QoS-aware discovery of web services

Zhou, Chen

2007

Zhou, C. (2007). QoS-aware discovery of web services. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/35747>

<https://doi.org/10.32657/10356/35747>

9529578

QoS-Aware Discovery of Web Services

Zhou Chen

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfilment of the requirement for the degree of
Ph.D

2007

Abstract

The web is evolving from a collection of pages to a collection of services. Different from original distributed computing system, Web Services are more dynamic because of its service discovery and run-time binding mechanism. As more and more web services are appearing on the web, Web Services' discovery mechanism becomes essential. However, the industry's standard service discovery protocol has limited support for value-added services. Its service discovery mechanism adopts keyword matching to locate published Web Services. It has neither well defined semantics nor QoS related information. Great efforts are required to manually compare and test on retrieved services, not to mention the automatic discovery, composition, execution and monitoring of the services.

According to these analysis, our research mainly focuses on the QoS-Aware service discovery portion of Web Services system. First part of our research is non-ontology based QoS-Aware service discovery system. We investigate on the key component of industry standard service discovery framework — the UDDI registry. We propose UDDI eXtension (UX), an enhancement for UDDI that facilitates requesters to discover services with QoS awareness. In this system the service requester invokes and generates feedback reports, which are received and stored in local domain's UX server for future usage. By sharing these experiences from those requesters in the local domain, the UX server summarizes and predicts the service's performance. A general federated service is designed to manage the service federation. The discovery between different cooperating domains is based on this general federated service. Therefore the links between domains are maintained dynamically. A prototype system is implemented to demonstrate the feasibility of the UX system.

We then further investigate the problem of semantics in the QoS-Aware service discovery. Various Web Services have quite different requirements for their service quality

and measurement. This requires the mechanism for extensible defining and sharing of QoS metrics, as well as the integration in the service discovery system. To address these problems, we choose ontology level solution since it provides a nature way for computers to share the domain knowledge. We design the OWL-QoS ontology together with its QoS-Aware services discovery framework. This novel ontology is designed as a complement for OWL-S ontology to offer QoS-Aware service discovery and measurement service. The designed model contains three concept layers: SLA layer, profile layer and metrics layer. A matchmaking algorithm is presented to rank the QoS profiles with multiple matching degrees. Well-defined metrics can be utilized by measurement organizations to check whether the service performs correctly. For different usage phases of the OWL-QoS ontology, we design the QoS matchmaking framework, the measurement framework and the measurement code generator. Semantics in this ontology level specification helps to achieve better interoperability, automation and extensibility. From the initial experimental results, we envision its potential usage in real e-business world.

Acknowledgments

First and foremost, I would like to express my great appreciation to my advisor, Dr. Chia Liang Tien, Clement, for his invaluable directions and support throughout my research efforts towards this dissertation. His insights and suggestions to the research area enlightened me in various detailed aspects throughout the work. His directions with regards to the development of my academic thinking and writing were inspiring and helpful for my current and future work.

Secondly, I would like to express my gratitude Dr. Lee Bu Sung, Francis, for his helpful discussions on the ideas in the work. Although he is very busy, he still gives me important guidance and insights on my research. I would also like to thank Dr. Bilhanan Silverajan for his kind help and comments on the research.

Furthermore, I would give my thankfulness to all professors and students of our PET-DEVICE++ group for their patience of listening and their sparkling minds. They're Dr. Deepu Rajan, Dr. Sourav Saha Bhowmick, Mrs. Wang Surong, Mr. Liu Song, Mr. Tang Guosheng, Mrs. Chen Ling, Mr. Yi Haoran, Mr Chu yang, et al.

Finally, I wish to say thanks to my family and all the people who provided me with continued support and understanding.

Contents

Abstract	i
Acknowledgments	iii
List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 An Overview of Web Service Discovery	1
1.2 Motivation for This Research	3
1.2.1 The Importance of Web Services and its Discovery Mechanism	4
1.2.2 Extension Requirements	6
1.3 Major Results	13
1.4 Dissertation Organization	14
2 Backgrounds	16
2.1 Web Services	16
2.1.1 Web Service Roles	17
2.1.2 Web Service Protocol Stack	19
2.2 Web Services Discovery	21
2.2.1 SOAP	21
2.2.2 WSDL	21
2.2.3 UDDI	22
2.3 Semantic Web	23
2.3.1 Layers of Semantic Web	23
2.3.2 The Ontology Language and Description Logics	24
2.4 Semantic Web Application for Service Discovery	28

2.5	Web Services QoS	32
2.5.1	QoS Metrics	32
2.5.2	QoS Bottlenecks	35
2.6	Summary	36
3	QoS-Aware Service Discovery without Ontology Support	37
3.1	Introduction	37
3.2	Related Work	38
3.3	Network Model and Design Considerations	40
3.3.1	Network Model	40
3.3.2	Design Considerations	41
3.4	Architecture	43
3.4.1	Components of UX System	43
3.4.2	QoS Metrics	46
3.4.3	Communications between Components	47
3.4.4	Customization	48
3.4.5	Important Policies	49
3.5	Federation Discovery Support for UDDI eXtension System	50
3.5.1	CSG Model	51
3.5.2	Federation Service for Web Services	54
3.5.3	UDDI eXtension Federation Service Design	58
3.6	Implementation and Experiment	60
3.6.1	Implementation	61
3.6.2	Experimental Result	66
3.7	Summary	72
4	QoS-Aware Service Discovery with OWL-QoS Ontology Support	74
4.1	Introduction	74
4.2	Related Works	77
4.3	Scenario	82
4.4	Design Principles for QoS Ontology	82
4.5	Model Design	84

4.5.1	SLA Layer	85
4.5.2	QoS Profile Layer	88
4.5.3	QoS Metrics Layer	90
4.5.4	Basic Metric Set	94
4.5.5	Reuse of Specification	95
4.6	Matchmaking for the OWL-QoS	96
4.6.1	Conformance Definition	97
4.6.2	Matchmaking Algorithm	100
4.6.3	Comprehensive Example	102
4.7	Discussion	106
4.8	Summary	108
5	Usage Phases of OWL-QoS Ontology and the Supporting Framework	110
5.1	Usage Phases of OWL-QoS Ontology	110
5.1.1	QoS Matchmaking Phase	112
5.1.2	SLO Agreement Phase	116
5.1.3	Measurement Preparation Phase	116
5.1.4	Measurement Phase	122
5.2	Experimental Results	127
5.3	Related Works	131
5.3.1	QoS Related Service Description without Ontology Definition . . .	132
5.3.2	QoS Related Service Description with Ontology Definition	134
5.3.3	Discussion of the Benefits	136
5.4	Summary	141
6	Conclusions	143
6.1	Major Contributions	145
6.2	Future Directions	147
	References	150
	List of Publications	157

List of Figures

1.1	Example Discovery Scenario	12
2.1	Typical Web Service Operations	18
2.2	Web Service protocol stack from IBM’s view	19
2.3	Semantic Web Layers	24
3.1	Network Model	41
3.2	Components of UX	44
3.3	Federated Server’s Local Tree Modification Sequence Diagram	55
3.4	Local Tree Change Interface	56
3.5	Federated Server’s Propagation Interface	57
3.6	UX System’s Interface	59
3.7	UX Server’s Federated Discovery Sequence Diagram	63
3.8	Federated Server’s Class Diagram	64
3.9	Percentage Deviation of Fast-Add Services Invocations from the Summary of Response Time	67
3.10	UX Federation Experiment Setup	70
3.11	The Network Overhead of UX Federation Experiment	71
4.1	Two Phases of the Service Discovery	77
4.2	Ontology Overview	86
4.3	QoS Profile Matchmaking for Service Requester and Service Providers	89
4.4	Ontology for Basic Metric Set	94
4.5	Advertisement Example of OWL-QoS Ontology	103
5.1	Overview of OWL-QoS Processing Phases	111

5.2 Matchmaker System Prototype 114

5.3 OWL QoS Matchmaker’s Inquiry Wizard 115

5.4 OWL QoS Matchmaker’s Inquiry Result 115

5.5 From Metrics to the Measurement 118

5.6 Measurement Partner’s Definition for Metrics 119

5.7 Measurement Architecture 123

5.8 Axis Message Path on the Server 124

5.9 Axis Measurement Handler Example 125

5.10 Advertisements’ Publish Time 128

5.11 Advertisements’ Inquiry Time 129

5.12 Integration of Data without/with a central ontology 138

List of Tables

2.1	Main functions of the UDDI Inquiry API	22
2.2	Semantics of Description Logic <i>SHIQ</i>	28
2.3	Comparison between Various Reasoners	29
3.1	Description of Functions in the Federated Discovery Sequence Diagram .	62
3.2	Class Descriptions in the Class Diagram	65
3.3	Mean Deviation of Summary Functions for Fast-Add Service	67
3.4	Mean Deviation of Summary Functions for Slow-Add Service	67
3.5	Mean Deviation of Summary Functions for Stock Service	68
3.6	User Voting on Service Discovery of Traditional and UX System	69
4.1	The Property List in SLA Layer	87
4.2	The Properties of Metric Class	91
4.3	The Properties of ComplexMetric Class	93
4.4	The Function Definitions	93
4.5	The Customization of Function	93
4.6	Voting on Understandability of Hierarchical and Flat Definitions for Metrics	106
4.7	Voting on Understandability of the Layered Design of OWL-QoS	106
5.1	Overhead of the Measurement Stubs	130
5.2	Voting on Usability of Inquiry and Return Result	131

Chapter 1

Introduction

The topic of this Ph.D dissertation is the discovery of Web Services. As more and more Web Services appear on the Internet, Web Services discovery mechanism becomes essential. Traditional service discovery mechanism should be enhanced to provide value-added services discovery. The focus of this research is the Quality of Service (QoS) aspect in the Web Services discovery. After the literature review, we present our non-ontology based QoS-Aware service discovery system — the UDDI eXtension (UX) system and its federated discovery framework. Then we introduce our ontology level QoS specification for Web Services and its service discovery and measurement framework.

This chapter provides a general introduction of this Ph.D dissertation. In section 1.1 we briefly review current Web Services discovery technologies and their relations. Section 1.2 explains the motivation for the researches on this topic and the focus of our research work. Then, we present the major results of our research work in section 1.3. Finally, section 1.4 describes the organization of this dissertation.

1.1 An Overview of Web Service Discovery

The web is evolving from a collection of pages to a collection of services. In both e-business and e-science, people are realizing that they can achieve significant cost savings and high efficiency by outsourcing and cooperation. There are increasing demands to

CHAPTER 1. INTRODUCTION

integrate services across distributed, heterogeneous and dynamic environments from different service providers. The emergence of Web Services technology supplements this trend. Its development helps to fasten the web's evolving from page-centric web to service-centric web.

The World Wide Web Consortium defines the Web Service as “a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL[1]). Other systems interact with the Web Service in a manner prescribed by its description using SOAP[2] messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.”¹ Here the WSDL means “Web Services Description Language”, and SOAP means “Simple Object Access Protocol”. Examples of Web Services are stock querying service, weather information inquiring service, on-line dictionary service, etc.

In the Web Services architecture, there are three main roles: a service provider, a service consumer, and a directory. This architecture enables a loosely coupling of service consumer and service provider: At first the service provider publishes descriptions of its services to the directory. After that the service consumer searches the directory to find information about appropriate provider and its service. Finally, the service consumer binds to the chosen provider and use its service. This loosely coupled architecture is performed dynamically, i.e., during run-time. This “publish-discovery-bind” model of service interaction increases the flexibility and agility of the Web Services. In addition, it raises the competition between service providers and forces them to provide better services. Figure 2.1 shows this “publish-discovery-bind” model. Services are implemented and published by service providers. They are discovered and invoked by service requesters. Information about a service may be kept within a service registry.

¹Web Services Glossary, available at: <http://www.w3.org/TR/ws-gloss/>

In this loosely coupled Web Services architecture, the discovery process becomes the key for the dynamic service binding and invocation, since the service requester needs to locate the most suitable Web Service first. In current Web Services practice, Universal Description Discovery and Integration (UDDI)[3] plays the role of directory for the Web Services' discovery. It not only consists of a data structure standard for the business-related and technical-related descriptions of services, but also interactive protocols that allow the service requester to gain access to the provider's information and service's description. After the service requester chooses the service, it uses the Web Services Description Language (WSDL) to get the service's binding information, and invokes the target Web Service by the Simple Object Access Protocol (SOAP) protocol. These three standards form the basis of Web Services. Web Services community requires the vendors to support these common standards to achieve the interoperability among Web Services. Additional protocols are designed on top of them to offer value-added services.

The Web Services' protocol stacks are still evolving. There have been a number of recent progresses related to the Web Services technology, in terms of both research and development. Although some of these efforts focus on the service discovery and QoS related topics, a lot of issues in this area remain untouched or have not been studied completely. This dissertation explores and addresses the service discovery portion of Web Services.

1.2 Motivation for This Research

The work described in this dissertation focuses on the enhancement of current Web Services discovery system to provide value-added services. The area being studied is: the discovery of Web Services, and we mainly address the Quality of Service (QoS) aspect of Web Services discovery.

At the beginning of our research for the discovery of Web Services, we find that the basic Web Services discovery protocols (i.e., UDDI, WSDL, and SOAP) do not provide adequate support for value-added service discovery, especially for the QoS aspects. Value-added extensions are necessary for maturing the discovery of Web Services. We identify which extensions are important goals for enhancing the service discovery process and how to achieve these goals. In the following, we first discuss the importance of Web Services discovery in the Web Services world. After that, we address some required extensions for the Web Services discovery.

1.2.1 The Importance of Web Services and its Discovery Mechanism

Driven by the business market, Web Services promise to be a new generation of business-to-business (B2B) and enterprise application integration (EAI) architectures. One goal for the Web Services is to assist the semi-automatic and run-time integration of heterogeneously distributed computing systems within or across the enterprise boundaries. Another important goal for Web Services is faster and cheaper development, organization, and maintenance of complex service workflow from basic Web Services. Historically, these same goals have been the main business motivations for several distributed computing technologies, such as RPC, CORBA[4]. Due to various technical and non-technical issues, previous technologies are still evolving and have not reached enough consensus and maturity. Web Services technology is now giving a huge push towards these goals. The main technical advantage of Web Services is to leverage the widely used technologies, such as XML for representing data content and Web protocols (HTTP[5], FTP[6], and so on) for message transportation. Web Services can smoothly reuse the existing systems by wrapping the legacy software components. As a result, small, medium or large sized businesses, with or without their existing computing and software system, can both adopt the Web Services system in their business processings. The main non-technical

CHAPTER 1. INTRODUCTION

advantage of Web Services technology is the investment and advocacy on Web Service protocols and products by all major software companies (IBM, Microsoft, Sun, HP, etc.). Furthermore, the Web Services protocols are controlled under several industrial standardization groups like W3C² and OASIS³. In addition, many higher level software systems have already chosen the standard Web Services as building blocks, such as Grid Services, Semantic Web enabled Web Services (SWWS)⁴. From these aspects, the Web Services technology is becoming the dominant choice for current distributed systems. Its adoption grows continuously in recent years and will likely maintain this trend in the future.

As more and more Web Services are published on the web and will appear in the future, the Web Services' discovery mechanism is one of the most important aspects for Web Services. First, the service requester will likely face a huge group of service offers, and each service has its own functionality. How to precisely locate the correct Web Service according to the service requester's demand is the prerequisite to dynamically bind and integrate the Web Services. Second, as the Web Services' market is becoming more competitive, service providers are trying to offer better quality services for their customers to achieve their competitive advantage. How to compare and discover the most satisfying service from the service list, not just the fitting ones, is important for the Web Services market. Last but not least, precise service discovery is important for the service composition process as well, especially the automated composition of Web Services. Computer Aided Service composition is a planning problem where service requester has complete information about the start and ending situation. The special feature of the composition problem is that: the plan has a bounded length, while on each planning step, the search space is very branchy, i.e., there are many services to

²The World Wide Web Consortium, available at: <http://www.w3.org>

³Organization for the Advancement of Structured Information Standards, available at: <http://www.oasis-open.org>

⁴Semantic Web Enabled Web Services, available at: <http://swws.semanticweb.org/>

choose from. Reducing the search space by preciser service discovery is an effective way to improve the speed of composition progress.

Different from the selection of services, the discovery approach combines the functionality and QoS matching in one process. It returns all the functionally matching services, ranked by their QoS properties. On the other hand, selection approach is a separate step, which chooses only one of the highest QoS among the functionally applicable services according to the requester's criterion.

1.2.2 Extension Requirements

According to the current status of Web Services, we focus on the QoS-Aware extension for Web Services discovery. Quality of Service (QoS) is originally a telecommunication term which is concerned with the network attributes of an end-to-end connection. In this dissertation, we use this term within the Web Services domain. It refers to a measure of performance for the Web Services that reflects the services' characteristics, such as the response time and reliability. Here we explain three important aspects for achieving QoS-Aware Web Services discovery:

- Support of QoS-Aware Web Services discovery. It extends the general Web services discovery by the matching of QoS related properties. Service requester should be able to find the Web Services according to their QoS information. Service provider can publish and advertise their services by describing their promised QoS level. This mechanism is a precondition to achieve the competition advantage for service providers.
- Well-defined Semantics for QoS-Aware Web Services discovery. Well-defined Semantics of QoS information are important for large-scale QoS-Aware service discovery. We need to define the QoS metrics classes as basis to compare among

different services. The semantics of the metrics should be precisely defined to avoid misunderstanding. The metrics model should be extensible to import new metrics. The semantics for the conjunction of QoS constraints should be specified as well to define the service level objective.

- Design of matchmaking algorithms, discovery protocols and the supporting framework. According to the semantics of service QoS information, corresponding matchmaking algorithms are needed to calculate the service's ranking. The discovery protocol and the supporting framework should be designed to provide the QoS-Aware service publish, discovery, and measurement.

In the following, we will explain these three points in more details.

1.2.2.1 Support of QoS-Aware Services Discovery

Web Services' discovery mechanism is more and more important as more and more web services are appearing on the Internet. Although the UDDI registry is the industry standard for service discovery, and all major vendors support this discovery mechanism, it is far from good enough to support some value-added mechanisms. UDDI defines the standard data structure, as well as standard service categories, for the business-related descriptions of services. However, the QoS-Aware service discovery is one limitation in UDDI because there is no service QoS related definition available in its data structure.

QoS-Aware service discovery is a prerequisite for sophisticated Web Services. Being able to provide QoS-Aware service discovery has several distinct advantages.

- It allows the discovery and execution of the service according to their QoS specifications. This pushes the service provider to offer more competitive service solutions and optimize service requester's satisfaction.

CHAPTER 1. INTRODUCTION

- It allows the Web Services to be applied in mission critical tasks. Although lots of Web Services have been developed and deployed on the web, many enterprises are still watching and evaluating this new technology. Before serious consideration of this new technology in the enterprise application, it must be proved whether Web Services can succeed in mission critical tasks. The minimum requirements of this are: the Web service should provide its performance promises during its execution; the invocation of the service should be monitored during its life circle to ensure the service conforms to the agreed service level objectives.
- The Web Services adaptation can be done according to the service requester's requirement. The unpredictable surrounding environment definitely influences the execution of Web Services. It is important to cater for adaptation, replacement, and rescheduling of the Web Services to meet the changing environment. The adaptation according to the initial QoS requirement is necessary for critical tasks as well to obey the service level agreement.

1.2.2.2 Well-Defined Semantics

Well-defined semantics in the service discovery involves the integration problems of the Web Services. Since Web Services are designed within the heterogeneous environments, the integration problem is a main concern for people to use this new technology. When different Web Services are trying to talk with each other, their invocation interfaces are required to be interoperable.

The integration problem is normally caused from two sources: the syntactic heterogeneity and semantic heterogeneity. Syntactic heterogeneity exists because the interface signatures use different data types and class hierarchies. Semantic heterogeneity originates from the different semantic meanings of the service interface parameters and its

CHAPTER 1. INTRODUCTION

functionalities. solve the problem of integration, both service requester and the service provider should have the same understanding of these meanings. The well-defined semantics tends to solve the problem of the semantic heterogeneity.

Without the definition of these semantics, it is a difficult problem for computer to automatically locate the interoperable target service that fully meets the service requester's demand. For the standard UDDI's keyword matching process, same service names can mean quite different services in the service registry. For example, the "stock" service may mean the stock price inquiry service, or stock buying service. In this case, UDDI's keyword matching algorithm has low precision to choose the correct stock query service. Neither UDDI nor WSDL provides precise semantic definitions for services' input and output. Hence the automatic service discovery and composition is not possible.

OWL-S/DAML-S [7, 8] tries to solve these problems by defining a general ontology for Web Services. The top level of this ontology contains the service profile ("what the service does"), the service model ("how it works"), and the service grounding ("how to access it"). It is designed to achieve automatic Web Service discovery, invocation, composition and interoperation. However, one limitation of OWL-S is that it does not provide a detailed set of classes and properties to represent quality of service metrics. Significant improvement for the QoS model should be made to support a realistic solution for OWL-S users.

To provide the QoS model for the OWL-S ontology, the semantics in the specification should be precisely defined to allow the QoS-Aware service discovery. This means the service provider and service requester should agree on how to evaluate, monitor and compare the concerned QoS metrics. The relationships between different QoS metrics should be organized to allow the comparison. All the general metrics, specific metrics and similar metrics should be grouped in the agreed hierarchy. We call this the metrics definition. In addition to this, the QoS specification of the service should be properly

designed for publishing and inquiry. We call this Service Level Objective's definition. For service requester, it means properly defined semantics to specify the requester's demands. For the service provider, it means properly defined semantics to specify the provider's QoS guarantees.

1.2.2.3 Matchmaking Algorithm

Matchmaking algorithm should be designed to help the service requester find the interoperable services in both syntactic and semantic aspect. As to the matching algorithm, traditional keyword matching algorithm is adopted in UDDI to discover related services. WSDL files can be retrieved from the matching services list, by which service requester binds and invokes the chosen service. However, neither UDDI nor WSDL offers precise service semantics in their description, such as the service's hierarchical position, Input, Output, Precondition, Effect (IOPE), and so on. As to the supporting framework, the matchmaking algorithm and its implementation should be developed to select the satisfying service by comparing different QoS specifications in the registry.

The importance of the QoS for the Web Services makes it necessary to improve the QoS-Aware service discovery. The service QoS information concerned in our work can be divided into two types.

- *service provider's declared advertisement.* The service provider should be provided with a formal way to define their service QoS information. At the beginning, our survey of this research area identifies some works on the concept of service descriptions, most of them are not designed specifically for Web Services nor based on XML level. This leads to the syntax level integration problem between these specifications and the Web Services descriptions. During the time of our research, some new Web Services QoS specifications appeared. We will discuss them in details in section 5.3. We try to design the QoS specification with precise and flexible definition to reduce the cost of integration.

CHAPTER 1. INTRODUCTION

- *service's pragmatic performance.* The service's advertisement may contain some inaccurate information. The runtime information provides the pragmatic aspect of the service QoS information for the service requester. This information may be combined with the declared advertisement to check whether the service providers conform to their promise.

The development of the Web Services discovery technologies is accompanied with the underlying infrastructure and supporting tools. One of our important design choices is not to reinvent the wheel. We try to extend the well-established Web Services infrastructures to support the QoS-Aware service discovery instead of completely reinventing the whole wheel. The choice of extension on current standard protocols helps to reuse the original work of the community, and to simplify the adoption of the new add-ons. Therefore, we would prefer to build up our implementation as add-ons over the well-established infrastructure to provide value-added services.

Here is an example for Web Services discovery and some QoS related scenarios. Figure 1.1 shows the sample discovery scenario. The Service requester wants to find a stock information inquiry service. According to this functional requirement, the service discovery procedure returns several potentially available services. Since all these services provide the stock inquiry function, service requester decides to select the most beneficial service according to his QoS requirement of the service, such as the performance. After this decision-making step, the service requester will then dynamically bind and invoke the target Web Service. If the service becomes inaccessible one day, or the service requester plans to pick up another service, the service requester needs to initialize the adaptation process by choosing the potential services list according to his QoS constraints for the alternative service.

The importance of the QoS-Aware discovery can be shown in this scenario. To make a sound decision in the service selection phase, the QoS information of the service should

Figure 1.1: Example Discovery Scenario

be collected and analyzed. If there is no QoS-Aware service discovery support in the service discovery framework, the service requester needs to inquire and compare the potential services one by one (in this figure S_1 to S_3) to find the most satisfying service. Current service matchmaker cannot help the service requester to make the decision since the QoS-Aware service discovery is not supported. Therefore, this wastes the service requester time and energy, and burdens the service provider and network resources as well. Furthermore, if the service requester has a stock buying workflow and wants to integrate this stock inquiry service into his workflow, he would have more problems in selecting the service. He knows neither the service QoS' guarantee nor its history of performance information. Hence, huge effort is required to check the service's QoS properties before invocation. Such service discovery mechanism is far from mature and limits the service's application in mission critical tasks.

According to these analysis, we can see that providing QoS-Aware service discovery for current Web Services system is important to make the service discovery more mature. In the following, the major results of our research is presented.

1.3 Major Results

The dissertation presents one of the earliest comprehensive studies on QoS-Aware Web Services discovery. The first part of our research is non-ontology based QoS-Aware service discovery system. We propose a QoS-Aware UDDI extension with federated discovery support and its prototype, based on the industry's Web Services discovery infrastructure. The UDDI extension includes the design of the feedback, collection and federated system for processing Web Services QoS information. The second part of our research is an ontology based QoS-Aware service discovery system. We develop an ontology to describe QoS specifications for service providers and service requesters, together with its related

matchmaking algorithms. It complements OWL-S[7] by providing detailed QoS metrics classes, the QoS-Aware discovery algorithm, and measurement framework.

Our efforts on QoS-Aware UDDI-extension with federated discovery have resulted in the following contributions:

- (i) Implementation of the pragmatic QoS feedback and collection system to gather service's QoS information.
- (ii) Design of the QoS summary and ranking system to provide QoS-Aware discovery extension for the UDDI registry.
- (iii) Invention of the federated node maintenance, failure recovery, message delivery, and QoS mapping system.

In the design of ontology based QoS-Aware service discovery, our research has resulted in the following advances:

- (i) Presenting of the ontology design for describing QoS specification and QoS metrics classes.
- (ii) Design and development of the semantic matchmaking algorithm for the QoS specification.
- (iii) Design and development of the QoS metrics code generator and measurement framework for the OWL-QoS ontology.

1.4 Dissertation Organization

The remainder of this dissertation is organized as follows. In chapter 2, we review the backgrounds of the Web Services protocols, their supporting framework, and their QoS issues. We describe the current Semantic Web technologies and its application for Web

CHAPTER 1. INTRODUCTION

Services. In the following chapters, we present our research works on the QoS-Aware service discovery. Our research works can be divided into two parts: the first one is the QoS-Aware service discovery without the ontology support, and the second one is the ontology level QoS-Aware service discovery and measurement.

The first part of our work, i.e., the QoS-Aware UDDI extension without ontology support, is introduced in chapter 3. We describe the design principles and the system components in this work. Its major purpose is to facilitate service requesters to discover services on UDDI with good performance. By sharing the service requester's experience, the system handles both the local and the federated UDDI inquiries. It provides a simple view for the service requester by hiding the complex details of the registries' inquiries for the whole federation. After that we present our ontology level QoS-Aware service discovery system.

