View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Saskatchewan's Research Archive

Integrating Peer-to-Peer into Web Services

A Thesis Submitted to the College of

Graduate Studies and Research

In Partial Fulfillment of the Requirements

For the Degree of Master of Science

In the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Weidong Han

© Copyright Weidong Han, August, 2006. All rights reserved.

Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

176 Thorvaldson Building

110 Science Place

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

S7N 5C9

Abstract

The Service Oriented Architecture (SOA) is emerging as a new standard for building large loosely coupled systems. Web Services, the dominant implementation platform for SOA, use a server-centric approach to manage all components. This limits the deployment of Web Services to static domains, since a service invocation will fail if the server component changes its availability or location.

This research focuses on the possibilities of integrating P2P technology into the Web Services environment as a means of increasing its robustness. A P2P-Web Services architecture is presented that enables service discovery and service invocations in dynamic environments. The corresponding experiments on the reference system and the simulation system present the characteristics and improvements of the hybrid system.

Acknowledgements

First of all, I would like to sincerely thank my supervisor, Dr. Ralph Deters, for his persistent support, guidance, help, and encourage during the whole process of my study and my thesis.

I would like to thank the members of my advisory committee: Dr. Julita Vassileva, Dr. John Cooke, and Dr. Chris Zhang (external) for their valuable suggestions and comments.

I would also like to thank the students, staff and faculty of the Computer Science Department and especially to the students of MADMUC lab for their support. Also thank you to Ms. Jan Thompson for her help and support.

Finally, I would like to thank my wife and daughter for their support and love all the time. I would like to dedicate my thesis to them.

TABLE OF CONTENTS

PERMISSION TO USE .... **i**

ABSTRACT .... **ii**

ACKNOWLEDGEMENTS .... **iii**

TABLE OF CONTENTS .... **iv**

LIST OF FIGURES .... **vi**

LIST OF TABLES .... **viii**

LIST OF ABBREVIATIONS .... **ix**

CHAPTER 1 INTRODUCTION **1**

CHAPTER 2 WEB SERVICES AND P2P **4**

2.1 Web Services 4
2.1.1 SOAP *5*
2.1.2 WSDL *6*
2.1.3 UDDI *7*
2.1.3.1 The UDDI Publication API 9
2.1.3.2 The UDDI Inquiry API 9
2.1.4 Drawbacks of Web Services *10*
2.2 P2P 11
2.2.1 Gnutella *11*
2.3 Linking P2P and Web Services 13

CHAPTER 3 LITERATURE REVIEW **14**

3.1 Web Services 14
3.1.1 Use of UDDI *14*
3.2 P2P 15
3.2.1 The Centralized Directory Model *15*
3.2.2 The Flooded Requests Model *16*
3.2.3 The Distributed Hash Table Model (DHT) *17*
3.3 Integrating P2P and Web Services 19
3.4 Service Discovery 20
3.5 Service Selection 22
3.6 Conclusions 26

CHAPTER 4 P2P-WEB SERVICES ARCHITECTURE 28

4.1 The P2P-Web Services Framework (PWSF) 29
4.2 The Distributed P2P UDDI (PUDDI) 32
4.3 The P2P-Web Services Gateway (PWSG) 33
4.4 Usage of PUDDI and PWSG 34
4.5 The QoS Based Selection 36
4.6 Deployment 39

CHAPTER 5 EXPERIMENT 1: OVERHEAD AND PERFORMANCE OF PUDDI AND PWSG 41

5.1 Experimental Setup 41
5.2 CPU Usage of PUDDI 44
5.3 CPU Usage of PWSG 46
5.4 Bandwidth Consumption of PUDDI 48
5.5 Bandwidth consumption of PWSG 50
5.6 Throughput and Response Time of PUDDI 51
5.7 Throughput and Response Time of PWSG 54
5.7 Conclusions 55

CHAPTER 6 CALIBRATING THE SIMULATION SYSTEM 56

6.1 Design of the Simulation System 57
6.2 Choice of System Parameters 59
6.3 Calibration 60
6.4 Conclusions 62

CHAPTER 7 EXPERIMENT 2: EFFECTS OF P2P PARAMETERS 64

7.1 Effects of TTL 64
7.2 Effects of the Number of Neighbors 67
7.3 Conclusions 69

CHAPTER 8 EXPERIMENT 3: EXAMINING QOS SELECTION 70

8.1 Experimental Setup 70
8.2 Experimental Results 73
8.3 Conclusions 76

CHAPTER 9 CONCLUSIONS AND FUTURE WORK 77

REFERENCES 82

LIST OF FIGURES

Figure 1.1: Topology of Web Services 1
Figure 2.1: The Web Services development/deployment scenario 4
Figure 2.2: Service invocation 6
Figure 2.3: SOAP message 7
Figure 2.4: A WSDL document 8
Figure 2.5: An example of the Gnutella network 11
Figure 2.6: Integrating P2P into Web Services 13
Figure 3.1: A centralized directory network 16
Figure 3.2: DHT network 18
Figure 3.3: Service selection 23
Figure 4.1: Using the proxy to manipulate communication 29
Figure 4.2: A working scenario using PWSF 29
Figure 4.3: The Software Structure of PWSF 30
Figure 4.4: Using PWSF as the base of PUDDI and PWSG 31
Figure 4.5: Using PUDDI in the proposed architecture to substitute UDDI 32
Figure 4.6: Using PWSG in the proposed architecture 33
Figure 4.7: Difference between two WSDL documents 35
Figure 4.8: The QoS Based Selection 36
Figure 4.9: The deployment scenario of the proposed architecture 39
Figure 5.1: Two-peer networking topology 42
Figure 5.2: Topology used for examining the PUDDI Publication API 42
Figure 5.3: Four-peer networking topology 43
Figure 5.4: CPU usage of the PUDDI Publication API 45
Figure 5.5: CPU usage of the PUDDI Inquiry API using two-peer topology 45
Figure 5.6: CPU usage of the PUDDI Inquiry API using four-peer topology 46
Figure 5.7: CPU usage of PWSG using two-peer topology 47
Figure 5.8: CPU usage of PWSG using four-peer topology 47
Figure 5.9: PUDDI bandwidth consumption on the consumer side using two-peer topology 48
Figure 5.10: PUDDI bandwidth consumption on the provider side using two-peer topology 48
Figure 5.11: PUDDI bandwidth consumption on the consumer side using four-peer topology 49
Figure 5.12: PUDDI bandwidth consumption on the provider side using four-peer topology 49
Figure 5.13: PWSG bandwidth consumption using two-peer topology 50
Figure 5.14: PWSG bandwidth consumption using four-peer topology 50
Figure 5.15: Throughput of the PUDDI Publication API 51
Figure 5.16: Response time of the PUDDI Publication API 52

