UDDI inquiry request can be hit in the caching proxy.

We have presented a brief discussion of the chain of influence among various parameters and metrics in this section. A more detailed analysis based on experimental results will be presented in the next chapter.

6.7 Summary

This chapter formally defined various parameters and performance metrics used in the experiments to demonstrate the improvement provided by the proposed approach for integrating the pre-fetching proxy and the caching proxy on the performance of Web

Service discovery systems. An analysis of the relationship among these parameters and performance metrics were also discussed in this chapter. The next chapter focuses on the experimental results.

Chapter 7: Experimental Results

7.1 Introduction

In this chapter, the experimental results that illustrate the relationship between Cache-Hit Ratio (H), Pre-fetch-Hit Ratio (H_p), Pre-fetching Rate (m), Response Time (r), EDT and UDDI inquiry requests over Time (t). In addition, we show diagrams that illustrate the relationship between Pre-fetch-Hit Ratio (H_p) and Cache-Hit Ratio (H), Cache-Hit Ratio (H) and Threshold on Edge Weight ($W_{threshold}$), Cache-Hit Ratio (H) and Search Depth of Conditional Breadth-First Search (C-BFS) (D), Cache-Hit Ratio (H) and Time Interval Threshold ($T_{threshold}$), Cache-Hit Ratio (H) and Overlap Rate of Applications (v). All of these experiments were conducted using three different experimental settings: both the caching proxy and pre-fetching proxy turned on (YCYP); only the caching proxy turned on (YCNP); and both the caching proxy and pre-fetching proxy turned off (NCNP).

7.2 Evolution of Cache-Hit Ratio (H) over Time (t)

7.2.1 Purpose of the Experiment

As described in Section 6.5.3, a higher value of the Cache-Hit Ratio H indicates that more original UDDI inquiry requests will encounter low latency, less workload in the remote UDDI registry server and less traffic through the networks.

In Section 6.5.5, we introduced three key parameters that have significant effects on the Pre-fetching Rate m, and therefore have a further influence on the Pre-fetch-Hit Ratio H_p and Cache-Hit Ratio H. These three parameters are Depth of Conditional Breadth-First Search D, Threshold on Edge Weight $W_{threshold}$ and Time Interval Threshold $T_{threshold}$. The experimental results presented in this section will show the impact of UDDI inquiry requests over time on the Cache-Hit Ratio for various values of D, $W_{threshold}$ and $T_{threshold}$. In Section 7.2.2.1 we analyze the influence of UDDI inquiry requests over time on the Cache-Hit Ratio for different values of Threshold on Edge Weight. The influence of UDDI inquiry requests over time on the Cache-Hit Ratio, for different values of Depth of Conditional Breadth-First Search, is then discussed in Section 7.2.2.2. The impact of Time Interval Threshold on performance is analyzed in Section 7.2.2.3. Finally, in Section 7.2.2.4, we analyze the influence of variations in Overlap Rate of Web Service Inquiry Requests between Applications (v) on the Cache-Hit Ratio.

7.2.2 Experimental Results and Analysis

7.2.2.1 The Effect of Varying the Threshold on Edge Weight ($W_{threshold}$)

Figure 7-1 shows the measured influence of UDDI inquiry requests on the Cache-Hit Ratio for different values of $W_{threshold}$ with the other parameters D, $T_{threshold}$ and v held at fixed values. The X-axis shows the UDDI inquiry requests over time while the Y-axis shows the mean of the measured Cache-Hit Ratio. Because $W_{threshold}$ is a property of the

pre-fetching proxy, the measurement has to be done using the experimental model "YCYP." There are five curves in this figure; the top four curves correspond to $W_{threshold}$=2, $W_{threshold}$=5, $W_{threshold}$=10 and $W_{threshold}$=15, while the bottom one corresponds to the experimental model "YCNP."

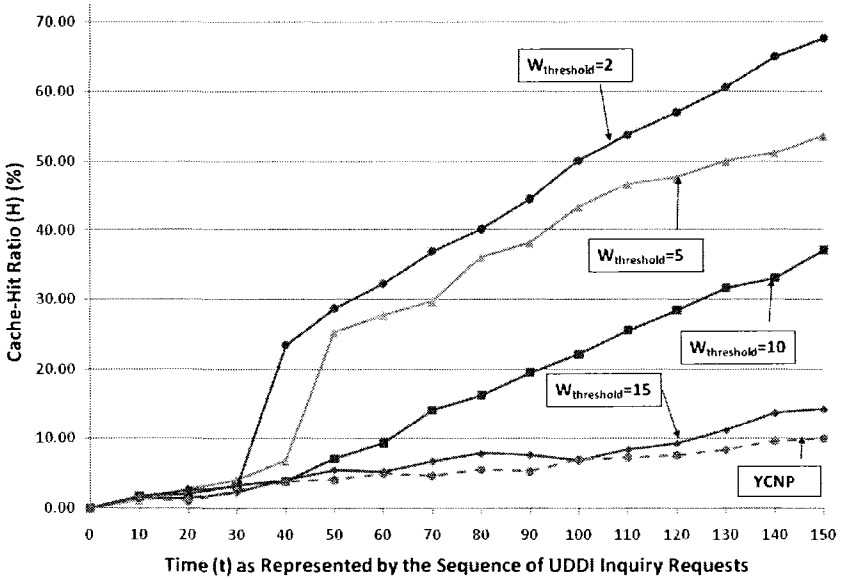

Figure 7-1 Influence of UDDI Inquiry Requests Over Time (t) on Cache-Hit Ratio (H) – Variations in $W_{threshold}$ (YCYP, $T_{threshold}$=10000ms, v=10%, D=2)

As shown in Figure 7-1, varying $W_{threshold}$ influences the Cache-Hit Ratio. For any given Threshold on Edge Weight, the Cache-Hit Ratio always increases over time. This is because there are more and more UDDI inquiry requests and the responses associated with them are cached in the caching proxy over time. The more UDDI inquiry requests and responses cached in the caching proxy, the greater the chance that subsequent UDDI inquiry requests will be hits in the caching proxy.

The Cache-Hit Ratio has a higher value when $W_{threshold}$ has a lower value. Moreover, a greater threshold value results in a curve that is closer to the "YCNP" curve; for example,

the curve when $W_{threshold}$=15 almost overlaps with the dashed curve in "YCNP." This is because the weight threshold indicates the minimum acceptable weight of the eligible directed edge in the pre-fetching graph. A higher weight threshold means that fewer directed edges will be eligible; therefore, fewer eligible additional UDDI inquiry requests associated with the end node of the eligible directed edges will be pre-fetched. A lower $W_{threshold}$ increases pre-fetching in the system and there is a greater chance that subsequent UDDI inquiry requests will be hits in the caching proxy. No eligible directed edge in the pre-fetching graph will be found and no additional UDDI inquiry requests will be pre-fetched when the weight threshold is very high. In this case, although the experiment is run using the "YCYP" experimental model, the pre-fetching proxy will pre-fetch nothing and simply forward the currently received UDDI inquiry request to the UDDI registry server without pre-fetching, similar to the way it occurs in the "YCNP" experimental model.

Figure 7-1 shows that the Cache-Hit Ratio H for $W_{threshold}$=2 and $W_{threshold}$=5 rises rapidly immediately after the initial process phase of the pre-fetching graph. However, the increase in H is less sharp for $W_{threshold}$=10 and $W_{threshold}$=15. The initial process phase of the pre-fetching graph is defined as the duration between the addition of the first UDDI inquiry request to the pre-fetching graph and the pre-fetching of the first additional UDDI inquiry request. The pre-fetching proxy only builds and maintains the pre-fetching graph according to the currently received UDDI inquiry request, and no additional UDDI

inquiry request will be pre-fetched during the initial process phase of the pre-fetching graph. The data shown in Table 7-1 presents an approximately 20 percentage points enhancement of the Cache-Hit Ratio immediately after the initial process phase of the pre-fetching graph when the weight threshold has a value of 2 (see boxed regions of Table 7-1). However, the enhancement of the Cache-Hit Ratio immediately after the initial process phase of the pre-fetching graph declines to approximately 18 percentage points when the weight threshold has a value of 5 and to approximately 4 percentage points when the weight threshold has a value of 10. Further, it is difficult to observe the enhancement of the Cache-Hit Ratio immediately after the initial process phase of the pre-fetching graph when the weight threshold has a value of 15. The reasonable explanation for this is that with a higher value of $W_{threshold}$, the pre-fetching operation starts later on the system and fewer additional UDDI inquiry requests are pre-fetched, thus reducing the slope of the curve (see Figure 7-1).

Figure 7-2 clearly illustrates the effect of the Threshold on Edge Weight on the Cache-Hit Ratio in the "YCYP" experimental model, when the other parameters are constants. There are three curves in this figure, corresponding to time point t=150, t=80 and t=10.

The bottommost curve indicates the effect of the weight threshold on the Cache-Hit Ratio when t=10. This curve keeps a very low Cache-Hit Ratio value, regardless of the changes in weight threshold. This is because t=10 is the beginning of the initial process

phase of the pre-fetching graph. During this phase, the pre-fetching proxy only builds and maintains the pre-fetching graph according to the most recently received UDDI inquiry requests, and no additional UDDI inquiry request will be pre-fetched. Therefore, the weight threshold has little effect on the Cache-Hit Ratio, as no pre-fetching takes place, and the enhancement of the Cache-Hit Ratio depends only on the contribution of the common UDDI inquiry requests within the Web Service application itself.

Time(t)	H-YCYP (%) ($W_{threshold}$=2)	H-YCYP (%) ($W_{threshold}$=5)	H-YCYP (%) ($W_{threshold}$=10)	H-YCYP (%) ($W_{threshold}$=15)
0	0.00	0.00	0.00	0.00
10	1.66	1.12	1.67	1.45
20	2.59	2.78	2.06	1.34
30	3.07	4.09	3.33	2.28
40	23.45	6.80	3.89	4.07
50	28.67	25.34	7.07	5.49
60	32.25	27.67	9.29	5.23
70	36.96	29.68	14.07	6.76
80	40.09	36.04	16.19	7.81
90	44.51	38.12	19.48	7.66
100	50.07	43.36	22.08	6.84
110	53.76	46.64	25.49	8.39
120	56.97	47.59	28.41	9.28
130	60.57	50.03	31.59	11.13
140	65.06	51.22	33.08	13.66
150	67.56	53.62	37.00	14.20

Table 7-1 Cache-Hit Ratio (H) vs. UDDI Inquiry Requests Over Time (t) – Variations in $W_{threshold}$ (YCYP, $T_{threshold}$=10000ms, v=10%, D=2)

For both t=80 and t=150, a higher weight threshold leads to a lower Cache-Hit Ratio. The result shown in this figure is consistent with that derived from Figure 7-1 that was discussed earlier in this section.

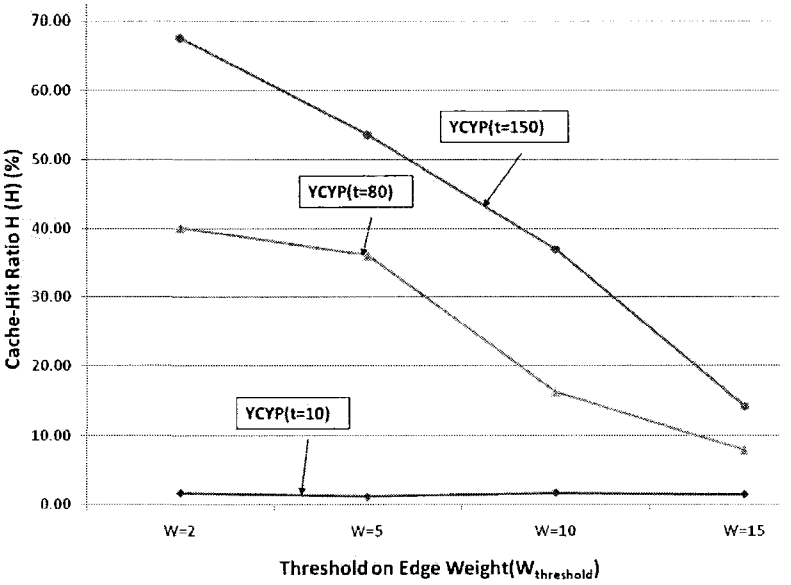

Figure 7-2 Influence of Threshold on Edge Weight ($W_{threshold}$) on Cache-Hit Ratio (H) (YCYP, $T_{threshold}$=10000ms, D=2, v=10%)

It is clear that the increase in weight threshold has a more significant influence on a Cache-Hit Ratio with a higher value of *t*. For example, a Cache-Hit Ratio with t=150 drops rapidly, by about 50 percentage points, when the weight threshold increases from $W_{threshold}$=2 to $W_{threshold}$=15, whereas a Cache-Hit Ratio with t=80 drops by only about 30 percentage points when the weight threshold increases from $W_{threshold}$=2 to $W_{threshold}$=15. A Cache-Hit Ratio with t=10 stays almost constant when the weight threshold increases from $W_{threshold}$=2 to $W_{threshold}$=15. Therefore, variation in weight threshold has a stronger influence on a Cache-Hit Ratio with a higher value of *t* when the pre-fetching graph has been able to record more system activities.

7.2.2.2 The Effect of Varying the Depth of C-BFS (*D*)

Figure 7-3 shows the influence of UDDI inquiry requests on the Cache-Hit Ratio when

the Depth of Conditional Breadth-First Search (D) parameter is varied and the other parameters, Threshold on Edge Weight, Time Interval Threshold and the Overlap Rate of Web Service Inquiry Requests between Applications, are held at fixed values. The X-axis indicates UDDI inquiry requests over time while the Y-axis shows the mean of the measured Cache-Hit Ratio. Because D is a property of the pre-fetching proxy, the measurement has to be done using the "YCYP" experimental model. There are four curves in this figure that correspond to D=5, D=4, D=3 and D=2.