Chapter 4 presents the second part of our work, that is, the design of our OWL-QoS ontology and its matchmaking algorithms. This ontology is a complement for OWL-S to deal with QoS-Aware service discovery and measurement. We describe the design principles and the layered structure of the ontology. Based on the specification refinement and conformance, we present the mechanism for the reuse of QoS specifications. The matchmaking algorithm is designed specifically for the ontology to provide multilevel of matching degrees. In chapter 5, we focus on the major usage phases for the OWL-QoS ontology. The matchmaking prototype, the measurement framework and the measurement code generator are presented in this chapter. From the initial experiments, we envisage the potential to utilize this ontology level QoS-Aware service discovery in real e-business scenarios. We further discuss a number of related works and compare them with our work.

Finally, in chapter 6 we conclude our works and give some future directions of this research work.

Chapter 2

Backgrounds

In this chapter, the backgrounds of Web Services technologies and discovery mechanisms are introduced. We further present the ontology and Semantic Web technologies. Some research works in Web services discovery are introduced. After that, the Web Services' QoS problems are discussed, and we find that the Web Services are more likely to be influenced by various QoS issues. QoS related improvements are necessary for matured service discoveries.

2.1 Web Services

Web Services are "self-contained, self-describing modular applications" [9]. They constitute software modules that "describe a collection of operations that are network-accessible through standardized XML messaging" [10].

A web service has some key behavioral characteristics [11, 12]: *self-describing*, *loosely-coupled*, and *coarse-grained*. *Self-describing* means there's an identifiable and bounded set of descriptions that contain information about service's contents and functionalities. *Loosely-coupled* means a consumer of a web service is not tied to the web service directly; the web service interface can change over time without compromising the client's ability to interact with the service. *Coarse-grained* means Web Services technology provides a

CHAPTER 2. BACKGROUNDS

natural way of defining coarse-grained services that access the right amount of business logic.

These characteristics require the application of XML language and well-defined semantics for the new Web Services specifications. To achieve advanced self-describing characteristic, it requires human understandable as well as computer processible. Therefore, the language of new Web services description is normally based on the XML level or higher levels, such as the RDF or OWL. Loosely-coupled characteristic makes the Web Services discovery mechanism essential.

Among the Web Services standards, the Web Services discovery mechanism plays an important role. In the following we provide a brief introduction for the Web Services architecture. Two different views are available for the Web Services architecture: the individual roles view and the protocol stack view.

2.1.1 Web Service Roles

One view point to see the web services is to examine the individual roles. There are three major roles in the Web Services architecture:

- The Service Provider: the business entity that provides software applications as Web Services. From a business perspective, the Service Provider is the owner of the Web Service; from a development perspective, this is the platform that hosts the Web Service.
- The Service Requester: the business entity that has a need which can be fulfilled by an available Web Service. It finds the service according to its needs so that a Service Provider may fulfill it. From a development perspective, this is the application which is looking for and invoking a Web Service.

CHAPTER 2. BACKGROUNDS

- The Service Registry: a searchable repository of Web Services descriptions where Service Providers publish their Web Services and Service Requesters locate Web Services and obtain binding information necessary to invoke the service.

Figure 2.1: Typical Web Service Operations

Three fundamental operations, which is shown in Figure 2.1, contain publish, find, and bind:

- **Publish:** Performed by the service provider to advertise the existence and capabilities of a service.
- **Find:** Performed by the service requester to locate a service that meets a particular need or technology fingerprint.
- **Bind:** Performed by the service requester to invoke the service being provided by the service provider.

The typical procedure in a web service discovery is as follows: Services are implemented and published by service providers. They are discovered and invoked by service requesters. Information about a service may be kept within a service registry. The registry publishes and locates services. To allow for service discovery, the registry also

Figure 2.2: Web Service protocol stack from IBM's view

provides standardized description facilities, e.g. taxonomies that allow the description of (i) the functionality of a service; (ii) its service provider; (iii) how to access and interact with the service. Figure 2.1 shows the major web service roles and how they interact with each other.

2.1.2 Web Service Protocol Stack

A second option for viewing the web service architecture is to examine the emerging web service protocol stack. The stack is still evolving, but it currently has following 5 major layers[13] as shown in Figure 2.2.

- **Network Protocols:** This layer is responsible for transporting messages between applications. Currently, this layer includes hypertext transfer protocol (HTTP)[5], Simple Mail Transfer Protocol (SMTP)[14], file transfer protocol (FTP)[6], and newer protocols, such as Blocks Extensible Exchange Protocol (BEEP)[15].
- **XML-Based Messaging:** This layer is responsible for encoding messages in a common XML format so that messages can be understood at either end. Currently, this layer includes XML-RPC[16] and SOAP[2].

- **Service Description:** This layer is responsible for describing the public interface to a specific web service. Currently, service description is handled via the Web Service Description Language (WSDL)[1]. The service profile of DAML-S[8] is a good candidate for the semantic service description and matchmaking.
- **Service Publication and Discovery:** The *service publication* is regarded as a business related description of a service; e.g. answering questions such as: What products are associated with this service? Which organization is offering this service? Universal Description Discovery and Integration (UDDI)[3] is in charge of the *service discovery* layer. UDDI registry contains data in a combination of white pages, yellow pages, and green pages. These data assist service requester to locate the proper service.
- **Service Flow:** This layer is responsible for prescribing an XML format for specifying service composition (also called service flow). Currently, several specifications are available: WSFL[17], XLANG[18], BPEL4WS[19], OWL-S[7].
- **Orthogonal Layers:** Several layers are orthogonal as shown in Figure 2.2 because issues like security, management and quality of service span over all the other layers.

Both of these views emphasize the importance of Web Services discovery. As mentioned in the service protocol stack, the quality of service (QoS) should span over all the other layers. This helps the service provider to achieve competitive advantage by offering better QoS. It protects the service provider's resource investment as well. Meanwhile, this is a prerequisite to incorporate the Web Services into the mission critical applications. Currently, the standard Web Services discovery framework does not provide value-added services for the QoS-Aware discovery. In the next section, we will describe in depth the Web services discovery protocols and framework to identify the limitations in the current state of Web Services discovery.

2.2 Web Services Discovery

2.2.1 SOAP

Simple Object Access Protocol (SOAP)[2] provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML. A SOAP message consists of an envelope containing an optional header and a required body. The header contains blocks of information relevant to how the message is to be processed. This includes routing and delivery settings, authentication or authorization assertions, and transaction contexts. The body contains the actual message to be delivered and processed.

Currently a number of SOAP implementations are available for the Web Services clients. Among them there are Apache Axis[20], MS .Net, Apache SOAP, etc. Among them Axis is the defacto Java Web Services' client. A number of major vendors, such as IBM, Sun, Oracle, etc. are using Axis as their base of the Web Services system. For the maximum compatibility with the industrial systems, we choose the Axis as our default Web Service client.

2.2.2 WSDL

Web Services Definition Language (WSDL) is a specification defining how to describe web services in a common XML grammar. WSDL describes four critical pieces of data: Interface, Data type, Binding and Address information. Given a WSDL file, a SOAP client can be manually created to invoke the service, or automatically initialized to invoke the service via a dynamic invocation tool[21, 22, 23]. WSDL, therefore, represents a cornerstone of the web service architecture, because it provides a common language for describing standard Web Services and sharing binding information.

Function Name	Description
Find_xxx Functions	
find_binding	Searches for bindings associated with a specified service
find_business	Searches for businesses that match the specified criteria
find_service	Searches for services associated with a specified business
find_tModel	Searches for tModels that match the specified criteria
get_xxx Functions	
get_bindingDetail	Retrieves a complete bindingTemplate record
get_businessDetail	Retrieves a complete businessEntity record
get_serviceDetail	Retrieves a complete businessService record
get_tModelDetail	Retrieves a complete tModel record

Table 2.1: Main functions of the UDDI Inquiry API

2.2.3 UDDI

UDDI, for Universal Description, Discovery, and Integration, is a specification that defines a service registry of available Web Services, serving as a global electronic "yellow pages". It allows a company to publish a description of available goods and services to the registry, thus announcing itself as a service provider. Service requesters can send requests as SOAP messages to the service registry to discover a service provider for obtaining goods or services. Upon finding a potential business partner, the service requester can easily integrate with a service provider to begin the e-business process.

Table 2.1 provides an overview of the main UDDI inquiry functions. Inquiry functions are further subdivided into two groups: find_xxx functions provide general search functionality, whereas get_xxx functions retrieve full records based on unique key values.

The publishing API is divided into three parts: authenticating users, saving data, and deleting data. Publishing UDDI data can only be performed by authorized users, whose token can be obtained by *get_authToken* request.

A number of UDDI client implementations are currently available, such as UDDI4J[24], SOAP:Lite[23]. They facilitate the SOAP client to search or publish UDDI data, without getting mired in the complexities of the UDDI API.

Although UDDI is currently the industry standard for the web service discovery, it has certain limits on QoS-Aware, checking, customization, Context-Aware, service description lease period, authorization and authentication, etc.¹ Therefore, value-added enhancement is required for the current Web Services discovery practices, and we will concentrate on the QoS-Aware and semantic expression issues for Web Services discovery.

2.3 Semantic Web

The Semantic Web provides a common framework that allows data to be shared and reused across application, enterprise, and community boundaries. It is a collaborative effort led by W3C with participation from a large number of researchers and industrial partners. It is based on the Resource Description Framework (RDF), which integrates a variety of applications using XML for syntax and URIs for naming.

2.3.1 Layers of Semantic Web

The Semantic Web principles are implemented in the layers of Web technologies and standards. The layers are presented in Figure 2.3. The Unicode and URI layers make sure the interoperability of international characters sets and provide means for identifying the objects in Semantic Web. The XML[25] layer provides a surface syntax for structured documents, but imposes no semantic constraints on the meaning of these documents. Namespace in XML are universally unique qualified names, which can be separated into a namespace prefix (universally managed URI) and a local part (document's own namespace). XML Schema is a language for restricting the structure of XML documents and also extends XML with datatypes. With RDF[26] and RDF Schema[27] it is possible to make statements about objects with URI and define vocabularies that can be referred to by URI. The ontology layer supports the evolution of vocabularies

¹In the new UDDI version 3 specification, a better level of data integrity and authenticity is delivered by the support of digital signature.

as it can define relations between the different concepts. Currently, OWL is the recommended ontology language for this layer. It adds more vocabulary for describing properties and classes: relations between classes (e.g., disjointness), cardinality (e.g., exactly one), equality, richer typing of properties, characteristics of properties (e.g., symmetry), and enumerated classes. With the Digital Signature layer for detecting alterations to documents, these are the layers that are currently being standardized in W3C working groups. The top layers: Logic, Proof and Trust, are currently being researched and simple application demonstrations are being constructed. The Logic layer enables the writing of rules while the Proof layer executes the rules and evaluates together with the Trust layer mechanism for applications whether to trust the given proof or not.

Figure 2.3: Semantic Web Layers

2.3.2 The Ontology Language and Description Logics

Ontology is adopted by AI researchers to describe formal domain models. It is defined as “an explicit specification of a conceptualization”[28], i.e., it is a explicitly described domain model (conceptualization). As formal models, ontologies represent knowledge in

a computer processable format, hence improve the communications between humans and computer programs or among computer programs.

An ontology normally contains classes (descriptions of concepts in a domain of discourse), properties (various features and attributes of each concept), and restrictions on the properties. An ontology together with a set of individual instances of classes constitutes a knowledge base.

Developing an ontology has not formed a universal method, but the practical way is discussed in [29]. The development normally includes: defining classes in the ontology, assigning the classes in a taxonomic (subclass-superclass) hierarchy, defining the properties and the constraints on these properties, and filling in the values for the instances. You can find more detailed information in [29]. Software tools are available to assist most aspects of ontology development, for example, OilEd[30] and Protégé[31]. These two tools contain the graphical user interface to view and edit the ontology. They also provide the inference engine to check the ontology and answer some queries.

In the following, we are to introduce more information about the available ontology languages and the description logics.

2.3.2.1 Ontology Languages

The World Wide Web (WWW) contains the largest amount of knowledge, while its presentation cannot be understood by the machine. To provide the machine understandable languages for the WWW, there are two types of ontology languages: one is the HTML-based ontology languages such as SHOE [32] and ontobroker [33]; the other is the ontology languages for the Semantic Web, such as the DAML+OIL [34] and OWL [35].

SHOE [32] is designed for distributed environments with the consideration of the ontology revision. It provides a simple HTML tag's extension, which provides the syntaxes for the ontology definition and revision. It uses description logic as its basic formalism for the inference.

CHAPTER 2. BACKGROUNDS

Ontobroker [33] presents the formalisms and tools for queries formulating, ontologies definition, and HTML annotating. Its inference engine relies on Frame-Logic and supports more complex inferencing for query answering, compared with the SHOE. For example, the negation is allowed in Ontobroker's clause but not the SHOE's. Here are some other differences between SHOE and Ontobroker: SHOE providers of information can introduce arbitrary extensions to a given ontology, while Ontobroker cannot. No central provider index is defined in SHOE, so that the web crawler may miss knowledge portions because it cannot parse the entire WWW. On the other hand, the non-central index and the revision support is well suited for the huge distributed environment.

In the languages designed for the Semantic Web, the ontology layer plays a key role by providing machine readable vocabularies. DAML+OIL [34] is an ontology language designed specifically to be used in the Semantic Web. It is based on RDF and RDF Schema and its well-defined semantics is similar to the description logic $SHIQ(D)$ [36]. DAML+OIL describes the structure of a domain in terms of classes and properties. Like $SHIQ(D)$, DAML+OIL (March 2001) also supports the use of datatypes in class description. By defining the service descriptions upon the semantics provided in DAML+OIL, we can utilize a Description Logic reasoner to make inference and classify these descriptions. The latest Semantic Web's ontology language OWL [35] has evolved to be the recommended ontology language to take the place of DAML+OIL. OWL facilitates greater machine interpretability of Web content than that supported by XML, RDF, and RDF Schema (RDF-S) by providing additional vocabulary along with a formal semantics.

Jena² is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF, RDFS and OWL. It has a well-established API to read and write RDF in RDF/XML, N3, and N-Triples. An OWL API is also supported in the Jena system. Currently it is the best parsing tool to process the Semantic Web

²Available at <http://jena.sourceforge.net/>

ontologies. A lot of other semantic web tools use Jena as the ontology parsing component, such as OilEd[30] and Pellet³. Hence, our system also use the Jena as the ontology's parsing module.

2.3.2.2 Description Logics

Description Logics (DL)[37] is a well-known family of knowledge representation formalisms based on the notion of concepts (classes) and roles (properties). The semantics of OWL and DAML+OIL are based on DL; this allows them to exploit formal results (e.g., w.r.t. the decidability and complexity of key inference problems[36]) and algorithm designs from DL research, as well as to use DL based knowledge representation systems to provide reasoning support for web applications.

Both of these ontology languages have their back-end semantics based on Description Logic (DL). The Description Logic is based on the notion of concepts (classes) and roles (binary relations), and is mainly characterized by constructors that allow complex concepts and roles to be built from atomic ones [38]. A DL reasoner can check whether two concepts subsume each other. In DL, an interpretation $I = (\Delta^I; \cdot^I)$ consists of a set Δ^I , called the domain of I, and a valuation $\cdot^I$ which maps every concept to a subset of Δ^I and every role to a subset of $\Delta^I \times \Delta^I$ such that, for all concepts, roles, and nonnegative integers, their semantics are satisfied[38].

The basic DL S does not fulfill our requirement of reasoning with cardinality restrictions on roles and datatypes. We therefore use the DL $\mathcal{SHIQ}(\mathcal{D})$ to reason with OWL/DAML+OIL descriptions, which include, e.g., cardinality restrictions on roles, and datatypes. A more detailed discussion of DL, which can be found in [37], is out of the scope of this chapter.

Modern DL reasoners apply the tableau algorithms [39] to check the concept's satisfiability problem in the ontology. Concept C is satisfiable iff there exists an interpretation

³Available at <http://www.mindswap.org/2003/pellet/>

Table 2.2: Semantics of Description Logic *SHIQ*

Construct Name	Syntax	Semantics	
atomic concept	A	$A^I \subseteq \Delta^I$	S
universal concept	$\top$	$\top^I = \Delta^I$	
atomic role	R	$R^I \in \Delta^I \times \Delta^I$	
transitive role	$R \in \mathbf{R}$	$R^I = (R^I)^+$	
conjunction	$C \sqcap D$	$C^I \cap D^I$	
disjunction	$C \sqcup D$	$C^I \cup D^I$	
negation	$\neg C$	$\Delta^I \setminus C^I$	
exists restriction	$\exists R.C$	$\{x \mid \exists y y.\langle x, y \rangle \in R^I \text{ and } y \in C^I\}$	
value restriction	$\forall R.C$	$\{x \mid \forall y y.\langle x, y \rangle \in R^I \text{ implies } y \in C^I\}$	
role hierarchy	$R \sqsubseteq S$	$R^I \subseteq S^I$	H
inverse role	R^-	$\{\langle x, y \rangle \mid \langle y, x \rangle \in R^I\}$	I
number	$\geq nR$	$\{x \mid \#\{y.\langle x, y \rangle \in R^I\} \geq n\}$	N
restrictions	$\leq nR$	$\{x \mid \#\{y.\langle x, y \rangle \in R^I\} \leq n\}$	
qualifying number	$\geq nR.C$	$\{x \mid \#\{y.\langle x, y \rangle \in R^I \text{ and } y \in C^I\} \geq n\}$	Q
restrictions	$\leq nR.C$	$\{x \mid \#\{y.\langle x, y \rangle \in R^I \text{ and } y \in C^I\} \leq n\}$	

I such that $C^I \neq \emptyset$. There are a number of DL reasoners developed by different research groups and organizations, from semantic web research community and description logic community. Some of the representative reasoners are RACER⁴, FaCT⁵, Pellet⁶ and KAON⁷. As a basic family of DL, *SHIQ* is supported by most of DL reasoners. To choose a most suitable one, we show a comparison for them in Table 2.3. In our system, we choose RACER version 1.9 taking into consideration of its easy to use interface and versatility in comparison with others.

2.4 Semantic Web Application for Service Discovery

This section will introduce the Semantic Web's utilization in Web services discovery area. The OWL-S ontology for services will be presented first, followed by some related works.

⁴Available at <http://www.sts.tu-harburg.de/~r.f.moeller/racer/>

⁵Available at <http://www.cs.man.ac.uk/~horrocks/FaCT/>

⁶Available at <http://www.mindswap.org/2003/pellet/>

⁷Available at <http://kaon.semanticweb.org/>

CHAPTER 2. BACKGROUNDS

	RACER	FaCT	Pellet	KAON
Logic	DL	DL	DL	
Language	Lisp Style	Lisp Style	Java	
Interface	DIG, Java, GUI	DIG, Cmd-Line	DIG, Java	Java API
query	nRQL		RDQL	spaRQL
Limitation	HTTP overhead	No Abox support	Performance	

Table 2.3: Comparison between Various Reasoners

OWL-S [7] is an OWL ontology for describing Web services. Through the tight connection with OWL, OWL-S aims to make Web Services computer-interpretable and to enable automated Web service discovery, invocation, composition and monitoring. It defines the notions of a Service Profile (what the service does), a Service Model (how the service works) and a Service Grounding (how to use the service).

The service profile is the key component for the service discovery. Its components can be divided into four sections:

- Service Profile provides a superclass of every type of high-level description of the service. There is a two-way relation between a service and a profile which are expressed by the properties *presents* and *presentedBy*.
- Service Name, Contacts and Description present some properties of the profile to provide human-readable information.
- Functionality Description Profile ontology defines the following properties of the Profile class for pointing to IOPE's (Input, Output, Precondition and Effect): *hasParameter*, *hasInput*, *hasOutput*, *hasPrecondition*, *hasEffect*.
- Profile Attributes define additional attributes of the service, including the quality guarantees that are provided by the service, possible classification of the service, and additional parameters that the service may want to specify.

OWL-S ontology combines the explicit and implicit representation for the Web Services[40]. OWL-S' process model explicitly defines the work process ontology of the Web Service. The Web Service operation is defined in the process ontology directly, and the process hierarchy's matching is a straightforward task. However, the process hierarchy's potential definition space is too big for the ontology's creation work, since the number of its varieties is exponential to the number of composed service. The alternative way to represent the service is to define the services according to their interfaces. The OWL-S' service profile implicitly defines the state transition of the Web Service's input to its output. This definition requires only the concepts that describe the domain of the service, hence it greatly reduces the domain experts' work for the service's definition. However, the matchmaking engine needs to infer the process from its implicit representation for the correct behavior matching.

OWL-S provides a good representation of service's functional capability. However, Cardoso et al. [41] point out that significant improvement for the QoS model should be made to supply a realistic solution to DAML-S users. One current limitation of DAML-S QoS model is that it does not provide a detailed set of classes and properties to represent quality of service metrics. The QoS model needs to be extended to allow a precise characterization of each dimension. This is an important motivation of this research work for the QoS ontology, and we later converted it to suit OWL-S as well.

Web Service Modeling Ontology (WSMO) [42] is another important work to define Semantics for Web Services. WSMO provides a conceptual framework and a formal language for semantically describing all relevant aspects of Web Services. There are four main elements: Ontologies, which provide the terminology used by other WSMO elements; Web Service descriptions, which describe the functional and behavioral aspects of a Web Service; Goals that represent user desires; and Mediators, which aim at automatic handling of interoperability problems between different WSMO elements. WSMO defines

CHAPTER 2. BACKGROUNDS

a list of non-functional properties for Web Services in its specification. The matchmaking algorithm for these non-functional properties needs further study. More research should be done on the measurement and monitoring system for service performance as well to make it feasible in Web Services practices.

Several research works for the Semantic Web Services' discovery is studied in the last few years. Here are some examples:

Paolucci, Kawamura et al.[43, 44] present an algorithm that deals with DAML-S specification language. It considers the matching of input/output concepts defined within the same ontology. The matching algorithm defines the matching degrees for the concepts in the ontology, including exact, plugIn, subsume, and fail. By calculating the matching degree of service's interface, the service score is computed for the comparison.

Based on Tversky's [45] feature-based similarity model, Cardoso [46] presents an ontology similarity matching algorithm that facilitates and assists the composition of Web Services. The main objective of this approach is to assess the similarity of terms and concepts in the services' interfaces. The workflow systems and discovery mechanisms can use this algorithm to find Web Services with desired interfaces and operational metrics.

Lei Li et al. [47] presents a software framework for matchmaking based on the Description Logic. The purpose of this framework is to support service advertisement and discovery in e-commerce. They argue that the information constraint in the DAML-S profile is too restrictive for effective information. As a result, they choose only the essential parts of the service profile for the matchmaking. The experiment result shows that the publish and classification for large numbers of advertisements are time consuming. However, the matching of queries to advertisements is performed very efficiently.

In conclusion, works for semantic Web Services mainly cover the definition of the Web Services ontology, design of matchmaking algorithm for functional and nonfunctional properties, implementation of the system prototype, etc. We will focus on the QoS

ontology design pattern, its matchmaking algorithm and measurement system in this thesis.

2.5 Web Services QoS

Web Services are more tend to be influenced by the QoS issues. Some important QoS metrics of Web Services are introduced. With the widespread proliferation of Web Services, QoS will become an important selling and differentiating point for these services.

2.5.1 QoS Metrics

QoS covers a whole range of techniques that match the needs of service requesters with those of the service provider's based on the available resources. It refers to characteristics of Web services that can be measured, improved, and, to some extent, guaranteed in advance. In this dissertation, we choose to design an extensible QoS model, which includes generic and domain specific metrics. The generic metrics are applicable for all Web services. Although the number of metrics introduced in the section is limited, we keep in mind that the model be extensible. New metrics, no matter generic or domain specific, can be added into the system without fundamentally altering the underlying matchmaking mechanism, as shown in chapter 3 and 4. In particular, it is possible to extend the QoS model to integrate the service data characteristics, or to integrate service QoS metrics for the business related criteria.

As to the generic QoS metrics, we choose the service level quality since it is the quality that a user perceives when actually using Web services. It contains performance metrics, including response time, throughput, and stability metrics, including availability, reliability, and accessibility.

Performance is how fast a Web service provider responds to any service request. The performance can be described in sub-metrics such as response time and throughput.

CHAPTER 2. BACKGROUNDS

- **Response Time:** It means the time taken to send a request and to receive the response. The Response Time is measured at an actual Web service call and it can be calculated by applying the following formula. The Response Completion Time is the time that all the data for response arrives at a user, while the User Request Time is the time when the user sends a request. In general, the Response Time is calculated by the mean value during a certain time.

$$\text{Response Time} = \text{Response Completion Time} - \text{User Request Time}$$

- **Throughput:** It means the max number of services that a platform providing Web services can process for a unit time. Throughput can be used as a performance index to evaluate a Web services provider. How many it can process also means how many users it can process concurrently in a web. The Maximum Throughput can be calculated with the following formula.

$$\text{Throughput} = \frac{\text{Complete Requests}}{\text{Unit Time}}$$

Stability means how stable and continuously Web services can provide services. The stability metrics include availability, reliability, and accessibility.

- **Availability** is defined as the ratio of time period in which a Web service exists or it is ready for use, that is, the Web service is maintained. Assuming that the time when a system is not available is 'Down Time' and the time when a system is available is 'Up Time'. the Availability is the average Up Time. To get Availability, instead of monitoring Up Time continuously, we suggest using the Down Time. Down Time could be obtained by monitoring system down events occurred in operation. The following formula calculates the Availability while unit time is a time to measure the time.

$$\text{Availability} = 1 - \frac{\text{Down Time}}{\text{Unit Time}}$$

- Reliability corresponds to the likelihood that the service will perform when the user demands it. We treat the Web Services as incontinuous processes. Each service has two distinct terminating states: one indicates that a web service has failed or aborted; the other indicates that it is successful or committed. We use the stable reliability model proposed by [48], for which the reliability of a web service is a function of the failure.

$$\text{Reliability} = 1 - \frac{\text{Number of Failure Messages}}{\text{Number of Request Messages}}$$

- Accessibility represents the degree that a system is normatively operated to counteract request messages without delay. In some cases, a Web service system could be accessible for external users to try accessing its resources even if its services are not available. We can know whether a Web service system is accessible by just inspecting that the system can returns an acknowledgment normally for a request message. Thus, Accessibility can be calculated as the ratio of number of acknowledgements received to the number of request messages.

$$\text{Accessibility} = \frac{\text{Number of Acks Received}}{\text{Number of Request Messages}}$$

The number of business related metrics can vary in various domains. For example, in phone service provisioning domain, the cost per minute and the fixed monthly charge are important decision factors for the users. In our sample application, the business related metrics include the cost per service.

- Cost is the amount of money which a service requester has to pay to the service provider to use a web service such as checking a credit, or the amount of money the service requester has to pay to the service provider to get a commodity like an entertainment ticket or a pay for the phone service. Web service providers either directly advertise the execution price of their services, or they provide means for potential requesters to inquire about it.

These are a list of metrics we considered in our system. We choose several key metrics from them in our prototype system to demonstrate the feasibility of our system. For the additional metrics (either generic or domain specific) required by the user, such as the information and data quality, they can be added without fundamentally influencing our underlying service discovery prototype. Since the sub metrics of stability has close meanings, we choose the reliability as the metric in our prototype for the demonstration. The metrics we have chosen include the response time, reliability, cost, throughput, etc.