Figure 5.17: Throughput of the PUDDI Inquiry API using two-peer topology............................52
Figure 5.18: Response time of the PUDDI Inquiry API using two-peer topology 53
Figure 5.19: Throughput of the PUDDI Inquiry API using four-peer topology 53
Figure 5.20: Response time of the PUDDI Inquiry API using four-peer topology.......................54
Figure 5.21: Response time of PWSG..54
Figure 5.22: Throughput of PWSG 55
Figure 6.1: The simulation system 57
Figure 6.2: Execution of the action 58
Figure 6.3: Decomposing a request process 59
Figure 6.4: Two-peer topology 60
Figure 6.5: Performance comparison using the two-peer topology 61
Figure 6.6: Three-peer topology 61
Figure 6.7: Performance comparison using the three-peer topology 61
Figure 6.8: Four-peer topology 61
Figure 6.9: Performance comparison using the four-peer topology...62
Figure 7.1: Number of reachable peers V.S. TTLs 65
Figure 7.2: Search time duration V.S. TTLs...66
Figure 7.3: Number of packages V.S. TTLs...66
Figure 7.4: Reachable peers V.S. #Neighbors 67
Figure 7.5: Search time V.S. #Neighbors..68
Figure 7.6: #Packages V.S. #Neighbors..68
Figure 8.1: Experimental result of Setting 1...74
Figure 8.2: Experimental result of Setting 2...74
Figure 8.3: Experimental result of Setting 3...75
Figure 8.4: Experimental result of Setting 4...76

LIST OF TABLES

Table 4.1: The parameters of QoS feedback.....37
Table 4.2: The parameters used in runtime context information.....37
Table 4.3: The notation of parameters.....38
Table 5.1: Hardware configuration of experimental machines.....43
Table 5.2: Web Service provided by each peer.....44
Table 7.1: Experimental configuration.....65
Table 7.2: Experimental configuration.....67
Table 8.1: Local-area network setting.....71
Table 8.2: Wide-area network setting.....71
Table 8.3: WAN with mixed servers setting.....72
Table 8.4: Heavy load WAN with mixed servers setting.....72

LIST OF ABBREVIATIONS

CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
DHT	Distributed Hash Table
HTTP	Hypertext Transfer Protocol
IP	Internet Protocol
LAN	Local-area Network
P2P	Peer-to-Peer
PUDDI	Distributed P2P UDDI
PWSF	P2P-Web Services Framework
PWSG	P2P-Web Services Gateway
QoS	Quality of Service
RDF	Resource Description Framework
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
TTL	Time to Live
UDDI	Universal Description, Discovery and Integration
URL	Uniform Resource Locator
WAN	Wide-area Network
WS consumer	Web Service consumer
WS provider	Web Service provider
WSDL	Web Services Description Language
XML	Extensible Markup Language

CHAPTER 1

INTRODUCTION

Web Services [28] are a distributed computing technology that allow applications to interact with each other over networks using Extensible Markup Language (XML, [7]) messaging. They are defined as a set of standards, including XML, Simple Object Access Protocol (SOAP, [27]), Web Services Description Language (WSDL, [11]), and Universal Description, Discovery and Integration (UDDI, [2]). Compared to other approaches, e.g., the Common Object Request Broker Architecture (CORBA, [29]) and the Distributed Component Object Model (DCOM, [25]), Web Services differ with respect to meta-data and platform independence. The service provider offers meta-data (WSDL document) for the potential client to examine its functionality at runtime. This self-describing feature of Web Services enables the automatic generation of support code that ensures a seamless interaction between the provider and the consumer.

Since Web Services use XML messaging, true platform and programming language independence is achieved, which in turn greatly simplifies the deployment of Web Services.

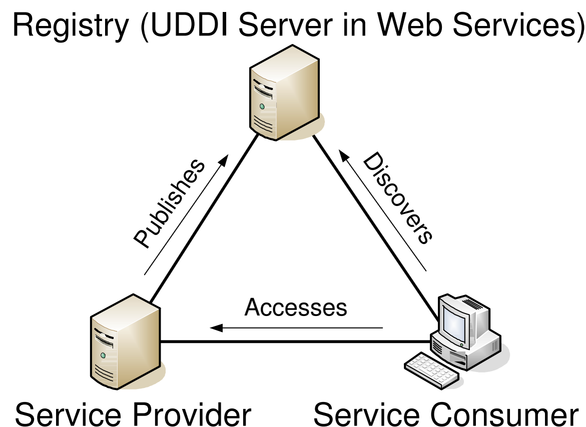

Figure 1.1: Topology of Web Services

Web Services are an implementation of the Service Oriented Architecture (SOA), an "architecture that represents software functionality as discoverable services on the

network” [9]. SOA enables loose coupling [30] among applications. Applications are not tightly bound together and can be developed and deployed separately. The message based interactions and standardized service definition schemas of SOA also give applications significant flexibility. SOA defines three essential components (Figure 1.1), i.e., the central registry, the service provider, and the service consumer.

The central registry provides a “yellow page” service. The service provider publishes its service information and network location in the registry. The service consumer discovers the service information from the registry. It then directly accesses the service of the provider using the discovered information.

The triangle structure of SOA (Figure 1.1) is server-centric and the servers are single points of failure. For the central registry and the service provider, any change in their availability or locations will incur an access failure for the service consumer. This failure can not be avoided in SOA because the service consumer always assumes that server components are available. Moreover, the information in the registry may be outdated due to this weakness of SOA. Therefore, SOA is not suited for dynamic environments in which all participants can enter and depart at any time without notice (e.g. wireless networks).

This thesis focuses on improving Web Services as an implementation of SOA to enable their deployment in dynamic environments. While Web Services support the integration of distributed components in a truly platform independent manner, they suffer from the brittleness of their server-centric design. P2P technologies exhibit remarkable robustness and are message based like Web Services; making a blend of Web Services and P2P an interesting option. If it were possible to integrate the P2P concepts transparently into the already established Web Services architecture, one would expect a more robust architecture enabling a more widespread deployment of Web Services.

This work will focus on evaluating the hypothesis that by transparently integrating P2P concepts (P2P, [26]) into Web Services, it will become possible to deploy them in dynamic environments like wireless networks without sacrificing any interoperability benefits. The remainder of this thesis is organized as follows. Chapter two provides background knowledge and a problem description. This is followed by the literature

review. Chapter four presents an architecture for solving the problems defined in Chapter two. Chapters five to eight give a set of experiments to evaluate the overhead, performance, and improvements of the architecture. Chapter nine presents a conclusion and outlook on future work.

CHAPTER 2

WEB SERVICES AND P2P

Web Services and P2P are two kinds of distributed computing technologies. Web Services adopt a triangle structure to manage all components. To support all functions, The server component must keep its availability and location immutable. P2P eliminates the need for the server component, and performs dynamic discovery to filter out unavailable resources. Integrating P2P into Web Services provides a means to improve system robustness.