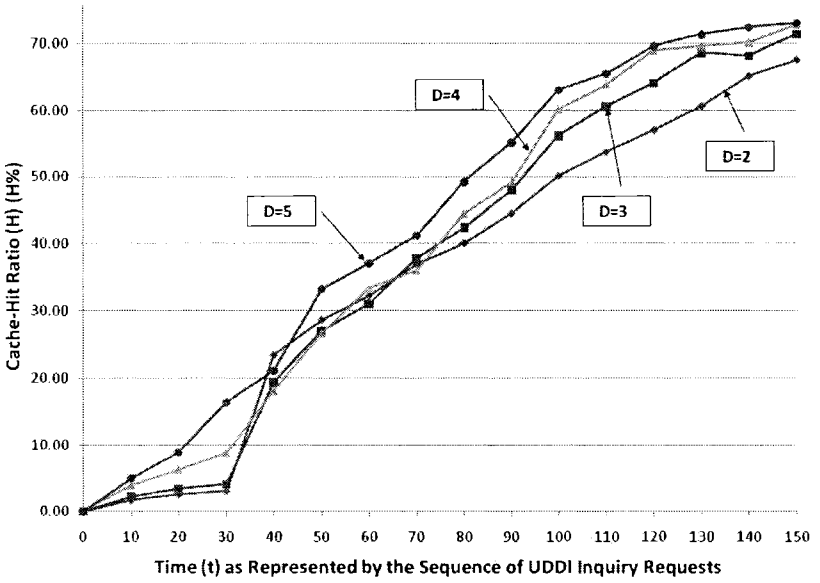

Figure 7-3 Influence of UDDI Inquiry Requests Over Time (t) on Cache-Hit Ratio (H) – Variations in D (YCYP, $T_{threshold}$=10000ms, v=10%, $W_{threshold}$=2)

As shown in Figure 7-3, varying the Depth of Conditional Breadth-First Search affects the Cache-Hit Ratio. The Cache-Hit Ratio tends to have a higher value when D has a higher value, because a higher value in this parameter indicates a deeper breadth-first search in the pre-fetching graph. With a deeper search in the pre-fetching graph, more eligible UDDI inquiry request nodes will be pre-fetched, and therefore more

additional UDDI inquiry requests and responses associated with them will be cached in the caching proxy. This increases the chance that the subsequent UDDI inquiry requests will be hit in the caching proxy. Figure 7-3 and Table 7-2 capture the non-linear relationship between *H* and *D*.

Time(t)	H-YCYP (%) (D=2)	H-YCYP (%) (D=3)	H-YCYP (%) (D=4)	H-YCYP (%) (D=5)
0	0.00	0.00	0.00	0.00
10	1.66	2.23	4.02	5.05
20	2.59	3.45	6.33	8.92
30	3.07	4.12	8.92	16.44
40	23.45	19.33	18.04	21.17
50	28.67	27.00	26.72	33.22
60	32.25	31.03	33.36	37.09
70	36.96	37.87	36.07	41.19
80	40.09	42.38	44.45	49.20
90	44.51	48.02	49.22	55.09
100	50.07	56.11	60.09	63.00
110	53.76	60.50	63.84	65.41
120	56.97	64.05	68.88	69.53
130	60.57	68.56	69.58	71.26
140	65.06	68.12	70.03	72.39
150	67.56	71.30	72.78	72.98

Table 7-2 Cache-Hit Ratio (H) vs. UDDI Inquiry Requests Over Time (t) – Variations in D (YCYP, $T_{threshold}$=10000ms, v=10%, $W_{threshold}$=2)

Figure 7-4 illustrates the effect of *D* on the Cache-Hit Ratio in the "YCYP" experimental model when the other parameters are constants. There are three curves in this figure, corresponding to the time points t=150, t=80, and t=10.

As shown in Figure 7-4, varying the *D* influences the Cache-Hit Ratio *H*. All three curves have the same trend; i.e., the Cache-Hit Ratio *H* goes up when the search depth

increases.

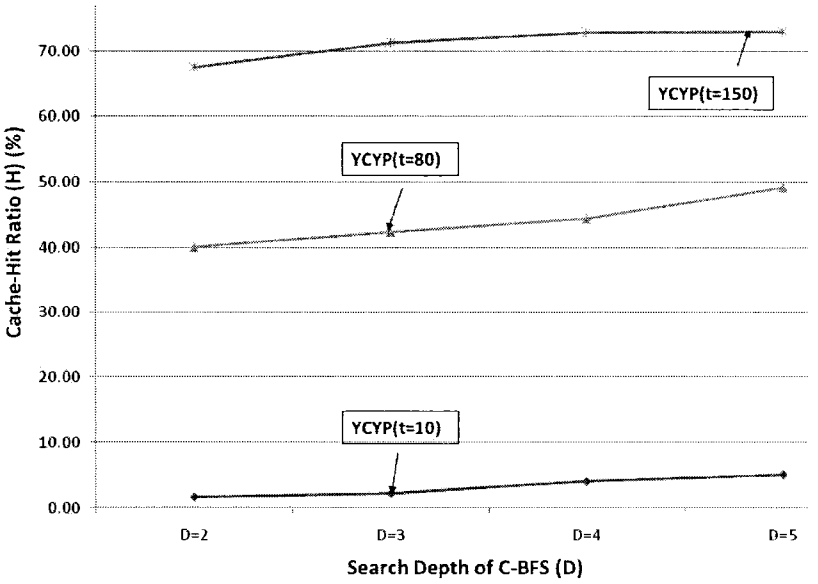

Figure 7-4 Influence of Searching Depth of C-BFS (D) on Cache-Hit Ratio (H) (YCYP, $T_{threshold}$=10000ms, $W_{threshold}$=2, v=10%)

7.2.2.3 The Effect of Varying the Time Interval Threshold ($T_{threshold}$)

Figure 7-5 shows the influence of UDDI inquiry requests on the Cache-Hit Ratio for various values of Time Interval Threshold ($T_{threshold}$) with the other parameters D, $W_{threshold}$ and v held at fixed values. The X-axis indicates UDDI inquiry requests over time while the Y-axis shows the mean of the measured Cache-Hit Ratio. Because $T_{threshold}$ is a property of the pre-fetching proxy, the measurement has to be done using the "YCYP" experimental model. There are five curves in this figure; the top four curves correspond to $T_{threshold}$=10000ms, $T_{threshold}$=5000ms, $T_{threshold}$=500ms and $T_{threshold}$=50ms, while the bottom one corresponds to the "YCNP" experimental model.

As shown in Figure 7-5, varying the Time Interval Threshold influences the

Cache-Hit Ratio. For any given the Time Interval Threshold, the Cache-Hit Ratio always increases over time. This is similar to the curves in Figure 7-1 and Figure 7-3 and was discussed in Section 7.2.2.1.

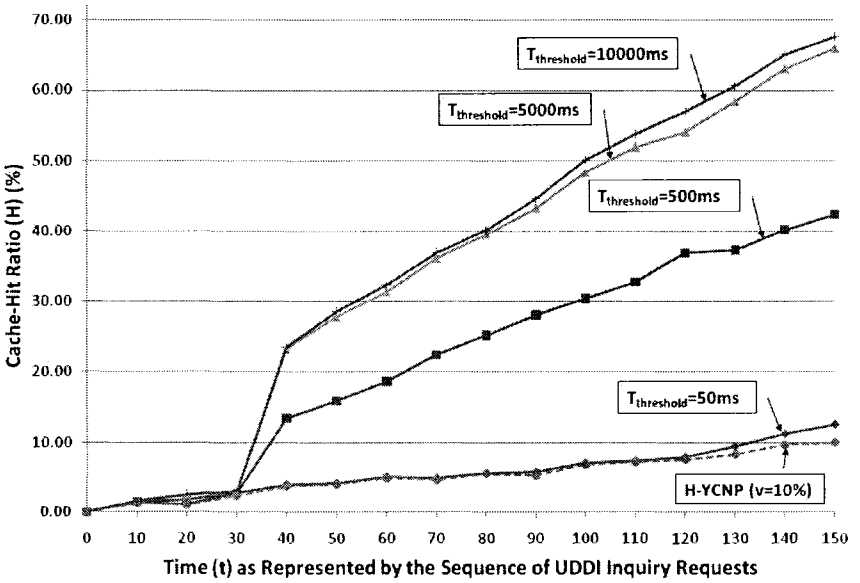

Figure 7-5 Influence of UDDI Inquiry Requests Over Time (t) on Cache-Hit Ratio (H) – Variations in $T_{threshold}$ (YCYP, v=10%, $W_{threshold}$=2, D=2)

The Cache-Hit Ratio H has a higher value when the Time Interval Threshold ($T_{threshold}$) has a higher value. Moreover, a Cache-Hit Ratio with a higher Time Interval Threshold increases at a rate faster in comparison to a system with a lower Time Interval Threshold. In addition, a lower Time Interval Threshold results in a curve that is closer to the "YCNP" curve; for example, a curve with $T_{threshold}$=50ms almost overlaps with the curve of "YCNP." $T_{threshold}$ indicates the maximum acceptable value of the time interval in milliseconds between two consecutively received original UDDI inquiry requests to add a directed edge (or increase the weight of an existing edge) to the pre-fetching graph. A lower Time Interval Threshold causes the pre-fetching proxy to ignore more edges. The

total number of edges in the pre-fetching graph therefore falls significantly. There is no directed edge in the pre-fetching graph and all the nodes in the pre-fetching graph are disconnected when the Time Interval Threshold decreases to a very large value. Therefore, no additional UDDI inquiry request will be pre-fetched. In this case, although the experiment is run using the "YCYP" experimental model, the pre-fetching proxy will pre-fetch nothing and simply forward the current received UDDI inquiry request to the UDDI registry server without pre-fetching. That is why the curve where $T_{threshold}$ =50ms in Figure 7-5 is close to the curve for the YCNP experimental model.

Time(t)	H-YCYP (%) ($T_{threshold}$=50ms)	H- YCYP (%) ($T_{threshold}$=500ms)	H- YCYP (%) ($T_{threshold}$=5000ms)	H- YCYP (%) ($T_{threshold}$=10000ms)
0	0.00	0.00	0.00	0.00
10	1.46	1.51	1.58	1.66
20	1.19	1.83	2.39	2.59
30	2.88	2.81	3.01	3.07
40	4.02	13.45	23.22	23.45
50	4.25	15.97	27.86	28.67
60	5.13	18.66	31.34	32.25
70	4.95	22.46	36.09	36.96
80	5.67	25.20	39.47	40.09
90	5.89	28.11	43.19	44.51
100	7.19	30.41	48.29	50.07
110	7.45	32.78	51.87	53.76
120	7.98	36.90	54.01	56.97
130	9.54	37.31	58.31	60.57
140	11.22	40.19	62.97	65.06
150	12.56	42.33	65.88	67.56

Table 7-3 Cache-Hit Ratios (H) vs. UDDI Inquiry Requests over Time (t) – Variations in $T_{threshold}$ (YCYP, v=10%, $W_{threshold}$=2, D=2)

As shown in Figure 7-5 and Table 7-3, for $T_{threshold}$=500ms, 5000ms and 10000ms

there is a sharp rise in H between t=30 and t=40. H is observed to increase at a smaller rate for higher values of t. Another interesting result observed in Figure 7-5 is that the curve where $T_{threshold}$ =5000ms is close to the curve where $T_{threshold}$ =10000ms. Increasing $T_{threshold}$ beyond 5000ms seems to produce only a small increase in H.

Figure 7-6 illustrates the effect of $T_{threshold}$ on the Cache-Hit Ratio in the "YCYP" experimental model when the other parameters are constants. There are three curves in this figure, corresponding to the time points t=150, t=80 and t=10.

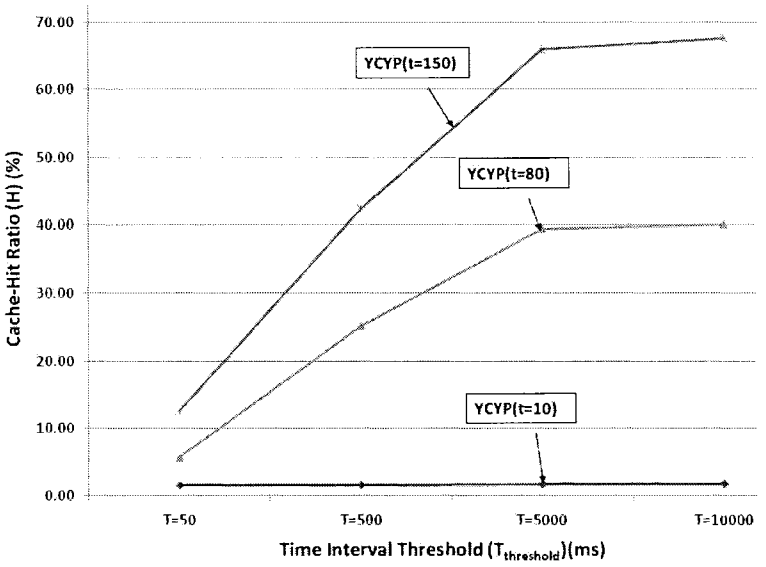

Figure 7-6 Influence of Time Interval Threshold ($T_{threshold}$) on Cache-Hit Ratio (H) (YCYP, D=2, $W_{threshold}$ =2, v=10%)

As shown in Figure 7-6, varying the Time Interval Threshold ($T_{threshold}$) influences the Cache-Hit Ratio H. The top two curves have the same trend: the Cache-Hit Ratio goes up initially when the Time Interval Threshold increases and then tends to flatten off around $T_{threshold}$=10000ms. However, the lowest curve, which shows the effect of the Time Interval Threshold on the Cache-Hit Ratio when t=10, keeps a very low, almost constant

of Cache-Hit Ratio value, regardless of the changes in the Time Interval Threshold. This observation is similar to that discussed in the context of Figure 7-2 and was in Section 7.2.2.1.