2.5.2 QoS Bottlenecks

Compared with the general software, Web Services will encounter more performance bottlenecks due to the limitations of the underlying messaging and transport protocols. The reliance on widely accepted protocols such as HTTP and SOAP, however, puts a tough burden on the QoS aspect of Web Services.

HTTP is a best-effort delivery service. It is a stateless data-forwarding mechanism which tends to create two major problems: 1), There is no guarantee of packets being delivered to the destination; 2), There is no guarantee of the order of the arriving packets.

SOAP performance is further degraded because of the following aspects: 1), Extracting the SOAP envelope from the SOAP packet is time-expensive; 2), Parsing the contained XML information in the SOAP envelope using a XML parser is also time-expensive; 3), There is not much possible optimization chances with XML data; 4), SOAP encoding rules make it mandatory to include typing information in all the SOAP messages sent and received. Encoding binary data in a form acceptable to XML results in overhead of additional bytes added as a result of the encoding, as well as processor overhead performing the encoding/decoding.

Other factors affecting Web Service performance include Web Server response time and availability, original application execution time like EJB/Servlets in Web application server, back-end database or legacy system performance, etc. Since many aspects

influence the Web Services QoS, a good service level takes service providers their extra resources and efforts. Proper service discovery mechanism is necessary for selling and differentiating point of these services to protect the service providers' benefits.

Due to these performance burdens on Web Services, the Web Services' QoS aspects are tend to be influenced by various conditions. This leads to the demand of QoS-Aware service discovery, which would provide value-added services to advertise/discover a proper service.

2.6 Summary

In this chapter, we have reviewed the basics of roles and protocols in Web Services architecture. The UDDI registry plays the central role in the industry standard for the service matchmaking. We introduce the OWL-S ontology for the semantic Web Services. The specific requirements for the Web Services' QoS are discussed. Compared with the traditional distributed systems, the QoS of Web Services is more sensible to various factors. This makes the QoS issues one of the most important considerations for Web Services.

QoS-Aware discovery of Web Services is an important issue. Different from the selection of services, the discovery approach combines the functionality and QoS phases in one process. It returns all the functional services, with the high performance services ranked higher. On the other hand, selection approach is a separate step that chooses the one of the highest quality of services among the functionally applicable services according to the requester's criterion. In the following chapters, we present our QoS-Aware service discovery systems. Chapter 3 presents the work of the QoS-Aware extension for UDDI registry. The OWL-QoS ontology, a complement ontology for OWL-S, will be described in chapters 4 and 5.

Chapter 3

QoS-Aware Service Discovery without Ontology Support

In this chapter we analyze the current QoS aspect problems in the Web Services discovery area. To address these problems, we design UDDI eXtension (UX), an enhancement for UDDI that facilitates requesters to discover services with QoS awareness. In this system the service requester invokes and generates feedback reports, which are received and stored in local domain's UX server for future usage. By sharing these experiences from those requesters in the local domain, the UX server summarizes and predicts the service's performance. A general federated service is designed to manage the service federation. The discovery between different cooperating domains is based on this general federated service, and therefore the links between domains are maintained dynamically. The system handles the federated inquiry, predicates the QoS difference among different domains, and provides a simple view over the whole federation. Meanwhile the UX server's inquiry interface still conforms to the UDDI Specification.

3.1 Introduction

With the industry's efforts on web services, a huge number of web services are being developed and made available on web. Organizations wish to be able to offer electronic

services worldwide and this creates several technical problems. First, being able to discover what services are available. Second, being able to determine which services match your specification. Third, being able to control which services are advertised to whom. Fourth, being able to assess previous and current service usage pattern for future selection.

Service requesters are presented with a group choice of service offers that provide similar services. Different offers may have different Qualities of Service (QoS). This will require more sophisticated patterns of service discovery and negotiation, for example, the trade-offs between quality and cost. This may involve the introduction of another service, a trading service involved in the process of determining the QoS of various service offers. Current UDDI [3] registries are neither accountable nor responsible for the QoS described in service offers.

The solution being proposed here is called UX (UDDI eXtension). In the system, QoS feedbacks made by service requesters are used to generate summaries for invoked services. These history summaries are then used to predict the services' future performance. The extended inquiry interface in UX is the counterpart of inquiry interface in UDDI and it conforms to the UDDI Specification. A lookup service interface is proposed for the UX framework that can be administratively federated across network boundaries. The lookup interface facilitates the discovery between different registries and the exchange of service QoS summaries.

3.2 Related Work

The UDDI specification [3] provides no QoS related inquiry in the discovery interface. Service requesters cannot filter the unqualified service nor can they get and compare between different services without testing them first. In addition, the approaches to Web services discovery can be classified as centralized and decentralized. Single UDDI falls

under fully centralized approach that supports replication, however replicating the UDDI data is not a scalable approach. QoS-Aware enhancement and decentralized registry approaches have been studied for Web Services discovery.

To solve QoS-Aware service discovery problem, works have been done to enhance the UDDI registry's inquiry/publish interface to embed the QoS information in the message. UDDIe project [49] is targeted mainly towards the QoS-supported interface enhancement for UDDI. UDDIe extends the UDDI registry to support searching on attributes of a service and develops the find method to enable queries for UDDI with numerical and logical (AND/OR) ranges. QoS management support is provided through the definition of QoS attributes in the extended UDDI API. The QoS information is provided by the service provider when they publish the service.

The semantics mapping from DAML-S service description into UDDI is proposed in [44]. It describes how the tModel can be designed to publish and search the DAML-S services in the UDDI registry. Since the DAML-S has limited metrics definition for QoS-Aware service discovery, the mapped tModel structure does not support this directly. The service discovery for Multi-Ontology in a federated registry environment is studied in [50]. This paper utilizes the JXTA[51] technology to connect the UDDI registries. They study the matchmaking algorithm designed for the Multi-Ontology environment, while the QoS issues and QoS differences between domains are unconsidered in this work.

Web Service QoS (WS-QoS)[52] defines the XML schema for Web Services to describe their services' high and low level QoS properties. The assistant framework is designed for the language specification to assist the service selection and publish. High level QoS requirement may be mapped to the actual QoS-enabled transport layer through its proxy. They provide assistant tools to help user to manage and inquiry the services descriptions. The WS-QoS adopts the provider's published QoS information. There is neither pragmatic information nor federated discovery available in this system.

As to the management of the registry network, the decentralized registries approach is studied in the related works to improve the scalability of the service discovery. In [53] the authors show that the cooperation between traders can be useful when a trader cannot provide sufficient service to a customer. Paolucci et al. [54] propose a pure P2P service discovery network and show how to perform matching capability between web services on the Gnutella network. This approach avoids a single point of failure and there's no danger of a bottleneck effects. Ping/Pong process is used to discover other server nodes. It is more appropriate in dynamic environments.

The inter-domains links can be statically established amongst individual registries nodes. Examples of static establishment include the CORBA Trading Service, ODP Trader[55], and DNS systems[56]. The DNS requires one to provide the names to bind to the target services, therefore it does not help the service requester to choose the best service. The CORBAR Trading Service and ODP Trader allow the service provider to advertise or export their service offers to the traders. They have been designed to find the servers for each requester according to a service type and a list of properties. However, they do not offer any tool for linking federated traders dynamically.

3.3 Network Model and Design Considerations

In this section, we discuss the abstracted network model and some design considerations applied in our UX architecture.

3.3.1 Network Model

In our system, the network model is abstracted into domains. Each domain has relatively high-bandwidth, low-latency, and uncongested connections. For the local domain's users, the Web Services QoS is mainly affected by the server's performance. However, the

Figure 3.1: Network Model

properties of connections between different domains are unknown. Different inter-domain connections can have quite different qualities. (See Figure 3.1)

In our system, these domains mainly stand for organization domains such as enterprises, universities, and so on. These organizations normally have good established internal networks as assumed in our network model. The connections between different organizations are uncertain. These organizations federate with each other by contracts. A local UDDI registry works in each domain for web service's discovery. It registers the local domain's services. To discover the external Web Services, the service requesters need to query external UDDI registry in the cooperative organization. Since different inter-domain connections have quite different qualities, the QoS-Aware discovery across domains need to be treated specifically.

3.3.2 Design Considerations

Our architecture incorporates five important design considerations that offer value-added services over standard UDDI:

CHAPTER 3. QoS-AWARE SERVICE DISCOVERY WITHOUT ONTOLOGY SUPPORT

- First, the architecture should be aware of the provider services' QoS. This can be achieved by sharing the service requester's experiences, i.e., the QoS reports of services are sent back and shared to predict the services' QoS. We rely on service level measurements, such as response time or reliability to help service requester's decision. New domain specific metrics can be added in our system without fundamentally change the underlying system. Many network level measurement system exists in which they use metrics such as routing metrics[57], link bandwidths[58], geographic locality[59], etc. to predict the relative performance of different hosts. Unfortunately, these metrics often do not correlate with service level performance. Servers' conditions such as load, popularity, etc. may also affect the service's performance.
- Second, the measurement results can be shared[60]. Requesters in the same domain can explicitly share the QoS reports by sending them to the local UX server. By sharing these experiences, the requesters are of better chance to get the knowledge of the service's condition, rather than making many test invocations. This saves the service requester's costs, and differentiates the qualified service providers. The decision to share measurements can be followed directly from our network model. Two requesters in the same domain are going to observe similar service performance, because of the good connection conditions in the domain. Measurements made in other domains may not be utilized directly, since the properties of the inter-domain connection is unknown. The way to process the QoS differences among different domains is described in section 3.5.3.2.
- Third, customization should be available in the discovery procedure[61]. Different requesters may have different preferences for the QoS-Aware service discovery. For example, some requesters prefer shorter service response time, while others prefer

lower cost. In order to help the requesters locate the most suitable services, our system should allow the requesters to describe their preference in their profiles. The result list is also generated according to their preference. The preference information can be stored in the user profile on server. If so, service requesters need to provide their identity in service discovery process to support the customization.

- Fourth, the extended inquiry interface is suggested to conform to the UDDI specification. The requesters hence can use their original discovery software to make queries. As to the federated discovery, the interface can still remains the same, since the requester still use the same way to inquiry the UX system, while the federated discovery is performed by the local UX system on behalf of the requester. If the requester wants to fully utilize the value-added features of the system, such as customization and authorization, client-side software support is needed.
- Last but not least, additional policies should be designed to manage behavior of the registry. For example, *intended_number* policy should be defined to describe the service requester's wanted number of services in the result. *hop_count* can be designed to control the depth of the query's propagation in the UDDI federation. Some other policies, such as cache's *living_time*, can also be designed to control the behavior of a registry. Our supported policies will be introduced in section 3.4.5.

3.4 Architecture

In this section we present the components of the UX system architecture.

3.4.1 Components of UX System

Figure 3.2 shows a diagram of the components of our architecture. It is comprised of service requester, local UDDI registry, Test host, and UX Server.

Figure 3.2: Components of UX

- *Service Requester* The service requester is the client who invokes the Web Service and sends the service’s performance feedback to the UX system. The performance feedback is generated when it invokes the Web Service. The feedback, containing QoS reports of service’s performance data, is send back in batches to reduce the traffic overhead. We offer the service requester an automatic tool to generate and send these QoS reports. The reporter works as a SOAP intermediary and no code modification is needed from the requester side.

To avoid the feedback report being manipulated by the service request, our system defines the policy to control the service requester’s feedback. Each requester needs to have a valid user id and password to successfully feedback the QoS reports. Furthermore, the id and password must be verified by the service provider to ensure that only the actual service requester of the service is allowed to feedback the service. The service provider can check the value of their QoS information in the UX system to assist their modification of the system.

- *Local UDDI Registry*

The local UDDI registry is a private UDDI registry that accepts the publication of the local domain's services. This registry is the backend service registry for the local domain's UX system. The UX system uses SOAP protocol to connect to this registry to ensure the generic of the registry.

- *Test Host* The test host is a Web Service client that periodically invokes the providers' services to generate the feedback reports. It is needed when some local domain's services are rarely invoked in the local domain to get the updated QoS report. We provide a test host tool so that it tests the service with random or pre-defined parameters to gain service reports. The test interval is carefully selected so that the test host will not create obvious overhead to the network and its QoS reports only occupy a small portion of the total reports.
- *UX Server* The UX server receives the service requester's inquiry, ranks the services, and returns the services list to the service requester in the order of their ranks. When the UX Server receives an inquiry from the requester, it searches the local UDDI for related results. If the number of items returned from the local registry is not enough according to the requester's requirement, the federated discovery is triggered for more results from the other domains, which will be discussed in detail in section 3.5. After collecting all these results, the UX Server merges these results and returns them back to the requester. If the inquiry is about the services rather than the provider detail or the service details, the UX server will sort these services according to their QoS ranks. The UX Server is also responsible for receiving the requester's QoS reports for the storage and processing.

3.4.2 QoS Metrics

As we discussed in the section 2.5.1, we use an extensible QoS model to include generic and domain specific metrics. The metrics we choose are response time, cost, and reliability. The definitions of these metrics can be found in section 2.5.1.

The UX Server generates a summary of the reports for each service regularly. It calculates the response times, terminating state and cost in each received QoS report to a summary. This summary records the information of response time, reliability, cost, timestamp, and report number.

To generate the summary, a function is needed to aggregate the received record list and draw a mean value. A number of mean functions are available, such as the weighted arithmetic mean ($\bar{x} = \frac{(\sum_{i=1}^n w_i * x_i)}{(\sum_{i=1}^n w_i)}$), and power mean ($\bar{x}(m) = \sqrt[m]{\frac{1}{n} * \sum_{i=1}^n x_i^m}$). Since each record is treated equally, we choose the weights and power numbers to be 1. In addition to the normal mean function, we select and compare low-pass filter functions as well for generating the summary's fields:

- (i) *average function*: $f = (\sum_{i=1}^n r_i)/n$, where r_i is the i^{th} received QoS report's field value during the interval of two summary, and n is the total reports number during the interval.
- (ii) *low-pass filter function*: $f_i = \alpha * f_{i-1} + (1 - \alpha) * r_i$ [62], where α is a smoothing factor, r_i stands for the i^{th} sorted record's field value during the interval of two summary, and f_i stands for the calculated field value when processing the i^{th} report. The value f_0 is initialised by the earliest summary's field value. To apply this function, QoS reports in the interval are first sorted by timestamp and then processed sequentially according to their position. This method helps the summary to include all the history measurement information. If α is set as 0.8, eighty percent of each new calculated f_i is gained from the previous calculation, and twenty percent is from the report's value.

- (iii) *median function*: median value of the related field during the interval of two summaries is selected.

The comparison and experiments for these summary functions will be presented in Section 3.6.2.

3.4.3 Communications between Components

The system uses a 'pull' approach to discover the services. Requester can find the related services by sending inquiry messages to the UX Server. The processing step is listed as follows (Figure 3.2):

Algorithm 1 Algorithm for Service Discovery

```

1: Requester.inquiry(UX_Server, query, maxNumber)
2: result = UX_Server.lookupCache(query)
3: if result is null then
4:   result = UX_Server.find(UDDI, query)
5:   if |result| < maxNumber then
6:     UX_Server.federatedLookup(query, maxNumber)
7:     result = UX_Server.federatedLookupResultMerge()
8:   end if
9: end if
10: UX_Server.saveCache(query, result)
11: if query.type = "find_server" then
12:   result = UX_Server.sortResult()
13: end if
14: UX_Server.returnResult()

```

The requester sends the inquiry to the UX Server first. This inquiry can contain the requester's profile identity information in the HTTP head, which is used to get the user's customized weights for various QoS metrics. The UX Server checks the cache to see whether the cache can provide the result. If the cache does not contain the result, the UX Server sends the query to the local UDDI registry and gets the result from it. After that, the UX Server checks the record number in the result. If the number is less than the

requester's intended number, it starts the federated discovery to get more results and then merges the results. If the service number satisfies the service requester's requirement, the federated service discovery will not be initiated. If the inquiry is service related, UX Server will sort the result according to QoS summary and requester's preference. Otherwise the UX Server returns the result directly. The format of the returned result conforms to UDDI specification.

3.4.4 Customization

Service requester may have different preference on the service's QoS metrics. By customization of his user-profile the requester can set up different weights for different metrics to stand for his preferences. Currently we provide a web interface to help the requester set up his profile. To utilize the user-profile, the service requester should put his identity in the inquiry's HTTP head. The UX server extracts his profile information from the database according to the identity. If no weight information is available, weight value 1 is used, and each metrics are treated equally.

If the inquiry is *find_service*, the UX Server will sort the service according to their ranks. The generation of the service's rank contains two steps: normalization of the QoS summary and score calculation.

In the UX Server's normalization phase, each metric field is mapped to a value between 0 and 1 and higher normalized value means better score. For the metrics which are worse for the higher value, such as Cost and Response Time, we use the inverse function $S_x = 1/(x + 1)$ for the normalization. x is the field value and S_x is the normalized value. For the metrics which are better for the higher value, we scale its value's range to $[0, 1]$ for the normalization. For example, Reliability value has already been in the range of 0 to 1, so that its value is kept.

In score calculation phase, the service score is calculated by weighted average function, i.e., $score = \frac{(\sum_{i=1}^n)w_i * x_i}{(\sum_{i=1}^n)w_i}$. Applying the metrics in the equation, we get:

$$score = \frac{w_{cost} * S_{cost} + w_{resptime} * S_{resptime} + w_{reliability} * S_{reliability}}{w_{cost} + w_{resptime} + w_{reliability}}$$

where S_{cost} , $S_{resptime}$, and $S_{reliability}$ are normalized summary scores; w_{cost} , $w_{resptime}$ and $w_{reliability}$ stand for the customized weights respectively.

The higher the score is, the better the service's quality. The result is sorted according to this score list so that the top item has the highest score. If some metric field's value happens to be unknown, random value is generated so that the service has a chance to be invoked by the requester. When the requester gets the result, it can easily choose among the several top services in the list.

3.4.5 Important Policies

intended_number policy describes how many resultant items the requester intends to get. It is normally contained in the standard UDDI's inquiry message. If it is not specified in the inquiry message, local only discovery is assumed. Otherwise, if the local discovery can't find enough service results compared to the user's *intended_number*, federated discovery will be executed which will be described in section 3.5.

To fasten the discovery procedure and improve the system's scalability, cache is used for UX Server. We use the LRU cache to store the discovery results and this cache is available only for each domain's requesters. When the user's inquiry is found in the cache table, the inquiry result will be returned directly. Otherwise, the inquiry is processed as the normal procedure. The cache consists of the inquiry, result and timestamp. *living_time* policy for cache is set to define the cache's living time. Any cache item which is older than the *living_time* portion is removed from the cache table. *cache_num* policy adjusts the cache's size. If the table size exceeds this size, the least used item will be removed from the cache table.

hop_count policy is used to control the depth of the query's propagation. When query is propagated one step further, the value of *hop_count* is decreased by one and when the *hop_count* reaches zero, the query is not propagated any more. The *hop_count* can be set statically or described in user's profile. It makes a tradeoff between response time and total result number.

3.5 Federation Discovery Support for UDDI eXtension System

Many enterprises prefer to publish their services on private UDDI registries for internal or cooperation usage. Federated discovery is a way to link these registries with each other and share information among these partners.

The federated discovery system requires the underlying framework to be able to scale and support a potentially huge number of requesters and services while adapting the underlying domains' changes (e.g., due to network partitions and node failure). This requires a proper link management and query propagation model. In this section we present our generic federation service based on the Cooperating Server Graph (CSG) model[63]. Through simple generalization on our federation service, Web Services can be integrated to form the service federation more easily. As far as we know, when we begin to do the research, there's no such generic federation service designed for Web Services.

The general federation service provides the link management and query propagation model. The domain specific query format and query propagation is left for the specific Web Services. We cooperate this design into our UDDI eXtension architecture to form the federation discovery services. It shows an example of how to use the federation service for normal Web Services.

3.5.1 CSG Model

In our system, the abstract network model is described in Chapter 3.3.1. Since the network is abstracted into domains, the links between network domains should be established and maintained for the cooperating partners. The inter-domains links can be manually or statically established amongst individual service nodes.

Our work assumes domain links are managed dynamically. We choose the CSG model[63] instead of pure p2p solutions for these enterprise domains because p2p's dynamic nature may influence the system's efficiency and stability. The p2p's freedom of joining and leaving the communication group also causes serious security threatens to those enterprise domains. In such domains, B2B peering contracts and policies would define the details on how companies can use each other's services. CSG[63] model's links should be established according to these contracts and policies and then managed dynamically.

The aim of the Cooperating Server Graph model (CSG) is to optimize and dynamically manage links between cooperating servers over a wide area network. The CSG model is designed to meet the following goals: (i) minimize the communication costs in the message propagation (according to the distance function); (ii) distribute the propagation costs between all the nodes and prevent bottlenecks; (iii) eliminate the cycles in graph and facilitate the propagation control.

The propagation's topology structure is carefully designed to meet these goals. The shortest path trees are not chosen to avoid generation of star trees, which creates the hot spots in the network. CSG approach uses a minimum-weight spanning tree to optimize links between UX Servers automatically. The weight is defined by a distance function to represent the communication burden between the couple of nodes (e.g., the inquiry latency, the hop number, etc.). Prim algorithm is used to calculate the minimum-weight

tree. In our general federated service design, each node in CSG is called a Federated Server (FS).

The CSG model and propagation tree have to adapt dynamically to the change of cooperating servers and the underlying network topology. In order to be more efficient for the graph's management and reduce the control overhead, different events are treated accordingly. The dynamic administration of CSG takes three levels of events into account:

- (i) **Alternative behaviour:** Once a Federated Server (FS) detects a failure of its neighbor nodes (i.e., when it can't propagate information to one of its neighbors in the tree), the FS uses the alternative behavior in case of failure: it propagates the information, on behalf of the failed neighbor (the communication failure may come from a failed FS or a network failure), to the neighbors of the failed neighbor in the tree. This behavior maintains the continuity of the service and it is feasible because of the global knowledge of the propagation tree.
- (ii) **Local reconfigurations:** The local reconfigurations level is used to take into account FS's long time failure, FS's long time failure recovery, as well as FS's addition and removal. Long time failure can be decided according to predefined failure time threshold. Instead of the propagation of a new CSG version, the local reconfiguration updates the local propagation tree and enables the CSG's minor change at a lower cost. A local reconfiguration is possible if and only if each node, after the reconfiguration, knows its own neighbors and the neighbors of its neighbors in the effective propagation tree. A node cannot participate in two local reconfigurations simultaneously. As a result the reconfiguration progress is made atomic. After accepting a local reconfiguration, a node buffers incoming requests so as to retransmit them at the end of the reconfiguration process. The following algorithm describes the local reconfiguration's process, which uses two phase commit protocol to keep the local propagation tree's integrity.

Algorithm 2 CSG Local Reconfiguration

```

1: FederatedService.multicast(LocalChangeGroup, addServer/removeServer)
2: for all nodes in LocalChangeGroup do
3:   if node.save(addServer/removeServer) then
4:     return success
5:   end if
6: end for
7: if success is returned from all nodes in LocalChangeGroup then
8:   FederatedService.multicast(LocalChangeGroup, commit)
9:   FederatedService.notifyChange(ChangingVersionFederatedServer)
10: else
11:   FederatedService.multicast(LocalChangeGroup, abort)
12: end if

```

- (iii) **Global change of the CSG version:** A Version Change Federated Server, chosen dynamically in the federation, triggers the version change of the CSG. Each FS sends its local long time reconfigurations to the Version Change Server, which includes the long time failures, long time distance changes, addition and removal of the nodes. Then it triggers a new CSG version when the degradation rate of the propagation tree (the sum of the weights of the degraded tree divides that of the minimum spanning tree which is still in use) goes past a given threshold. The new version is then propagated to all the domains' FSs via the propagation tree. Two nodes (sender and receiver) have to agree on a version before they can communicate through CSG. If the receiver has an older version, the request is buffered on the receiver until its version is updated. On the other hand, if the sender has an older version, it will send the request back to the source node which in turn will reinitiate a propagation after updating its CSG version.

This model can tolerate a great number of failures. All the CSG updates are made dynamically. The number of CSG version changes is greatly reduced by defining the different levels of events. In case of too many failures leading to dividing the CSG into several isolated classes, server cooperation is limited inside each class.

3.5.2 Federation Service for Web Services

The Federation Service takes the Web Services layer approach instead of network layer approach to achieve the extensibility and easy adoption. There are two basic communication semantics in the federation service: The local tree modifications and the propagation of messages. When the local tree modification happens, the local tree modification has to be made coherently on a group of neighbor servers which are one and two steps away from the center coordinator of the modification. This modification should guarantee that either all these neighbor server nodes make the modification or none of them does. A service node cannot participate in two local tree modifications simultaneously. In short, the behavior of modification is kept consistent and atomic.

Different from the local tree modification, the query propagation does not need to guarantee the atomicity or the message's ordering. Each federated server in the propagation's chain becomes a coordinator when it receives the information to propagate. It has to forward the information to all its neighbors in the propagation tree, except for the one from which information was forwarded. If one neighbor does not acknowledge, the propagation should still go on and the coordinator triggers the alternative behavior in case of the neighbor's failure.

According to the two different communication semantics, we divide the federated service design into two service groups: the LocalChangeGroup and the PropagationGroup. The LocalChangeGroup provides the atomic and consistent invocation for the local tree modification. The PropagationGroup offers the propagation invocation for the normal messages. The message order and the atomic of this invocation are not ensured.

The local reconfiguration procedure is shown in Figure 3.3. The algorithm CSG Local Reconfiguration shows the details of this procedure. To ensure the atomic modification semantic, the LocalChangeGroup uses a two phase commitment method to perform the reconfiguration. At first the LocalChangeGroup is told to multicast the save method to

CHAPTER 3. QoS-AWARE SERVICE DISCOVERY WITHOUT ONTOLOGY SUPPORT

the neighbors of the federated node and the neighbors of its neighbor. A boolean result is returned to indicate whether the LocalChangeService is ready to process the modification. If all neighboring LocalChangeServices are ready, the commit method is invoked on all the neighboring LocalChangeServices which in turn invoke their add/remove method to modify the local graph. Otherwise, the abort method is invoked and the multicast method returns to indicate the failure of the local reconfiguration. The interface definition is shown in Figure 3.4. All the successful local reconfiguration information is sent to ChangingVersionFederatedServer, which controls the CSG version. It triggers a new CSG version if the degradation rate of the propagation tree (the sum of weights of the degraded tree divides by that of the minimum spanning tree which is still in use) goes past a given threshold.

Figure 3.3: Federated Server's Local Tree Modification Sequence Diagram

The propagation invocation for the normal messages tries their best to deliver the message rather than using the aromatic semantic for the message delivery. To deliver


```
public class LocalChangeAdminMessageData {
    public String adminType; // addServer or removeServer
    public String nodeIdentity;
    public EdgeDetail[] edgesDetail;
}
public class LocalChangeGroup{
    public synchronized boolean multicast(String messageContent);
    public boolean isCommitted(String messageID);
    ...
}
public class LocalChangeService
{
    public synchronized boolean save(String messageID, String messageContent);
    public void commit(String messageID);
    public void abort(String messageID);
}
```

Figure 3.4: Local Tree Change Interface

the message, the PropagationGroup invoke the multicast method to propagate the message to its neighbor nodes. When the neighbor nodes receives the propagated message, it will send the acknowledgment message back. If no acknowledgment is received from the target server, the target server is deemed as temporarily failure and the Propagation-Group triggers the AlternativeBehavior method to bypass the failure node. The following algorithm shows the processing procedure of the tree’s propagation.