2.1 Web Services

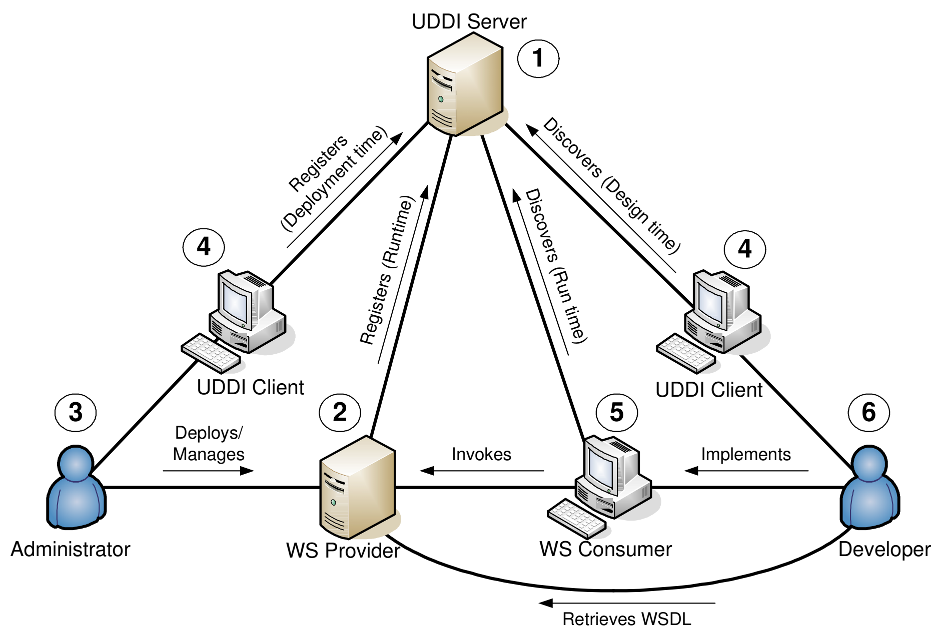

Figure 2.1: The Web Services development/deployment scenario

Figure 2.1 shows a standard Web Services development/deployment scenario, which consists of six elements.

1) The UDDI server, which is a central registry providing the yellow page service.

2) The Web Service provider (WS provider), which hosts the service.

3) The Administrator, who deploys and manages the WS provider.
4) The UDDI client, which provides an easy-to-use interface for users to manipulate data in the UDDI server.
5) The Web Service consumer (WS consumer), which invokes the service of the WS provider.
6) The Developer, who is responsible for developing the WS consumer.

The elements are related as follows:

- The administrator deploys the WS provider on a physical machine and uses a UDDI client to publish the information about the service to the UDDI server. The WS provider can also update its existing service information in the UDDI server after its state is changed.
- Typically, the developer performs a lookup operation through a UDDI client (design-time discovery) to find a desirable service from the UDDI server. After finding a suitable service, the interface definition (WSDL document) can be retrieved from the WS provider.
- The developer implements a WS consumer based on the retrieved interface definition. The WS consumer interacts with the WS provider directly to invoke the service.
- If the WS consumer fails to invoke the service of the provider, it tries to obtain updated information about the provider from the UDDI server again (run-time discovery). The WS consumer can arrange the next request according to the newly fetched information.

SOAP, WSDL, and UDDI provide the core functionality of Web Services and will be explained in the following sections.

2.1.1 SOAP

The most important feature of Web Services is interoperability between heterogeneous applications. Web Service applications can easily publish and invoke services no matter

what programming languages they are written in, and what platforms they are running on due to the XML-based SOAP protocol.

Figure 2.2 shows a service invocation occurring between a WS provider and a WS consumer. The WS consumer sends a SOAP request to the WS provider through an HTTP connection. The provider processes the request and returns a SOAP message as the reply to the consumer. Then, the provider closes the HTTP connection to finalize the invocation.

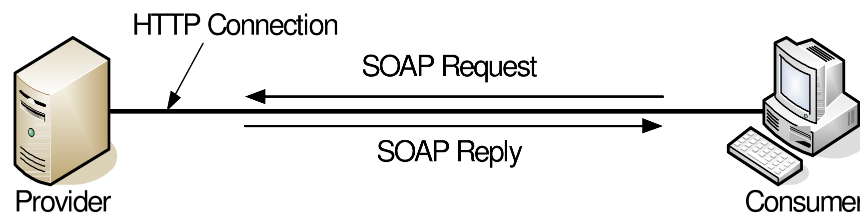

Figure 2.2: Service invocation

Figure 2.3 shows a SOAP request and a SOAP reply used in a service invocation. The SOAP request carries a call to the method "*getResponse*" with an input argument "*Service1 Test*". The SOAP reply carries a response "*Service1 Reply*" to the call. To maximize the compatibility, SOAP hides all language and platform specific information and presents the message in a universal way, so any application can parse the information without any ambiguity. Since SOAP is a textual protocol, it can be enveloped by any other protocol.

2.1.2 WSDL

Self-description is an important feature of Web Services. Most distributed computing environments prior to Web Services, e.g., CORBA and DCOM, do not provide a truly platform independent means to publish the interface definitions of services. In Web Services, WSDL standardizes the service description using XML. A WSDL document, as shown in Figure 2.4, describes the following aspects of a service:

- ***message***: defines the data type of the input/output message used by the service
- ***portType***: defines all operations of the service, each of which contains an input and an output messages

SOAP Request

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
                   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
                   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <i2:getResponse id="ref-1" xmlns:i2="http://schemas.microsoft.com/clr/nsassem/Service1">
      <input id="ref-3">Service1 Test</input>
    </i2:getResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

SOAP Reply

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
                   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                   xmlns:clr="http://schemas.microsoft.com/soap/encoding/clr/1.0"
                   SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <i2:getResponseResponse id="ref-1"
     xmlns:i2="http://schemas.microsoft.com/clr/nsassem/Service1">
      <return id="ref-3">Service1 Reply</return>
    </i2:getResponseResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 2.3: SOAP message

- ***binding***: defines the protocol used to invoke the service
- ***service***: defines the name and the port (see below) of the service
- ***port***: defines the address of the service

After obtaining the WSDL document describing the service, a developer can build a WS consumer immediately because the WSDL document contains sufficient information for service invocation.

2.1.3 UDDI

The UDDI server is the key component connecting WS consumers and WS providers, and is itself a set of Web Services supporting the description and discovery of services on three meta-data levels:

```
<definitions name='WSProvider' targetNamespace='http://…' xmlns='http://…'
   <message name='WSProvider.getResponseInput'>
      <part name='a' type='xsd:int'/>
   </message>
   <message name='WSProvider.getResponseOutput'>
   </message>

   <portType name='WSProviderPortType'>
      <operation name='getResponse' parameterOrder='a'>
         <input name='getResponseRequest' message='tns:WSProvider.getResponseInput'/>
         <output name='getResponseResponse' message='tns:WSProvider.getResponseOutput'/>
      </operation>
   </portType>

   <binding name='WSProviderBinding' type='tns:WSProviderPortType'>
      <soap:binding style='rpc' transport='http://schemas.xmlsoap.org/soap/http'/>
      <operation name='getResponse'>
         <soap:operation soapAction='http://…'/>
         <input name='getResponseRequest'>
            <soap:body use='encoded' encodingStyle='http://…' namespace='http://…'/>
         </input>
         <output name='getResponseResponse'>
            <soap:body use='encoded' encodingStyle='http://…' namespace='http://…'/>
         </output>
      </operation>
   </binding>

   <service name='WSProviderService'>
      <port name='WSProviderPort' binding='tns:WSProviderBinding'>
         <soap:address location='http://192.168.0.1/service1'/>
      </port>
   </service>
</definitions>
```

Figure 2.4: A WSDL document

1) businesses, organizations, and other units, which are represented by the *businessEntity* data structure,

2) service abstracts describing the functionalities of services, which are represented by the *businessService* data structure, and

3) technical information defining the location of the provider, which is represented by the *bindingTemplate* data structure.