7.2.2.4 The Effect of Varying the Application Overlap Rate (v)

Figure 7-7 shows the influence of UDDI inquiry requests on the Cache-Hit Ratio H when the Overlap Rate of Web Service Inquiry Requests between Applications (v) is varied with the other parameters D, $T_{threshold}$ and the $W_{threshold}$ held at fixed values. The X-axis indicates UDDI inquiry requests over time while the Y-axis shows the mean of measured Cache-Hit Ratio H. Because v is a property of the Web Service application, the measurements are done using all the "NCNP," "YCNP" and "YCYP" experimental models. There are nine curves in this figure; the top four curves correspond to v=10%, v=15%, v=20% and v=25% in the "YCYP" experimental model, while the lowest four dashed curves correspond to v=10%, v=15%, v=20% and v=25% in the "YCNP" experimental model. The bottommost straight line on the X-axis corresponds to the "NCNP" experimental model.

As shown in Figure 7-7, varying v does not influence the Cache-Hit Ratio when the "NCNP" experimental model is used. This is because the caching proxy never caches any UDDI inquiry request and response associated with it when the "NCNP" experimental model is used. Therefore the Cache-Hit Ratio is always zero and all the UDDI inquiry

requests have to be forwarded to the pre-fetching proxy.

However, varying v influences the Cache-Hit Ratio H when both the "YCNP" and "YCYP" experimental models are used. For any given overlap rate and experimental model, the Cache-Hit Ratio always increases over time. However, after the initial process phase of the pre-fetching graph, the Cache-Hit Ratio of the "YCYP" experimental model increases much faster than the Cache-Hit Ratio of the "YCNP" experimental model with the same overlap rate. For v=25% for example, the gradient of the curve in the "YCYP" experimental model is steeper than that of the curve in the "YCNP" experimental model. This is because only the time factor enhances the Cache-Hit Ratio in the "YCNP" experimental model, whereas the enhancement of the Cache-Hit Ratio in the "YCYP" experimental model is affected by not only the time factor, but also the pre-fetching factor. In the "YCYP" experimental model, additional UDDI inquiry requests are pre-fetched and responses associated with them are cached in the caching proxy after the initial process phase of the pre-fetching graph. The additional UDDI inquiry requests and responses yield a higher chance for the subsequent UDDI inquiry requests to be a hit in the caching proxy.

In both the "YCNP" and "YCYP" experimental models the Cache-Hit Ratio increases when v is increased. Note that the overlap rate indicates the percentage of UDDI inquiry requests common among the different Web Service applications. Thus, with a higher overlap rate, there is a greater chance that the subsequent UDDI inquiry requests will

generate hits in the caching proxy.

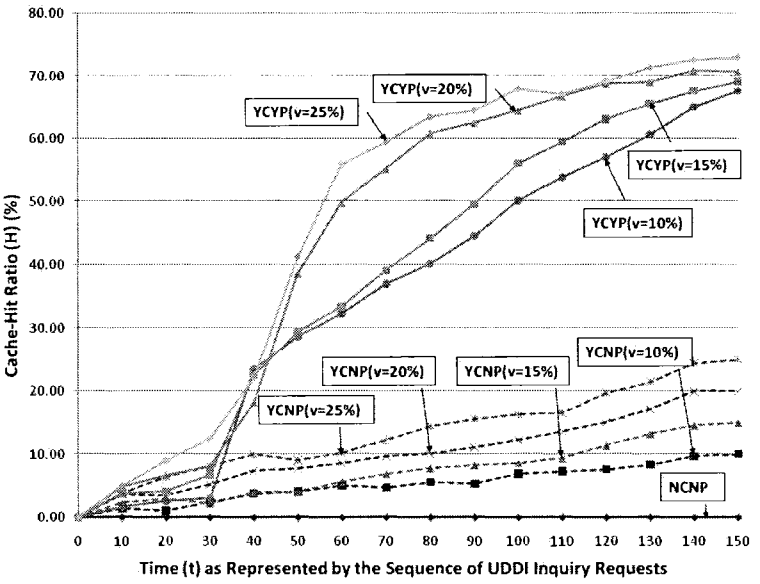

Figure 7-7 Influence of UDDI Inquiry Requests Over Time (t) on Cache-Hit Ratio (H) – Variations in v (YCYP, $T_{threshold}$=10000ms, $W_{threshold}$=2, D=2 & YCNP & NCNP)

The Cache-Hit Ratio in the "YCNP" experimental model finally reaches the overlap rate, such as the curve for v=25% in the "YCNP" experimental model reaches the 25% Cache-Hit Ratio when t=150 (see Figure 7-7).

Figure 7-8 and Table 7-4 capture the effect of *v* on *H* when the "NCNP," "YCNP" and "YCYP" experimental models are used. There are five curves in this figure; the solid lines correspond to t=150 and t=20 under the "YCYP" experimental model, while the dashed lines correspond to t=150 and t=20 in the "YCNP" experimental model. The dotted line on the X-axis corresponds to the curve of the "NCNP" experimental model.

As shown in Figure 7-8, varying *v* influences the Cache-Hit Ratio *H*. All curves have the same trend, i.e. that the Cache-Hit Ratio goes up when the overlap rate increases, except for the "NCNP" curve; as expected the Cache-Hit Ratio of this experimental

model keeps a constant value at zero irrespective to the overlap rate.

Time	H-NCNP	H-YCNP (%)				H-YCYP (%)			
(t)	(%)	v=10%	v=15%	v=20%	v=25%	v=10%	v=15%	v=20%	v=25%
0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
10	0.00	1.45	2.28	3.66	3.89	1.66	3.78	4.88	5.02
20	0.00	1.06	2.98	3.44	6.39	2.59	4.09	6.67	8.93
30	0.00	2.48	2.22	5.22	8.15	3.07	6.77	8.11	12.44
40	0.00	3.78	3.99	7.38	9.91	23.45	22.49	18.18	22.23
50	0.00	4.12	4.05	7.71	9.08	28.67	29.48	38.59	41.19
60	0.00	4.98	5.58	8.55	10.19	32.25	33.31	49.74	55.83
70	0.00	4.69	6.91	9.66	12.24	36.96	39.06	55.07	59.37
80	0.00	5.56	7.77	10.04	14.44	40.09	44.11	60.67	63.43
90	0.00	5.33	8.29	11.11	15.58	44.51	49.48	62.49	64.47
100	0.00	6.92	8.58	12.24	16.28	50.07	55.93	64.39	67.87
110	0.00	7.27	9.36	13.59	16.54	53.76	59.41	66.73	67.13
120	0.00	7.59	11.33	15.06	19.69	56.97	63.03	68.77	69.12
130	0.00	8.39	13.22	17.18	21.44	60.57	65.52	69.05	71.29
140	0.00	9.68	14.48	19.97	24.38	65.06	67.49	70.78	72.45
150	0.00	10.00	15.00	20.00	25.00	67.56	69.03	70.67	72.88

Table 7-4 Cache-Hit Ratios (H) vs. UDDI Inquiry Requests Over Time (t) – Variations in v (YCYP, $T_{threshold}$=10000ms, $W_{threshold}$=2, D=2 & YCNP & NCNP)

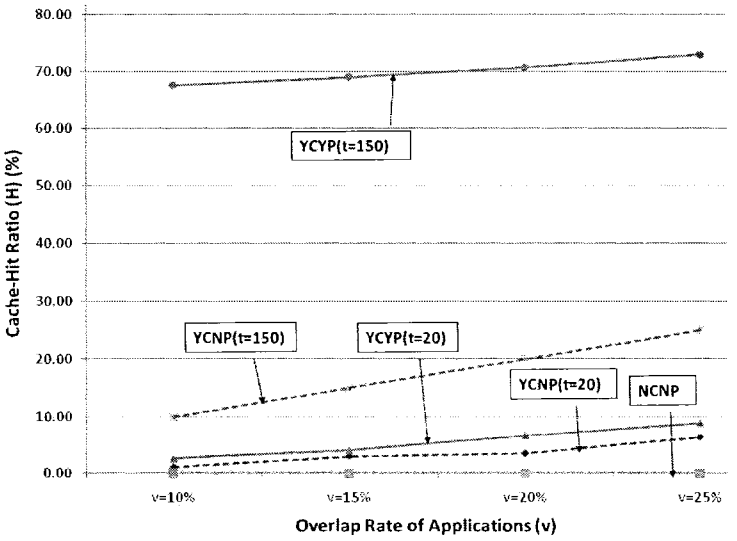

Figure 7-8 Influence of Overlap Rate of Applications (v) on Cache-Hit Ratio (H) (YCYP, D=2, $W_{threshold}$=2, $T_{threshold}$=10000ms Vs. YCNP Vs. NCNP)

7.2.2.5 Influence of Pre-fetching Rate (*m*) on Cache-Hit Ratio (*H*)

Figure 6-5 shows that the pre-fetching graph parameters D, $W_{threshold}$ and $T_{threshold}$ affect the performance metrics Cache-Hit Ratio through the middleware metric m. This section discusses the relationship between the Pre-fetching Rate and the Cache-Hit Ratio.

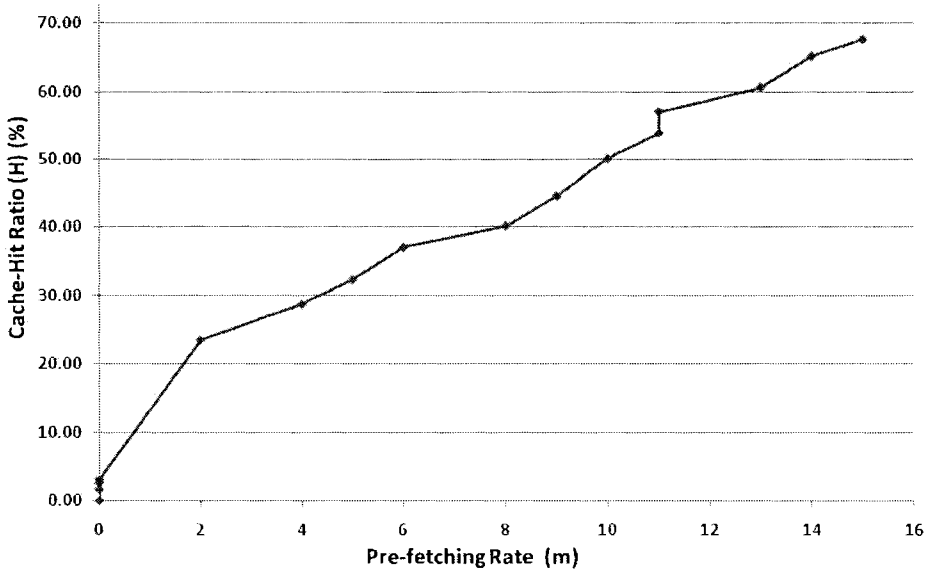

Figure 7-9 Relationship between Pre-fetching Rate and Cache-Hit Ratio (YCYP, $W_{threshold}$=2, D=2, $T_{threshold}$=10000ms, v=10%)

Figure 7-9 shows the influence of Pre-fetching Rate on the Cache-Hit Ratio when v=10%, D=2, $T_{threshold}$=10000ms and the $W_{threshold}$=2. As shown in this figure, increasing pre-fetching rate results in raising the Cache-Hit Ratio. A higher Pre-fetching Rate m corresponds to more additional UDDI inquiry requests being pre-fetched in the pre-fetching proxy and cached in caching proxy, leading to an increase in Cache-Hit Ratio. It is interesting to note that H increases with m sharply at first. However the rate of increase seems to be smaller at higher values of m.

7.2.3 Summary

In this section, we discussed the impact of UDDI inquiry requests on Cache-Hit Ratio for various values of D, $W_{threshold}$, $T_{threshold}$ and v. The proposed approach improves performance by increasing the Cache-Hit Ratio. In addition, varying the D, $W_{threshold}$, $T_{threshold}$ and v parameters influences system performance of the proposed approach. Higher values of Depth of Conditional Breadth-First Search, Time Interval Threshold and Overlap Rate of Web Service Inquiry Requests among Applications tend to improve the Cache-Hit Ratio, whereas a lower Threshold on Edge Weight seems to increase it.

7.3 Evolution of Pre-fetching-Rate (m) Over Time (t)

7.3.1 Purpose of the Experiment

In this section, we present the evolution of the Pre-fetching Rate m over Time t as represented by the sequence of UDDI inquiry requests. A higher value of Pre-fetching Rate indicates that more additional UDDI inquiry requests will be pre-fetched and responses associated with them cached in the caching proxy, leading to a greater chance that the subsequent UDDI inquiry request will be a hit in the caching proxy. In the previous section, we discussed the effect of UDDI inquiry requests over Time t on the Cache-Hit Ratio H for different parameter values. A significant part of the improvement in the Cache-Hit Ratio when the parameters are changed can be attributed to an increase

in the number of pre-fetched UDDI inquiry requests in the pre-fetching proxy. D, $W_{threshold}$, $T_{threshold}$ and v influence m that in turn influences H. For conservation of space, we present a discussion of the effect of $W_{threshold}$ and v on m in Section 7.3.2.1 and Section 7.3.2.2, respectively. Discussion of the influence of D and $T_{threshold}$ on m is similar to the discussion of the relationship between D and H presented in Section 7.2.2.2, and between $T_{threshold}$ and H presented in Section 7.2.2.3 respectively. The graphs corresponding to the experiments investigating the relationship between each of these parameters and m are presented in Appendix C.