Algorithm 3 CSG Tree Propagation

```
1: FederatedService.multicast(PropagationGroup, request)
2: for all nodes in PropagationGroup do
3:   if node has no response then
4:     FederatedService.alternativeBehavior()
5:   end if
6: end for
```

Figure 3.5 shows the interface design for the PropagationGroup. The FederatedMes-

CHAPTER 3. QoS-AWARE SERVICE DISCOVERY WITHOUT ONTOLOGY SUPPORT

sageHeadData describes the head data format for the propagation message. This head data are set by the original sender. The content is only related to the special services to deliver its application data. The messageType describes the kind of message which is being propagated. It contains the application related message type and the control message type. The queryID is set to uniquely identify the query so that the repetition and orderless messages can be reorganized in our system. The PropagationCondition describes the propagation policies for the federation, such as the hop_number.

```
public class FederatedMessageHeadData {
    public String version; // the CSG version.
    public String messageType; // the message type    "UXLookup",
    "VersionUpdate", "SendbackSinceLowerVersion"
    public String queryID;
    public String originalSender; // correspond to the node Identifier
    public String lastSender;
    public PropagationCondition pCondition;
}
public class PropagationGroup
{
    public void multicast(GraphNode bypassedNode, ArrayList theNodes, String head,
String content);
    public void alternativeBehavior(GraphNode failedNode, GraphNode preFailedNode,
String head, String content);
}
```

Figure 3.5: Federated Server’s Propagation Interface

The federated service can be applied to a wide range of web services, especially some message propagation and inquiry services. Armed with federated service, the service provider focuses more on their own business logics while leave the network link maintainance and message propagation problem to our federation system. Our UDDI eX-tension system is an example. We extends the original Federated Server and build our special UXFederatedServer class through inheritance. In this UXFederatedServer class,

we add "UXLookup" message type for federation propagation and define special handling logic for "UXLookup" message. Additional propagation condition can also be extended in federatedLookup method to control the propagation logic. This design achieves the separation of underlying network links' management and UDDI lookup logic's processing, therefore the code is reused for the federated services. The FederatedServer manages the message propagation and failure control, while the UXFederatedServer deals with the special UDDI lookup logic for service discovery. We will introduce our UDDI eXtension system in more detail in the next section.

3.5.3 UDDI eXtension Federation Service Design

We build our UDDI eXtension system based on the reusable Federated Server to provide the specific query services for UDDI. The extended class is called UXFederatedServer, and this section presents how this Server interacts with other UX Servers across network domains to execute federated service discovery.

3.5.3.1 UDDI Federation

We reuse the federated service by inheriting the common routines in FederatedServer and extends the service inquiry related services in UXFederatedServer. In addition to the underlying model, a lookup interface between UX Servers is presented to support the federated discovery. This interface for the UX Server is presented in Figure 3.6. The interfaces contain the UX Server's extended inquiry interface, lookup interface, the original UDDI publish interface and the Admin interface. The extended inquiry interface provides the QoS-Aware Web Services discovery over the original UDDI and local domain's service requesters query through the local UX Server. Lookup interface is designed to support federated discovery between UX Servers. The original UDDI publish interface is kept for service provider to publish their business services. Extended admin interface manages the domain links and policies.

Figure 3.6: UX System's Interface

The input to the Lookup interface is shown in Figure 3.5, and the message head is rendered in the format of XML. The Lookup interface formalises the federated query information for other domain's UX servers' process. It contains mainly query ID, hop number, original sender, last sender and query content. The response from other domain's UX server is also a string of XML that describes the query result and related QoS summary if the query is service related. It contains mainly query ID, sender, query response and QoS summary.

When the original UX Server finds that its local UDDI registry does not have enough results returned, it begins to propagate queries. When the original UX Server receives the results from an external UX Server, it accumulates the result number and checks it with the *intended_number*. The result collection ends when the result number reaches the *intended_number* or the timeout is reached. After the collection, it merges these results into a single result set for the service requester. If the query is service related, the result set is sorted according to the QoS summaries.

3.5.3.2 QoS similarity domains

The similarity of QoS between two domains is a measure of the differences for these two domains' views for the same service. By analyzing the QoS summaries of the local domain, we learn the QoS similarities between the local domain and other domains. Notice that the local domain's feedback QoS reports contain two parts. First part records the QoS of services that is registered in local domain while second part records the QoS of services that is registered in other domain. When some other domain's service QoS summary is returned in the federated discovery, this service's corresponding summary information in the second part of local QoS summaries is located. The difference between these two is calculated for the similarity judgment. The QoS similarity measurement is recored on a regular basis to update the similarity information.

According to our network model 3.3.1, the difference of a service's performances between two domains is mainly caused by the network connection between these two domains. Therefore, a linear function is used to predict the metric response time's value, i.e., to add the average difference directly to the received QoS summary as the predicted summary. The server then uses this predicted summary to sort the result.

3.6 Implementation and Experiment

We have implemented a prototype of the local UX system. The system uses Apache Axis[20], UDDI4J[24] and WSIF[21] as basic components. The Microsoft public UDDI registry is actually used in our experiment as the established local domain UDDI registry. The UX Server is implemented on the Apache Axis platform. Because the service and tModel list is returned in id forms, a mapping between the id and the service's location is generated by the test host and stored in the database for UX Server's usage.

3.6.1 Implementation

The components of the system includes Service Requesters, Test Host and UX Server. In the following, we'll introduce these components' implementation in more details here:

On the requester side, an assistant tool is designed to facilitate the service QoS measurement and feedback. It is a SOAP intermediary which extracts and stores the service's access point by parsing the SOAP message. The service's response time and ending status can be decided by this tool. The requester can further specify the cost information of the service.

Due to the security and customization considerations, the requesters need to have a valid user id and password to execute, feedback and inquiry the service. We choose the basic HTTP authentication[64] to carry the user's id and password since this approach is commonly supported in general HTTP client and server. For example, the latest version of UDDI4J supports this mechanism in its configuration file.

According to the deployment definition, the summary is built regularly. The UX Server processes the newly received feedbacks and the last summary to generate the new summary. The new summary is calculated according to the summary functions introduced in section 3.4.2. After generating the summary, the information will be stored on the backend database for the UX Server's reference.

A web portal is built for the service requesters to modify their profiles and preferences. Requester can login to change their password and metrics' weight values, and these form the user profile. The profile information is used when the UX server sorts the result. The sort algorithm used weighted average algorithm to calculate each service's score, which is described in section 3.4.4.

The algorithm for service discovery presents the way the UX Server processes the requester's inquiries. When the UX Server failed to get enough results for the requester,

CHAPTER 3. QoS-AWARE SERVICE DISCOVERY WITHOUT ONTOLOGY SUPPORT

federated service discovery procedure is triggered. PropagationGroup handles the message multicast and other domain's UX Server will return the results directly back to the original UX Server. The cache can fasten the procedure and improve the performance. Figure 3.7 shows the UML sequence diagram for service discovery in UX Server. The UX Server triggers the tree propagation and multicast the inquiry message in the CSG network. If network error happens, CSG performs the alternative behavior to bypass the failure node. When the inquiry arrives the target UX Server, the server get the results and return them back to the original UX Server. We conclude each major process for the sequence diagram in Table 3.1.

Function Name	Description
getUserProfile	Get the user's profile, which contains the preference of the user.
lookupCache	Check the local cache and load the cache item if exists.
inquiryLocalUDDI	Inquiry the local UDDI registry.
getQoSSummary	Find the QoS Summary of the Discovered Service.
federatedLookup	Start the federated Lookup.
multicast	Propagate the lookup message to the neighborhoods.
specialRequest	Invoke the special request on the neighborhood.
lookup	Look up the registry.
federatedLookupResultMerge	Merge the federated discovered service.
saveCache	Save the discovered cache.
sortResult	Sort the discovered results.

Table 3.1: Description of Functions in the Federated Discovery Sequence Diagram

The UML class diagram of the FederatedServer is given in Figure 3.8. The FederatedServer uses a composite design pattern to compose the LocalChangeGroup and the PropagationGroup. It is itself a generalization of GraphNode for Graph so that each FederatedServer can be easily combined into the Federation Graph. The FederatedServer controls the addition and removal of service nodes, encapsulates the propagation interface and accepts the requester messages from special services. Table 3.2 describes each class in the federated system. The line with triangle ending means the inheritance relationship.

CHAPTER 3. QoS-AWARE SERVICE DISCOVERY WITHOUT ONTOLOGY SUPPORT

Figure 3.7: UX Server's Federated Discovery Sequence Diagram

CHAPTER 3. QoS-AWARE SERVICE DISCOVERY WITHOUT ONTOLOGY SUPPORT

The line with diamond ending stands for the composition relationship. The normal line represents the association between two classes.

Figure 3.8: Federated Server’s Class Diagram

Take the stock service inquiry mentioned in section 1.2.2 as an example: service requester "Steve" has already setup his customization profile on the UX Server. In this profile he specifies that his weight values on response time, reliability, and cost are 2, 1, and 1 respectively. When he sends *find_service* inquiry to UX Server with name = "stock" and *intended_number* = 10, UX Server checks the cache and finds that there's no

CHAPTER 3. QoS-AWARE SERVICE DISCOVERY WITHOUT ONTOLOGY SUPPORT

Function Name	Description
Node	Records the information of the individual UDDI registry.
Graph	The connection information of nodes.
LocalChangeService	The basic services provided for each nodes in the local change group.
LocalChangeGroup	The services provided in the local change group.
PropagationGroup	The services provided in the propagation group.
FederatedView	The logic view for the federation.
FederatedServer	The server implementation for each federated nodes.
UXFederatedServer	The UX Server implementation that inherits the FederatedServer.
ChangeVersionFederatedServer	The version control server of the federation.

Table 3.2: Class Descriptions in the Class Diagram

such inquiry recently. Hence UX Server looks for the local UDDI registry, and it gets 6 returned services as well as their QoS summary. Because the result number is less than the *intended_number*, federated discovery is triggered. The original UX Server sets the timeout as 5*10 seconds (each hop waits for 10 seconds), *hop_number* as 5, and starts the collection of returned results. The first neighbor UX Server gets the federated inquiry, reduces the *hop_number* by 1, propagates the inquiry, and returns 5 serviceInfos together with the QoS summary. The original UX Server receives this service list, and stops the collection to improve the processing speed, since the number of discovered services is already greater than the *intended_number*. As the original UX Server works out the mapping and merging of the QoS summaries, merged results and QoS summaries are stored in the cache, and then sorted according to Steve's preference. This service list is then returned back to Steve.

The UX system's implementation follows the design considerations. The feedback mechanism is aware of the provider services' QoS. In the same domain, the QoS information is shared by the requesters. The requester can do the customization through their profile. We provide a web page for this configuration. The inquiry interface of UX still follows the UDDI specification. Some additional policies are designed to manage the UX

system's behavior.

3.6.2 Experimental Result

Several tests are made to check the validation of predicting according to the metric's summary information. Comparison of summary functions shows that the low-pass filter function has the best predicting result.

In the local test, we set up two similar web services for comparison. The difference between these two services is their working speed, and one of them is faster than the other. The test is conducted in the university's LAN with a node/server pair. During each service invocation, the UX Server collects its service information. Both of these services' costs are zero. More than 400 feedback data are collected in 10 hours.

The summary functions, i.e., the average, the median, and the low-pass filter function, are compared. Figure 3.9 shows the percentage deviation of fast-add services invocations from the feedback value to the summarized response time using the summary functions.

The weighted average of the deviation is calculated to show the difference between the summarized value to the feedback records. Each deviation is the percentage away from the summarized value to the invocation value. Each weight is the percentage of invocations that have the same response time. To judge whether summarized function is good or not, we use the mean deviation as the criteria, since lower deviation means a better estimation of the record values. By comparison of three functions' mean deviations, we find that the low-pass filter function f_i has the minimum deviation (See Table 3.3). Furthermore, if α is chosen as 0.6960, f_i will reach the minimum value of 12.64. Therefore, the low-pass filter function f_i is chosen as the summary function in UX Server.

We further calculated the deviation value of the Slow-Add services, and the low-pass filter function again has the minimum deviation.

Figure 3.9: Percentage Deviation of Fast-Add Services Invocations from the Summary of Response Time

Function Name	Mean Deviation (%)
Average Function	19.03
Median Function	17.38
Low-pass Filter Function with $\alpha = 0.8$	15.10

Table 3.3: Mean Deviation of Summary Functions for Fast-Add Service

Function Name	Mean Deviation (%)
Average Function	2.307
Median Function	2.307
Low-pass Filter Function with $\alpha = 0.8$	2.305

Table 3.4: Mean Deviation of Summary Functions for Slow-Add Service

CHAPTER 3. QoS-AWARE SERVICE DISCOVERY WITHOUT ONTOLOGY SUPPORT

We then choose the public Xmethods' stock service ¹ for the test. More than 900 service invocations' data are collected in 24 hours' experiment. More than 90 percent of the service invocations are within 10 percent away from the summarized value. The reason of the better convergence is that: in the remote web service's test, a substantial part of the response time is due to the network transportation time, which is much higher than the LAN's test. The variance in the remote test is therefore relatively small. Among the three summary functions, the low-pass filter function with $\alpha = 0.8$ still gets the best results (See Table 3.5).

Function Name	Mean Deviation (%)
Average Function	11.57
Median Function	4.24
Low-pass Filter Function with $\alpha = 0.8$	4.12

Table 3.5: Mean Deviation of Summary Functions for Stock Service

User evaluation is conducted to compare the traditional service discovery and the QoS-Aware service discovery. We setup five Web services with same function of adding two numbers. Among these services, one is very fast (response time is less then 10 ms), while another is very slow (response time is more than 60 seconds). The other three is of normal speed (response time is 5 seconds or so). Two inquiry services: the traditional service discovery registry and the QoS-Aware are provided for the user. In this evaluation, a user study is carried out for 20 students who are selected from computer engineering department. Through the user study we evaluate user's satisfaction with the following two cases.

- (i) Services in the traditional registry are returned, of which the first one is chosen.
- (ii) With QoS-Aware discovery support by UX Server, the sorted services are returned to the user, of which the first one is chosen.

¹<http://www.swanandmokashi.com/HomePage/WebServices/StockQuotes.asmx>

The students are asked to try and compare the service from traditional registry to the service from UX Server. Table 3.6 shows the voting result. Some users are not satisfied with the traditional discovery. This is probably caused by the invoking of the slowest service. The users who evaluate the UX system are satisfied with the results.

	Bad	Fair	Good
Traditional	15 %	60 %	25 %
QoS-Aware	0 %	40 %	60 %

Table 3.6: User Voting on Service Discovery of Traditional and UX System

The UX federation system’s performance complexity is analyzed as follows: To the CSG model’s management, if the size of the CSG is N , the size of the adjacent Matrix will be N^2 , and the size of propagation edges will be $N-1$. On the ChangingVersionFederatedServer, prim algorithm’s computation complexity is $O(N^2)$, and the space complexity is $O(N^2)$ to store the adjacent Matrix. On each UX Server, only propagation edges are stored, therefore the space complexity is $O(N)$. The neighbor of each UX Server is pre-computed, therefore the computational complexity is kept offline. To each UX Server’s inquiry processing, if the number of all returned results for one inquiry is S , the complexity for merging the received result will be $O(S)$, and the sorting computational complexity will be $O(S \log(S))$. As a result, the total computational complexity for UX server’s query processing is within $O(S \log(S))$. The performance will likely be affected by the propagation delay between UX Servers. However, this can be reduced by the cache mechanism (See section 3.4.5).

Federated experiment is performed to check the system’s network overhead for service discovery. During the federated experiment, we setup four UX Server nodes in our test environment to form the UX federation. On each server we run the UX Server in the Tomcat servlet container with Axis deployed. These server nodes’ network topology is shown in Figure 3.10. Nodes A, C, and D join the federation through node B. Node D

CHAPTER 3. QoS-AWARE SERVICE DISCOVERY WITHOUT ONTOLOGY SUPPORT

receives the inquiry in its domain. If the local inquiry does not provide enough services for the requester, the federation discovery is triggered by node D. By this experiment, we try to check out the percentage of federation inquiries among the total inquiries received by node D, and the network overhead for these nodes caused by the federated discovery. Since the network has good connections within the local domain, we are more interested in the network overhead among cross domain connections. The cache mechanism is disabled in this experiment to gain a precise result.

Figure 3.10: UX Federation Experiment Setup

During the experiment, the inquiries in node D's domain are randomly generated with two alphabet letters. It issues a series of inquiries, by which we record the network usage across the domain links. The hop number is set to be 2 so that all nodes in the federation can receive the inquiry. When the federated discovery is performed, the inquiry is delivered to node B, C, and D through the propagation tree. The experiment

shows that the percentage of federated discovery in the all inquiries is about 43%. Figure 3.11 shows the network overhead of each cross domain network link for the inquiry. The X axis is the index of the invocation, and the Y axis is the average bandwidth overhead on each cross domain connection caused by this federation query. The zero points in the figure means that the federation discovery is not executed in this inquiry. From the figure we can see that the average network overhead for each inquiry is about 1.5 KB on each cross domain network link. This overhead is less than a normal web page's size, which is affordable for the normal discovery in most industries. The cache mechanism can be turned on to avoid the duplicated invocations across enterprise domains. Since the requesters in enterprise domain are often interested in some common services, the real world inquiry's cache hit rate could be good.

Figure 3.11: The Network Overhead of UX Federation Experiment

Compared with UDDIe project, our system does not modify the standard UDDI interface and the client side software can transparently plug on to our system. Our system defines a set of consensus QoS metrics. It continuously collects the feedback reports to get the services' real performance. The service provider does not need to worry about the publish of proper QoS attributes in UDDI as well. When the service provider and service

requester are located in different network domains, our system calculates the influence according to the connection condition. The prediction of the service performance is therefore more precise than its centralized model.

Our federation approach sits between the pure P2P mechanism and the static configuration. P2P systems are more appropriate in dynamic environments such as ubiquitous computing. Serious security threats exist in pure P2P for enterprise domain's usage. On the contrary, static configuration does not provide good fault tolerance and needs a bit of management work. It is only suitable in fixed environments where information is persistent. For enterprise domains, the cross-domain connections are less dynamic than P2P networks, but they still need some flexibility to allow easy link managements. The proposed federation service locates within this category. It has good load distribution and tolerance for network or node failures. The topology is stable and the system has the knowledge about the global federation. Each node can be reached, hence no service discovery information will be missed during the search. As discussed in section 3.3.1, in a centralized model or a cloud model based on replica, the UDDI registry cannot provide a precise prediction for the real service performance because the performance is influenced by the service requesters' different connection conditions. Our federated service considers the network connection conditions between different domains and provides a more reasonable estimation.

3.7 Summary

In this chapter we have presented a set of challenges for web services discovery when large amount of web services become available on the Internet. A number of similar web services are also emerging on the Internet and they're competing to offer better services. Mechanisms are required to efficiently discover and compare such services and to cooperate among registries.

CHAPTER 3. QoS-AWARE SERVICE DISCOVERY WITHOUT ONTOLOGY SUPPORT

Our work is tightly related to the web services discovery standards, QoS's prediction and the federated servers' management. We present a UX architecture that is QoS-aware and facilitates the federated discovery for web services. We describe the network model and design choices that we have made during the implementation of our architecture and feedbacks from service requesters are used to predict the service's performance. Customization is also provided for the service requesters to describe their preferences for discovery. A general federated service is designed according to the CSG model. It maintains the links among federated servers and deals with the message propagation. It can tolerate a great amount of node failures so that the global version change is reduced considerably. Based on the CSG model, UX server supports federated discovery across domains. The method to process different domain's difference based on QoS summaries as well as some additional policies are incorporated to support the system.

Compared with the original UDDI system, our system is aware of the basic service performance information with relatively small overhead from the feedback. The federated discovery helps the system to perform the discovery in the wider areas and estimate the QoS difference between domains dynamically. The replication consistency problem in normal UDDI is absent in our system as federated discovery is processed by UX Servers. One drawback of the system is the metrics system adopted in UX system. Adding new metric into current system is not an easy process. We need to update the specification to inform the users the updates, to write new feedback stubs and to integrate with the original summary functions. If various Web Services requires quite different set of metrics, the sharing of the metrics semantics and feedback stubs will become almost impracticable. In the next chapter, we would introduce our research work on the semantics definition for metrics and QoS specification to address these adding and sharing problems. The ontology level work provides a practical solution for integrating different Web Service's discovery requirements and measurement issues.

Chapter 4

QoS-Aware Service Discovery with OWL-QoS Ontology Support

In this chapter we investigate on the ontology level service descriptions. For service selection and management purpose, it is necessary to explicitly, precisely, and flexibly specify QoS constraints and QoS metrics for Web services descriptions. We describe our design principles for QoS ontology first. OWL-QoS ontology is designed as a complement to OWL-S Ontology to provide QoS-Aware service discovery and offer a practical QoS metrics model. Three separate layers are designed in the ontology, accompanied with development role descriptions for each of them. We provide precise semantics for the definition of QoS constraints, QoS metrics, and the specifications' conformance rule. The matchmaking algorithm for QoS profiles is presented for service discovery purpose. A comprehensive example is shown at the end of this chapter to demonstrate the ontology's usage.

4.1 Introduction

System integration through Web Services are gaining more and more importance as the focus of the internet business model is moving towards EAI (Enterprise Application Integration) and B2B with lower integration fee. Driven by this motivation, the industry's

advocation of Web Services technology has resulted in fast growth of Web Services. The Web Services technology is a good candidate for solving the integration problems, thanks to its self-describing, loosely-coupled, and coarse-grained features. The self-describing feature defines the standard interface for the Web Services during the service integration. The loosely-coupled feature enables the dynamic service invocation and increases the flexibility in the integrated system. Compared with the traditional information exchange systems, such as the Electronic Data Interchange (EDI), Web Service has a lower integration cost for enterprise applications.

The proper selection of Web Services is one key problem for those loosely-coupled systems during system integration. Because of the dynamic nature of the Web Services, they are not tightly connected. The target service must be located first before dynamic invocation. This problem becomes more and more serious as Web Services continuously appear on the web. In the current standard and research work, there are two matchmaking levels for the Web Services' discovery: syntax level and semantic level. Syntax level algorithm matches a target service if it has some matching keywords in its description. Semantic level algorithm matches a target service if its description's semantics are compatible with the requester's requirement. The current service discovery mechanism (i.e., UDDI) adopts keyword-matching technology to locate the published Web Services. However, matchmaking on the syntax level, that is, matching according to a set of weighted keywords, is a less-than-desirable situation. Service requester may have different understanding of the keywords, hence the returned discovery result might not satisfy the requester's intended requirements. The service provider also has problem in creating and sharing of the new keyword definition with partners. This leads to a bit manual works to choose the proper service. A mature service discovery framework requires to understand the services' semantics to improve the integration process.

A service's description normally can be divided into two parts: functional and non-functional properties. To understand and match against the target service, the selection

CHAPTER 4. QoS-AWARE SERVICE DISCOVERY WITH OWL-QoS ONTOLOGY SUPPORT

criteria should at least include the service's functional and nonfunctional requirements. To support the semantic level matchmaking for the functional and nonfunctional properties during service discovery, Web Services' domain-specific criteria should be clearly and shared among partners. This requires domain-specific knowledge.

In this light, Semantic Web technology is a promising innovation for service discovery. It requires that data are not only human readable, but also machine understandable. Semantic Web concepts can help form a unifying system that minimizes misunderstandings among different partners. The OWL-S[7] ontology is one effort by a group of Semantic Web researchers to describe Web Services. It aims to enable automated Web Service discovery, invocation, composition, and monitoring. It defines the notions of a service profile (what the service does), a service model (how the service works), and a service grounding (how to use the service). Because the service profile precisely defines the Web Service's functional aspect, it meets the requirements for the functional aspect of service discovery.

The nonfunctional service discovery is an important aspect in service discovery. After the service requesters filter out the unsatisfying services according to their functional properties, they still face a group of offers that provide similar functions. These similar services may differ significantly in QoS and cost during the service invocation. This services list can potentially contain thousands of Web Services. Therefore, one cannot expect the requester to manually test through all of the Web Services on the list and select the most suitable one. Comparing the QoS aspect of these services is preferred, and this requires more sophisticated patterns of service discovery and selection.

In practice, we can divide service discovery into two phases as shown in Figure 4.1. In the first phase, a requester discovers the service using the functional aspect of the service—what the service does, its input and output parameters, preconditions, and effects. This phase ensures that the returned services meet the requester's requirements

for service's function. The second phase is to identify the most-appropriate service for the current task; it accounts for the nonfunctional information, such as QoS, about each service returned in the first phase. This second phase has become an important research area in process combination and service discovery because dynamic binding and invocation are preferable.

Figure 4.1: Two Phases of the Service Discovery

Domain knowledge is required to describe the basic metrics used in the Web Services. Based on some previous works[65, 66], we design the basic QoS metrics set for the general Web Services, such as the response time, reliability, cost, and so on. We need to mention that, this basic metric set is an example for the prototype system. Following the design pattern, it can be extended to support diverse services. In addition, we study the service measurement issues to validate the agreed service levels.

The main motivation of this work is to enhance the service discovery system with matchmaking mechanisms for QoS-Aware service discovery and integration. This novel method complements the OWL-S ontology to provide semantic level QoS discovery, integration, and measurement using OWL-QoS ontology.

4.2 Related Works

The OWL-S' profile portion addresses the first phase of the service discovery. It specifies the concepts of the service's input, output, precondition, effect, and actors, etc. The

CHAPTER 4. QoS-AWARE SERVICE DISCOVERY WITH OWL-QoS ONTOLOGY SUPPORT

following list shows an partial example for the OWL-S' profile definition.

```
<profileHierarchy:BookSelling
  rdf:ID="Profile_Congo_BookBuying_Service">

  <!-- reference to the service specification -->
  <service:presentedBy
    rdf:resource="&congoService;#ExpressCongoBuyService"/>
  <profile:has_process
    rdf:resource="&congoProcess;#ExpressCongoBuy"/>

  <profile:serviceName>Congo_BookBuying_Agent</profile:serviceName>
  <profile:textDescription>
    This agentified service provides the opportunity to browse a
    book selling site and buy books there
  </profile:textDescription>

  <profile:contactInformation>
    <actor:Actor rdf:ID="CongoBuy_contacts">
      <actor:name>ExpressCongoBuy</actor:name>
      <actor:title> Service Representative </actor:title>
      ...
    </actor:Actor>
  </profile:contactInformation>
  <profileHierarchy:deliveryRegion
    rdf:resource="&country;#UnitedStates"/>
  <profile:hasInput
    rdf:resource="&congoProcess;#ExpressCongoBuyBookISBN"/>
  ...
  <profile:hasPrecondition
    rdf:resource="&congoProcess;#ExpressCongoBuyAcctExists"/>
  ...
  <profile:hasResult
    rdf:resource="&congoProcess;#ExpressCongoBuyPositiveResult"/>
  ...
  <profile:hasOutput
    rdf:resource="&congoProcess;#ExpressCongoBuyOutput"/>
</profileHierarchy:BookSelling>
```

This profile definition explicitly associates the Congo buying Web Service's interface¹ with concept definitions. Congo progress is the main progress model of the Congo service. It provides the concept definitions of the input, output, etc. One of the service's inputs

¹Congo Service in OWL-S 1.0 is available at <http://www.daml.org/services/owl-s/1.0/examples.html>

is ExpressCongoBuyBookISBN, which is defined in the Congo Process. The output, precondition, and result are also linked to the related concepts. These concept definitions provide the semantic level information to assist the automatic service discovery and integration.