The UDDI server organizes all data structures hierarchically. Each data structure has a unique ID representing the business key, the service key, or the binding key.

The latest version of UDDI (V3, [3]) provides six API sets, i.e., Inquiry, Publication, Security, Custody and Ownership Transfer, Subscription, and Replication. The Inquiry

and Publication APIs are essential, as they enable a WS consumer to discover a WS provider.

2.1.3.1 The UDDI Publication API

The UDDI Publication API provides the UDDI client and the WS provider with a method to publish and update information according to the formats of *businessEntity*, *businessService*, and *bindingTemplate*.

The most important methods in the API are *save_business*, *save_service*, and *save_binding*. They all have a parameter carrying a unique ID representing the business key, the service key, or the binding key respectively. According to the value of the ID, the UDDI server performs either a publishing or an updating operation. If the UDDI client invokes one of the three methods with an empty ID, the UDDI server will perform a publishing operation to create a new record in its repository. A new ID will be assigned to the record and returned to the client as the reference of the record. If the UDDI client invokes one of the three methods with a pre-fetched ID, the UDDI server will update the existing record using the ID as the index.

2.1.3.2 The UDDI Inquiry API

The UDDI inquiry API allows the UDDI client and the WS consumer to browse the UDDI server and discover meta-data from it. The most essential methods of the API are *find_business*, *get_businessDetail*, *find_service*, *get_serviceDetail*, *find_binding*, and *get_bindingDetail*. The "find" methods browse or query information in the repository by specifying keywords. The "get" methods obtain certain information by specifying record IDs (index key). The Inquiry API supports three patterns to access the repository, namely, the browse pattern, the drill-down pattern, and the invocation pattern.

- ***The browse pattern***

  The UDDI client uses "find" methods and specified keywords to look for records about businesses, services, and bindings. It obtains information about the provider and the service from retrieved records.

- ***The drill-down pattern***

The client uses "get" methods and a specified record ID to obtain detailed information about a business, service, or binding from the UDDI repository.

- ***The invocation pattern***

If the WS consumer fails to fulfill a service invocation, it should use the *get_bindingDetail* method to fetch fresh *bindingTemplate* information. Then, it can rearrange the service invocation according to the fetched information.

The differences between the drill-down pattern and the invocation pattern are:

- The drill-down pattern is the activity following the browse pattern, while the invocation pattern is an independent activity and only works with the *get_bindingDetail* method.
- The drill-down pattern is used in the design-time discovery by the UDDI client, while the invocation pattern is used in the run-time discovery by the WS consumer.

2.1.4 Drawbacks of Web Services

Web Services still adopt a server-centric approach to organize components. The WS consumer assumes that the states of all server components, i.e., the WS provider and the UDDI server, are immutable, because it has been bound in the design time with a service definition (WSDL) which finally maps to a specific WS provider after deployment. If there is any change in the provider's IP address or service definition, the consumer will fail to perform the service invocation. This assumption prevents Web Services from being used in a dynamic networking environment, in which all participants can autonomously enter and depart at any time without notice. An example is a wireless ad hoc network consisting of a number of mobile devices that communicate with each other through wireless connections. All participants may change their availability or network locations at any time due to changes in their physical positions.

The UDDI server aims to assist the WS consumer in locating server components at run-time. However, it does not solve the problem perfectly because it is also a single

point of failure. Moreover, the information in the UDDI server may be outdated because the UDDI server is unable to probe the change in the provider. If, for instance, a WS provider has crashed due to a hardware error, it will still be listed as a valid service provider by the UDDI server until the provider itself refreshes the information. Consequently, any WS consumer having discovered the provider from the UDDI server will fail to perform the invocation.

2.2 P2P

P2P is a technology that does not suffer the above problem since it eliminates the need for central servers to sustain the whole system. A P2P network only consists of peers that all provide common services and differ only in the resources they own. The peer providing a resource to others is the provider peer, while the peer consuming the resource is the consumer peer. All peers in a P2P network are organized by themselves using a specific protocol, by which they can publish and find resources in a cooperative pattern. Since P2P performs dynamic discovery to filter out unavailable resources, it is very robust in a dynamic networking environment.

The rest of the section presents the Gnutella protocol to explain the working mechanism of a P2P network. Gnutella is also implemented in the proposed architecture (Chapter four) and examined in the experiments (Chapter five and seven) due to its high flexibility and robustness (Chapter three).

2.2.1 Gnutella

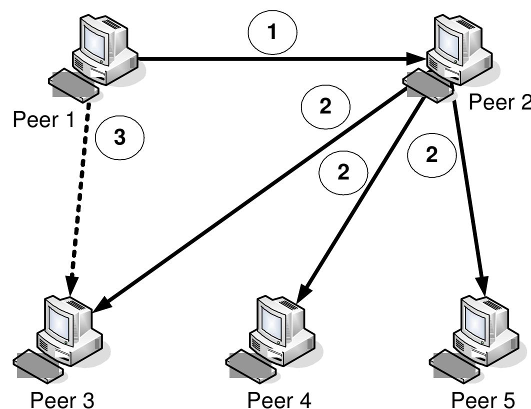

Figure 2.5: An example of the Gnutella network

Each Gnutella peer has four basic operations, Ping, Pong, Query, and QHit. Ping and Pong are used to find existing peers in the network, and Query and QHit are used to find desirable resources.

A Gnutella peer should propagate an incoming request (Ping, Query) to those peers that it has direct connections with (those peers are usually called as its neighbors), and send back the response (Pong, QHit) to the peer issuing the request. Based on Figure 2.5, the working mechanism of Gnutella can be explained in the following four steps.

1. If Peer 1 wants to join a P2P network, it will first connect to a known peer, e.g., Peer 2 in Figure 2.5.
2. After a connection to Peer 2 is established, Peer 1 will send a Ping request to Peer 2 to find other peers. Peer 2 responds with a Pong message to Peer 1. The Ping request is also propagated to Peer 2's neighbors (e.g., Peer 3, 4, 5). All neighbors respond the Ping and send Pong messages back to Peer 1 through Peer 2. At this stage, Peer 1 knows Peer 2 – Peer 5 and vice versa.
3. Since a Gnutella peer always keeps a certain number of active connections (usually ≥ 5) to other peers, Peer 1 will try to establish more connections. For instance, Peer 1 may connect to Peer 3.
4. Peer 1 sends Peer 2 a Query request to find a desirable resource. In addition, Peer 2 propagates the Query request to its neighbors. If a peer has the requested resource, it sends a QHit message back to Peer 1 along the incoming path. In this way, Peer 1 knows all peers owning the requested resource. How Peer 1 accesses the resource of other peers depends on different implementations. For most systems, exchanging resources will use a dedicated connection instead of the one transferring requests.

The Ping-Pong mechanism of Gnutella keeps detecting any change in the peer while the Query-QHit mechanism ensures dynamic lookups.