7.3.2 Experimental Results and Analysis

7.3.2.1 The Effect of Varying the Threshold on Edge Weight ($W_{threshold}$)

Figure 7-10 shows the evolution of the Pre-fetching Rate as represented by the sequence of UDDI inquiry requests. The Threshold on Edge Weight is varied while the other parameters D, $T_{threshold}$ and v are held constant. The experiment is run for the "YCYP" experimental model because the Pre-fetching Rate is relevant only when the pre-fetching proxy is enabled. The X-axis indicates the UDDI inquiry requests over time while the Y-axis shows the Pre-fetching Rate.

As shown in Figure 7-10, varying $W_{threshold}$ influences the Pre-fetching Rate. For any given Threshold on Edge Weight, the Pre-fetching Rate increases after a certain t value that corresponds to the completion of the initial process phase for the pre-fetching graph.

Figure 7-10 also shows that a lower $W_{threshold}$ leads to a higher Pre-fetching Rate. Higher the value of this parameter, higher is the value of t at which m starts increasing from a zero value. For example, for $W_{threshold}$=2 this occurs at t=30, while for $W_{threshold}$ =15 at t=110. This is because the weight threshold indicates the minimum acceptable weight of the edge required for pre-fetching to occur. Thus a higher $W_{threshold}$ leads to a longer time before pre-fetching starts on the system.

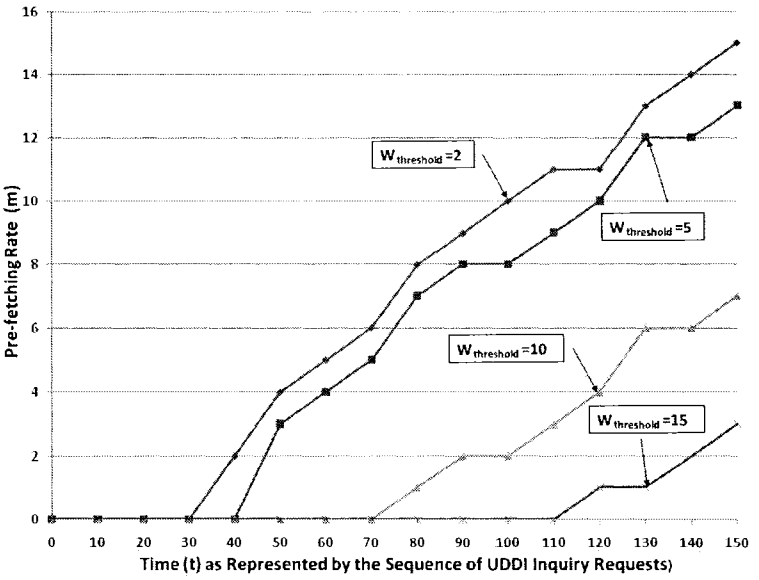

Figure 7-10 Evolution of Pre-fetching Rate (m) Over Time (t) as represented by the sequence of UDDI Inquiry Requests – Variations in $W_{threshold}$ (YCYP, $T_{threshold}$ =10000ms, v=10%, D=2)

7.3.2.2 The Effect of Varying the Application Overlap Rate (v)

Figure 7-11 shows the evolution of the Pre-fetching Rate as represented by the sequence of UDDI inquiry requests. The Overlap Rate of Web Service Inquiry Requests among Applications (v) parameter is varied while the other parameters $W_{threshold}$, $T_{threshold}$ and D are held constant. The X-axis indicates the sequence of UDDI inquiry requests

over time while the Y-axis shows the Pre-fetching Rate. There are four curves in this figure, corresponding to v=25%, v=20%, v=15% and v=10%.

As shown in Figure 7-11, varying *v* has only small influence on the Pre-fetching Rate. It is interesting to observe that an increase in *v* leads to only a marginal increase in *m*. The overlap rate only contributes to the improvement of the Cache-Hit Ratio which was discussed in Section 7.2.2.4.

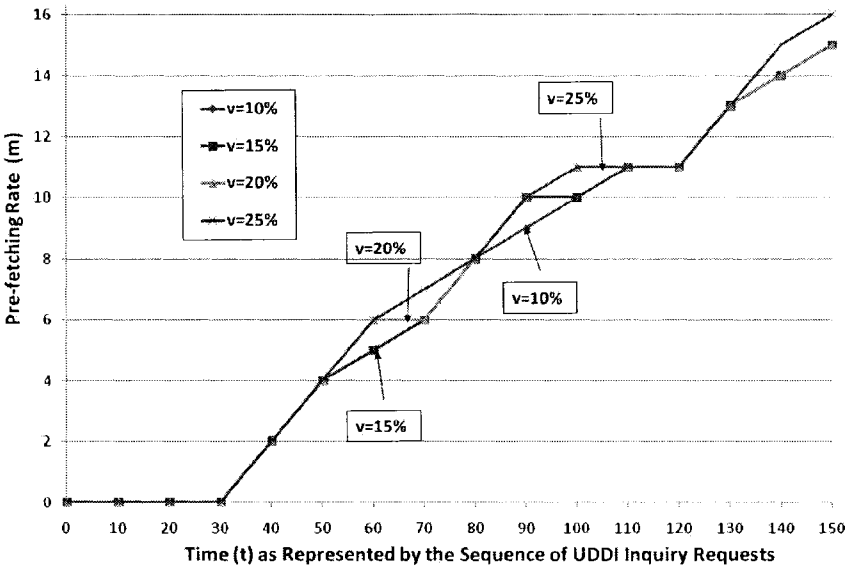

Figure 7-11 Evolution of Pre-fetching Rate (m) Over Time (t) as represented by the sequence of UDDI Inquiry Requests – Variations in v (YCYP, $T_{threshold}$=10000ms, $W_{threshold}$=2, D=2)

7.3.3 Summary

In this section, we analyzed the effect of *D*, $W_{threshold}$, $T_{threshold}$ and *v* on the Pre-fetching Rate. The proposed approach improves performance by increasing the Pre-fetching Rate, thereby increasing the Cache-Hit Ratio. Moreover, variations in the parameters of *D*, $W_{threshold}$, $T_{threshold}$ and *v* influence the performance of the proposed approach. Higher *D*

and $T_{threshold}$ values result in more UDDI inquiry requests being pre-fetched and responses associated with them cached in the caching proxy, thereby improving the Pre-fetching Rate and the Cache-Hit Ratio. Overlap Rate of Web Service Inquiry requests has only a small contribution to improving the Pre-fetching Rate. An increase in Pre-fetching Rate, however, leads to additional cost that is discussed in Section 7.5.

7.4 Evolution of Response Time over Time (t)

7.4.1 Purpose of the Experiment

In this section, we present the evolution of Response Time r over Time t as represented by the sequence of UDDI inquiry requests. Similar to Section 7.2, the results of experiments in this section will show the impact of Depth of Conditional Breadth-First Search, the Threshold on Edge Weight, the Time Interval Threshold and the application Overlap Rate v on Response Time r. As shown in Figure 6-5, we know that the impacts of D, $W_{threshold}$ and $T_{threshold}$ on r and EDT are actually results of the influence of these parameters on the Pre-fetching Rate and the Cache-Hit Ratio. The explanation of the experimental results related to Response Time is very similar to that presented in Section 7.2. Therefore, for conservation of space, we only provide a quantitative analysis of the influence of varying $W_{threshold}$ on the Response Time r in Section 7.4.2.1 and on EDT in Section 7.4.2.2. The graphs capturing the impact of D, $T_{threshold}$ and v on Response Time

are included in Appendix C.

7.4.2 Experimental Results and Analysis

7.4.2.1 Varying the Threshold on Edge Weight ($W_{threshold}$)

Figure 7-12 shows the influence of UDDI inquiry requests over time on the UDDI inquiry request Response Time r when the Threshold on Edge Weight parameter is varied and the other parameters D, $T_{threshold}$ and v are held constant. The X-axis indicates UDDI inquiry requests over time while the Y-axis shows the Response Time r. Because $W_{threshold}$ is a property of the pre-fetching proxy, the measurement has to be done using the "YCYP" experimental model. There are five curves in this figure: the top, dashed curve corresponds to the "YCNP" experimental model, while the lower curves correspond to $W_{threshold}$=15, $W_{threshold}$=10, $W_{threshold}$=5 and $W_{threshold}$=2 in the "YCYP" experimental model.

As shown in Figure 7-12, varying the Threshold on Edge Weight influences the Response Time. For any given Threshold on Edge Weight, the Response Time keeps decreasing over time. This is because as more and more UDDI inquiry requests are generated and the responses associated with them are cached in the caching proxy over time. The more UDDI inquiry requests and responses cached in the caching proxy, the higher the chance that subsequent UDDI inquiry requests will give rise to a cache hit. The response associated with the cache-hit UDDI inquiry request in the caching proxy can be

directly retrieved from the caching proxy and returned to the client. The UDDI inquiry request does not need to be forwarded to the pre-fetching proxy and the UDDI registry server. In our proposed approach, the caching proxy is located as close to the client as possible, as presented in Section 4.2.2, therefore the Response Time is significantly shorter.

For a given t, the Response Time has a lower value when $W_{threshold}$ has a lower value. In addition, a higher threshold on edge weight leads to a curve that is much closer to the dashed curve that corresponds to the "YCNP" experimental model (see Figure 7-12). Section 7.2.2.1 explained why a greater weight threshold value results in a Cache-Hit Ratio curve that is closer to the "YCNP" curve. The same explanation can be applied here. A higher weight threshold value leads to fewer eligible additional UDDI inquiry requests that can be pre-fetched in the pre-fetching proxy and cached in the caching proxy. Therefore the chance of a subsequent UDDI inquiry requests being hits in the caching proxy drops, and the UDDI inquiry request Response Time goes up.

Note also that the UDDI inquiry request Response Time with $W_{threshold}=2$ and $W_{threshold}=5$ fall rapidly, immediately after the initial process phase of the pre-fetching graph. However, the rapid fall in Response Time is not observed for higher values of $W_{threshold}$ (see Figure 7-12 and Table 7-5). This observation is consistent with the observation in the context of H discussed in Section 7.2.2.1. Thus, no further discussion is presented in this section.

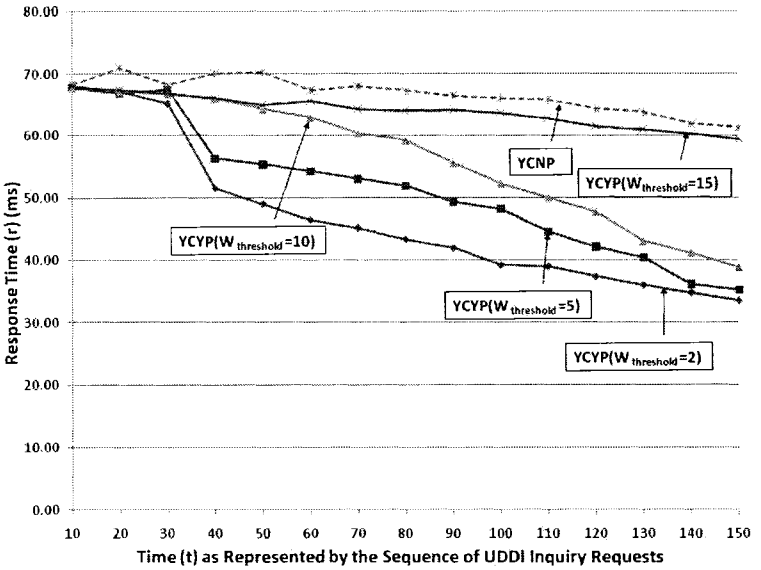

Figure 7-12 Influence of UDDI Inquiry Requests over Time (t) on Response Time (r) – Variations in $W_{threshold}$ (YCYP, $T_{threshold}$=10000ms, v=10%, D=2)

Time(t)	r-YCYP (ms) ($W_{threshold}$=2)	r-YCYP (ms) ($W_{threshold}$=5)	r-YCYP (ms) ($W_{threshold}$=10)	r-YCYP (ms) ($W_{threshold}$=15)
0	0.00	0.00	0.00	0.00
10	68.02	67.70	67.81	67.92
20	67.11	66.78	67.11	67.34
30	65.12	67.46	66.71	66.91
40	51.48	56.23	65.94	66.11
50	48.97	55.28	64.27	65.00
60	46.39	54.19	62.87	65.65
70	45.11	52.95	60.33	64.33
80	43.29	51.84	59.12	64.03
90	41.95	49.23	55.56	64.23
100	39.21	48.17	52.24	63.67
110	38.95	44.47	49.92	62.81
120	37.38	42.10	47.71	61.43
130	36.01	40.39	43.00	60.91
140	34.78	36.19	41.11	60.33
150	33.56	35.23	38.81	59.41

Table 7-5 Response Time (r) vs. UDDI Inquiry Requests over Time (t) – Variations in $W_{threshold}$ (YCYP, $T_{threshold}$=10000ms, v=10%, D=2)

7.4.2.2 Investigation of EDT

The discussion presented in Section 7.4.2.1 focused on the Response Time. In this section we focus on EDT. The difference between EDT and Response Time was discussed in Section 6.5.1.