For the second discovery phase, OWL-S has defined quality rating ontologies to describe the service's QoS information. However, as Cardoso et al. [41] have pointed out, these ontologies require significant improvements to provide a realistic solution to OWL-S users. Although OWL-S has quality rating property in the profile class, one current limitation of OWL-S' QoS model is that it does not provide a detailed set of classes and properties to represent QoS metrics. This motivates us to design the QoS ontology to compensate for this shortcoming of OWL-S in the second phase.

Using the ontology to define the QoS metrics is the best practice for services[67, 52]. Different Web Services normally have quite different types of metrics to deal with, and some of them have relationships between each other. Definition through ontology makes it possible to specify the meaning of these concepts, as well as relationships between them. This makes the ontology a good candidate for defining the QoS metrics. The metrics also require a list of measurement properties to precisely define the method to calculate their values. These definitions need to answer the when, how, and where questions. For instance, what does response time actually mean? Is the response time of every data item measured? Is it an average, or is every nth item measured? Is it the actual time that is important or just the % of items that are beyond a certain range? Traditionally the syntax level of defining metrics often uses natural language to describe such questions, rather than properly define the related properties for computer's understanding. A result of this is that the measurement and calculation portion is hard coded in the measurement system. This means that it is not easy to extend the predefined schema to include new metric properties. On the other hand, ontology nicely supports the precise definition of

the metrics. It can be easily extended to describe new metrics and properties as well. As a result, it is natural to choose ontology as the best practice to define metrics.

Several research works for the Semantic Web Services' discovery is studied in the last few years. Here are some examples:

Paolucci, Kawamura et al.[43, 44] present an algorithm that deals with DAML-S specification language. It considers the matching of input/output concepts defined within the same ontology. The matching algorithm defines the matching degrees for the concepts in the ontology, including exact, plugIn, subsume, and fail. By calculating the matching degree of service's interface, the service score is computed for the comparison.

Based on Tversky's [45] feature-based similarity model, Cardoso [46] presents an ontology similarity matching algorithm that facilitates and assists the composition of Web Services. The main objective of this approach is to assess the similarity of terms and concepts in the services' interfaces. The workflow systems and discovery mechanisms can use this algorithm to find Web Services with desired interfaces and operational metrics.

Lei Li et al. [47] presents a software framework for matchmaking based on the Description Logic. The purpose of this framework is to support service advertisement and discovery in e-commerce. They argue that the information constraint in the DAML-S profile is too restrictive for effective information. As a result, they choose only the essential parts of the service profile for the matchmaking. The experiment result shows that the publish and classification for large numbers of advertisements are time consuming. However, the matching of queries to advertisements is performed very efficiently.

These three works tries to match the target service based on the semantic level. By judging the service's input, output, and functionality, these works can locate the target service with proper functional properties.

Clustering technology is also used to discover the Web Services according to their text information. Various approaches have been tried. One of these works creates the

domain text vector space according to the domain ontology, and then clusters the normal WSDL/UDDI's textual descriptions to get the matching services[68]. The other way is to apply the clustering technology in the data-intensive service description's sampling process [69] to gain the source-biased summary. By ranking the summary with the source, the target services can be identified. Different from the semantic level matchmaking, these service discovery works apply the clustering technology on the text vector space to achieve the matching result. We view these approach as a type of syntax level matchmaking for the services. The goodness is that the clustering process on the text can be processed automatically, and no domain knowledge is explicitly required on such technologies. However, their matching precision is not as good as the methods based on the semantic level matchmaking.

Syntactic matching is a useful tool when the domain ontology is not available for the Web Services' descriptions. In addition to the clustering technology, traditional information retrieval technology can be applied on the Web Services' description to perform the syntactic matching. For example, the TF-IDF[70] method can calculate a similarity score between two documents according to the terms' frequency. The limitation of the syntactic matching is that it does not understands the description's semantics, which leads to a lower precision to the semantic matching.

According to the above analysis, the semantic matching is a machine-understandable mechanism for the service matching, and the syntactic matching is a useful supplement for service matching. Since the semantic matching for the OWL-QoS has some limitations on the QoS-Aware service discovery, our work is to improve its QoS model, and to design the proper matching algorithm for the service discovery process.

4.3 Scenario

We show an example scenario in which the OWL-QoS ontology is needed. A service requester is looking for a book shopping service to locate and browse the latest popular books. Let us assume that the requester has precisely defined the functional requirement of the service as book locating service. After issuing this requirement in the first phase of service discovery, the service requester gets a list of book locating services, such as S1, S2, and S3, from the service registry. Although these services can be integrated into the service requester's system since they provide the required functions, they probably differ greatly in their nonfunctional properties. The service requester wants the most responsive and reliable one from this list. Without the QoS-Aware service discovery support, to choose the satisfying service to meet the service requester's performance constraint and financial target, he has to manually check the suitable services one by one from this list. The size of this service list may be much larger in the real world. How to share the semantics for the service's nonfunctional properties and automatically match them according to the requester's demand remains a problem. This is the place where our ontology tries to start from.

4.4 Design Principles for QoS Ontology

We call our ontology OWL-QoS; it provides detailed QoS information as well as the QoS measurement information for service partners.

Based on ontology approach, we aim to reduce the integration cost for Web Services. The service integration problem inherits from syntactic heterogeneity and semantic heterogeneity. Syntactic heterogeneity exists because the interface description uses different descriptions, data types and class hierarchies. Semantic heterogeneity originates from the different semantic meanings of the service interface attributes and its functionalities. We

try to reduce the syntactic heterogeneous by using uniform ontology syntax, i.e., standard OWL ontology language. As to the semantic heterogeneous, we try to reduce the misunderstanding of the service's functionality by OWL-S, as well as the nonfunctional aspects of the service by OWL-QoS. Sharing the same ontology domain is required in our design for service partners to avoid ontology mapping problem.

Nonfunctional aspects describe constraints such as service level, management statements, security policies, pricing information, and other contracts between Web Services and their users. While the security of Web Services is an extremely important issue, we believe that security itself is a separate research area. Our work, including the QoS specification, is not about the security of Web Services and the related protocols.

We conclude several design principles to guide our ontology design. When a Web Service is designed and developed by developers before its launch, it is said to be in the design-time. On the other hand, when a service is running, or executing, it is said to be in run-time. During design-time, developers would normally focus on the system's functional aspects rather than nonfunctional aspects. However, the best practice is to keep QoS issues in mind during early stages of the design phase. This requires a related QoS specification to describe the overview of the system's QoS requirement. During run-time, QoS specification should provide additional information to support dynamic service discovery, composition, integration, and monitoring, which requires appropriate language support as well. As a QoS specification for Web Services, our OWL-QoS ontology's design principles include:

- *Ease of use.* Developers should find the specification easy to use and understand.

A clear specification helps developers to understand the system's nonfunctional aspects and guides them toward choosing a proper system design pattern early on in development.

- *Precision and flexibility.* To allow value-added services for dynamic service discovery as well as automatic and customizable composition and integration, the QoS specification should have precision and flexibility. Precision means that it should answer “when, which, where, what, and how” of the measurement evaluation questions against Web Services [71]. Flexibility permits the specification of customized metrics for diverse Web Services.
- *Object-oriented style.* Because of the widespread acceptance of object-oriented design principles in the service’s implementation, it would help if the definitions for QoS requirements are also object-oriented. When developers know the overall interface’s QoS requirements, it makes sense for lower-level interfaces to inherit QoS requirements and reuse the QoS constraints from the super interfaces.
- *Automatic validation.* As a project grows, automatic validation of the syntax and semantics of the QoS specification to avoid development mistakes becomes important. More specifically, it is desirable to automatically check whether the specification’s syntax conforms to the grammar, and there is no conflicts in its semantics.
- *Separation of design duties.* A good design should separate its measurement details from its metric concept definition. It is the measurement partners’ task to fill in the metric measurement details according to the service execution environment. A measurement partner is the group that deploys the measurement handlers and performs the service measurement tasks. The service provider or requester can take on this role, or it can be outsourced to a third party.

4.5 Model Design

According to these design principles, we divide the ontology into a three-layered representation:

- The Service Level Agreement (SLA) Layer defines the schema for a general service level agreement. Its individual defines the detailed instance of an agreement.
- QoS Profile Layer defines the constraints of a service's QoS properties. It describes certain service level objectives and it can be used for service discovery purpose. Its individual is defined by measurement partner for monitoring purpose.
- The Metrics Layer defines QoS metrics and their measurement information. We provide a set of common QoS metrics for general Web Services.

Figure 4.2 shows the overview of the OWL-QoS ontology design. In the top of the figure it shows the SLA Layer of the ontology. The concepts in the middle of the figure stand for the profile Layer of the ontology. The bottom side of the figure presents the metrics Layer. In the following subsections, we will introduce these three layers in more details.

4.5.1 SLA Layer

The Service Level Agreement (SLA) is a contract between the vendor and the user that specifies the level of service expected during its term. The SLA Layer defines the schema for a general SLA. Its individual defines the instance of an agreement.

The SLA Layer provides a way to link the service description to the QoS constraints specified in the QoS Profile Layer. We reuse some core concepts in the OWL-S ontology to describe the service partner's information. For example, inherited by the Actor concept, the service provider, service requester, measurement partner concepts are taken to represent the service partners. Table 4.1 shows the main properties list in the SLA Layer.

Our ontology does not limit the service partners to use OWL-S for their service description. The QoS constraints of the service are defined separately in the QoS Profile

CHAPTER 4. QoS-AWARE SERVICE DISCOVERY WITH OWL-QoS ONTOLOGY SUPPORT

Figure 4.2: Ontology Overview

CHAPTER 4. QoS-AWARE SERVICE DISCOVERY WITH OWL-QoS ONTOLOGY SUPPORT

Property Name	Range	Description
hasServiceProfile	ServiceProfile	Point to the Service Profile individual in OWL-S.
hasScope	WsdAtomic ProcessGrounding	Point to the service scope described by OWL-S' grounding ontology.
slaPartner	Provider/Requester	Point to the subclass of OWL-S' Actor concept
measurementPartner	MeasurementPartner	Point to the subclass of OWL-S' Actor concept
startTime	date	The start time of the measurement for Service Level Description.
endTime	date	The end time of the measurement for Service Level Description.
MonthOfYearMask	octet string	The months of the year on which the measurement is valid.
DayOfMonthMask	octet string	The days of the month on which the measurement is valid.
DayOfWeekMask	octet string	The days of the week on which the measurement is valid.
TimeOfDayMask	string	The range of times at which measurement is valid.
priceStatement	string	The statement describes about the price of the service.
penaltyStatement	string	The statement describes about the penalty of the service.

Table 4.1: The Property List in SLA Layer

Layer, which is isolated from the SLA Layer. For those service providers who use OWL-S to specify their service description, SLA Layer in our ontology provides a direct way to link their description to our QoS Profile Layer. For those who use other languages to specify their service description, they can utilize our ontology's QoS Profile Layer to specify the QoS constraint and link it to their description.

Currently our prototype framework is a discovery and measurement system. The service execution history is recorded in the logs and the violation will be notified. However the auditing system has not been supported in our system. The price/penalty statement properties are descriptive information rather than machine processable penalty rules. They provide human readable information to describe the subtle constraint and responsibility of the Web Services. More detailed constraint definition for price/penalty can be one of our future works on the price auditing system for Web Services.

4.5.2 QoS Profile Layer

The QoS Profile Layer defines the Service Level Objective (SLO) for the Web Service interface. SLO is normally a set of parameters and their values to specify QoS constraints. In our system, each SLO specifies the detailed QoS constraints of the service. It is defined as a QoS profile concept in OWL-QoS.

Each QoS property definition in the QoS Profile Layer constrains the property's domain and range information. Each property's domain is defined as the QoSProfile class or its subclasses. The range of the QoS property is defined as the QoS metric class, or its subclass. The QoS metric classes are defined in the QoS Metrics Layer (See section 4.5.3). The QoS profile class is defined by the conjunction of these property constraints.

Numeric constraints of service QoS are defined over these QoS properties. By combining the information of property type, range and numeric constraint, our matchmaking algorithm tries to order these profiles according to their relative goodness. We describe

this relative goodness in more details in section 4.6.1, and it is called the service profile’s conformance rule.

In our model we declare the class *QoSProfile* as a common superclass for all QoS specifications. Additional QoS constraints arise from the *QoSProfile*’s inherited concepts. Figure 4.3 gives an example scenario for the QoS-Aware service discovery: The service requester requires a Web Service that implements the service interface and satisfies the QoS constraints in concept named *InquiryQoS*. The service providers promise to implement the service interface with the QoS constraints stated in the concept named *ProviderQoS1*, *ProviderQoS2*, and so on. These concepts are subclasses of *QoSProfile* and defined by service requesters and service providers.

Figure 4.3: QoS Profile Matchmaking for Service Requester and Service Providers

A service provider defines classes of QoS profiles for service requesters to select from. These predefined QoS profiles benefit the service provider by separating the market into detailed segmentations. The agreement between service providers and service requesters is made upon the published profile rather than individually customized QoS profiles. This chosen set of predefined QoS profile classes simplifies the service provider’s management system and saves the service provider’s limited underlying resources. Defining

individually customized QoS profiles for each requester is not chosen in our design. If the service provider supports individual customized QoS profile, the service provider needs to customize each service requester's QoS requirements and choose the runtime system resources bundles, deployment scripts and topology design. This normally causes a substantial overhead and negates the benefits of customization.

4.5.3 QoS Metrics Layer

The QoS Metrics layer provides QoS metrics' definition for service requesters and service providers to describe their inquiries and advertisements. Furthermore, these definitions are shared by service measurement partner to measure the service performance and check against the guarantee. A measurement partner is the group that deploys the measurement handlers and performs the service measurement tasks. The service provider or requester can take on this role, or it can be outsourced to a third party in case the service provider and requester do not trust each other.

The service QoS metrics are divided into AtomicMetrics and ComplexMetrics, which are showed as follows:

$$\begin{aligned} Metric &\sqsubseteq \top \\ AtomicMetric &\sqsubseteq Metric \\ ComplexMetric &\sqsubseteq Metric \end{aligned}$$

The Metric class is a common superclass for all metrics. The part B in Figure 4.5 shows some Metrics Layer's classes and their properties. The Metric class has related properties *hasUnit*, *value*, *metricName*, etc. Table 4.2 provides a summary description for these properties.

The Metric class has two subclasses: AtomicMetric and ComplexMetric. All metrics are subclasses of these two metrics and should inherit all the properties from one

Property Name	Description	Range
hasUnit	the metric's unit	Unit Class
value	initial value of the metric measurement handler	integer
metricName	the name of the metric	string
windowSize	the amount of history data the metric should keep	integer
pushPoint	the address for sending the collected data	string

Table 4.2: The Properties of Metric Class

of these two metrics. Based on these metrics definition, we designed a prototype measurement system for Web Services. In our measurement prototype, each AtomicMetric corresponds to some measurement handler, while each ComplexMetric corresponds to a report collector.

AtomicMetric represents the metrics that collect the first hand data from managed resources using measurement handlers. The AtomicMetric's property *measureAt* answers the "where" question of the measurement, for instance, the question like "where is the response time being measured?" It can be set as "Server", "Client", and so forth. According to the AtomicMetric's individual definition, the measurement party generates the corresponding measurement handler and deploys the handler at the proper location. The handler pushes its measured data to the collector located at the *pushPoint*.

ComplexMetric defines the way to combine lower level metrics (AtomicMetric or ComplexMetric) according to the function definition. It has properties *hasFunction*, *operand*, *windowPos*, and so on. Table 4.3 provides a brief description of these properties.

The *windowPos* property is the parameter offset for the operand metric's data series, from which the function will find its parameter. Negative *windowPos* refers to the history data in the data series.

The *operand* property points to the function's operand, and each of these operands is one metric definition. For example, the AverageResponseTimeMSMetric uses the average function to operate on its operand ResponseTimeMSMetric. Our system uses this composition method to calculate the high business level metrics from low level metrics.

The *hasFunction* property points to the Function class. The Function class contains the Arithmetic function, Boolean function, and Aggregate function. Each function has its function name and the function definition. Currently we have defined the syntax for general function's definition. A series of Function classes are declared in our ontology. Table 4.4 shows a partial list of our predefined Function classes. Arithmetic function provides basic mathematical operations for the system. Boolean function provides basic Boolean operations for the system. The zero value is deemed as false while non-zero values are deemed as true. The aggregate function can be used to describe the accumulated characteristics of the service, such as Average, Max, Min, etc. The Aggregate function has two specific properties: *window* and *timeWindow*. *window* property describes the window size for the operand's records. The stored number of operand's records (*windowSize*) should be larger or equal to the *window* plus the absolute value of *windowPos*. *timeWindow* property describes the timestamp window size for the operand's records. Each operand's record is accompanied with a timestamp. All the records within the period of *latest_record* and *latest_record* - *timeWindow* will be processed by the aggregate function. These two properties provide a refined constraint for processing the operand's data records.

User can define their customized function definition by using the syntaxes listed in Table 4.5. User can also define their own Function handler in java code to process their customized functions. The Function's property definition and composition style provide a detailed definition for high level metrics. The "what and how" question of the measurement is answered by these property definition in the Metrics individuals. For example, the questions like: "What aspect of the Web Service is measured? How the average response time is calculated? How many response time records are used?", etc.

Property Name	Description	Range
hasFunction	points to the function to calculate ComplexMetric's value	Function Class
operand	the contained metrics that the function operates on	Metric Class
windowPos	the offset for the operand's data series	integer

Table 4.3: The Properties of ComplexMetric Class

Function Type	Function Names
AggregateFunction	Average, Max, Min, Size, Median, and so on.
ArithmeticFunction	Add, Subtract, Divide, Multiply, Remainder, Sin, Cos, and so on.
BooleanFunction	Greater, Less, Equal, And, Or, XOR, and so on.

Table 4.4: The Function Definitions

Imbeded Types	Supported Syntaxes
Unary operator	+ -
Binary operator	^ * / MOD + - >= <= AND XOR OR (from the highest to the lowest priority)
Condition operator	IF THEN ELSE
Embeded Function	LN LOG EXP SQR SQRT COS SIN TAN ACOS ASIN ATAN COSH SINH TANH INT ABS OPP NOT
Constants	PI FALSE TRUE

Table 4.5: The Customization of Function

4.5.4 Basic Metric Set

The QoS metrics need to be defined precisely in the Web Services domain. We use an extensible QoS model to include generic and domain specific metrics. The generic metrics are applicable for all Web services. To facilitate the speedy startup in using the QoS ontology, we designed a basic metric set for the generic metrics. Figure 4.4 shows the Ontology hierarchy definition for the basic Metric set. A corresponding measurement handler generator for this basic set is also developed in our system. New metrics, no matter generic or domain specific, can be added into the system without fundamentally altering the underlying matchmaking mechanism.

Figure 4.4: Ontology for Basic Metric Set

The basic metric set contains the performance metrics, reliability metrics, availability

metrics, and so on.² The definition of these metrics can be found in section 2.5.1.

Using the basic metric set, service provider and service requester can share this knowledge to publish and inquiry QoS profiles for general Web Services. The standard measurement stubs can be defined for the basic metric set and this facilitates the measurement partners to start up the service measurement. The specific metrics can be defined by the domain experts. The measurement framework will be further discussed in section 5.1.4.

4.5.5 Reuse of Specification

QoS profile's reuse is an important requirement for the service requesters and providers. The object-oriented design principle is widely used in the service component's development. Since the profile's definitions are linked directly to the service's interfaces, object-oriented style is a natural choice for the corresponding profile's definition. Corresponding to the software components' inheritance, we use conjunction rules to refine the specific QoS requirement for the inherited interface and reuse all the inherited QoS specifications described in the super interface.

The QoS profile concept and all its sub concepts are reusable in object-oriented style. The newly defined QoS profile concept can reuse any predefined QoS profile concept by inheriting this QoS profile concept and then refine it with addition property constraint definition. According to the description logic's semantics, the newly defined concept is interpreted as the conjunction of all these inherited super concepts and its newly added property constraints. Therefore, each property's constraint in the predefined QoS profile is inherited in the newly defined profile concept. The software developers with OOP experience are normally familiar with this inheritance style. This shortens the learning curve for these developers. Furthermore, the metrics are defined as concepts. They are

²To keep the compactness and save space, we do not write detailed ontology definition for these QoS properties. Please visit <http://www.ntu.edu.sg/home5/pg04878518/OWLQoSontology.html> to get more detailed information about the property and QoS metrics definition.

reusable through inheritance as well. If the service requester indicates one general metric in his query, the metric's sub concepts will also be processed by the reasoner according to the matchmaking algorithm.

The different scenarios for the creation and maintenance of the QoS Ontology are as follows:

- *Green field*: In this scenario, the developer starts completely from scratch, creating all the layer's Ontologies. With this approach, QoS designer first designs the customized QoS property, and then selects or defines the QoS metrics for the property's range constraint. The new QoS metrics are put in the metric taxonomy. The QoS metric's individuals are then defined by measurement organizations. The constraints for the QoS properties are set by the service provider.
- *Bottom up*: The bottom up scenario follows along the same lines as the green field, with the exception that the QoS metrics and properties have already been defined by the QoS designer or by the Web services vendor. The provided basic profile is located in this category. The remaining thing is to setup the numeric constraints for the properties by the service provider.
- *Top down*: The top down scenario is a bit different. The property domain and constraints have already been defined. The QoS metrics are quite specific and not properly defined or the original QoS metrics are not suitable. The service provider will select one metric in the taxonomy or define a new metric. The newly defined metric's individuals will be further specified by the measurement organization.

4.6 Matchmaking for the OWL-QoS

Matchmaking here is defined as a process that requires a repository to take an inquiry as input, and to return all the published advertisements which satisfy the QoS requirements

specified in the input inquiry. Each QoS profile describes their service level objective (SLO) and can be deemed as an SLO profile. To determine whether the advertisement satisfies the inquiry's requirement, conformance definition between SLO profiles is required.

4.6.1 Conformance Definition

Constraint conformance judges whether one SLO specification conforms to another SLO specification. A SLO specification conforms to another SLO specification only if it is stronger than, or equally strong as, the constraints in the other specification. The stronger definition is relative to the metric's type. Roughly speaking, the SLO S_1 conforms to S_2 when each of S_1 property constraints conforms to each of S_2 's. This can be formally defined as follows.

We define the conformance for the metric property's restriction first. We say $S_i(p)$ is the metric property's constraint definition which has the form " $op_i n_i p$ " in which $op_i \in \{\leq, \geq, =\}$, $i = 1, 2$. n_i is non-negative integer that represents the constraint value. Operator $\prec$ represents the conformance relationship. The statement $S_1(p) \prec S_2(p)$ denotes that constraint $S_1(p)$ conforms to $S_2(p)$ with respect to the metric property p .

$$\frac{n_1 \leq n_2}{S_1(p) \prec S_2(p)}, op_1 \in \{=, \leq\}, op_2 \in \{\leq\} \quad (\text{Eq. 4.1})$$

$$\frac{n_1 \geq n_2}{S_1(p) \prec S_2(p)}, op_1 \in \{=, \geq\}, op_2 \in \{\geq\} \quad (\text{Eq. 4.2})$$

$$\frac{n_1 = n_2}{S_1(p) \prec S_2(p)}, op_1 \in \{=\}, op_2 \in \{=\} \quad (\text{Eq. 4.3})$$

The first rule (Eq. 4.1) states that when the property points to a decreasing metric, such as the response time, smaller number means stronger conformance than larger

numbers. n_1 is smaller than n_2 so that $S_1(p)$ conforms to $S_2(p)$. The rule (Eq. 4.2) states the constraint conformance for the increasing metric. The rule (Eq. 4.3) targets the enumeration metrics. Same number means that $S_1(p)$ conforms to $S_2(p)$.

We define the SLO specification's conformance in terms of individual metric property's conformance. This is described in Rule (Eq. 4.4). If all the property definition in S_2 conforms to the definition in S_1 , we say that the specification S_1 conforms to S_2 .

$$\frac{\forall p \text{ in } S_2 : S_1(p) \prec S_2(p)}{S_1 \prec S_2} \quad (\text{Eq. 4.4})$$

Above definition presents the conformance rules for general SLO specifications. We use description logic's numeric constraints to present the SLO specification's constraints. In the following we will show that general conformance relationship $\prec$ between two SLO specifications can be mapped to the description logic's subsumption relationship. As a result, we can utilize the description logic reasoner to judge the general conformance relationships.

Lemma 4.6.1 *The conformance relationship is transitive, i.e., if $S_1 \prec S_2$, and $S_2 \prec S_3$, we have $S_1 \prec S_3$.*

Proof: For any property p , if $S_1(p) \prec S_2(p)$, and $S_2(p) \prec S_3(p)$, we want to prove that: a), $\prec$ is transitive for each property's constraint definition $S(p)$, i.e., $S_1(p) \prec S_3(p)$. b), $\prec$ is transitive for the SLO specification.

For the point a), if p points to a decreasing metric, Eq. 4.1 shows that $n_1 \leq n_2$, and $n_2 \leq n_3$. This leads to $n_1 \leq n_3$, and we get $S_1(p) \prec S_3(p)$ according to Eq. 4.1. This result suits the decreasing metric and enumeration metric as well.

As to point b), from $S_1 \prec S_2$ and $S_2 \prec S_3$, we have $\forall p \text{ in } S_2, S_1(p) \prec S_2(p)$ and $\forall p \text{ in } S_3, S_2(p) \prec S_3(p)$. From these two conditions, $\forall p \text{ in } S_3, S_1(p) \prec S_3(p)$ is a nature conclusion from point a). According to Eq. 4.4, we have $S_1 \prec S_3$. Therefore $\prec$ is a transitive relationship.

Theorem 4.6.2 *The conformance problem of two SLO specifications is equivalent to the concept subsumption problem in Description Logic.*

Proof: “ $\Rightarrow$ ” Let S_1 and S_2 be the stronger and weaker constrained SLO specifications respectively. According to the conformance definition Eq. 4.4, for each property p in S_2 , $S_1(p) \prec S_2(p)$. Without loss of generality, we assume each property P points to an increasing metric. According to the Eq. 4.2, the numbers in S_1 are larger than those in S_2 . The concept representation of S_1 and S_2 can be expressed as:

$$\begin{aligned} S_1 &\doteq (\geq k_1 P_1.C_1) \sqcap (\geq k_2 P_2.C_2) \sqcap \dots \sqcap (\geq k_l P_l.C_l) \sqcap \dots \sqcap (\geq k_m P_m.C_m) \\ S_2 &\doteq (\geq n_1 P_1.C_1) \sqcap (\geq n_2 P_2.C_2) \sqcap \dots \sqcap (\geq n_l P_l.C_l) \end{aligned}$$

in which $P_1, P_2, \dots, P_l$ are the properties listed in S_2 . Compared with S_2 , properties $P_{l+1}, \dots, P_n$ are the additional properties list in S_1 . According to Eq. 4.2 for any p in S_2 , $S_1(p) \prec S_2(p)$, we have $k_1 \geq n_1, \dots, k_l \geq n_l$. Furthermore, S_1 has additional numeric constraints $(\geq k_l P_l.C_l) \sqcap \dots \sqcap (\geq k_m P_m.C_m)$. According to these two concepts' definition, S_1 has more constrained definition than S_2 . As a result, we can draw that concept $S_1 \sqsubseteq S_2$, that is, the concept of strong SLO specification is subsumed by the concept of weak SLO specification.