2.3 Linking P2P and Web Services

Web Services do not work properly in a dynamic networking environment even when a UDDI server is present. By contrast, P2P works effectively in the dynamic networking environment. This raises the question if it is possible to integrate P2P into Web Services to form a hybrid system, which is able to perform service discovery and service invocations?

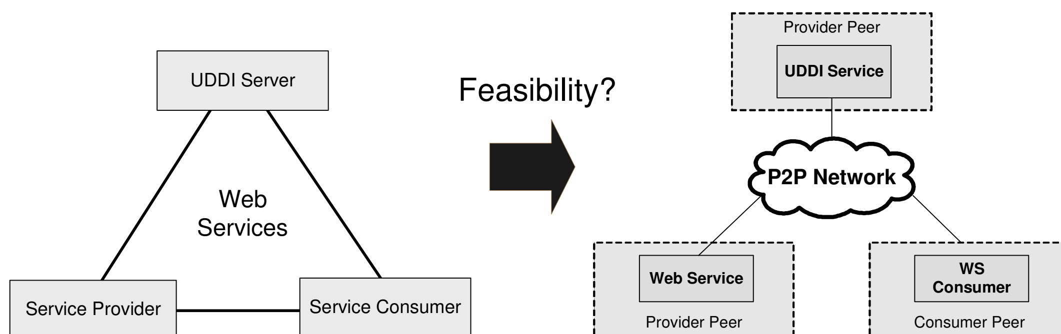

Figure 2.6: Integrating P2P into Web Services

As mentioned in section 2.2, peers in the P2P network have the same functionality. They differ in whether or not they provide resources. To utilize P2P in Web Services, each service of the WS provider must be treated as a service resource of the P2P network. Then, a WS consumer can be transformed into a peer inquiring the service resource in the P2P network (consumer peer), and a WS provider to a peer providing the service resource in the P2P network (provider peer). Figure 2.6 illustrates this transformation on a concept level.

The XML-based SOAP message enables easy manipulation. It can be enveloped by a P2P protocol and then transmitted over a P2P network. Therefore, the provider peer and the consumer peer are able to communicate with each other in the P2P network. This significantly improves the feasibility of integrating P2P into Web Services. Moreover, the provider/consumer peer can be augmented to support more functionality because they are not constrained by Web Services standards.

CHAPTER 3

LITERATURE REVIEW

Integrating P2P into Web Services is a means to improve the dependability and flexibility of Web Services. Some studies in this area have presented methodologies of integration on a concept level. Most studies focus on specific aspects of integration, e.g., distributed discovery and autonomic selection.

3.1 Web Services

By far, most efforts in Web Services focus on improving the functionality of the core standards that mainly include SOAP, WSDL and UDDI. The first version of SOAP [6], v1.1, was released in May of 2000 and was upgraded to the version 1.2 [27] in June of 2003. WSDL was originally released in March of 2001 (version 1.1, [11]). The working draft of its successor, WSDL 2.0 [4], was released in March of 2004, which involves many substantial changes from WSDL 1.1, e.g., supporting interface inheritance. The first version of UDDI, 1.0 [5], was released in June of 2002. Shortly afterwards in July of 2002, the most widely used version, UDDI 2 [2], was published. UDDI 3.0.2 [3] was drafted in October of 2004, adding support for multi-registry environments, digital signatures, and a new subscription API.

These improvements do not solve the inherent questions of Web Services, e.g., poor reliability in the dynamic networking environment and outdated information provided by UDDI. This is because these improvements and additions still use the server-centric structure that potentially causes these questions.

3.1.1 Use of UDDI

IBM, Microsoft, and Oracle all have public UDDI servers running for commercial purposes. There are also many organizations developing third party servers for users to establish their private UDDI servers. jUDDI [16] is such one open source Java

implementation of the UDDI server and uses a relational database as the backend repository.

The UDDI client offers two approaches to access the UDDI server: either through a standalone application providing an easy-to-use interface for developers; or through a software library working with the WS consumer or provider. UDDI Browser [43] is an open-source UDDI client following the first approach. A developer can use the application to browse, search, and even change information in the UDDI server. UDDI4J [42] is an open-source Java library providing a set of APIs for the WS consumer/provider to interact with UDDI. Hagge [13] presents in detail how a WS consumer discovers and invokes Web Services at run time using UDDI4J.

3.2 P2P

P2P networks and corresponding protocols emerged at the end of 1990s as a direct consequence of improved bandwidth, connectivity, and available system resources. At that time, P2P networks were used in instant messaging systems and scientific research systems. After Napster [41] first introduced the technology in its file sharing system, P2P networks have managed to establish themselves as an independent track of distributed computing. Milojicic et al. [26] divides P2P networks into three types according to the discovery/communication model: the centralized directory model, the flooded requests model, and the document routing model (usually called the distributed hash table model). The latter two have been gaining most attention from researchers due to their better flexibility and dependability than the first one.

3.2.1 The Centralized Directory Model

This is the simplest P2P model. A central node works as the registry (as shown in Figure 3.1). All peers register their addresses and resources to the registry. By searching the registry, a peer can find other peers that have the desired resource. In this model, the central registry is still a single point of failure and introduces performance bottlenecks in the system.

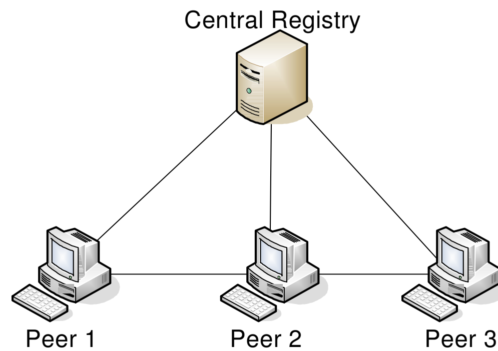

Figure 3.1: A centralized directory network

3.2.2 The Flooded Requests Model

Gnutella is the best-known example of this model (described in subsection 2.2.1). According to its propagation mechanism, peers generate many redundant messages which are transferred over the whole network. This is the reason that the model is called "flooded requests". Time to Live (TTL) is an approach used to reduce the number of redundant messages transferred over the P2P network. Every message has a numeric parameter called TTL indicating how many hops the message can be routed. When a message reaches a peer, the TTL is decremented. If TTL equals 0, the message is expired and will not be routed anymore.

Ritter [34] showed a close relationship among reachable peers, TTL, and the number of neighbors. He found that increasing TTL and the number of neighbors (marked as N) would also increase the number of reachable peers geometrically. In a large-scale network, if the values of TTL and N are not big enough, a peer can only access a small part of the network. However, high TTL and N incur high bandwidth consumption. He presented the bandwidth consumption numerically. As mentioned in his report, to propagate an 83 byte Query package in a 10885 peer network (TTL = 5, N = 7), a total of 1.8 megabytes of data will be transferred over the network. Therefore, to minimize the negative impact, TTL and N should be carefully chosen according to the scale of the network.