Figure 7-13 shows the measured values of EDT over time. Note that as described in Section 6.4 five different applications were run in different sequences and the X-axis corresponds to the average completion times of the 1^{st}, 2^{nd}, 3^{rd}, 4^{th}, and 5^{th} applications in the sequence. The figure shows that time has small impact on EDT when the "NCNP" experimental model is used. However, running the Web Service discovery applications over time influences the EDT when the "YCNP" and "YCYP" experimental models are used. The EDT for the "YCYP" experimental model decreases much faster than that of the "YCNP" experimental model. This result shows that the Web Service discovery application that runs later will get much more benefit from the activities of the previous applications when the "YCYP" experimental model is used. Thus a Web Service discovery application can experience a shorter discovery time if it executes later.

7.4.2.3 Influence of Cache-Hit Ratio (H) on Response Time (r)

As shown in Figure 6-5, the pre-fetching graph parameters D, $W_{threshold}$, $T_{threshold}$ and application related parameter v as well as Time t affect the target metrics r and EDT by affecting the middleware metric H. This section discusses the influence of Cache-Hit Ratio on the Response Time r and EDT.

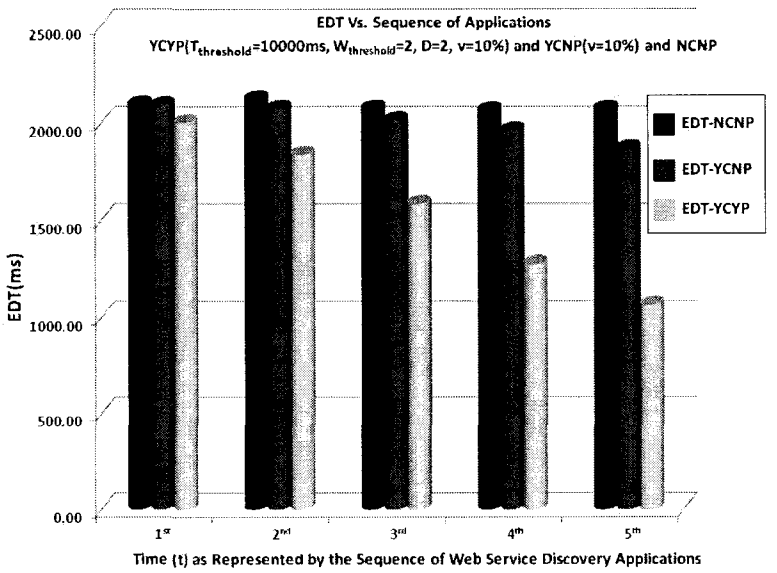

Figure 7-13 Influence of Web Service Discovery Application over Time (t) on EDT – (YCYP, $T_{threshold}$=10000ms, D=2, $W_{threshold}$=2, v=10% & YCNP, v=10% & NCNP)

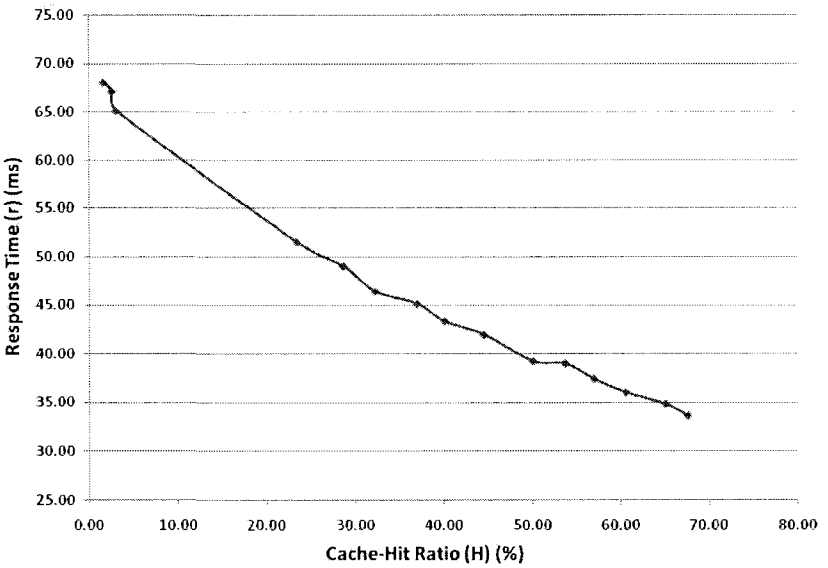

Figure 7-14 Influence of Cache-Hit Ratio (H) on the Response Time (r) (YCYP, $W_{threshold}$=2, D=2, $T_{threshold}$=10000ms, v=10%)

As shown in Figure 7-14, an increase in Caching-Hit Ratio leads to a lower Response Time. This is because that a higher Cache-Hit Ratio results in more original UDDI inquiry requests finding cached responses associated with them in caching proxy. This reduces the Response Time significantly. It is interesting to note that the relationship

between these parameters is approximately linear: a decrease in H is accompanied by a proportional increase in r.

7.4.3 Summary

In this section, we analyzed the impact of D, $W_{threshold}$, $T_{threshold}$ and v on performance. The proposed approach improves performance by shortening the Response Time. In addition, variations in the parameters of D, $W_{threshold}$, $T_{threshold}$ and v influence the performance of the proposed approach. A higher Depth of Conditional Breadth-First Search, Time Interval Threshold, Overlap Rate of Web Service Inquiry Requests among Applications values and a lower Threshold on Edge Weight shorten the Response Time. The observation of the evolution of EDT with time demonstrated how applications that run later on the system can benefit from the activities of previously running applications.

7.5 Interrelationship between Pre-fetch-Hit Ratio (H_p) and Cache-Hit Ratio (H)

7.5.1 Purpose of the Experiment

In this section, we analyze the relationship between two important performance metrics: H_p and H. According to the definition of the Pre-fetch-Hit Ratio (H_p) given in Section 6.5.4, the Pre-fetch-Hit Ratio reflects the usefulness of the pre-fetched UDDI inquiry

requests. This is an important performance metric, as we have to achieve a balance between incurring additional overheads and gaining performance benefits from using the proposed approach. The additional overheads caused by the extra pre-fetched UDDI inquiry requests are:

- More UDDI inquiry requests are sent to the UDDI registry server with the original UDDI inquiry request when the Pre-fetching Rate is greater than one. This may increase the workload of the server and the data traffic between the pre-fetching proxy and the server.
- More UDDI responses are cached in the caching proxy, and the cache size requirement may therefore increase.

The benefits are:

- More UDDI inquiry requests are cached in the caching proxy, so the Cache-Hit Ratio increases, more original UDDI inquiry requests encounter low latency. The cache hits tend to lower the workload in the remote UDDI registry server and the traffic through the networks.

7.5.2 Experimental Results and Analysis

7.5.2.1 Influence of UDDI Inquiry Requests over Time (t) on Pre-fetch Rate (m)

As shown in Figure 7-15, the Pre-fetching Rate always increases over time. This is because the pre-fetching proxy keeps updating the pre-fetching graph according to the

UDDI inquiry requests received, by either increasing the edge weight, if a directed edge already exists in the pre-fetching graph, or adding a new directed edge if it does not. Therefore, the C-BFS search will find more eligible edges and pre-fetch the end nodes associated with these eligible directed edges.

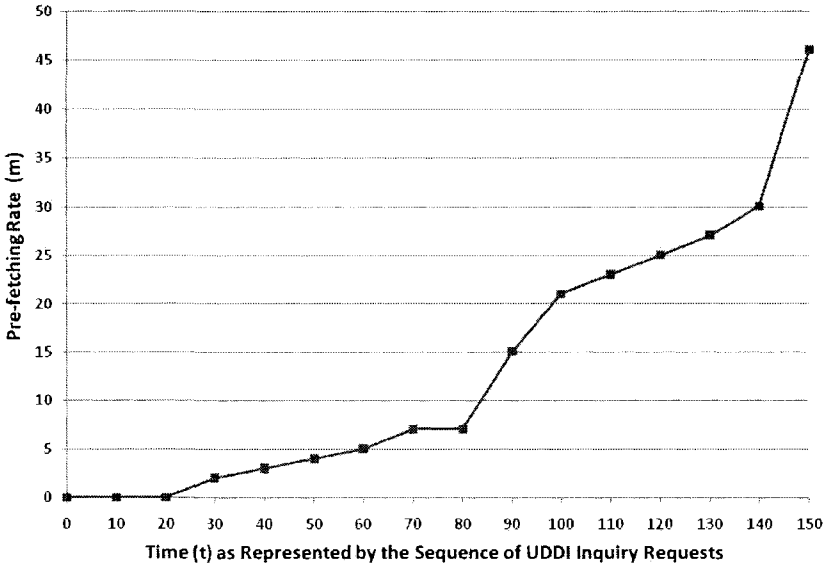

Figure 7-15 Influence of UDDI Inquiry Requests over Time (t) on Pre-fetch Rate (m) – (YCYP, $W_{threshold}$=2, D=5, $T_{threshold}$=10000ms, v=10%)

7.5.2.2 Influence of UDDI Inquiry Requests over Time (*t*) on the Number of Pre-fetch hits (*h*)

Figure 7-16 shows the influence of UDDI inquiry requests over time on the Number of Pre-fetch hits (*h*) when the Threshold on Edge Weight is 2, the Depth of Conditional Breadth-First Search is 5, the Time Interval Threshold is 10000ms and the Overlap Rate of Web Service Inquiry Requests among Applications is 10%. The X-axis indicates UDDI inquiry requests over time, while the Y-axis shows *h*.

As shown in Figure 7-16, the Number of Pre-fetch hits is 0 during the initial process

phase of the pre-fetching graph. This is because during this phase no UDDI inquiry requests are pre-fetched. From t=90 to t=140, the Number of Pre-fetch hits does not increase from 2. A similar step-like behavior is observed from t=30 to t=80. The usefulness of pre-fetched information is analyzed further in the next subsection.

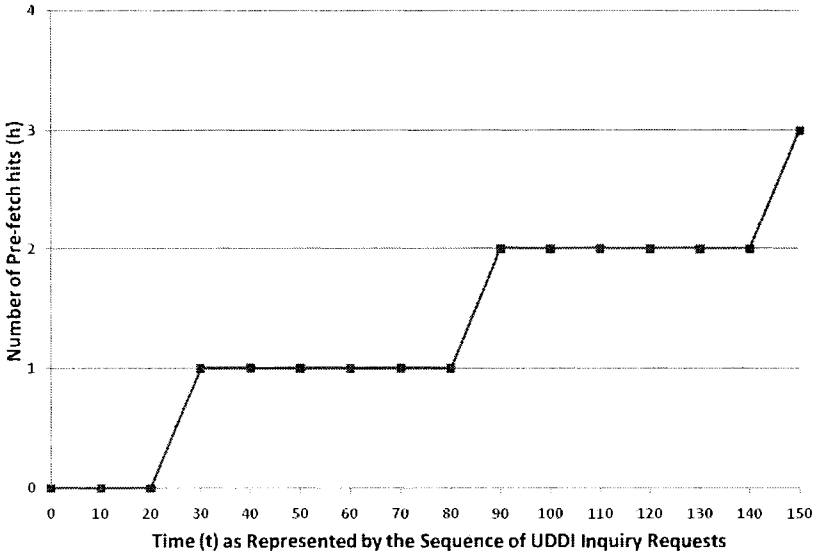

Figure 7-16 Influence of UDDI Inquiry Requests over Time (t) on Number of Pre-fetch hits (h) – (YCYP, $W_{threshold}$=2, D=5, $T_{threshold}$=10000ms, v=10%)

7.5.2.3 Relationship between Cache-Hit Ratio (*H*) and Pre-fetch-Hit Ratio (H_p)

Figure 7-17 captures the interrelationship between the Cache-Hit Ratio H and the Pre-fetch-Hit Ratio H_p when the Threshold on Edge Weight $W_{threshold}$ is 2, the Depth of Conditional Breadth-First Search is 5, the Time Interval Threshold parameter value is 10000ms and the Overlap Rate of Web Service Inquiry Requests among Applications is 10%.

As shown in Figure 7-17, the Pre-fetch-Hit Ratio keeps dropping with t while the

Cache-Hit Ratio increases with *t*. Although the number of pre-fetched UDDI inquiry requests keeps increasing rapidly after the initial process phase of the pre-fetching graph along with the UDDI inquiry requests over time (see Figure 7-15), the Number of Pre-fetch hits go up slowly (see Figure 7-16). This indicates that a few original UDDI inquiry requests are Pre-fetch hits defined in Section 6.5.4, although many additional UDDI inquiry requests are anticipated and cached in the caching proxy. Although these additional UDDI inquiry requests do not give rise to Pre-fetch hits, these pre-fetched UDDI inquiry requests make a significant contribution to the Cache-Hit Ratio. That is why the Pre-fetch-Hit Ratio goes down quickly, while the Cache-Hit Ratio keeps going up.

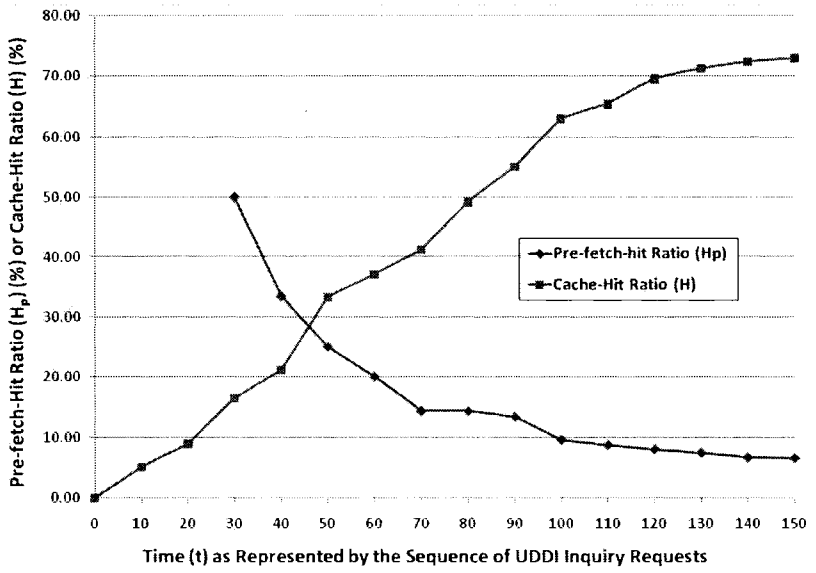

Figure 7-17 Relationship between Cache-Hit Ratio (H) and Pre-fetch-Hit Ratio (H_p) – (YCYP, $W_{threshold}$=2, D=5, $T_{threshold}$=10000ms, v=10%)

7.5.3 Summary

In this section, we discussed the influence of UDDI inquiry requests received over Time t on the Pre-fetch Rate m and on the Number of Pre-fetch hits h. We also analyzed the interrelationship between the Pre-fetch-Hit Ratio and the Cache-Hit Ratio, and concluded that the proposed approach leads to an inverse relationship between the Pre-fetch-Hit Ratio H_p and Cache-Hit Ratio H.