“ $\Leftarrow$ ” Let concept S_1 be subsumed by concept S_2 ($S_1 \sqsubseteq S_2$), and we want to prove that the SLO specifications S_1 conforms to S_2 . Without loss of generality, we assume each property p points to an increasing metric.

If S_1 and S_2 have the same l properties in their concept definition, S_1 will have stronger numeric constraints than S_2 . If this is not true, there must be some property $(\geq k_t P_t.C_t)$ in S_1 and $(\geq n_t P_t.C_t)$ in S_2 in which $k_t < n_t$, $1 \leq t \leq l$, from which we have $(\geq n_t P_t.C_t) \sqsubseteq (\geq k_t P_t.C_t)$. According to the numeric constraint's definition, we can interpret the property in such a way that: $x \in \Delta^I$, $x \in (\geq k_t P_t.C_t)^I$, $x \notin (\geq n_t P_t.C_t)^I$

and $x \in (\geq k_i P_i.C_i)^I$ where $i = 1, 2, \dots, l$ and $i \neq t$. Hence $x \in S_1^I$ while $x \notin S_2^I$, and this contradicts the premise $S_1 \sqsubseteq S_2$. Therefore, we get the result that $k_t \geq n_t$, and $S_1 \prec S_2$.

If S_1 has all the S_2 's properties as well as some additional properties, we select all the properties' numeric constraints in S_1 whose property appears in S_2 as well, and then define the conjunction of these S_1 's numeric constraints as SC . Obviously we know $S_1 \sqsubseteq SC \sqsubseteq S_2$. According to the above proof, we know that the SLO specification $SC \prec S_2$. Since each numeric constraint in SC and S_1 is same, and S_1 may have some additional properties, we get $S_1 \prec SC$. According to Lemma 4.6.1, we have $S_1 \prec S_2$.

According to the theorem 4.6.2, we know that the conformance problem of the SLO specification can be converted to the concept's subsumption problem. Therefore, we design the corresponding matchmaking algorithm for this DL representation of SLO specification. The novelty of the QoS matchmaking algorithm is that it transforms the problem of service QoS constraints conformance into the problem of concept subsumption relationship. By this matchmaking algorithm, we can determine whether the guarantee given in the provider's advertisement is stricter than the one inquired by the service requester, and whether these specifications have conflicts in them or not.

4.6.2 Matchmaking Algorithm

The relationship between the conformance problem of SLO specifications and the subsumption problem of DL concepts shows the way to match against the QoS Profiles. Formally the matchmaking can be specified as: For a given inquiry P , the matchmaking algorithm should return the set of all the published advertisements which are compatible. Two QoS ontology descriptions, say $C1$ and $C2$, are compatible iff their intersection is satisfiable:

$$compatible(C1, C2) \Leftrightarrow \neg(C1 \sqcap C2 \sqsubseteq \perp)$$

All the compatible advertisements will be added to the resultant set. However, we need to introduce the definition of the degree of match to distinguish different advertisements. The sequence of matching degrees in our algorithm is different from [43, 47]. The matching degree's definitions are as follows:

- *Subsume*: If request R is a super-concept of advertisement A , i.e., $A \sqsubseteq R$. We call the match Subsume;
- *Exact*: If advertisement R and request A are equivalent concepts, formally $R \equiv A$.
- *PlugIn*: If request R is sub-concept of advertisement A , we call this match PlugIn; that is, $R \sqsubseteq A$.
- *Intersection*: If the intersection of advertisement A and request R is not empty, that is, $\neg(A \sqcap R \sqsubseteq \perp)$.
- *Disjoint*: Otherwise it is disjoint; i.e., $A \sqcap R \sqsubseteq \perp$.

Subsume matches are considered the preferred match, since we can expect that the advertisement conforms to the request; Exact matches are the next best, because the advertisement is exactly the same as the request; PlugIn matches are considered to be the third best, as the advertisement does not fully provide the required QoS level according to the request; Intersection is supposed to be the fourth best, since it means that the advertisement is not incompatible with the request; and Disjoint is the worst case, because it shows the conflict between the advertisement and the request. This is a failed match.

The validation of the specification's correctness is important when the size of a project grows. Two types of validation are required in the development: the syntax and semantics validation. The syntax correctness of the specification is checked by the ontology parsers, such as Jena parser[72]. As to the semantic correctness, if the specification's SLO concept

is equivalent to $\perp$ concept, there are some contradictions in the specification and it is deemed as semantically incorrect. The validation algorithm judges the specification as valid if the SLO concept is not equivalent to $\perp$ concept.

4.6.3 Comprehensive Example

We can demonstrate our approach with a comprehensive scenario that involves a book shopping service for browsing and locating a target book. The book locate service is defined by the OWL-S cargo service. Figure 4.5 shows this example's main definition diagrams. Suppose that we want to specify the provider's advertisement for the service in which the average response time is no more than 5 seconds and the cost is no more than 1 dollar. The part A in Figure 4.5 shows the example of this QoS advertisement. According to our ontology design, this advertisement can be written in description logic's syntax as:

$$\begin{aligned}
 \textit{ProviderQoS} &\doteq \textit{QoSProfile} \\
 &\sqcap (\leq 100 \textit{ cost}) \sqcap (\forall \textit{cost}.\textit{CostUSCentMetric}) \\
 &\sqcap (\leq 5000 \textit{ averageResponseTimeMS}) \\
 &\sqcap (\forall \textit{averageResponseTimeMS}.\textit{AverageResponseMSMetric})
 \end{aligned}$$

Similar to this advertisement, service requester defines an inquiry in which the average response time of the service should be no more than 10 seconds and the cost should be no more than 5 dollars. This inquiry can be expressed as:

CHAPTER 4. QoS-AWARE SERVICE DISCOVERY WITH OWL-QoS ONTOLOGY SUPPORT

Figure 4.5: Advertisement Example of OWL-QoS Ontology

$$InquiryQoS \doteq QoSProfile$$

$$\sqcap(\leq 500 \text{ cost}) \sqcap (\forall \text{cost}. \text{CostUSCentMetric})$$

$$\sqcap(\leq 10000 \text{ averageResponseTimeMS})$$

$$\sqcap(\forall \text{averageResponseTimeMS}. \text{AverageResponseMSMetric})$$

In this example we have $ProviderQoS \sqsubseteq InquiryQoS$, therefore the service provider's $ProviderQoS$ meets the service requester's requirement defined in $InquiryQoS$. In the functional service discovery phase, the service requester finds the services list according to the service's functional properties in the registry and gets the cargo service. In the nonfunctional service discovery phase, the matchmaking algorithm returns the *Subsume* degree for this service's advertisement, which means the service is a satisfying candidate for the service requester. Hence the service requester selects this service.

The constraint definitions in these QoS profiles cover property domain, range, and cardinality. More specifically, in Racer's syntax, the advertisement can be expressed as:

```
(define-concept ProviderQoS (AND QoSProfile (AT-MOST 100 cost)
(ALL cost CostUSCentMetric) (AT-MOST 5000 averageResponseTimeMS) (ALL
averageResponseTimeMS AverageResponseMSMetric)))
```

Since the OWL parsing tool has limited support for the XML schema types, we use the DL's number restriction instead of the concrete domain's restriction to constrain the properties to avoid integration problems with other systems. Our solution limits the precision of the numeric constraints, however this problem can be solved by proper choosing of the metric's unit. It is one of our future work to invest the solution based on the concrete domain, when the XML schema gains better support from the OWL specification and its parser tools.

We also want to mention that the tasks for defining the metric concepts and individuals are separated among service partners and measurement partners. As shown in

part D of Figure 4.5, the details of metrics individuals are defined by the measurement partners. After the service requester and service provider agree upon the QoS profile, it is the measurement partner's responsibility to define the metrics individuals by filling in their property values. On the other hand, service requester and service provider focus on the profile and metrics concepts definition in the T-Box. As shown in part B of Figure 4.5, the service provider chooses the proper metric concepts to form their QoS profiles during their system design. Consequently their system developers will choose the proper system design pattern and resource bundles according to the selected QoS profile. Service requester chooses their favorite metrics to discover and does not need to deal with metrics individuals.

Users' opinions on the model's layered design are criteria for the application of the model. For this case, a user study is held for 20 students who are from computer engineering department. Through the user survey we evaluate the users' satisfaction with the following two cases.

- (i) Compare the hierarchical definition of the metrics with the flat definition of the metrics according to the understandability.
- (ii) Understand the layered (SLA, profile, and metrics) design of our QoS ontology.

In the case (i), users are presented the hierarchical definition of the metrics in Figure 4.4 and the flat descriptions of the metrics. Users are asked to compare the understandability of hierarchical and flat definitions for the metrics. Table 4.6 shows the voting result. Users find that the hierarchical definitions of the metrics are easier to be understood. We think its reason is that the hierarchical definition uses the graph to provide additional information of metrics and their relationships.

In the case (ii), users are presented with our layered design and role introductions of the SLA. They are told to have a comparison for its understandability and the mono

	Bad	Poor	Fair	Good	Excellent
Hierarchical	0 %	0 %	20 %	80 %	0 %
Flat	0 %	70 %	30 %	0 %	0 %

Table 4.6: Voting on Understandability of Hierarchical and Flat Definitions for Metrics design’s understandability. Users’ opinions on whether they can understand the function and usage of this layered design is recorded. Table 4.7 presents the voting results. We find that some users has obviously preference to the layered design. One reason is that they are quite familiar with the network protocol’s layered design and think that the layered design as a must for complex systems.

	Bad	Poor	Fair	Good	Excellent
Understandability	0 %	0 %	20 %	65 %	15 %

Table 4.7: Voting on Understandability of the Layered Design of OWL-QoS

4.7 Discussion

Our OWL-QoS ontology focuses on the service discovery aspect of the Web Services. This decision is followed by the dynamic nature of Web Services. Instead of elaborating the detailed contract statements between agreement partners, our SLA layer can be viewed as a lightweight Service Level Agreement to provide the basic contract information. For the service discovery purpose, we specify the service constraints in the QoSProfile layer so that service provider can publish their promised SLO in the service registry, and service requester can discover the service with wanted quality from the service registry as well. The matching/discovery ability is often missing in other SLA works, however this becomes important due to dynamic nature of Web Services. The related works for the SLA specifications will be presented in the chapter 5.3.

The matchmaking algorithm presented in this chapter provides the way to process the QoS-Aware service discovery for the semantic Web Services. Defining the SLO constraints

in the DL by conjunction of numeric constraints, we prove that the conformance problem of SLO is equivalent to the concept subsumption problem in description logic. Hence, we can represent the SLO in the DL style and reuse the modern DL reasoners to process the QoSProfile concepts. The experimental result of the matching algorithm and the related works will be presented in the chapter 5. This approach has the following advantages:

- *Enable QoS-Aware Service Discovery:* The OWL-QoS ontology complements the OWL-S ontology to provide QoS-Aware service discovery for the Web Services. Based on our ontology, service providers can publish their service's performance information in the service advertisement, and service requester can match and locate the proper services by specifying the QoS requirement in their inquiry. This emphasizes the importance of QoS in Web Services, which is a precondition for the Web Services' application in mission critical tasks.
- *Multiple Matching Degrees:* The matching algorithm provides multiple matching degrees rather than strict matching/unmatching result. The traditional mathematics matching algorithm returns only stronger level matching results, while our algorithm provides the *SUBSUME*, *EXACT*, *PLUGIN*, and *INTERSECTION* results. This gives user a better idea about the matching level, and more alternative choices as well.
- *Reuse of the Reasoning Engine:* The concept subsumption problem is a general task for the DL reasoners. The modern DL reasoners have optimized performance for such reasoning problems and we can rely on its correctness and robustness on processing the matchmaking algorithm. We use standard interface to communicate with the engine, and this saves the cost for the development of new matching component, and when new reasoning engine is available, it can be plugged into the current system directly.

- *Aware of Metric's Hierarchy Structure:* Our algorithm makes use of the hierarchical structure for the metric concepts. The subclass metrics are treated as a type of super class metric. The service requester can specify the super class metric in his inquiry and get the related matching result from the matchmaking algorithm.

These are some features of our matchmaking algorithm. Our design principle for this ontology level QoS specification leads to its specific advantages, and we will do the discussion in chapter 5.

4.8 Summary

The Web Services market grows continuously by the advocacy of vendors and demand of consumers. As more and more services appear on the Web, service discovery is one of the key problems for the dynamic service invocations and integrations. To help the integration of services, the service's discovery process can be divided into two phases: the functional properties discovery phase and non-functional properties discovery phase. The correct understanding of the service's semantics is important for service integration. The ontology approach to define the domain knowledge has been proved to be a feasible solution for the integration problems. The OWL-S ontology describes the Web Services' functional properties on the semantic level, however, it did not provide a detailed set of classes and properties to represent quality of service metrics. Furthermore, for service selection and management purpose, it is necessary to precisely and flexibly specify QoS constraints and metrics definitions for Web Services descriptions.

This chapter provides a novel ontology as a complement for OWL-S ontology to support QoS-Aware service discovery and measurement service. In our model, three concept layers (SLA layer, profile layer and metrics layer), the profile individual and the metrics individual help to separate the design duties for partners. Formal semantics are defined

CHAPTER 4. QoS-AWARE SERVICE DISCOVERY WITH OWL-QoS ONTOLOGY SUPPORT

to describe the QoS constraints. A matchmaking algorithm for QoS property constraints is presented with multiple matching degrees. Well-defined metrics can be further utilized by measurement organizations to check whether the service provider conforms to the agreement. The service discovery and measurement phases will be introduced in chapter 5, together with our prototype framework implementation.

Chapter 5

Usage Phases of OWL-QoS Ontology and the Supporting Framework

This chapter presents the usage phases of OWL-QoS ontology, which include the QoS matching phase, SLO agreement phase, measurement preparation phase, and measurement phase. We present the underlying framework implementation for the OWL-QoS, together with the experimental results for matchmaking component and measurement component. The measurement code generation algorithm is provided after that. As to our knowledge, this is the first ontology level research work to cover the whole QoS life phases for Web Services. From our experimental results, we envisage its potential usage in the real e-business world.

5.1 Usage Phases of OWL-QoS Ontology

In this chapter, we provide a general overview of the usage scenarios for OWL-QoS ontology as shown in Figure 5.1. This approach consists of four sequentially related phase: *QoS Matchmaking*, *SLO Agreement*, *Measurement Preparation*, and *Measurement*. After the user discovers the proper services by the OWL-S matchmaker, a service list will be delivered to the OWL-QoS matchmaker. The *QoS Matchmaking* phase matches the QoS profiles among this service list and returns the service list together with its match

CHAPTER 5. USAGE PHASES OF OWL-QoS ONTOLOGY AND THE SUPPORTING FRAMEWORK

level. According to the match level, the service requester makes his selection decision. Then he signs the *SLO Agreement* with the service provider and issues the contract. A measurement partner is then responsible for the run-time monitoring of the service to ensure its conformance to the SLO agreement. The QoS profile ontology and QoS metrics are partially outsourced to the measurement partner. In the *Measurement Preparation* phase, the measurement details should be filled in by the measurement partner and agreed upon by service provider and requester. According to these detailed definition, measurement partner generates the measurement handler and deploys them on the measurement framework. The measurement framework provides the *Measurement* services. Logs and notifications will be created by the measurement framework to reflect the service's condition.

Figure 5.1: Overview of OWL-QoS Processing Phases

The approach described here allows a service developer to take advantage of the complementary strength of both OWL-S and our OWL-QoS ontology design model. On one hand (service profile side), service developer benefits by making use of OWL-S for semantic matchmaking of service capability, as well as the well defined process model and the grounding information. On the other hand (QoS profile side), the developer benefits

from the use of OWL-QoS's QoS profile layer for QoS matchmaking, as well as the QoS metric layer's definition for the QoS measurement.

5.1.1 QoS Matchmaking Phase

The QoS matchmaking phase takes the inputs of the OWL-S matchmaker's service list and the service requester's QoS profile. There are a few Java libraries for parsing OWL-S service profiles, for example the Mindswap's OWL-S API¹, the TUB OWL-S matcher², etc. Since our focus is the nonfunctional discovery for the Web Services discovery, we use single service profile directly, and we choose the OWL-S matcher library to parse basic OWL-S service profile and get the matched list of service profiles. After that, in the QoS matching phase we use the matchmaking algorithm introduced in section 4.6.2 to select the satisfying QoS profiles. To express the matching process more precisely, we list the matchmaking pseudocode here. Since the EXACT match is a special case of SUBSUME, the EXACT match is judged before the SUBSUME match. The rank of the matching level is as follows: SUBSUME > EXACT > PLUGIN > INTERSECTION > DISJOINT.

When the service list is passed to our matchmaking engine, we use *getQoSProfile* function to read the inputted service list and then generate its related QoS profile list. As to the service requester's InquiryProfile, we designed a wizard page to assist the requester to generate the inquiry QoS profile ontology. Our matchmaking engine then uses the algorithm *match* to calculate the ranked profile list and return the result back to the service requester. We view the QoS criteria as a soft criteria rather than a hard criteria. Hence in addition to the Subsume and Exact level, other matching levels are also returned back to the user to avoid missing any potential service. Since each service

¹OWL-S API, available at: <http://www.mindswap.org/downloads/>

²TUB OWL-S Matcher (OWLSM), available at: <http://kbs.cs.tu-berlin.de/ivs/Projekte/owlsmatcher/>

Algorithm 4 match(serviceList, inquiryProfile)

```

1: providerProfileList = getQoSProfiles(serviceList)
2: for all providerProfile in providerProfileList do
3:   if concept-equivalent(inquiryProfile, providerProfile) then
4:     put EXACT match
5:   else if concept-subsumes(inquiryProfile, providerProfile) then
6:     put SUBSUME match
7:   else if concept-subsumes(providerProfile, inquiryProfile) then
8:     put PLUGIN match
9:   else if concept-subsumes( $\neg$ inquiryProfile, providerProfile) then
10:    put DISJOINT match
11:   else
12:     put INTERSECTION match
13:   end if
14: end for

```

profile has a computed matching level, the additional results do not increase the inquiry time noticeably.

Figure 5.2 describes the system diagram of our matchmaker prototype. The prototype system contains an ontology repository, a converter and an ontology reasoner. We have four interfaces designed for the system: the inquiry interface, the publish interface, the browse interface and the administration interface. Some tools are used to implement our system: Jena parser [72] is used to parse the OWL ontology; the parsed ontologies are temporarily stored and processed in the memory through the OilEd's [30] internal ontology data structure; in the reasoner part, Racer[73] is selected as the ontology reasoner to classify the ontologies.

When the service provider publishes their service QoS profile through the publish interface, the ontology will be parsed by the Jena parser first. If the parsing process ends successfully, the ontology is stored in the server's ontology repository. Meanwhile the ontology is also loaded into the main memory and represented in the form of OilEd's ontology data structure. It is then rendered into Racer's KRSS[74] description through the Racer Render. After we finish the parsing and the converting process of the ontology,

Figure 5.2: Matchmaker System Prototype

the Racer engine accepts this KRSS description and publishes the profile concept to its knowledge base. By classifying based on its knowledge base, the Racer engine can organize the ontologies' taxonomy and improve the query speed. The publish and the classification process is relatively slow but this can be processed offline. Section 5.2 describes in details our prototype's performance in the experiment.

When the service requester submits an inquiry, the inquiry is compared against each related QoS profile. As described in section 4.6.2, the matchmaker gets all of the *Subsume* match, *Exact* match, *PlugIn* match, *Intersection* match and *Disjoint* match list. These result lists are sorted according to their ranks and then sent back to the service requester. We have designed a wizard page to assist the requester to the inquiry QoS profile. The wizard page let the user choose the name of the QoS profile concept, and then sets up the QoS properties and their numeric constraints. Figure 5.3 shows the inquiry wizard and Figure 5.4 shows the ranked list of the inquiry.

The browse interface enables the service requester to browse the ontologies list in the repository or view the content of a profile by the profile name or by the unique ID

CHAPTER 5. USAGE PHASES OF OWL-QoS ONTOLOGY AND THE SUPPORTING FRAMEWORK

Figure 5.3: OWL QoS Matchmaker's Inquiry Wizard

Figure 5.4: OWL QoS Matchmaker's Inquiry Result

that represents the ontology. The Administration interface provides the functionality to initialize or reload the whole knowledge base from the repository.

5.1.2 SLO Agreement Phase

The service requester selects his preferred QoS profile from the QoS matchmaker's output and then assigns the agreement contract with the service provider by filling in the details for SLA individual. The SLA individual represents the contract between the service provider and the service requester, and it defines the agreement's scope, the start time, the end time, and so forth. The industry level SLA examples are like WSLA[75]. Our SLA individual can be viewed as a simple version of Service Level Agreement. The properties of its definition is listed in Table 4.1.

Among the properties of the agreement, as listed in Table 4.1, the *hasServiceProfile* and *hasScope* properties specify the agreement's range, i.e., what Web Service the partners are agreeing with. These properties link to the OWL-S' service profile definition. The *hasQoSProfile* property points to the agreed QoSProfile individual, which specifies the SLO of the Web Service. The *startTime*, *endTime*, and the time masks properties define the SLA's measurement duration. For more detailed definition of the time masks, please refer to RFC3060[76]. The *priceStatement* describes the price declaration for the service's execution. The *penaltyStatement* describes the penalty to be paid if the service violates the agreement. Price and penalty information is quite delicate, hence these two statements are written as human readable natural language. Our system does not provide support for the automatic price/audit processing yet.

5.1.3 Measurement Preparation Phase

Measurement partner is the group that deploys the measurement handlers and performs the service measurement tasks. Normally it is outsourced to a third party since the

service requester and provider do not trust each other. Measurement partner's main task is to define the measurement details for the service invocation, when the service provider and service requester sign the SLA agreement. Figure 5.5 gives an overview for the measurement partner's activities, which include the detailed measurement definition, measurement code generation, deployment, and service monitoring.

After signing the Service Level Agreement by service requester and service provider, measurement partner fills in the details of the measurement for the SLA. It is done by setting the individuals for the metric concepts and the QoS profile concept (see step 1 in Figure 5.5). *ProfileMonitorIndividual* is the individual of a QoS profile concept. *MetricIndividual* is the individual of a Metric concept. *ComplexMetricIndividual* is the individual of a ComplexMetric concept. *FunctionIndividual* is the individual of a function concept. Tables 4.2, 4.3, and 4.4 show the properties for these individual definitions. The measurement partner need at least to specify the metrics' basic properties to answer how the measurement result is retrieved and how to report this to the measurement system. If the metric individual is an instance of ComplexMetric, the measurement partner needs to further fill in the definition of its operand metrics individuals together with its function individuals to answer how the value of the metric is calculated. Take part D in Figure 4.5 as an example, the QoS profile has one *averageResponseTime* property and one *cost* property. The measurement partner defines an *AverageResponseTimeMSMetric*'s individual. Since the *AverageResponseTimeMSMetric* is a ComplexMetric, the measurement partner specifies its operand *ResponseTimeMSMetric* and the average function. The average response time aggregates 100 response time's history data to calculate the metric's value. The *CostUSCentMetric* is also defined and the cost is a fixed value 100 cent. The detailed definition is available in Figure 5.6.

The function class defines the measurement logic for the system. It is written in the function individual rather than hard-coded in the measurement framework. This achieves

CHAPTER 5. USAGE PHASES OF OWL-QoS ONTOLOGY AND THE SUPPORTING FRAMEWORK

Figure 5.5: From Metrics to the Measurement

CHAPTER 5. USAGE PHASES OF OWL-QoS ONTOLOGY AND THE SUPPORTING FRAMEWORK

```

<qosmetrics:CostUSCentUnit rdf:ID="CostUSCentUnit1"/>
<qosmetrics:TimeMSUnit rdf:ID="TimeMSUnit1"/>

<qosmetrics:CostUSCentMetric rdf:ID="CostUSCentMetric1">
  <qosmetrics:pushPoint>http://155.69.150.102:9080/axis/urn:collector</qosmetrics:pushPoint>
  <qosmetrics:measureAt>
    <xsd:string xsd:value="Server"/>
  </qosmetrics:measureAt>
  <qosmetrics:value>
    <xsd:integer xsd:value="100"/>
  </qosmetrics:value>
  <qosmetrics:hasUnit rdf:resource="#CostUSCentUnit1"/>
</qosmetrics:CostUSCentMetric>

<qosmetrics:Average rdf:ID="Average100">
  <qosmetrics:functionName>
    <xsd:string xsd:value="Average100"/>
  </qosmetrics:functionName>
  <qosmetrics>window>
    <xsd:integer xsd:value="100"/>
  </qosmetrics>window>
  <qosmetrics:functionDefinition>
    <xsd:string xsd:value="Average"/>
  </qosmetrics:functionDefinition>
</qosmetrics:Average>

<qosmetrics:ResponseTimeMSMetric rdf:ID="ResponseTimeMSMetric1">
  <qosmetrics:hasUnit rdf:resource="#TimeMSUnit1"/>
  <qosmetrics>windowSize>
    <xsd:integer xsd:value="200"/>
  </qosmetrics>windowSize>
  <qosmetrics:value>
    <xsd:integer xsd:value="1000"/>
  </qosmetrics:value>
  <qosmetrics:measureAt>
    <xsd:string xsd:value="Client"/>
  </qosmetrics:measureAt>
  <qosmetrics:pushPoint>http://155.69.150.102:9080/axis/urn:collector</qosmetrics:pushPoint>
</qosmetrics:ResponseTimeMSMetric>

<qosmetrics:AverageResponseTimeMSMetric rdf:ID="AverageResponseTimeMSMetric1">
  <qosmetrics:hasFunction rdf:resource="#Average100"/>
  <qosmetrics:operand1 rdf:resource="#ResponseTimeMSMetric1"/>
  <qosmetrics:hasUnit rdf:resource="#TimeMSUnit1"/>
  <qosmetrics>windowSize>
    <xsd:integer xsd:value="15"/>
  </qosmetrics>windowSize>
  <qosmetrics>windowPos1>
    <xsd:integer xsd:value="-1"/>
  </qosmetrics>windowPos1>
  <qosmetrics:value>
    <xsd:integer xsd:value="1000"/>
  </qosmetrics:value>
  <qosmetrics:pushPoint>http://155.69.150.102:9080/axis/urn:collector</qosmetrics:pushPoint>
</qosmetrics:AverageResponseTimeMSMetric>

<myqos:MyQoS rdf:about="myqos:MyQoS">
  <qosmetrics:averageResponseTimeMS rdf:resource="#AverageResponseTimeMSMetric1"/>
</myqos:MyQoS>

```

Figure 5.6: Measurement Partner's Definition for Metrics

CHAPTER 5. USAGE PHASES OF OWL-QoS ONTOLOGY AND THE SUPPORTING FRAMEWORK

the sharing of the measurement knowledge by sharing of the function definitions, while still keeps enough flexibility for future's extension and modification.

After filling in all the metrics individual definition, measurement partner generates the measurement handler according to these individuals' definition. This is shown as step 2 in the Figure 5.5. The measurement partner can create their own measurement code manually or using some code template. Since all important information is already defined in the shared ontology, tools can assist the measurement partner to create the code stubs for measurement as well. To prove this, we have designed a sample code generator for the basic metric set (see section 4.5.4). The code generating process is listed as the following: after the metrics individuals are parsed into the memory, the code generator starts from the QoS Profile individual and gets the top level metric individuals according to the QoS properties (see algorithm *generateCode*). After that the second level metrics contained in the first level ComplexMetrics are traversed (see algorithm *generateComplexMetric*), and so on. Once all the metric levels are visited, the code generation process finishes. During the code generation, each AtomicMetric individual generates one measurement handler, each ComplexMetric individual generates one collector, each function individual generates one function parser, and each QoS profile individual generates one evaluator.