Lv et al. [20] investigated the characteristics of this model using four overlay topologies: Power-Law Random Graph, Normal Random Graph, Gnutella Graph, and

Two-Dimensional Grid. They found that high connectivity of Power-Law Random Graph and Gnutella Graph results in a high redundant message ratio and high load on the peer. The Normal Random Graph was found most suitable for the flooded requests model. They also proposed two approaches to improve the efficiency of discovery, i.e., *expanding ring* and *random walks*. Using *expanding ring*, a peer first discovers a resource using a small TTL. If discovery fails, it will increase TTL and perform new discovery. Therefore, the number of redundant requests can be reduced for the hot resource (the popular resource) because the hot resource is widely replicated and easily reached. Using *random walks*, a peer propagates a request to a randomly chosen neighbor. This approach reduces the number of redundant messages significantly but increases the delay of discovery. Lv et al. suggest that the requesting peer sends a request to more than one neighbor to reduce the delay.

Chawathe et al. [10] present three more approaches to improve the scalability of Gnutella besides the *random walks*. *Dynamic topology adaptation* enables low degree (low capacity) peers to connect with a high degree (high capacity) peer closely. It utilizes processing abilities of high degree peers. *Active flow control* throttles requests according to the capacity of a peer to avoid overloading it. *One-hop replication* enables a peer to keep the service information of its neighbors. The experiment shows that using these four approaches together to modify the Gnutella protocol achieves three to five orders of magnitude improvement in the capacity of the system.

Although the flooded requests model suffers from high bandwidth consumption, it is still preferred in research for three reasons. First, its algorithm is simple and eliminates the need for the central server. Second, it is very robust in dynamic environments even when peers are transient. Finally, it is flexible enough to support different search algorithms.

3.2.3 The Distributed Hash Table Model (DHT)

In the DHT network, every peer has a unique peer ID and is organized by a ring shape link table (as shown in Figure 3.2). The peer's position in the link table is determined by a hash function and its peer ID. Each peer knows several other peers in the link table to

build its routing table. The peers in the routing table are chosen from the link table by different step lengths. For example, Pastry [35] uses 2^{bn} to determine step lengths. Where, the peer ID uses base 2^b, and n stands for the step number. If L stands for the location of the current peer in the link table, the first chosen peer is located at $L + 2^b$, and the second one is at $L + 2^{2b}$.

Each resource, such as the shared file, is also given a unique ID by a specific algorithm. The peer whose peer ID is closest to the resource ID must own the resource or know the location of the resource. When a peer issues an inquiry for a resource, it should first route the inquiry based on the routing table to the peer whose peer ID is closest to the resource ID. Every peer receiving the inquiry will repeat the process until the peer whose ID is globally closest to the resource ID is reached. The last found peer is supposed to own or know the resource.

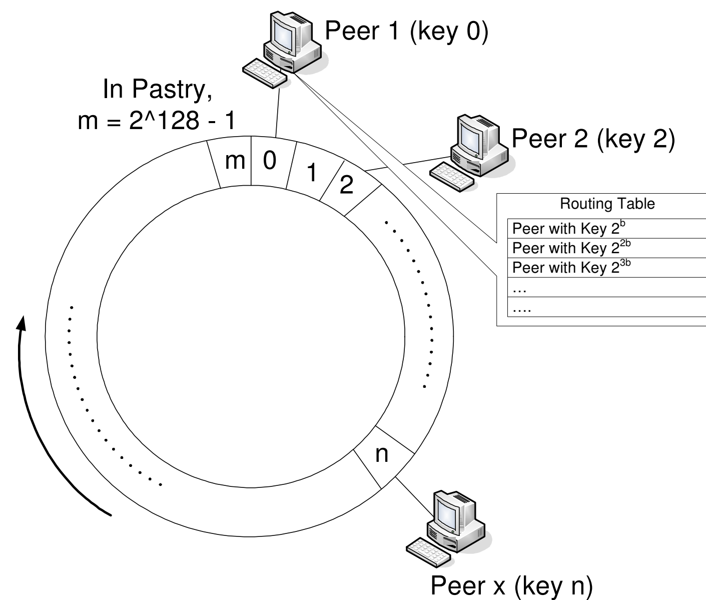

Figure 3.2: DHT network

Handling the arrival and departure of peers in the DHT network varies from one protocol to another. Pastry, for example, handles the arrival and departure of peers in the following way:

Arrival: A new peer X must know at least one peer (peer A) in the Pastry network before it joins the network. Then, X issues a “join” message with its peer ID to A. Each peer that receives the “join” message should perform three operations.

- The peer routes the “join” message based on the routing table to the peer A_0 whose peer ID is numerically closest to the peer ID of X.
- The peer returns the information about A_0 to X to build the routing table of X.
- If the peer ID of X is closer to the peer than any peer is in the routing table, the peer updates the routing table using X.

X has joined the network if the peer that receives the “join” message is already the closest one and cannot route the message anymore. At this time, other peers know X and X has its routing table built.

Departure: A peer may depart without notification. To avoid any interruption of routing messages, the routing table keeps the information about those peers whose IDs have the same prefix. If the peer finds that a peer X is unavailable, it turns to inquire those peers with the same ID prefix as X for an alternate peer. The alternate peer should be the closest one to X in all returned peers. Then, the routing table is updated.

Chawathe et al. [10] talk about weaknesses of DHT compared to Gnutella. DHT networks require more actions and time to process the arrival and departure of the peer. Frequent status changes in peers will impair the dependability and performance of the system. It is difficult to perform keyword discovery in DHT networks because the resource ID must be known before the discovery. Gnutella is better at discovering replicas than DHT.

3.3 Integrating P2P and Web Services

Schneider [38] discusses the convergence of P2P and Web Services. He compares the JXTA [15] protocol with Web Services in several aspects, namely the conceptual architecture, the wire protocols (the connecting mechanism), security, discovery, reliability, and business standards. He concludes that P2P and Web Services adopt different discovery mechanisms, i.e. decentralized vs. centralized, but they are able to utilize WSDL and SOAP. The convergence of P2P and Web Services is a way to increase efficiency and decrease cost.

Samtani et al [36] identify several potential problems in integrating P2P and Web Services into commercial applications, namely network bandwidth, security, and complex architectures and maintenance. The main contribution in the paper is that it identifies three approaches for integrating P2P and Web Services:

- Using Web Services protocols as the basic protocol in the P2P network;
- Using the P2P technology to transform the centralized UDDI registry into the decentralized mode;
- Using XML as business processes in P2P;

However, concern about the first and third problems may not be necessary. Using a properly selected P2P protocol and specific algorithms, e.g., *random walks*, will effectively reduce the bandwidth consumption to an acceptable level. On the other hand, since P2P is designed to work in a kind of environment in which the network node may not be stable, it requires less maintenance than other systems when confronted with malfunction.

3.4 Service Discovery

UDDI is a standard component in Web Services to provide a discovery service. It supports multiple UDDI servers through the replication API (UDDI Version 2 and 3 specifications) to improve the reliability. Each UDDI server keeps the same set of data in the repository to avoid a single point of failure. However, using the replication API to organize multiple servers involves a high cost in deployment and maintenance. Decentralizing UDDI using the P2P technology does not suffer from the same problem.

An implementation of a distributed UDDI is the system PETERPAN proposed by Laoveerakul et al. [17]. PETERPAN uses the Gnutella protocol and acts as middleware connecting GRID and Web Services.