Chapter 8: Conclusions

8.1 Summary of Research

This thesis proposed an effective approach for pre-fetching and integrating it with caching, to improve the performance of the discovery process in Web Service registry systems. We conducted various experiments under different experimental models, varying system and workload parameters to demonstrate the improvement in performance achieved by applying the proposed approach to the Web Service discovery systems.

8.2 Conclusions

Based on the experimental investigations presented in this thesis, the following conclusions can be drawn:

- Because the Web Service inquiry requests and associated responses are all SOAP messages, it is beneficial to reuse as many of the existing caching and pre-fetching techniques in traditional Web application and Web Service fields as possible in the context of Web Service registry systems, to improve the performance of Web Service discovery.
- The proposed pre-fetching approach significantly improves Response Time by improving the Cache-Hit Ratio. More specifically, the critical parameters for the

pre-fetching graph—Depth of Conditional Breadth-First Search, Threshold on Edge Weight, Time Interval Threshold and Overlap Rate of Web Service Inquiry Requests among Applications—influence the performance of the proposed approach as measured by the Cache-Hit Ratio and Response Time. Higher values of D, $T_{threshold}$ and v increase the Cache-Hit Ratio. $W_{threshold}$ has an opposite effect on H: lower the value of $W_{threshold}$, higher is the value of H. Increasing pre-fetching and the Cache-Hit Ratio by selecting these parameters appropriately may increase the storage space used in the caching proxy. However, Response Time decreases when some of this cached information is accessed by subsequent UDDI inquiry requests. The choice of these parameters should strike an appropriate tradeoff between the increase in storage space and reduction in Response Time.

- A Web Service discovery application that runs later will benefit much more from previous applications when the proposed approach is used. Thus a Web Service discovery application achieves a better discovery time if it executes later.
- A higher Cache-Hit Ratio indicates a lower Response Time for UDDI inquiry requests. However, a higher Pre-fetching Rate does not necessarily indicate an improvement in performance. If the future discovery requests are not anticipated correctly, a higher Pre-fetching Rate may result in more useless UDDI inquiry request objects and associated response objects being cached in the caching proxy

and may give rise to a greater burden of handling UDDI inquiry requests on the UDDI registry server. Precision of anticipation was evaluated via the Pre-fetch-Hit Ratio parameter. A higher precision of anticipation leads to a drop in the cost associated with the proposed approach. Especially, in a system where the cache size is limited, the wrong prediction in the subsequent inquiry requests can heavily penalize the system.

- Some of the above conclusions can be applied to not only the domain of Web Services registry system but also other domains, such as traditional Web applications, or even domains outside of the context of the Web. The proposed graph-based pre-fetching approach could be adapted to any domain relying on a request-response sequence model.

8.3 Future Work

The work presented in this thesis has expanded our understanding of the Web Service discovery and registry systems, and provides insights into areas that require further exploration and analysis.

In Section 4.3.1, we described the makeup of pre-fetching graph G. A node $v \in V$ in G represents a Web Service inquiry request that has been received by the pre-fetching proxy. A node is uniquely identified by the name of the Web Service inquiry request object and the hash value of all the parameters of the Web Service inquiry request.

However, the pre-fetching graph can grow unnecessarily big, as it is rare for two UDDI inquiry requests to have exactly the same parameters. The proposed approach can be improved, to detect analogous UDDI inquiry requests and merge them in the pre-fetching graph. For example, the UDDI inquiry request *find_business("New York", "Music Store", "On-line")* and *find_business("Ottawa", "Music Store", "On-line")* can be treated as the same UDDI inquiry request, regardless of some non-essential parameters, while the proposed approach treats them as different UDDI inquiry requests. Similarly, the improved approach would consider the UDDI inquiry request *find_service("New York Music Store Business Key", "On-line Order")* and *find_service("Ottawa Music Store Business Key", "On-line Order")* as similar UDDI inquiry requests. Therefore the pre-fetching graph is updated to contain one directed edge *find_business("", "Music Store", "On-line")* → *find_service("", "On-line Order")* with a weight value of 2, rather than two separate directed edges with a weight value of 1, respectively. This makes building the pre-fetching graph faster and shortens its initial process phase while keeping the graph to a reasonable size. In order to further improve the performance of our proposed approach, more research is required on such condensed pre-fetching graphs.

Moreover, we can further improve the proposed C-BFS pre-fetching algorithm. A problem in the algorithm is that edge weight never decreases. Thus if a wrong pre-fetching decision is made it is likely to be made again in the future. As the result, the pre-fetching proxy may keep on predicting irrelevant nodes. There are several solutions

to improve the algorithm. We can maintain a communication between the caching proxy and pre-fetching proxy: the caching proxy can inform the pre-fetching proxy that some of the pre-fetched additional inquiry requests associated with the latest received original inquiry request are useless to subsequent inquiry requests. The pre-fetching proxy in turn can revise the relevant edge weight such that such irrelevant nodes are not pre-fetched in the future. Another improvement is to use a threshold on the relative weight of outgoing edges from a given node, instead of the absolute weight. For example, if the current node has three adjacent edges with weights 5, 5 and 50 respectively, the current C-BFS returns all three edges if the threshold on edge weight is 4, while the improved C-BFS only returns the edge with weight 50 if the threshold is 40 percent. This is because the edge with weight 50 is the only edge that has a proportion greater than 40 percent.

In the experiments reported in the thesis, the cache size used was very large such that information from the cache was never discarded. Investigating system performance for various different cache sizes forms an interesting direction for future research.

Bibliography

[1] A. Abhari, S. P. Dandamudi and S. Majumdar, "Web Object-based Storage Management in Proxy Caches," *Journal of Future Generation Computer Systems*, Volume 22, Number 1, January 2006, pp. 16–31.

[2] A. Abhari, S. P. Dandamudi and S. Majumdar, "Using Web Object to Improve the Performance of Proxy Caching," in *Proceedings of the Fourth International Workshop on Web Engineering at the World Wide Web WWW10 Conference*, Hong Kong, China, May 2001, pp. 82–92.

[3] O. Azim and A. K. Hamid, "Cache SOAP Services on the Client Side," http://www.javaworld.com/javaworld/jw-03-2002/jw-0308-soap_p.html, last accessed on March 26, 2006.

[4] A. Bestavros, "Using Speculation to Reduce Server Load and Service Time on the WWW," in *Proceedings of the Fourth ACM Conference on Information and Knowledge Management*, Baltimore, Maryland, November 1995, pp. 403–410.

[5] A. Bhattacharjee and B. K. Debnath, "A New Web Cache Replacement Algorithm," in *Proceedings of the 8th International Conference on Computer and Information Technology (ICCIT2005)*, Dhaka, Bangladesh, December 2005, pp. 420–423.

[6] L. Clement, A. Hately, C. V. Riegen, and T. Rogers, "UDDI Spec Technical Committee Draft, UDDI Version 3.0.2, Dated 20041019," http://uddi.org/pubs/uddi_v3.htm, last accessed on May 25, 2006.

[7] A. G. Sylvest, "Address Resolving Service: OIO Service Oriented Infrastructure

version 0.8," *Technical Report,* National IT and Telecom Agency, Copenhagen, Denmark, March 2006.

[8] G. Govato, "UDDI is Yellow Pages of Web Services," http://www.networkworld.com/news/tech/2002/0527tech.html, last accessed on June 2, 2006.

[9] J. S. Gwertzman, "Autonomous Replication in Wide-Area Internetworks," Harvard College, Cambridge, Massachusetts, USA, http://www.eecs.harvard.edu/vino/web/push.cache/thesis.html, last accessed on May 10, 2006.

[10] J. S. Gwertzman and M. Seltzer, "World Wide Web Cache Consistency," Microsoft Corporation and Harvard University, http://www.eecs.harvard.edu/vino/web/usenix.196/, last accessed on June 18, 2006.

[11] V. Holmedahl, B. Smith and T. Yang, "Cooperative Caching of Dynamic Content on a Distributed Web Server," *in Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, Chicago, USA, July 1998, p. 243.

[12] T. Johnson and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB'94)*, Santiago de Chile, Chile, September 1994, pp. 439–450.

[13] S. M. Kim and M. C. Rosu, "A Survey of Public Web Services," in *Proceedings of the 13th International Conference on the World Wide Web (WWW 2004)*, New York, USA, May 2004, pp. 312–313.

[14] A. Kumar, A. El-Geniedy, and S. Agarwal, "A Generalized Framework for Providing QoS-Based Registry in Service-Oriented Architecture," in *Proceedings of the 2005 IEEE International Conference on Services Computing (SCC 2005)*, Volume 01, Orlando, USA, July 2005, pp. 295–306.

[15] E. P. Markatos and C. E. Chronaki, "A Top-10 Approach to Pre-fetching the Web," in *Proceedings of the 8th Annual Conference of the Internet Society (INET'98)*, Geneva, Switzerland, July 1998.

[16] J. Myllymaki and B. Reinwald, "Database Support for Web Service Caching," *Technical Report RJ 10327 (A0311-052)*, IBM Research Division, Almaden Research Center, San Jose, California, November 2003.

[17] Network Working Group, "Hypertext Transfer Protocol—HTTP/1.1," http://www.w3.org/Protocols/rfc2616/rfc2616.html, last accessed on June 18, 2006.

[18] W. Onstine and R. Hightower, "Build SOA with Web Services using WebSphere Studio, Part 3: WSDL and UDDI Publishing Using WebSphere," IBM DeveloperWorks, https://www6.software.ibm.com/developerworks/education/ws-soa-3/ws-soa-3-ltr.pdf, last accessed on May 15, 2006.

[19] V. Ramasubramanian and D. B. Terry, "Caching of XML Web Services for Disconnected Operation," *Technical Report MSR-TR-2004-139*, Microsoft Research Group, December 2004.

[20] A. Rousskov and V. Soloviev, "A Performance Study of Squid Proxy on HTTP/1.0," *Journal of World Wide Web*, Special Edition on WWW Characterization and Performance and Evaluation, Volume 2, Numbers 1-2, 1999, pp. 47–67.

[21] S. Seltzsam, R. Holzhauser and A. Kemper, "Semantic Caching for Web Service," University Passau, Passau, Germany, http://webkemper1.informatik.tu-muenchen.de:8080/research/publications/conferences/ICSOC2005.pdf, last accessed on March 11, 2006.

[22] S. Seltzsam, R. Holzhauser and A. Kemper, "Semantic Caching for Web Services (Extended Version of the ICSOC 2005)," in *Proceedings of Service-Oriented Computing—ICSOC 2005, Third International Conference*, Amsterdam, Netherlands, December 2005, pp. 324–340.

[23] T. Takase, Y. Nakamura, R. Neyama and H. Eto, "A Web Services Cache Architecture Based on XML Canonicalization," in *Proceedings of the 11th International World Wide Web Conference (WWW 2002)*, Honolulu, Hawaii, USA, May 2002.

[24] T. Takase and M. Tatsubori, "Efficient Web Services Response Caching by Selecting Optimal Data Representation," in *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, Tokyo, Japan, March 2004, pp. 188–197.

[25] W.G. Tang, C.Y. Chang and M.S. Chen, "Integrating Web Caching and Web Pre-fetching in Client-side Proxies," *IEEE Transactions on Parallel and Distributed Systems*, Volume 16, Issue 5, May 2005, pp. 444–455.

[26] J. Z. Wang and V. Bhulawala, "Design and Implementation of a P2P Cooperative Proxy Cache System," in *Proceedings of the 2005 IEEE/WIC/ACM International Conference on Web Intelligence (WI 2005)*, Compiegne, France, September 2005, pp. 508–514.

[27] K. Y. Wong, "Web Cache Replacement Policies—A Pragmatic Approach," *IEEE*

Network, Volume 20, Issue 1, Jan.-Feb. 2006, pp. 28–34.

[28] "Advancing Web Services Discovery Standard," http://www.uddi.org/about.html, last accessed on June 2, 2006.

[29] "Web Services Definitions, tModel," http://searchwebservices.techtarget.com/sDefinition/0,,sid26_gci805760,00.html, last accessed on June 3, 2006.

[30] A. Kodaypak, "Adaptive Admission Control Policy for Web Caching," M.C.S. Thesis, School of Computer Science, Carleton University, August, 2007

[31] M. Arlitt, R. Friedrich and T. Jin, "Performance Evaluation of Web Proxy Cache Replacement Policies," in *Proceedings of the 10th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*, Palma De Mallorca, Spain, September 1998, pp. 193–206.

[32] "Apache Project—jUDDI," http://ws.apache.org/juddi/, last accessed on June 22, 2007

[33] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, R. Neyama, "Building Web Services with Java—Making Sense of XML, SOAP, WSDL and UDDI," 1st Edition, Pearson Education, New Jersey, USA, December, 2001.