Algorithm 5 generateCode(individualQoSProfile)

```

1: for all metricProperty in individualQoSProfile do
2:   individualMetric  $\leftarrow$  metricProperty's metric individual
3:   if individualMetric isa AtomicMetric then
4:     generateAtomicMetric(individualMetric)
5:   else if individualMetric isa ComplexMetric then
6:     generateComplexMetric(individualMetric)
7:   end if
8: end for
9: generateEvaluatorCode()

```

Each metric has its corresponding measurement component in the measurement system. An AtomicMetric corresponds to a measurement handler, whose code is generated

Algorithm 6 generateComplexMetric(individualComplexMetric)

```

1: for all operand in individualComplexMetric do
2:   individualMetric  $\leftarrow$  operand's metric individual
3:   if individualMetric isa AtomicMetric then
4:     generateAtomicMetric(individualMetric)
5:   else if individualMetric isa ComplexMetric then
6:     generateComplexMetric(individualMetric)
7:   end if
8: end for
9: generateComplexMetricCode()

```

from template handler by the `genAtomicMetricCode()` method in algorithm *generateCode*. A `ComplexMetric` corresponds to a collector, whose code is generated by algorithm *generateComplexMetricCode*. During the `genComplexMetricCode` process, the function handler will be generated to process the operand metrics. If the `ComplexMetric`'s function is an `Aggregate` function, the operand metric will be viewed as data series in the system. The `ComplexMetric` will maintain a local data queuing for each operand metrics. The length of the queuing is decided by the function's *window* property. The aggregate function operates on the queuing during the run-time. The profile individual corresponds to a performance evaluator, whose code is generated by the `generateEvaluatorCode()` in algorithm *generateCode*. We will introduce these measurement components in more details in section 5.1.4.

Defining the metrics in concepts and measurement details in individuals simplifies the integration complexity. First, the metric concepts in ontology style facilitates the service provider, requester and measurement partner to share the basic metric knowledge and avoid the misunderstanding. Second, the details of metric definition is self encapsulated in the metric individual. Same metric concept can be reused in different measurement cases by defining different individuals. This simplifies the integration by sharing the same metric concept definition, while still keeps enough customization ability for the measurement by defining the metric individuals. Third, if outsourcing is required, ontology

naturally supports partially outsourcing the definition of concepts and individuals. This is a possible solution to avoid security and management problems if multiple partners are involved in the measurement tasks. Only the related metric individuals and concept definitions should be outsourced to the corresponding measurement partner. Fourth, the integration complexity is reduced since all the syntaxes are in the uniform syntax in OWL languages rather than wrappers for multiple QoS languages. A number of tools have already been developed for the OWL language's editing and reasoning.

5.1.4 Measurement Phase

Some basic requirements for the measurement system are: the service measurement should allow the involvement of third parties; the measurement system itself should be able to plug into the target monitoring Web Service system with minimal configuration/modification influences for it.

Keeping these requirements in mind, our major components in the measurement system contain: measurement handler, collector and evaluator. Each measurement handler corresponds to one AtomicMetric individual. Each collector corresponds to one ComplexMetric individual while the evaluator corresponds to the profile individual. Figure 5.7 shows our measurement system's architecture.

The details of AtomicMetric individual helps to generate the measurement handler. The measurement handler measures the first hand data from the deployed environment and then pushes the data to the collector. To make the push interface interoperable between partners, we define the signature of the push operator. The push point is described in the metric's pushPoint property. *void setMetricValue(String metricName, String strUID, String value)* is the push Web Service's signature. The *metricName* stands for the metric individual's full name. *strUID* is the unique id for the measured service invocation. *value* is the measurement reporting value.

Figure 5.7: Measurement Architecture

The measurement system prototype is based on the Axis SOAP engine[20]. Poorly designed measurement system can cause some serious influences on the target system, such as the change of the business logic, modification of the message head/content, etc. To minimize the influence on the target Web Service, the measurement code is wrapped as Axis handler. The benefits of this approach is that, it can be processed at run-time and does not change the target Web Service's code or processing logics. The basic idea of it is that, it adds a handler configuration in the service deployment and re-deploys the service for monitoring. Figure 5.8 shows the Axis chain structure on the server. The small cylinders represent Handlers and the larger, enclosing cylinders represent Chains. A Chain is a Handler container consisting of a sequence of Handlers which are invoked in turn. When a message arrives, if a transport request Chain exists, it will be invoked, followed by a global request Chain, and then the service Chain if configured. The client side has a similar Axis Chain system while the invocation sequence is the service Chain, the global Chain and then the transport Chain. All of our measurement handlers are located before or after the service interface pivot. Therefore, the measurement code does not change the service's implementation. In the code generation process, the Atomic-

Metric generates the measurement handler together with its deployment description for the handler's deployment. To distinguish between different service invocations, a UID-Handler is required in the handler chain to assign unique identity in the SOAP message's head and control the evaluation period.

Figure 5.8: Axis Message Path on the Server

Figure 5.9 shows an example of measurement handler system to help the user's understanding. The measurement handlers in this figure are generated according to the metrics individual example shown in Figure 5.6. Three types of measurement handlers are created and deployed in this example: the UIDHandler, the ResponseTimeHandler and the CostHandler. The UIDHandler checks the measurement period. If the current time is within the measurement period, it will generate the unique identity for each service invocation. Other measurement handlers will execute and push the measurement data only if the unique identify is available in the SOAP head. The ResponseTimeHandler measures the response time while the CostHandler measures the cost. All these handlers do not make change on the target service since they are deployed before or after the service pivot. Such handler system minimizes the influence on the original Web Services.

Each collector corresponds to one ComplexMetric and it stores the received data series for function calculation. Since all the measurement components are loosely coupled,

Figure 5.9: Axis Measurement Handler Example

other measurement partners can conveniently participate in the measurement. The collector receives the reports pushed from different measurement handlers and calculate the higher level metrics from low level metrics. JeksParser[77] framework is used in collector for function syntax's parsing. It is a Java library that parses, compiles and computes expressions and functions. The aggregate functions are implemented as user defined function handlers based on the jeks' function interface. The measurement partner can extend these original power function parser and define their own function handler as well.

The evaluator corresponds to the SLO profile's individual. It collects all related metric report in the profile. The evaluator's main task is to generate the measurement profile concept and then compare it with the agreed SLO profile ontology. If the measurement profile concept is subsumed by the agreed SLO profile, it is deemed as a compliance. Otherwise there is a violation. For example, we have the ProviderQoS as the agreed SLO profile. The profile describes the agreement that the cost of the service is no more than one dollar and the average response time is no more than five second. The measured data shows that the cost is one dollar and the average response time is four minute. Therefore,

CHAPTER 5. USAGE PHASES OF OWL-QoS ONTOLOGY AND THE SUPPORTING FRAMEWORK

$MeasureQoS \sqsubseteq ProviderQoS$ and no violation for the Web Service. If any violation happens, a notification will be created and sent out to the locate mentioned in the *notification* property to indicate the violation. The evaluator generates the measurement report in concept style. The benefit of this method is that different partners can share the report without misunderstanding. The meaning of the concept and related metrics are precisely defined. The outsourcing and integration can be achieved more naturally. The run-time burden of the evaluation system is affordable. From the experiment we find that the inference time for the violation check algorithm is normally less than one ms.

$$\begin{aligned}
 ProviderQoS &\doteq QoSProfile \\
 &\quad \sqcap(\leq 100cost.CostUSCentMetric) \\
 &\quad \sqcap(\leq 5000averageResponseTimeMS. \\
 &\quad \quad AverageResponseMSMetric) \\
 MeasureQoS &\doteq QoSProfile \\
 &\quad \sqcap(= 100cost.CostUSCentMetric) \\
 &\quad \sqcap(= 4000averageResponseTimeMS. \\
 &\quad \quad AverageResponseMSMetric)
 \end{aligned}$$

Because of our limited development resources, the prototype did not target all types of Web Services. Instead we provide support for the basic metric set (see section 4.5.4) together with their measurement handler templates. This set is designed to speed up measurement handler's development for generic Web Services. If an AtomicMetric is within the basic metric set, this AtomicMetric is supported by our code generator. According to the AtomicMetric's property values and its code template, the code generator

generates the metric handler and its deploy scripts. The *measureAt* property will affect the selection of the code template to affect the generated codes.

We suggest a market place for trading the measurement stub templates for the Web Services. Since all metrics concept definitions are shared, the service provider, service requester, and measurement partner can reuse this knowledge base and the designed measurement stubs. The templates can be put on the market place for reuse and trading. Measurement partners firstly search for the ontology knowledge base to get the predefined metrics. After that, they search the market place to find the proper stub templates. For most Web Services, all these stuffs are shared among partners. In case of a matching failure, they have to design the new metric concept, and then share this knowledge by publishing the ontology definition to the knowledge base. The corresponding measurement stub templates can be shared on the market place as well.

5.2 Experimental Results

The purpose of our experiment is to prove the feasibility of the matchmaking algorithm, to evaluate the performance of this approach, and to check the overhead of our measurement framework. With the experimental results, we want to find out whether its performance can be applied in the e-business world with online live Web Services.

The first part of our experiment focuses on the matchmaking portion of the system. This is the key point in the QoS-Aware service discovery process. Two types of experiments are conducted: the publish and the inquiry. The testing system for the experiment is Dell PowerEdge Server with Xeon 2.8-GHz CPU, 1024M memory, and operating system Windows 2003. We use Racer as the backend DL reasoner. Actually the Racer engine did not utilize these computing resources fully in the experiment. It takes about 8M memory and the CPU usage is close to zero percent in most of the publishing time. As

CHAPTER 5. USAGE PHASES OF OWL-QoS ONTOLOGY AND THE SUPPORTING FRAMEWORK

for the advertisement and inquiry concept, we established five hundred profile concepts for the experiment. Each profile has five randomly generated metric constraints.

Figure 5.10 shows the publish time for inserting a profile into the knowledge base. The X axis shows the number of the advertisement profiles in the knowledge base before the publish. The Y axis shows the publish time for the new advertisement profile. To publish the concept, we first parse the profile concept and convert it into Racer’s KRSS[74] format. This KRSS definition is then published to the Racer engine. After that the new knowledge base is classified. Hence, the publish time contains the time for parsing the advertisement profile concept, defining the concept in the knowledge base and classifying the knowledge base. From the figure, we can see that the publish time is relatively long when there are large number of advertisement profiles. However, the publish is a one time procedure and can be processed offline. Furthermore, the curve increases almost linearly on the graph. The linear increment as observed from the graph shows the potential of the Racer engine to deal with larger published dataset when more advertisements are added to the knowledge base.

Figure 5.10: Advertisements’ Publish Time

In the inquiry experiment, we setup a list of similar services with different QoS prop-

erties. In the experiment, the inquiry profile concept is parsed and converted into KRSS description. This description is then compared against each profile concept in knowledge base using the algorithm *match*. Therefore, the inquiry time is the inquiry concept’s parsing time plus the total subsumption reasoning time for each advertisement. In the experiment, the precision of the inquiry is 100%, since all the returned result are compatible with the inquiry.

Figure 5.11 shows the inquiry time for the published knowledge base. In the figure, the X axis shows the number of advertisement profiles in the knowledge base and the Y axis shows the inquiry time for the inquiry profile. From the figure, we can see that the inquiry process is much faster than the publish process. The inquiry time for the whole five hundred records is less than eight hundred ms and the inquiry can be further processed in parallel because they do not change the knowledge base. Since the number of similar services are normally limited, the searching’s space is constrained. This experiment shows the chance to use the OWL-QoS in the real e-business world using semantic web technologies.

Figure 5.11: Advertisements’ Inquiry Time

The next portion of the experiment tests the overhead of our measurement system.

CHAPTER 5. USAGE PHASES OF OWL-QoS ONTOLOGY AND THE SUPPORTING FRAMEWORK

This experiment is conducted on the same machine as the previous experiment. The experiment's service is deployed in the Tomcat server. We use the Axis SOAP engine as the Web Service engine. In the experiment, the measurement configuration is shown in Figure 5.9. On the client side's request flow, we deployed the SimpleUIDHandler and the ResponseTimeHandler. On its response flow, we deployed the ResponseTimeHandler. On the server side's request flow, we deployed the SimpleUIDHandler. On its response flow, the SimpleUIDHandler and CostHandler are deployed.

To compare the overhead, we firstly evaluate the average response time and memory usage of 50 service invocations without the measurement handler deployed. After this, we measure its performance again with the measurement handler deployed. Table 5.1 shows the experiment results. From this experiment, we can see that the response time's overhead is relatively small, compared with the normal service invocation time in seconds. This overhead can be further reduced by cache and batch feedback technology. The memory usage overhead creates 56 percent overhead. This memory overhead contains the memory usage of collector and evaluator's as well. This overhead can be further reduced if the measurement partner deploys the collector and evaluator on separated application server.

Overhead	Value	Percentage
Memory(Mb)	11.4	56 %
Response Time (ms)	29	2 %

Table 5.1: Overhead of the Measurement Stubs

From the user's view point, the response time is one of the most important performance influence for the service. From this experiment, we can see that the deployment of the monitoring service does not put a significant overhead to the service's performance. This overhead is normally acceptable for the Web Services and e-business in real world.

The users' opinions on the using and result of the OWL-QoS discovery system are then evaluated. A user study is carried out for 20 students who are selected from the

CHAPTER 5. USAGE PHASES OF OWL-QoS ONTOLOGY AND THE SUPPORTING FRAMEWORK

computer engineering department. We setup five Web services with same function for adding two numbers. Among these services, one is very fast, while another is very slow. The other three is of normal speed. Through the user study we evaluate the users' satisfaction with the following two cases.

- (i) Understand and use the inquiry GUI to query the services.
- (ii) Evaluate the services result returned by the QoS-Aware service discovery system.

The purpose of this evaluation is to check the design of the inquiry interface and the service's result quality. The users are asked to fill in their satisfaction of the OWL-QoS system. Table 5.2 shows the users' evaluation result. Users are relatively satisfied with the inquiry GUI. Some users are less satisfied with the result. A possible reason is that the result returned in the top of the list is of the subsume level, but not necessarily the fastest service.

	Bad	Poor	Fair	Good	Excellent
Inquiry GUI	0 %	0 %	35 %	45 %	20 %
Return Result	0 %	0 %	55 %	40 %	5 %

Table 5.2: Voting on Usability of Inquiry and Return Result

5.3 Related Works

There're many research works that target the description, advertising and signing up of Web Services at defined QoS levels. These are normally application layer specifications which are hardware and platform independent. They include aspect oriented approach such as QuO framework [78] and its QoS Description Language (QDL); object-oriented approach such as HP's QoS Modeling Language (QML) [79]; XML based QoS

languages such as HP's Web Services Management Language (WSML) and its framework [71], IBM's Web Service Level Agreement (WSLA) language [75] and its supporting framework, the Web Services Offer Language (WSOL) [67], WS-Policy [80], WS-QoS Framework[52], and UDDI eXtension (UX) project; ontology level works, such as Web Service Modeling Ontology (WSMO)[42], Mid-level Ontologies for Quality (MOQ)[81], and QoS Ontology (QoSOnt)[82].

The following two sections describe these research works in more details. We can divide these research works into two types: non-ontology based works and ontology related works.

5.3.1 QoS Related Service Description without Ontology Definition

QoS Description Language (QDL)[78] is the description language for the Quality of Service for CORBA Objects (QuO) framework, which extends the CORBA functional IDL. It adopts aspect-oriented programming (AOP)[83] as its design choice. Through aspect-oriented approach, an application can be decomposed into functional components and aspects, where different aspects can be programmed in different languages. The QDL language consists of three sub-languages: the contract description language (CDL), the structure description language (SDL), and the resource description language (RDL). The CDL specifies a QoS contract. It is not based on XML since it is designed for CORBA[4] IDL. It is not compatible with Web Services descriptions. The SDL and RDL are not further described in this paper.

Quality of Service Modeling Language (QML) [79] is a non-XML based specification for defining multi-category QoS specifications for components in distributed object systems. Through object oriented approach, it provides specification refinement and simple contract types such as reliability and performance. Complex QoS specification can be expressed, for example, using percentiles, variance and frequency aspects. QML supports

the specification reusability through contract and profile refinement. The well-defined semantics and reusability makes it a good OOP style QoS specification. It is not based on XML as well, which leads to interoperability and integration problems with the Web Services protocols stacks.

WS-Policy[80] is a general framework for the specification of policies for Web Services. The details of the specification for particular policies will be defined in specialized languages. It is flexible because its policies are not limited in certain places and its specification is extensible through additional specifications[84]. However it did not provide a detailed specification for the QoS related policies. We are waiting for WS-Policy's newest specification for the QoS related issues.

Web Services Management Language (WSML)[71] and Web Service Level Agreement (WSLA)[75] are sophisticated Service Level Agreement Languages developed for the XML-based specification of custom-made SLAs for Web Services. They define SLAs which contain QoS constraints, prices, and management information. Appropriate management infrastructures are accompanied with these specifications since they are oriented towards management applications in enterprise scenarios. These specifications try to provide a precise and flexible solution for SLA definition. Support for templates is available in WSML and WSLA to provide the specification reusability. Compared with WSLA and WSML, our work can be viewed as a lightweight SLA. It is mainly designed to achieve good extensibility and integration ability. WSLA and WSML contain more sophisticated language definitions about the QoS guarantees, price/penalty statements, and partner information. Our specification focuses more on the service discovery portion. We provide the QoS profile matchmaking algorithm with multiple matching degrees. Working with OWL-S, the management system tends to offer better automation because of the precise matching of service functional and nonfunctional properties. Although WSLA and WSML have template and macro structures for the reuse of specification, we use the

object-oriented approach to define the QoS profile. We believe this is a natural way to combine the QoS profile with the service components, since the service components are probably designed through object-oriented style. Sharing of ontology knowledge achieves good flexible and agreeable reusability. Some ontology editors have been developed in Semantic Web community, such as Protege[31], OilEd[30], etc. This can be used as auxiliary tools for our profile modification. Furthermore, export/import of partial information is supported in OWL ontologies.

UDDI eXtension(UX)[85] is proposed in our previous work described in chapter 3. It facilitates requesters to discover services with good performance. In each network domain, the requesters' QoS feedback is collected and summarized in a local UX server. By sharing these experiences from all the requesters in the local domain, the system predicts the service's future performance according to the service's historical performance. However, to precisely define and share new metrics remains a problem. Therefore, we start our work on the formal definition of QoS metrics.

5.3.2 QoS Related Service Description with Ontology Definition

The Web Service Modeling ontology (WSMO)[42] provides a conceptual framework and a formal language for semantically describing all relevant aspects of Web Services. There are four main elements: Ontologies, which provide the terminology used by other WSMO elements, Web Service descriptions, which describe the functional and behavioral aspects of a Web Service, Goals that represent user desires, and Mediators, which aim at automatically handling interoperability problems between different WSMO elements. Their nonfunctional properties define a list of general properties definitions, such as performance, financial, and so on. A semantic matchmaking algorithm is provided to match provider's and requester's goals. Compared with our profile design, their QoS constraint details are defined in the instance level. How to reuse these individuals has not been

clearly defined in their paper. No further detailed information is described in WSMO ontology for measurement purpose. We define the QoS profiles in concept level and support high reusability. In our system, our metrics contain all the necessary information for measurement partners to generate the measurement handlers, to integrate the measurement information, and to validate the service's conformance to the agreement.

Web Services Offer Language (WSOL)[67] provides formal representation of various constraints as well as management statements. Its major feature is its rich set of reusability constructs, lightweight management infrastructure, and dynamic switching protocol. The definition of QoS metrics and how they are measured or computed are done in external ontologies. Compared with WSOL, we do not supported the run-time switching mechanism. Our run-time framework is a lightweight measurement and feedback system. The adaptation mechanism is left to the service partners' choice. Our specification uses the uniform ontology language rather than the hybrid solution to achieve easier integration. We have designed the comparison algorithm for different QoS profiles to assist the QoS-Aware service discovery and performance validation. We also support the code generator for the basic QoS metrics set to assist the measurement. Furthermore, the validation of specification on syntax and semantic level reduces the developer's burden, especially when the project is big.

Web Service QoS (WS-QoS)[52] defines the XML schema for Web Services to describe their services' high and low level QoS properties. The assistant framework is designed for the language specification to assist the service selection and publish. High level QoS requirement may be mapped to the actual QoS-enabled transport layer through its proxy. It uses ontology style xml schema to define custom metrics. Two levels of metrics, the service performance level metrics and the transport level metrics, are defined in their system. However, it is in fact an XML level language with no formally defined semantics. No formal reasoning support has been provided by sophisticated logic reasoners, which could lead to integration problems.

Mid-level Ontologies for Quality (MOQ)[81] tries to minimize the ambiguity in QoS evaluation by defining a formal semantics for QoS using a FOL style logic definition. It defines the conformance of a requirement by checking whether all sub-requirements of its composition are met. It also identifies the importance of the Traceability to check where the problem arises. Although it describes the general semantics for a QoS Ontology's consideration, it does not present an Ontology vocabulary and taxonomy of QoS terms. Hence it cannot be applied yet to solve the practical issues on QoS-Aware service discovery and measurement, unless enough QoS vocabulary definitions are provided.

QoS Ontology (QoSOnt)[82] is an OWL level QoS specification designed specifically for Web Services. It has some common features as our Ontology. It contains links to OWL-S and concentrates upon the definition of metrics and the requirement matching. However, due to the poor data type support in OWL and therefore in OWL tools, an associated XML language is used in QoSOnt to express the user-defined data type and QoS requirements. This is not a standard way for defining the requirement semantics in OWL, which lead to integration problems. In the new version of OWL 1.1 draft, the data type problem is better mentioned. Hence the new definition for its requirements may be designed more formally, and its parsing and reasoning can be done in a standard way by other OWL tools.

5.3.3 Discussion of the Benefits

Our work is one of the first researches to apply the ontology on the QoS related area and the result is OWL-QoS ontology and its supporting framework. We design the related matchmaking algorithm for the ontology to meet the design principles.

Web Services systems have their unique features that require the ontology knowledge to define its description and perform the matchmaking. Web Services are loosely coupled so that they are out of service requester's control. This makes the QoS aspect more important for the Web Services. Some issues of the service descriptions are: although basic

QoS metrics are commonly known to each service, their meanings are often ambiguous for the questions like how, where, and so on. Precise definition is premise for the service discovery. Furthermore, for the QoS-Aware service discovery, different Web Services have various QoS requirements. As a result, service partners need to extend the metrics set to allow this flexibility. Hence, easy to incorporate the new metrics, as well as to share these metrics knowledge are important requirements for Web Services.

Our work provides an ontology design pattern for Web Services. This work contributes to the service description and service discovery in following aspects.

- *Lower Integration Cost*: For the system with centralized ontology repository, the number of required mappings is of proportional relationship with the number of systems for integration. If such a central ontology is not used, the number of the required mappings grows quadratically, depending on the number of systems or interfaces to be integrated. This is the benefits of ontology to reduce the integration cost. See Figure 5.12 for the comparison. Semantic web based specification provides a way for different systems to speak the same language, hence integration cost is reduced. Uniformed data also provides a better view for decision-making and matchmaking.
- *Extensibility*: Different Web Services have various requirements for their QoS descriptions. This requires the QoS descriptions' data structure extensible to allow easy modification and addition of new metrics/properties/constraints. The OWL-QoS ontology provides a base for users to add new metrics or new metric properties to the knowledge base. Since it is based on the ontology rather than the hard coded data schema, new definitions are relatively easy to be added into the knowledge base. Furthermore, the openness of ontology definition facilitates the share of development experiences and fastens the development cycle.

Figure 5.12: Integration of Data without/with a central ontology

- Reuse of the Knowledge.* Different measurement partners often have various measurement systems and data structure for the target Web Services. For example, the measurement frameworks include the information system, monitoring system, and so on. Since various Web Services often have various QoS requirements, it is important to allow the share and reuse of new metrics definitions to reduce the development fees. Separate development of measurement frameworks and related data structure involve a bit research/development cost. Our ontology definition provides a reusable domain knowledge for creating and sharing these new metrics for various Web Services. Since this shared knowledge base provides precise and flexible definitions for these metrics, it is best practices for the partners to use such knowledge to avoid the misunderstanding and mistakes. Furthermore, we provide a sample measurement code generator for them to check and reuse the measurement tools.
- Validation.* Validation is a necessary function for large project. When a project grows big, it tends to have more errors and bugs in the service QoS descriptions.

Locating and checking such problem are very expensive if the validation must be done manually without proper assistant tools. We have focused on two types of validations: the syntax validation and the semantics validation. The syntax validation checks whether the specification's syntax is correct. The semantics validation checks whether the semantics of the specification is correct. We provide the tools to assist the developers to do the validation. The ontology parsing tool helps to validate the syntax correctness for the OWL-QoS. A validation algorithm is designed to check whether the semantics of the OWL-QoS is correct. We develop the implementation for this algorithm and support these two types of validations to fasten the project development.

- *Partial Outsourcing.* Web Services often involve multiple partners during its invocation. These partners often do not trust each other, or they want to keep part of the service information as secret. Therefore, it is a general requirement that only the required part of the description information is shared among partners. Partial outsourcing the descriptions to the partners demands that the description can be divided into smaller pieces while still keeping its syntax and semantics to be processible by the management tools. Compared with the other XML/text based descriptions, the ontology based descriptions offer a low cost way to do the outsourcing since its concepts (stands for the data schema) and individuals (stands for the data structure) can be partially exported respectively while still keeping its syntax and semantics processible by other partners.
- *Intelligent Matchmaking.* Special matchmaking algorithm is designed for the service descriptions according to the ontology's specific features. Although there are mathematical matching algorithms to check whether the required QoS metric value agrees to the advertised QoS metric, such algorithms do not take advantage of the

ontology's hierarchical structure. Our algorithm utilizes the relationships between the metric concepts and takes such relationships into the consideration. Hence, the subclass metrics are treated as a type of super class metric. Meanwhile, our matching algorithm not only matches those QoS descriptions with the strict satisfying conditions, but also the partially satisfying descriptions as lower matching degree. This provides the users with more alternative service choices.

- *Support of Object-Oriented Style.* Object oriented programming (OOP) is widely used in the software development world. This design style provides good reusability when user required new features are added onto the original system. In such situations, the OOP way provides the ability to inherit all of the visible super object/interface's functionality, hence the super object implementation can be reused with minimum development cost. Such reusability is normally not supported in the traditional QoS descriptions. We support such OOP style in our specification definition and design the corresponding matchmaking algorithms for the description comparison and validation as well.
- *Separate the design duties.* Separating the design duties is one of the major considerations of software engineering to improve the development efficiency and reduce the error rates. We support this separation by proper designing of the layered structure in the ontology. Different groups take separate tasks in the system design and development. System developers define the metrics layer to specify the metric definition. Service provider specifies the offered service QoS constraints on these metrics. The measurement partner specifies the measurement details in the metric individuals for the service monitoring. Hence different groups can perform their experts in the related development area. Although these various concepts and individuals definition are designed by different groups, these separate works can be

imported into the knowledge base for the sharing and processing. This improves the efficiency and reduces the cost of the development.