Papazoglou et al. [32] propose a UDDI-enabled super-peer registry concept to group service providers according to their offerings and subscriptions. A group of service providers is a service syndication, in which any provider either relies on a service provided by another provider, or provides a service to another provider in the same group.

Each service syndication has a super-peer providing publication and subscription services. All super-peers are attached to the central UDDI server and communicate in a centralized directory P2P manner. The provider in the service syndication appears as a service-peer that communicates with other peers in the syndication in a Gnutella manner. If a new provider needs to join a proper syndication, it first publishes the services it provides, and subscribes to the services it requires (as a precondition) on a known super-peer. Then, this super-peer will try to find a destination super-peer that satisfies either of the following two criteria; 1) in the syndication managed by the destination super-peer, there is a provider having the service required by the new provider; 2) in the syndication managed by the destination super-peer, there is a provider requiring the service provided by the new provider. If this destination super-peer can be found, the new provider will be registered to the super-peer and added to the corresponding syndication. The approach in this paper enables service providers to group autonomously according to their service offerings and requirements.

Castro et al [8] propose building a universal ring in a DHT P2P network to support service advertisement, service discovery, and code binding. These functions are based on three operations, namely the persistent store, the application-level multicast, and the distributed search. Every exposed service (a piece of code) has a code certificate to identify the correctness of the service (code binding). Since there may be many Web Services with the same functionality and name globally, the code certificate will help the user find the correct service to invoke. It is essential for finding the correct service that the information used to generate the service key for discovery should be the same as the one used to generate the service key for advertisement.

Banaei-Kashani et al. [1] suggest a P2P service discovery method using the Gnutella protocol and semantic technologies. There is no central registry in the system. Every service provider or consumer is a peer. The service provider publishes its service locally. The service consumer originates a query based on keywords or ontology, and propagates it in a Gnutella manner. Although this method is intuitive and simple, it can cause high network consumption and lower the possibility of finding a service.

METEOR-S Web Services Discovery Infrastructure (MWSDI, [44]) is a scalable publication and discovery environment involving semantic Web Services and the P2P topology. In the MWSDI environment, there are many standard UDDI registries running in different Operator peers respectively. Every Operator peer exposes Operator Services to other peers. Operation Services are a set of services based on standard UDDI specification and enhanced by semantic technologies. The communication between peers relies on the P2P protocol. The user who needs to publish or discover services should call the Operator Services through a client peer, which is a transient interpreter between the user and MWSDI. To build the relationship between the Web Service and the ontology, MWSDI uses WSDL and predefined tModels (the metadata structures in UDDI describing the common concept and understanding) to present taxonomies of inputs and outputs.

Toma et al. [40] propose a P2P discovery environment based on the Web Services Execution Environment (WSMX, [14]), a test bed supporting Semantic Web Services. They choose a flooding request P2P protocol, HyperCuP [37], as the networking technology. Each WSMX peer has a registry which keeps the information of the local service. All peers are organized by the HyperCuP protocol. The Service Requestor sends out a request for service to a known WSMX peer. The WSMX peer searches in the local registry for the service and sends the result back to the requestor if it has the service. Otherwise, it sends the request to other peers according to the HyperCuP protocol. The most interesting aspect in the paper is that the HyperCuP protocol reduces the bandwidth consumption by building ontology concept clusters. The request is routed to the corresponding clusters according to its domain ontology concepts.

3.5 Service Selection

When dynamic discovery is used in Web Services, it is common that the result of the discovery contains more than one provider. Unlike the file sharing P2P system in which a file download can be split into many small tasks running in multiple peers, a service invocation occurs between a provider and a consumer. As shown in Figure 3.3, the WS consumer must pick only one from all candidate providers to perform the invocation. Even for a composite Web Service consisting of many atomic Web Services, the

selection issue still needs to be addressed when there are multiple providers available for an atomic service.

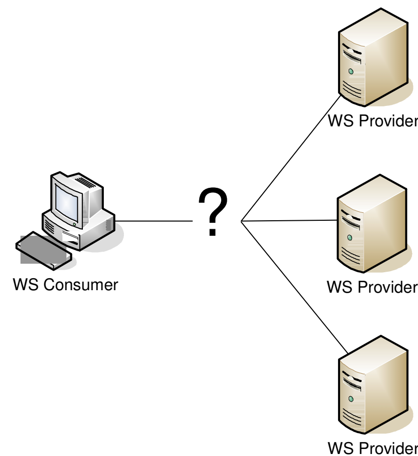

Figure 3.3: Service selection

Padovitz et al. [31] present three models to collect information dynamically in a mobile/wireless environment. The collected information is the base for selecting the best provider. The first is the RPC based model in which the WS consumer sends out an inquiry to all direct providers to collect information. If a provider is dependent on other providers to perform its service, it will send the inquiry to the dependent provider. This process repeats until the final provider or the depth constraint for the inquiry is reached. This model works like a contract-net [39] in which the further decision is delegated to the sub-contractor. The second is the mobile agents based model, in which the mobile agent works as the information collector moving from one provider to another provider. Once the complete series of service providers has been browsed by the mobile agent, it will go back to the consumer. Security is a major concern when applying this approach. The third is the circulating mobile agent model in which the mobile agent itself moves from one provider to another continuously to collect necessary information. When a cycle is done, the mobile agent has visited all providers and goes back to the consumer to report all collected information. In addition to the security issue, this model may suffer from waste cycles when the consumer does not need any information in the idle state.

QoS is a means to enable selection and filter out unqualified providers. Vu et al. [45] propose P2P-based Web Services discovery that uses semantics to find WSDL and QoS data to select a provider. In the QoS part, the registry, which resides in the peer, keeps

Web Service publications from providers and QoS feedback from consumers. To avoid cheating on the feedback from the consumer, the system employs trusted agents to monitor both service providers and consumers. They monitor the service and create QoS reports in the registry. Based on the QoS reports, they evaluate the feedback from consumers. This approach may cause three potential issues; 1) continuous probing of the service consumes processing resources and network bandwidth of the target machine; 2) agents may not be accurate or correct in evaluating QoS reports from consumers because they use different criteria to measure a service; 3) as mentioned in the paper, those agents are usually resource consuming. The complex configuration and maintenance counteract the benefit of adopting P2P.

Some QoS-based selection methods merely use the experiences of each individual service consumer. When a service consumer has had no experiences with service providers, it cannot make a good choice. If service consumers can share their experiences, they can build up the reputation of service providers and find desired services. Maximilien et al. [22, 23, 24] propose an agent-based approach where each Web Service is associated with an agent responsible for collecting feedbacks (e.g. service response time, availability, price) from the consumers of the service. When a service consumer wants to know whether this web service is good or not, it can access the reputation-related information from the agent, calculate the reputation of the service provider, and make a decision taking its own QoS preference into account. For instance, some consumers prefer a service with short response time, while some consumers may care more about the reliability of a service. Unfortunately, Maximilien et al. do not provide details about how to compute the reputation.