[34] "Standardized Web Services," http://www.sis.pitt.edu/~vkhenat/Standardized_web_services.pdf, last accessed on June 28, 2007.

[35] "Service-Oriented Architecture—SOA,"

http://www-306.ibm.com/software/solutions/soa/, last accessed on May 18, 2007.

[36] "Web Services Architecture," http://www.w3.org/TR/ws-arch/, last accessed on May 22, 2007.

[37] Jeff Hanson, "Creating Your Own Private UDDI Registry," http://www.devx.com/Java/Article/21390, last accessed on June 26, 2007

[38] E. Christensen, F. Curbera, G. Meredith, S. Weerawarana, "Web Services Description Language (WSDL) 1.1," http://www.w3.org/TR/wsdl, last accessed on June 28, 2007.

[39] "UDDI Version 3.0.2, UDDI Spec Technical Committee Draft, Dated 20041019," http://uddi.org/pubs/uddi-v3.0.2-20041019.htm, last accessed on June 28, 2007.

[40] "UDDI Version 2.0.4 API Specification," http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.htm, last accessed on June 28, 2007.

[41] "Java Caching System," http://jakarta.apache.org/jcs/index.html, last accessed on Mar 09, 2008.

[42] Y.Y. Jiang, M.Y. Wu, and W. Shu, "Web Pre-fetching: Costs, Benefits and Performance," in *Proceedings of the 7th International Workshop on Web Content Caching and Distribution (WCW)*, Boulder, Colorado, August 14–16, 2002.

[43] "UDDI4j Version 2.0.5," http://uddi4j.sourceforge.net/index.html, last accessed on March 16, 2008.

[44] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, "Introductions to Algorithms," 2nd Edition, The MIT Press, Massachusetts, USA, 2001.

[45] "Tomcat Version 6.0.2," http://tomcat.apache.org/, last accessed on March 11, 2008.

Appendix A

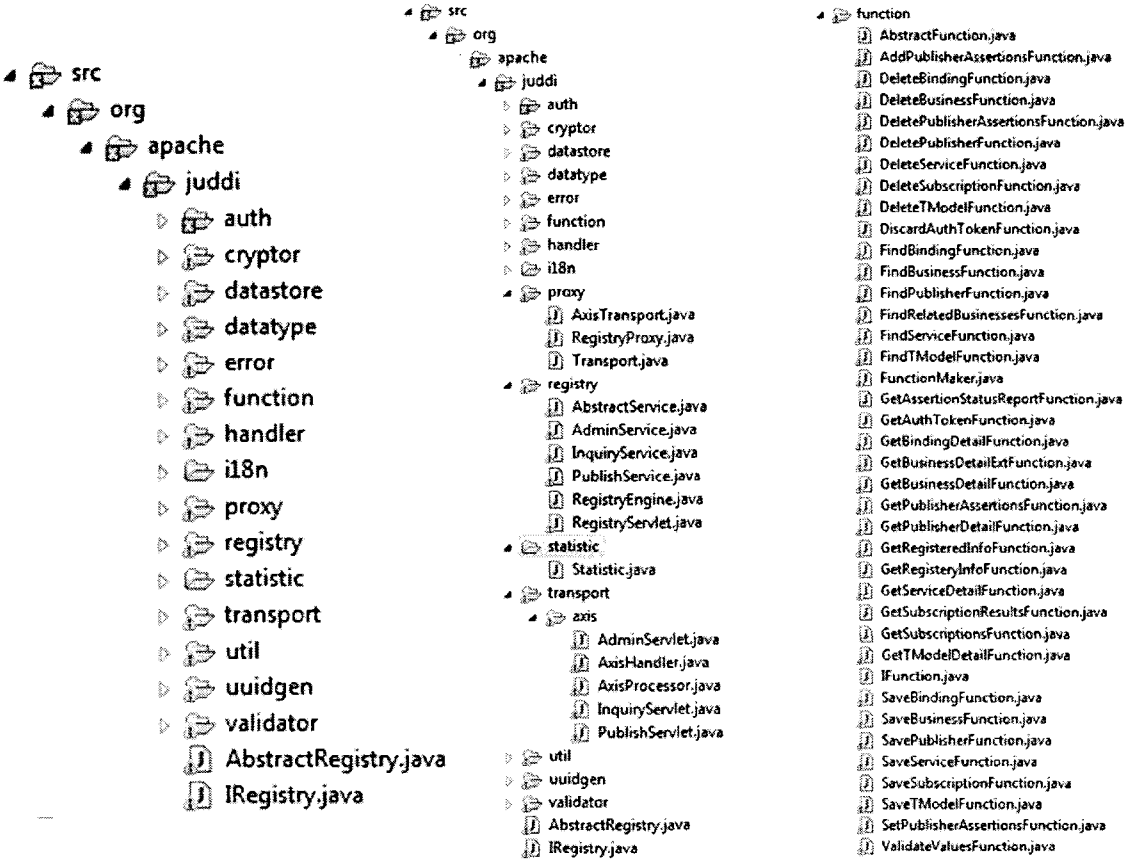

Figure A-1 Structure of jUDDI (Part I)

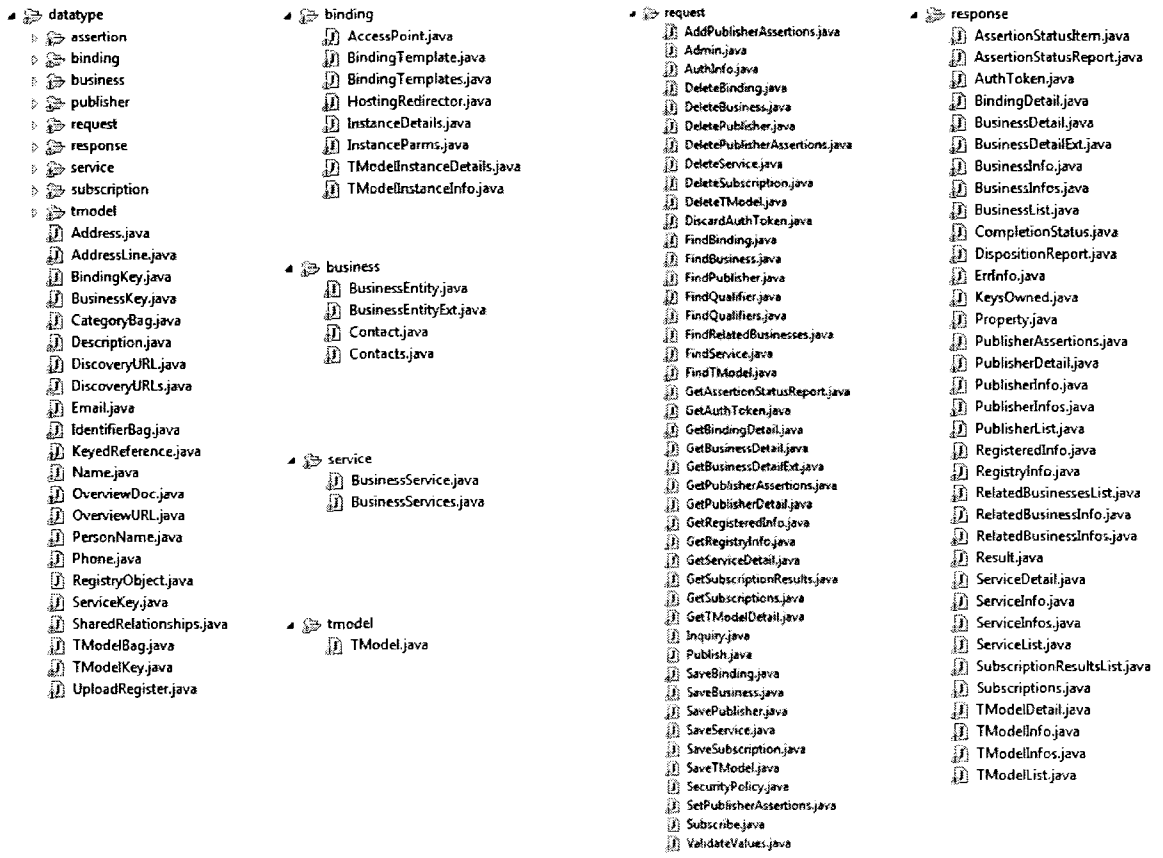

Figure A-2 Structure of jUDDI (Part II)

Appendix B

JCS configuration code that is contained in "cache.ccf" JCS configuration file:

```
# Licensed to the Apache Software Foundation (ASF) under one
# or more contributor license agreements.  See the NOTICE file
# distributed with this work for additional information
# regarding copyright ownership.  The ASF licenses this file
# to you under the Apache License, Version 2.0 (the
# "License"); you may not use this file except in compliance
# with the License.  You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing,
# software distributed under the License is distributed on an
# "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
# KIND, either express or implied.  See the License for the
# specific language governing permissions and limitations
# under the License.

# ####################################################################
# ################# A.DEFAULT CACHE REGION ######################
# [0]sets the default aux value for any non configured caches region
jcs.default=DC
jcs.default.cacheattributes=org.apache.jcs.engine.CompositeCacheAttributes
jcs.default.cacheattributes.MaxObjects=500001
jcs.default.cacheattributes.MemoryCacheName=org.apache.jcs.engine.memory.lru.LRU
MemoryCache
# Adding memory shrinking as cache attributes
jcs.default.cacheattributes.UseMemoryShrinker=true
jcs.default.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.default.cacheattributes.ShrinkerIntervalSeconds=30
jcs.default.cacheattributes.MaxSpoolPerRun=100
# Adding element attributes
jcs.default.elementattributes.IsSpool=true
jcs.default.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.default.elementattributes.IsEternal=false
jcs.default.elementattributes.MaxLifeSeconds=3600
```

```
jcs.default.elementattributes.IsRemote=false
jcs.default.elementattributes.IsLateral=false
jcs.default.elementattributes.IdleTime=1800

# ##############################################################
# ################# B.CACHE REGIONS AVAILABLE ##################
# Regions preconfirgured for caching
# [0]Define Pre-defined Region "uddiCache":
jcs.region.uddiCache=DC
jcs.region.uddiCache.cacheattributes=org.apache.jcs.engine.CompositeCacheAttributes
jcs.region.uddiCache.cacheattributes.MaxObjects=500001
jcs.region.uddiCache.cacheattributes.MemoryCacheName=org.apache.jcs.engine.memory
.lru.LRUMemoryCache
# Adding memory shrinking as cache region attributes
jcs.region.uddiCache.cacheattributes.UseMemoryShrinker=true
jcs.region.uddiCache.cacheattributes.MaxMemoryIdleTimeSeconds=3600
jcs.region.uddiCache.cacheattributes.ShrinkerIntervalSeconds=30
jcs.region.uddiCache.cacheattributes.MaxSpoolPerRun=100
# Adding element attributes
jcs.region.uddiCache.elementattributes.IsSpool=true
jcs.region.uddiCache.elementattributes=org.apache.jcs.engine.ElementAttributes
jcs.region.uddiCache.elementattributes.IsEternal=false
jcs.region.uddiCache.elementattributes.MaxLifeSeconds=3600
jcs.region.uddiCache.elementattributes.IsLateral=false
jcs.region.uddiCache.elementattributes.IsRemote=false
jcs.region.uddiCache.elementattributes.IdleTime=1800

# ##############################################################
# ################# C.AUXILIARY CACHES AVAILABLE ###############

# Define Auxiliary "DC": Adding a disk cache to my default parameters,
# Primary Disk Cache-- faster than the rest because of memory key storage
jcs.auxiliary.DC=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheFactory
jcs.auxiliary.DC.attributes=org.apache.jcs.auxiliary.disk.indexed.IndexedDiskCacheAttri
butes
jcs.auxiliary.DC.attributes.DiskPath=C:\Program
Files\apache-Tomcat-6.0.2\webapps\juddi\cache
jcs.auxiliary.DC.attributes.MaxPurgatorySize=10000000
jcs.auxiliary.DC.attributes.MaxKeySize=1000000
```

jcs.auxiliary.DC.attributes.MaxRecycleBinSize=5000
jcs.auxiliary.DC.attributes.OptimizeAtRemoveCount=300000
jcs.auxiliary.DC.attributes.ShutdownSpoolTimeLimit=60

Appendix C

C.1 Evolution of Pre-fetching-Rate (m) over Time (t)

C.1.1 The Effect of Varying the Depth of Conditional Breadth-First Search (*D*)

Figure C-1 shows the evolution of the Pre-fetching Rate over Time *t* as represented by the sequence of UDDI inquiry requests, with variations in the Depth of Conditional Breadth-First Search parameter while the other parameters, such as the Threshold on Edge Weight, the Time Interval Threshold and the Overlap Rate of Web Service Inquiry Requests among Applications, are held constants.