- *Interoperability*: In a complex scenario like the Enterprise application, there are a variety of management tools to assist the system's execution and maintainance. Semantic web based specification provides a way for different systems to speak the same language, hence basic integration is simplified. Uniformed data also provides a better view for decision-making and matchmaking. In our solution, a common ontology should be shared so that all partners in the cooperation speak and understand the same words. Previous supporting system normally uses wrappers to solve the interoperability problems but this is a costly solution.

5.4 Summary

For service selection and management purpose, it is necessary to precisely and flexibly specify QoS constraints and metrics definitions for Web Services descriptions. This chapter presents the usage scenarios and functional phases for the OWL-QoS ontology. When an inquiry is received, the OWL-S matchmaker generates a matching service list and passes it to our OWL-QoS matchmaking module. With the assistant of our matchmaker in the *QoS Matchmaking* phase, service requester locates the satisfying service according to his request profile definition. Then in the *SLO Agreement* phase he signs the service agreement with service provider by filling the details in the QoS profile's individual definition. Later in the *Measurement Preparation* phase, the metrics individual definitions are outsourced to the measurement partners. We design the code generation algorithm to assist measurement partners to create the measurement handler, collector and evaluators. Finally the measurement partners deploy and perform the measurement tasks in *Measurement* phase. Different part of the ontology are outsourced to partners and definitions can be uniformly integrated together in OWL.

CHAPTER 5. USAGE PHASES OF OWL-QoS ONTOLOGY AND THE SUPPORTING FRAMEWORK

Based on our design principle, our approach has some unique advantages: lower integration fee, extensibility, reuse of the knowledge, support of validation, partial outsourcing, intelligent matchmaking, separate the design duties, etc. Combined with OWL-S, it tries to improve the business process automation and reduce the integration cost. From the initial experiments, we see the potential chance to utilize this ontology level QoS-Aware service discovery in real e-business scenarios. In the future, we believe the Semantic Web technology will help to integrate system components and tools into a more unified system.

Chapter 6

Conclusions

The topic of this Ph.D dissertation is the Discovery of Web Services, and our focus is the QoS-Aware service discovery for Web Services. In this chapter, we first summarize our research works on the area of QoS-Aware service discovery. We then conclude the achieved major contributions in section 6.1. Finally, section 6.2 gives an overview on the possible future directions of this research work.

In the beginning of this dissertation, we review the background of general Web Services' technology and their supporting frameworks in chapter 2. Compared with the general software, Web Services will encounter more performance bottlenecks due to the limitations of the underlying messaging and transport protocols. Meanwhile it is poorly supported for the QoS related issues in general Web Services. The UDDI is the industry's service discovery standard, but it supports only the keyword matching algorithm, and it provides no QoS related data structure for the QoS-Aware service discovery.

We first design the UDDI eXtension (UX) system to address the QoS-Aware service discovery problem. Chapter 3 presents this extension system for UDDI and discusses our initial design principles for it. Its major purpose is to facilitate service requesters to discover services with good performance. In each enterprise domain, the requesters' QoS feedback is received and then stored in a local UX server. By sharing these experiences from all the requesters in the local domain, the UX system can predict the service's

CHAPTER 6. CONCLUSIONS

future performance. To support the discovery between different cooperation domains, we design a general federated service and then enhance the UX system's federated discovery ability based on this federated service. The system handles the federated inquiry and provides a simple view over the whole federation. Meanwhile the UX server's inquiry interface will still conform to the UDDI 2 Specification. Compared with the original UDDI system, our system is aware of the basic service performance information with relatively small overhead from the feedback. The federated discovery helps the system to perform the discovery in the wider areas, and to estimate the QoS difference between domains dynamically.

Chapter 4 introduces the design of our OWL-QoS ontology for service discovery and measurement. We originally designed this ontology during the time of the DAML-S ontology. Therefore it is originally written in the DAML+OIL language, and later converted into OWL formatted ontology. We discuss the design principle of this ontology first. Based on these design principles, we present this ontology model's design pattern to assist the user in describing the service's QoS properties. In this model, the ontology can be divided into three layers: SLA Layer, QoS Profile Layer and Metrics Layer. SLA Layer defines the schema for a general service level agreement. QoS Profile Layer defines the constraints of a service's QoS properties. The Metrics Layer defines QoS metrics and their measurement information. Formal semantics are defined to describe the QoS constraints. A matchmaking algorithm for QoS property constraints is presented with multiple matching degrees. Well-defined metrics can be further utilized by measurement organizations to check whether the service provider conforms to the agreement. This ontology level solution facilitates the service partners to share and reuse the service knowledges during the service discovery and measurement process.

The usage phases of OWL-QoS are presented in the chapter 5, as well as the experimental results. The service measurement consists a series of tasks rather than a single

step. For example, these tasks contain: defining SLA by partners, specifying the measurement details by measurement partners, and deploying the measurement system in the current Web Services architecture. Based on the analysis, we divide the ontology's use cases into four phases: *QoS Matchmaking*, *SLO Agreement*, *Measurement Preparation*, and *Measurement*. To prove the feasibility of this theoretical ontology and its corresponding algorithms, we design the prototype system for the matchmaking system and measurement system. From the initial experiments, we see the potential to utilize this ontology level QoS-Aware service discovery in real e-business scenarios. We further discuss a number of related works and make some comparison between these works and our work. Based on the ontology level solution, we achieve the benefits of lower integration fee, extensibility, reuse of the knowledge, automatic validation, separation of the design duties, etc. We believe such solutions will help to resolve the interoperability problems for enterprise Web Services systems.

6.1 Major Contributions

The Web Services are evolving continuously and will probably become the dominant distributed computing technology. The QoS aspect of Web Service is important since this is the precondition for Web Services to apply in the mission critical applications. To summarize, the main contributions of this dissertation are listed as follows.

- Support of QoS-Aware Web Services discovery. It extends the general Web services discovery by the matching of QoS related properties. Service requester should be able to find the Web Services according to their QoS information. Service provider can publish and advertise their services by describing their promised QoS level. This mechanism is the precondition to apply the Web Services in mission critical applications. Chapters 3 and 4 present the QoS-Aware service discovery system with and without ontology support respectively.

CHAPTER 6. CONCLUSIONS

- The development of OWL-QoS ontology to support well-defined Semantics for QoS-Aware Web Services discovery. Well-defined Semantics of QoS information are important for large-scale QoS-Aware service discovery. In chapter 4 we define the QoS metrics classes as basis to compare among different services. The semantics of the metrics are precisely defined to avoid misunderstanding. The model of the metrics are extensible to allow the definition of metrics in the new domain. We further present the semantics and conformance rules for our QoS specification. Based on this definition, the service level objectives can be formally presented and compared.
- Design of matchmaking algorithms, discovery protocols and the measurement framework. According to the semantics of service QoS information, corresponding matchmaking algorithms are presented in section 4.6 to calculate the service's matching degree. The discovery protocol and the measurement framework presented in chapter 5 cover the general usage phases for the QoS-Aware Web Services discovery, agreement, and measurement. The experimental results show its potential application in the real e-business world.
- Lower the Integration Cost. Our ontology provides the basis for the service provider and service requester to share the knowledge. It reduces the number of the required mappings for the different service partners. Meanwhile, it works as a compliment of OWL-S to provide the QoS-Aware service discovery and measurement support for Web Services. Better automation and sharing of knowledge help the service partners to achieve lower integration fee.

There are also some limitations exist in our work:

- No disjunction of the inquiry conditions. The QoS profile concepts for the matchmaking is defined by the conjunction of the separated constraints. However, the

CHAPTER 6. CONCLUSIONS

users are not able to express the disjunction of the constraints, since the concepts defined in this way are not necessarily follows the conformance rules. This inability forces the users to issues multiple inquiries instead of one disjunction inquiry.

- Reasoning time is still slow for large data set. The matchmaking of the QoS profiles requires the DL reasoner to do the subsumption check for the concepts. The speed of the subsumption checking is still slow, compared with the traditional rule engines in our experiments. When the number of the advertisement concepts are huge, the reasoning time of the system will affect the user's experience.
- The complexity of the ontology development. Although the concept can express the domain's knowledge in machine understandable style, the defining of the new concepts in the ontology is not an easy task. In our example, we need the QoS domain experts to define and check the structure of the ontology for their domain's metrics. This takes a bit of learning and training for the end users and providers. This is partially solved by well-designed tools, which can assist the normal users to make use of the system.

6.2 Future Directions

In the future, Semantic Web technology will help to integrate system components and tools into a more unified system. Precisely defined semantics and unified system view will reduce the system integration cost and reuse the well-established domain knowledge. New trends in Semantic Web technology will form new solutions for system management. Here we list some important future directions of our research work.

- Ontology definition. The definition and sharing of ontology in the new domain is the necessary step if new domain's web applications tend to adopt the OWL-QoS ontology. To reduce the cost of this process, the automatic translation and

CHAPTER 6. CONCLUSIONS

alignment of predefined ontology to the OWL-QoS' ontology is a preferred solution. However, the ontology translation and alignment is itself a research topic. A number of research works deal with the ontology translation[86] and ontology integration[87]. By utilizing these research results, more cost-effective solutions may be achieved by the service providers to create their domain specific ontology base.

As to the generic metrics used in current OWL-QoS, we plan to invest more metrics definition for the web based applications. In our current work, a basic set of metrics are setup, while the domain specific metrics are left to the user's definition. Some metrics, such as the security and data quality, may be incorporated into our generic ontology when we get more knowledge about its definitions. Additional general metrics for users' Web Services are also considered, such as the stability metrics, additional performance metrics, etc. Meanwhile, we are paying close attention to the current research works on the QoS ontologies, such as the QoSOnt[82], to improve our ontologies' quality.

- Support of arbitrary data type. Due to the limited support of XML data types in the OWL 1.0, arbitrary XML data types are not used in our QoS constraints to gain the interoperability. In the new version of OWL 1.1 draft, the data type problem is better mentioned, and this may lead to finer requirements definition. Meanwhile, according to the latest development of the Pellet¹ system, it has start to provide the support for a rich set of basic datatypes including numeric types, strings, and date/time types. User-defined simple datatypes are also supported primitively in Pellet. This gives us a direction to solve the arbitrary data types problem.

¹Available at <http://www.mindswap.org/2003/pellet/>

CHAPTER 6. CONCLUSIONS

- Service adaptation. It is a natural requirement to automatically choose and adapt a new service for the target service when the target service fails to satisfy the QoS requirement. The automatic adaptation requires some prerequisites: the well-defined matchmaking engine to discover the services with the same functional properties and proper non-functional properties; the service QoS measurement framework to detect the service's execution status; the definition of the event triggering rules to activate the service switch process; The adaptation rules and the supporting framework are important elements for the system's work. The first two conditions are partially solved in our OWL-QoS framework. The third condition requires proper definition of the rule system. The problem is that the OWL ontology language has no proper constructors for the rules' definition. Some research works has proposed the solution for the rules definition in the ontology. For example, SWRL[88] may be a candidate to help the expression of QoS adaptation rules and system management policies. Some further work could be the audit and pricing system, which is preferred for a mature Web Services system.

References

- [1] World Wide Web Consortium, “Web Services Description Language (WSDL) 1.1,” *available at: <http://www.w3.org/TR/wsdl>*, 2001.
- [2] XML Core Working Group, “Simple Object Access Protocol (SOAP) 1.1,” *<http://www.w3.org/TR/SOAP/>*, 2000.
- [3] OASIS, “UDDI Version 2 API Specification,” *available at: <http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.pdf>*, 2002.
- [4] Object Management Group and Open Group, “Common Object Request Broker Architecture (CORBA/IIOP) 3.0 , *<http://www.omg.org/cgi-bin/apps/doc?formal/02-12-02.pdf>*,” 2002.
- [5] Network Working Group, “Hypertext Transfer Protocol (HTTP) 1.1,” *<http://www.w3.org/Protocols/rfc2068/rfc2068>*, 1997.
- [6] Network Working Group, “File Transfer Protocol (FTP),” 1985.
- [7] OWL-S Coalition, “OWL-S: Semantic Markup for Web Services,” *available at: <http://www.daml.org/services/owl-s/1.0/owl-s.html>*, 2003.
- [8] The DAML Services Coalition, “DAML-S: Semantic Markup for Web Services,” *available at: <http://www.daml.org/services/daml-s/0.7/daml-s.pdf>*, 2002.
- [9] J. Martin, “Web services: The next big thing,” *XML Journal 2*, *available at: <http://www.sys-con.com/xml/>*, 2001.
- [10] H. Kreger, “Web Services Conceptual Architecture (WSCA 1.0),” *available at: <http://www-4.ibm.com/software/solutions/webservices/>*, 2001.

REFERENCES

- [11] D. Chappell and T. Jewell, *Java Web Services*, O'Reilly, March 2002.
- [12] E. Cerami, *Web Services Essentials*, O'Reilly, February 2002.
- [13] T. Sollazzo, S. Handschuh, S. Staab, and M. Frank, "Semantic Web Service Architecture - Evolving Web Service Standards toward the Semantic Web," 2002.
- [14] J. B. Postel, "Simple Mail Transfer Protocol (SMTP)," 1982.
- [15] Network Working Group, "The Blocks Extensible Exchange Protocol Core," 2001.
- [16] D. Winer, "XML-RPC Specification," available at: <http://www.xmlrpc.com/spec>, 1999.
- [17] F. Leymann, "Web Services Flow Language (WSFL) Version 1.0," available at: <http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf>, 2001.
- [18] S. Thatte, "XLANG Web Services for Business Process Design," available at: http://www.getdotnet.com/team/xml_wsspecs/clang-c/default.htm, 2001.
- [19] T. Andrews and F. Curbera et al., "Business Process Execution Language for Web Services version (BPEL4WS) 1.1," available at: <http://www-106.ibm.com/developerworks/library/ws-bpel/>, 2003.
- [20] Axis Development Team, "Apache Axis 1.0," <http://ws.apache.org/axis/>, 2002.
- [21] Apache Software Foundation, "WSIF: Web Services Invocation Framework 2.0," available at: <http://ws.apache.org/wsif/>, 2002.
- [22] Mind Electric, "GLUE," available at: <http://www.themindelectric.com/>, 2002.
- [23] Paul Kulchenko, "SOAP::Lite for Perl," available at: <http://www.soaplite.com/>, 2001.
- [24] IBM, "UDDI4J Version 2.0, Java interaction implementation of UDDI," available at: <http://www.uddi4j.org/>, 2001.
- [25] World Wide Web Consortium, "Extensible Markup Language (XML) 1.0," available at: <http://www.w3.org/TR/REC-xml>, 2000.

REFERENCES

- [26] O. Lassila and R. R. Swick (eds.), “Resource Description Framework (RDF) Model and Syntax Specification,” *available at: <http://www.w3.org/TR/1999/REC-rdf-syntax-19990222>*, Feb 1999.
- [27] R.V. Guha (eds.) D. Brickley, “Resource Description Framework (RDF) Schema Specification 1.0,” *available at: <http://www.w3.org/TR/2000/CR-rdf-schema-20000327>*, Mar 2000.
- [28] Thomas R. Gruber, “A translation approach to portable ontology specifications,” *Knowledge Acquisition*, vol. 5, no. 2, pp. 199–220, 1993.
- [29] Natalya F. Noy and Deborah L. McGuinness, “Ontology development 101: A guide to creating your first ontology,” Tech. Rep., Stanford University SMI, 2001.
- [30] Sean Bechhofer, Ian Horrocks, Carole Goble, and Robert Stevens, “OilEd: A reasonable ontology editor for the semantic Web,” in *24th German / 9th Austrian Conference on Artificial Intelligence (KI 2001)*, 2001.
- [31] Holger Knublauch, Ray W. Ferguson, Natalya F. Noy, and Mark A. Musen, “The protege owl plugin: An open development environment for semantic web applications,” in *Third International Semantic Web Conference - ISWC 2004*, 2004.
- [32] Jeff Heflin and James Hendler, “Dynamic ontologies on the web,” in *17th National Conference on Artificial Intelligence(AAAI-2000)*, 2000.
- [33] S. Decker, D. Fensel, M. Erdmann, and R. Studer, “The technical core of ontobroker,” .
- [34] Dan Connolly, Frank van Harmelen, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein, “The DAML+OIL Language,” 2001.
- [35] W3C Committee, “OWL Web Ontology Language Reference,” *available at: <http://www.w3.org/TR/owl-ref/>*, Feb 2004.
- [36] I. Horrocks and U. Sattler, “Ontology reasoning in the SHOQ(D) description logic,” in *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001.

REFERENCES

- [37] Franz Baader, Diego Calvanese, Deborah McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, Eds., *The Description Logic Handbook: Theory, Implementation, and Applications*, Cambridge University Press, 2003.
- [38] Ian Horrocks, Ulrike Sattler, and Stephan Tobies, “Practical reasoning for expressive description logics,” in *Proceedings of the 6th International Conference on Logic for Programming and Automated Reasoning (LPAR’99)*, 1999, pp. 161–180.
- [39] Franz Baader and Ulrike Sattler, “An Overview of Tableau Algorithms for Description Logics,” *Studia Logica*, vol. 69, no. 1, pp. 5 – 40, 2001.
- [40] Katia P. Sycara and Massimo Paolucci, “Ontologies in Agent Architectures,” in *Handbook on Ontologies*, pp. 343–364. 2004.
- [41] J. Cardoso, A. Sheth, and J. Miller, “Workflow Quality of Service,” in *Proceedings of the International Conf. on Enterprise Integration and Modeling Technology and International Enterprise Modeling*, 2002.
- [42] Holger Lausen, Axel Polleres, and Dumitru Roman (Eds.), “Web Service Modeling Ontology (WSMO),” available at: <http://www.w3.org/Submission/WSMO/>, Jun 2005.
- [43] M. Paolucci, T. Kawamura, T. Payne, and K. Sycara, “Semantic matching of web services capabilities,” in *First International Semantic Web Conference*, 2002, To appear.
- [44] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne, and Katia Sycara, “Importing the Semantic Web in UDDI,” in *Web Services, E-Business and Semantic Web Workshop*, 2002.
- [45] Amos Tversky, “Features of similarity,” in *Psychological Review*, 1977, vol. 84, pp. 327–352.
- [46] Antonio Jorge Silva Cardoso, *Quality of Service and Semantic Composition of Workflows*, Ph.D. thesis, University of Georgia, Athens, GA, 2002.

REFERENCES

- [47] L. Li and I. Horrocks, "A Software Framework For Matchmaking Based on Semantic Web Technology," in *the International World Wide Web Conference (WWW2003)*, 2003.
- [48] E.C. Nelson, "A statistical basis for software reliability assessment," *TRW Systems*, 1973.
- [49] Ali Shaikhali, Omer F. Rana, Rashid J. Al-Ali, and David W. Walker, "UDDIe: An extended registry for web services," *Symposium on Applications and the Internet Workshops (SAINT'03 Workshops)*, 2003.
- [50] K. Sivashanmugam, K. Verma, and A. Sheth, "Discovery of Web services in a federated registry environment," in *IEEE International Conference on Web Services*, 2004, pp. 270–278.
- [51] "JXTA Technology," available at: <http://www.sun.com/software/jxta/>.
- [52] M. Tian, A. Gramm, H. Ritter, and J. Schiller, "Efficient Selection and Monitoring of QoS-aware Web services with the WS-QoS Framework," in *International Conference on Web Intelligence (WI'04)*, Sep 2004.
- [53] M. Christoffel, "Cooperations and Federations of Traders in an Information Market," in *AISB Symposium Intelligent Agents in Electronic Commerce*, 2001.
- [54] Massimo Paolucci, Katia Sycara, Takuya Nishimura, and Naveen Srinivasan, "Using daml-s for p2p discovery," in *International Conference on Web Services (ICWS03)*, 2003.
- [55] ISO/IEC IS 13235-1, "ODP Trading Function," in *ITU/T Draft Rec X950 - 1*, 1997.
- [56] M. P, "Domain names - implementation and specification," in *STD 13, RFC 1035*, 1987.
- [57] P. Francis, S. Jamin, V. Paxson, L. Zhang, D. F. Gryniwicz, and Y. Jin, "An architecture for a global internet host distance estimation service," in *Infocom '99*, 1999.

REFERENCES

- [58] R. L. Carter and M. E. Crovella, "Dynamic server selection using bandwidth probing in wide-area networks," Tech. Rep., Computer Science Department, Boston University, 1996.
- [59] J. Gwertzman and M. Seltzer, "The case for geographical push-caching," in *Proc. Fifth IEEE Workshop on Hot Topics in Operating Systems*, May 1995.
- [60] Mark Stemm, Srinivasan Seshan, and Randy H. Katz, "A network measurement architecture for adaptive applications," in *INFOCOM (1)*, 2000, pp. 285–294.
- [61] J. S. Davis, C. Bisdikian, W. F. Jerome, and D. M. Sow, "Emerging research opportunities in service discovery," in *NYNET01 - New York Metro-Area Networking Workshop*, 2001.
- [62] Richard S. W., *TCP/IP Illustrated*, Addison Wesley Longman, Inc., 1994.
- [63] D. Belaid, N. Provenzano, and C. Taconet, "Dynamic Management of CORBA Trader Federation," in *Proceeding of the 4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1998.
- [64] Network Working Group, "RFC 2617, HTTP Authentication: Basic and Digest Access Authentication," available at: <http://www.ietf.org/rfc/rfc2617.txt>, 1999.
- [65] Anbazhagan Mani and Arun Nagarajan, "Understanding Quality of Service for Web Services," Tech. Rep., IBM Software Labs, 2002.
- [66] Jorge Cardoso and Amit Sheth, "Semantic e-workflow composition," *J. Intell. Inf. Syst.*, vol. 21, no. 3, pp. 191–225, November 2003.
- [67] Vladimir Tomic, Bernard Pagurek, and Kruti Patel, "WSOL - A Language for the Formal Specification of Classes of Service for Web Service," in *the Int. Conf. on Web Services (ICWS03)*, 2003.
- [68] Xu Bin, Wang Yan, Zhang Po, and Li Juanzi, "Web services searching based on domain ontology," in *SOSE '05: Proceedings of the IEEE International Workshop*, Washington, DC, USA, 2005, pp. 51–58, IEEE Computer Society.

REFERENCES

- [69] James Caverlee, Ling Liu, and Daniel Rocco, "Discovering and ranking web services with basil: a personalized approach with biased focus," in *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, New York, NY, USA, 2004, pp. 153–162, ACM Press.
- [70] Gerard Salton and Chris Buckley, "Term weighting approaches in automatic text retrieval," Tech. Rep., Ithaca, NY, USA, 1987.
- [71] Akhil Sahai, Anna Durante, and Vijay Machiraju, "Towards Automated SLA Management for Web Services," HPL-2001-310 (R.1), 2002.
- [72] HP Labs Semantic Web Programme, "Jena - A Semantic Web Framework for Java," available: <http://jena.sourceforge.net/index.html>, 2002.
- [73] Volker Haarslev, Ralf Moller, and Michael Wessel, "Racer: Renamed abox and concept expression reasoner," available at: <http://www.sts.tu-harburg.de/~r.f.moeller/racer/>, 2003.
- [74] P.F. Patel-Schneider and B. Swartout, "Description-Logic Knowledge Representation System Specification from the KRSS Group," available at: <http://www-db.research.bell-labs.com/user/pfps/papers/krss-spec.ps>, Nov 1993.
- [75] Heiko Ludwig, Alexander Keller, Asit Dan, Richard P. King, and Richard Franck, "Web Service Level Agreement (WSLA) Language Specification, v1.0," available at: <http://www.research.ibm.com/wsla/WSLASpecV1-20030128.pdf>, Jan 2003.
- [76] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen, "Policy core information model," available at: <http://www.faqs.org/rfcs/rfc3060.html>, Feb 2001.
- [77] eTeks company, "Jeks and jeksparser," available at: <http://www.eteks.com/jeks/en/>.
- [78] John A. Zinky, David E. Bakken, and Richard E. Schantz, "Architectural support for quality of service for CORBA objects," *Theory and Practice of Object Systems*, 1997.
- [79] S. Frolund and J. Koistinen, "QML: A Language for Quality of Service Specification," Tech. Rep., HPL-98-10, Feb 1998.

REFERENCES

- [80] Maryann Hondo and Chris Kaler, "Web Services Policy Framework (WS-Policy) Version 1.0," December 2002.
- [81] Henry M. Kim, Arijit Sengupta, and Joerg Evermann, "MOQ: Web Services Ontologies for QOS and General Quality Evaluations," in *European Conference Conference on Information Systems*, May 2005.
- [82] Glen Dobson, Russell Lock, and Ian Sommerville, "QoSOnt: a QoS Ontology for Service-Centric System," in *31st EUCROMICRO Conference on Software Engineering and Advanced Applications*, Aug 2005.
- [83] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-oriented programming," in *Proceedings European Conference on Object-Oriented Programming*, Mehmet Akşit and Satoshi Matsuoka, Eds., vol. 1241, pp. 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.
- [84] Microsoft, IBM, BEA, and SAP, "Web Servies Policy Attachment (WS-PolicyAttachment)," available at: <http://www.verisign.com/wss/WS-PolicyAttachment.pdf>, 2002.
- [85] Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, "QoS-Aware and Federated Enhancement for UDDI," *International Journal of Web Services Research (JWSR)*, vol. 2, pp. 58–85, 2004.
- [86] Dejinj Dou, Drew Macdermot, and Peishen Qi, "Ontology translation on the semantic web," *LNCS Journal of Data Semantics*, vol. 2, no. 3360, pp. 35–57, 2004.
- [87] M Lenzerini D Calvanese, G De Giacomo, "A framework for ontology integration," in *the First Semantic Web Working Symposium*, 2001.
- [88] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosf, and Mike Dean, "Swrl: A semantic web rule language combining owl and ruleml," available at: <http://www.w3.org/Submission/SWRL/>, May 2004.

List of Publications

Book Chapter

1. Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, Web Services Discovery and QoS-Aware Extension, *Semantic Web Technologies and e-Business*. (to appear)
2. Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, QoS-Aware Web Services Discovery with Federated Support for UDDI, *Modern Technologies in Web Services Research*. (to appear)
3. Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, Web Services Discovery with OWL-QoS Ontology, *Modern Technologies in Web Services Research*. (to appear)

Journal

1. Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, Semantics in Service Discovery and QoS Measurement, *IEEE IT Professional Magazine*, Vol. 7, No. 2, pp. 29-34, 2005
2. Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, Web Services Discovery with DAML-QoS Ontology, *International Journal of Web Services Research (JWSR)*, vol 2(2), 44-67, 2005
3. Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, QoS-Aware and Federated Enhancement for UDDI, *International Journal of Web Services Research (JWSR)*, vol 1(2), 58-85, 2004

Conference

1. Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, QoS Measurement Issues with DAML-QoS Ontology, *in the proceeding of the IEEE International Conference on e-Business Engineering (ICEBE 2005)*, Oct, 2005

REFERENCES

2. Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, Service Discovery and Measurement based on DAML-QoS Ontology, *in the proceeding of Fourteenth International World Wide Web Conference*, May, 2005
3. Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee, DAML-QoS Ontology for Web Services, *in the proceeding of the International Conference on Web Services 2004 (ICWS04)*, Jul, 2004
4. Chen Zhou, Liang-Tien Chia, Silverajan Bilhanan, and Bu-Sung Lee, UX - An Architecture Providing QoS-Aware and Federated Support for UDDI, *in proceeding of the first International Conference on Web Services (ICWS03)*, Jun, 2003