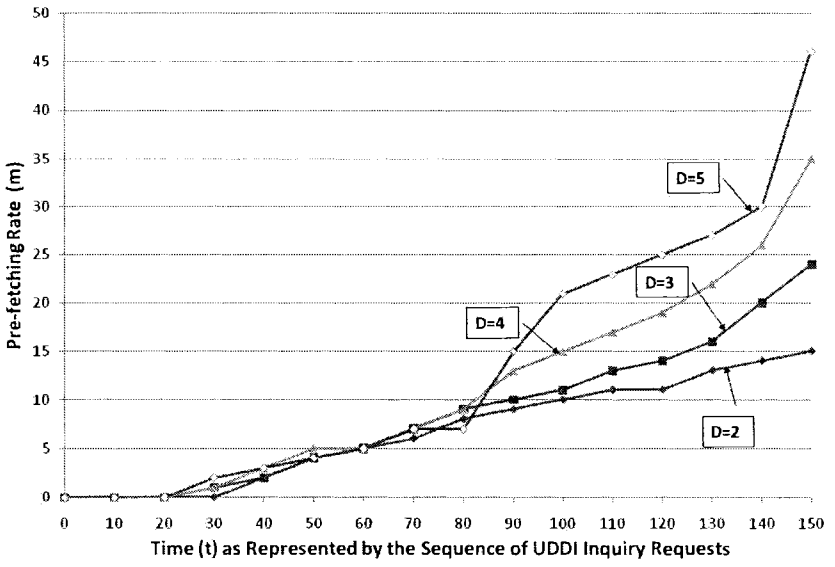

Figure C-1 Evolution of Pre-fetching Rate (m) over Time (t) as Represented by the Sequence of UDDI Inquiry Requests (t) – Variations in D (YCYP, $T_{threshold}$=10000ms, v=10%, $W_{threshold}$=2)

C.1.2 The Effect of Varying the Time Interval Threshold ($T_{threshold}$)

Figure C-2 shows the evolution of the Pre-fetching Rate over Time *t* as represented by the

sequence of UDDI inquiry requests with variations in the Time Interval Threshold parameter, while the other parameters Depth of Conditional Breadth-First Search, the Threshold on Edge Weight and the Overlap Rate of Web Service Inquiry Requests among Applications, are held constants.

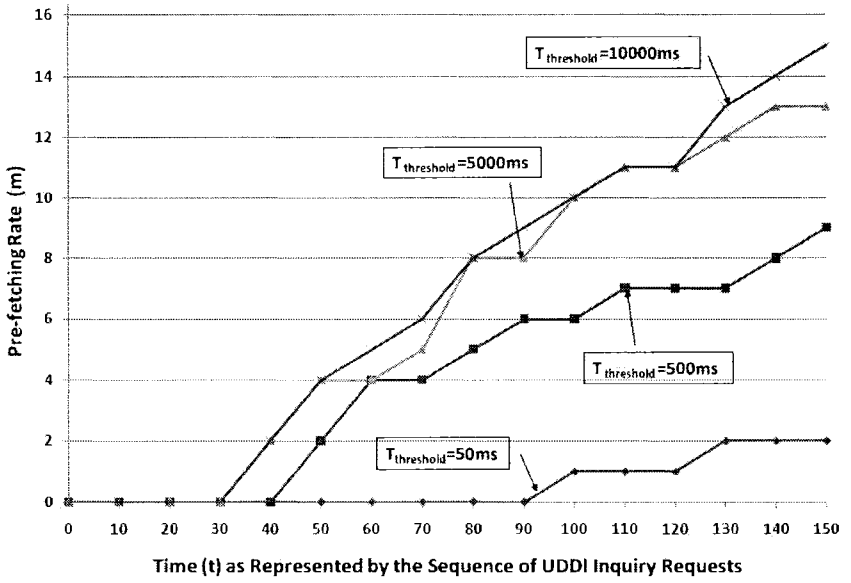

Figure C-2 Evolution of Pre-fetching Rate (m) over Time (t) as Represented by UDDI Inquiry Requests – Variations in $T_{threshold}$ (YCYP, v=10%, $W_{threshold}$=2, D=2)

C.2 Evolution of Response Time (r) over Time (t)

C.2.1 The Effect of Varying the Depth of Conditional Breadth-First Search (*D*)

Figure C-3 shows the evolution of Response Time over time as represented by the sequence of UDDI inquiry requests when there are variations in the Depth of Conditional Breadth-First Search (*D*) parameter and the other parameters, the Threshold on Edge Weight, the Time Interval Threshold and the Overlap Rate of Web Service Inquiry Requests among Applications, are held constants.

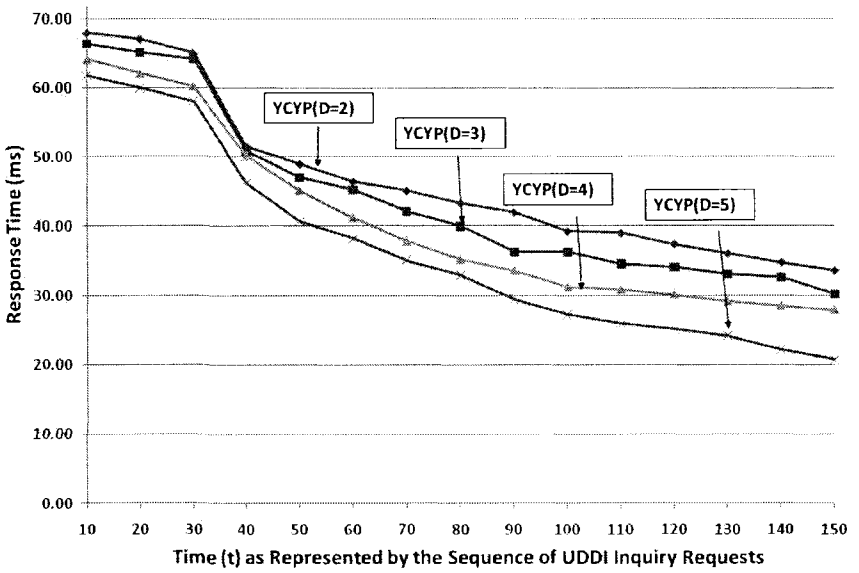

Figure C-3 Influence of UDDI Inquiry Requests over Time (t) on Response Time – Variations in D (YCYP, $T_{threshold}$=10000ms, v=10%, $W_{threshold}$=2)

Time(t)	r-YCYP (ms) (D=2)	r-YCYP (ms) (D=3)	r-YCYP (ms) (D=4)	r-YCYP (ms) (D=5)
0	0.00	0.00	0.00	0.00
10	68.02	66.47	64.26	61.87
20	67.11	65.22	62.18	60.03
30	65.12	64.29	60.28	58.11
40	51.48	50.87	50.22	46.23
50	48.97	47.09	45.17	40.69
60	46.39	45.26	41.28	38.25
70	45.11	42.12	37.87	35.10
80	43.29	40.01	35.29	32.98
90	41.95	36.33	33.65	29.49
100	39.21	36.29	31.29	27.30
110	38.95	34.56	30.88	25.91
120	37.38	34.11	30.22	25.24
130	36.01	33.05	29.19	24.21
140	34.78	32.67	28.55	22.23
150	33.56	30.23	27.96	20.78

Table C-1 Response Time vs. UDDI Inquiry Requests over Time (t) – Variations in D (YCYP, $T_{threshold}$=10000ms, v=10%, $W_{threshold}$=2)

C.2.2 The Effect of Varying the Time Interval Threshold ($T_{threshold}$)

Figure C-4 shows the evolution of Response Time over time as represented by the sequence of UDDI inquiry requests when there are variations in the Time Interval Threshold ($T_{threshold}$) parameter and when other parameters, the Depth of Conditional Breadth-First Search, the Threshold on the Edge Weight and the Overlap Rate of Web Service Inquiry Requests among Applications, are held constants.

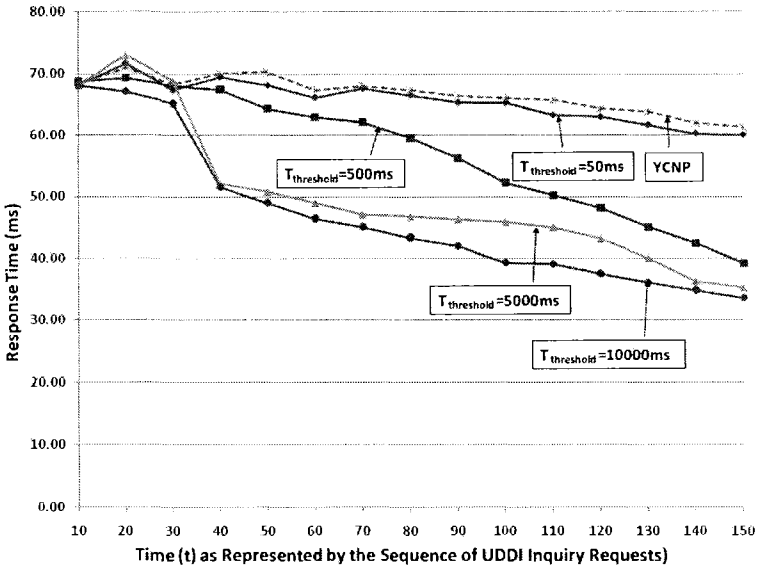

Figure C-4 Influence of UDDI Inquiry Requests over Time (t) on Response Time – Variations in $T_{threshold}$ (YCYP, v=10%, D=2, $W_{threshold}$ =2)

Time(t)	r-YCYP (ms) ($T_{threshold}$=50ms)	r-YCYP (ms) ($T_{threshold}$=500ms)	r-YCYP (ms) ($T_{threshold}$=5000ms)	r-YCYP (ms) ($T_{threshold}$=10000ms)
0	0.00	0.00	0.00	0.00
10	68.12	68.68	68.10	68.02
20	71.58	69.34	72.96	67.11
30	67.33	68.00	68.78	65.12
40	69.39	67.45	52.11	51.48
50	68.12	64.29	50.85	48.97
60	66.09	62.87	49.02	46.39
70	67.57	62.11	47.12	45.11
80	66.44	59.49	46.81	43.29
90	65.25	56.18	46.33	41.95
100	65.28	52.31	45.93	39.21
110	63.24	50.15	45.04	38.95
120	62.92	48.18	43.23	37.38
130	61.58	45.01	40.04	36.01
140	60.21	42.39	36.29	34.78
150	60.03	39.19	35.22	33.56

Table C-2 Response Time vs. UDDI Inquiry Requests over Time (t) – Variations in $T_{threshold}$ (YCYP, v=10%, D=2, $W_{threshold}$=2)

C.2.3 The Effect of Varying the application Overlap Rate (v)

Figure C-5 shows the evolution of Response Time over time as represented by the sequence of UDDI inquiry requests when there are variations in the Overlap Rate of Web Service Inquiry Requests among Applications (v) parameter and the other parameters, the Depth of Conditional Breadth-First Search, the Time Interval Threshold and the Threshold on Edge Weight, are held constants.

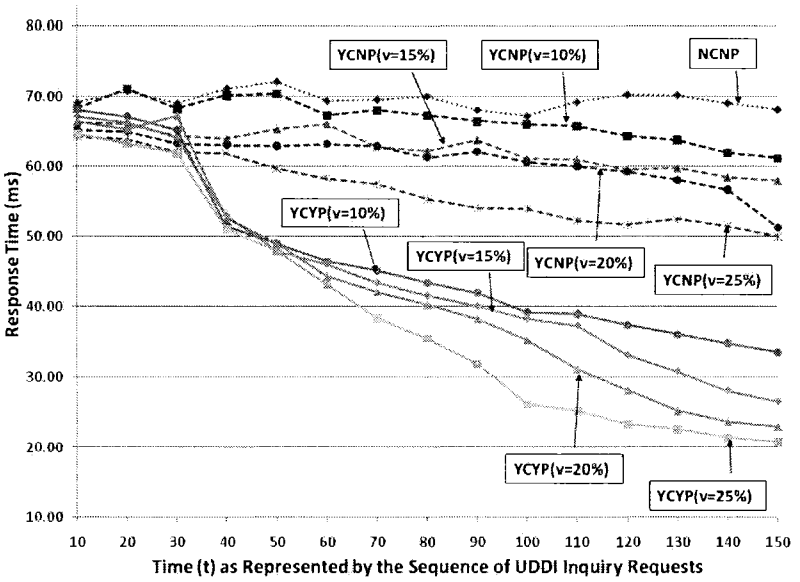

Figure C-5 Influence of UDDI Inquiry Requests over Time (t) on Response Time – Variations in v (YCYP, $T_{threshold}$=10000ms, D=2, $W_{threshold}$=2 & YCNP & NCNP)

Time (t)	r-NCNP (ms)	r-YCNP (ms)				r-YCYP (ms)			
		v=10%	v=15%	v=20%	v=25%	v=10%	v=15%	v=20%	v=25%
0	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
10	69.08	68.22	66.29	65.23	64.21	68.02	67.11	66.48	64.67
20	71.72	72.98	66.02	65.01	63.89	67.11	66.28	65.33	63.23
30	68.99	68.19	64.36	63.33	62.11	65.12	64.11	67.26	61.89
40	71.07	70.11	63.98	63.12	61.88	51.48	53.03	52.68	51.07
50	72.01	70.28	65.29	62.92	59.67	48.97	47.87	48.97	47.88
60	69.33	67.33	66.01	63.21	58.29	46.39	46.02	44.27	43.11
70	69.54	68.01	62.87	62.95	57.55	45.11	43.39	42.05	38.29
80	70.01	67.29	62.22	61.32	55.39	43.29	41.55	40.18	35.44
90	68.04	66.42	63.79	62.12	54.13	41.95	40.02	38.19	31.85
100	67.21	66.04	61.12	60.66	53.97	39.21	38.19	35.22	26.09
110	69.19	65.77	60.98	60.03	52.33	38.95	37.29	31.08	25.16
120	70.22	64.29	59.56	59.28	51.63	37.38	33.11	27.99	23.21
130	70.11	63.78	59.79	58.11	52.49	36.01	30.77	25.11	22.54
140	69.01	61.89	58.48	56.69	51.48	34.78	28.03	23.55	21.33
150	68.14	61.23	58.03	51.29	50.02	33.56	26.56	22.88	20.76

Table C-3 Response Time vs. UDDI Inquiry Requests over Time (t) – Variations in v (YCYP, $T_{threshold}$=10000ms, D=2, $W_{threshold}$=2 & YCNP & NCNP)