In Web Services, a consumer performs discovery only in the UDDI server. UDDI does not provide QoS information to the consumer. To address this issue, Ran [33] proposes a model for Web Services discovery. The model introduces a new certifier role into the conventional triangle structure of Web Services. A service provider first sends its QoS claim to the certifier to get the claim verified. Then it registers its service and certified claim to the UDDI server. A service consumer discovers a service with specific QoS constraints from the UDDI server. The returned result from UDDI contains services satisfying QoS constraints. To present the QoS information, Ran extends the UDDI data

structure by attaching *qualityInformation* to *businessService*. He introduces categories used in *qualityInformation*, which can be used in the proposed architecture in this thesis to define QoS. The weakness of the model is that all existing components in Web Services must be re-implemented.

Using generic criteria to measure QoS of Web Services is not sufficient in some domains. Liu et al. [19] address the problem by using an extensible QoS model. They divide quality criteria into generic ones and business related ones. Then the QoS computation is to apply two phases of normalization on the QoS criteria matrix. The proposed approach in the paper is uniform. It is able to process an unlimited number of criteria and allow setting parameters to bias the selection.

Day et al. [12] have discussed related work regarding service selection. They argue that there are two ways to get the information necessary for service selection: provider-side and consumer-side augmentations. The provider-side augmentation allows the service provider to describe the guarantee about its service. The service consumer may select a suitable provider according to the description. However, if the description of the service is not consistent with its performance, the selection will be wrong. By contrast, the consumer-side augmentation, which allows the service consumer to record the experience about each service invocation, has more advantages. Each experience record describes QoS in three aspects: whether or not the service is available, whether or not the expected result is returned, and the roundtrip time. The consumer treats the service provider as a black box and assesses it according to those history experiences.

They proposed two approaches, namely the rule-based and the naive Bayesian reasoners, to fulfill the autonomous service selection using the client side augmentation. Specially, they chose Resource Description Framework (RDF, [21]), which provides knowledge management using XML based metadata models, as the underlying repository. The rule-based reasoner takes the QoS experiences as the input and outputs a series of values representing candidate services respectively. The service with the highest value will be the best. The naive Bayesian reasoner takes more attributes as the input besides each experience. These extra attributes, e.g. processor load, total memory used, and the number of processes, represent the runtime context of the consumer and may influence

the manner of the service invocation. The output is one of five classes the service belongs to, namely excellent, good, acceptable, poor, and terrible. The reasoner always chooses the service in the highest available class as the best one. If there is more than one service in the highest available class, it chooses randomly.

Those attributes mentioned by Day et al. to represent the runtime context of a consumer influence a service invocation locally, but have nothing to do with the invocation issued by any other consumer in the system. They will not help other consumers choose the most suitable providers.

3.6 Conclusions

Web Services will benefit from the P2P technology in service discovery and service invocations. Using P2P to distribute the UDDI service began to gain attention early. The distributed UDDI service provides higher reliability than the standard one, and is compatible with existing Web Service components. Banaei-Kashani et al. [1] present a feasible and simple design in which multiple peers cooperate to provide the UDDI service.

Using P2P in service invocations brings more functionality to the system. When the service consumer requests a service, an agent peer will receive the request and perform the request in a P2P manner. This process may involve a service discovery operation (dynamic discovery), and a selection operation when there are multiple services available. Many studies have been done in this area.

QoS is the most studied criterion used in the selection operation. Due to a lack of QoS support in Web Services, researchers have proposed the use of agents [30, 44, 12] to collect QoS information. They tend to use consumer-side augmentations that treat the provider as a black box. The only information collected from a provider is its static QoS statement. However, the runtime context of a provider will influence QoS significantly. Treating the service provider as a black box prevents the consumer from assessing the provider's characteristics globally. The consumer may be misled due to the common experiences disregarding the current context of the provider. Using a provider-side agent, it is possible to represent the runtime context of the provider.

Although these studies provide valuable theories and techniques that can be used to solve the problem of improving the reliability of Web Services in a dynamic environment through P2P technologies, they only focus on a part of the whole problem, either service discovery or service selection. They do not manage the system as a whole. Moreover, the approaches above disregard the life cycle of Web Service applications (e.g., development and deployment) and require extra modifications in Web Service applications.

The research presented in this thesis focuses on the development of a framework that will support the transparent integration of P2P concepts into the Web Services. The framework is fully compatible with existing Web Services applications. It provides Web Service applications life cycle support.

CHAPTER 4

P2P-WEB SERVICES ARCHITECTURE

The P2P-Web Services architecture enables Web Services to work effectively in a dynamic networking environment. It consists of two functional components, the distributed P2P UDDI (PUDDI) and the P2P-Web Services Gateway (PWSG).

PUDDI is a substitute for UDDI in the proposed architecture to overcome the drawbacks of UDDI. It provides all functions listed in the UDDI specification V3 [3]. The UDDI client, the WS provider, and the WS consumer interact with PUDDI to fulfill UDDI operations in the proposed architecture. From their perspectives, there is no difference between PUDDI and UDDI. However, PUDDI has no central registry and works in a distributed manner.

The service invocation between the WS provider and the WS consumer is performed via PWSG. In the proposed architecture, PWSG plays the role of the WS consumer for the WS provider, and the WS provider for the WS consumer. PWSG controls every aspect of the service invocation to take advantage of P2P.

PUDDI and PWSG both play two roles: a Web Services component and a P2P peer. They are Web Services components when interacting with other Web Services components, and they are P2P peers when transferring messages over the P2P network. This similarity motivates the idea of the P2P-Web Services framework (PWSF) to reuse the same structure/functionality and minimize the effort to build PUDDI and PWSG.

This chapter is outlined as following. Section 4.1, 4.2, and 4.3 describe PWSF, PUDDI, and PWSG respectively. Section 4.4 presents the way to use both PUDDI and PWSG. Section 4.5 details the functions used by PWSG to improve service invocations. Finally, the deployment of the architecture is presented in Section 4.6.

4.1 The P2P-Web Services Framework (PWSF)

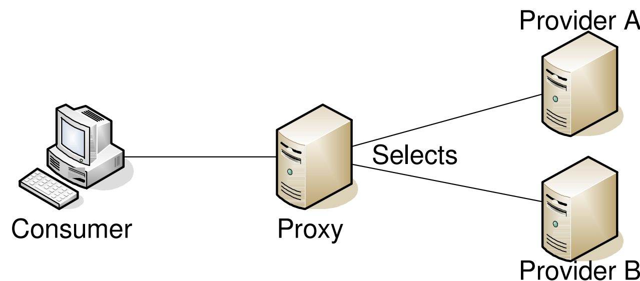

Figure 4.1: Using the proxy to manipulate communication

In Web Services, the SOAP message is the base of the inter-application communication. Its textual format provides an opportunity to manipulate the communication by using a proxy that intercepts SOAP messages passing through it. Figure 4.1 shows the concept of using a transparent proxy between a WS consumer and a WS provider, in which the proxy plays two roles. From the perspective of the consumer, the proxy is the service provider. From the perspective of the provider, the proxy is the consumer. Since the consumer sends the request to and receives the response from the proxy, the proxy can fully control the communication between the consumer and the provider. In Figure 4.1, the proxy can “select” a service provider dynamically without notifying the consumer.

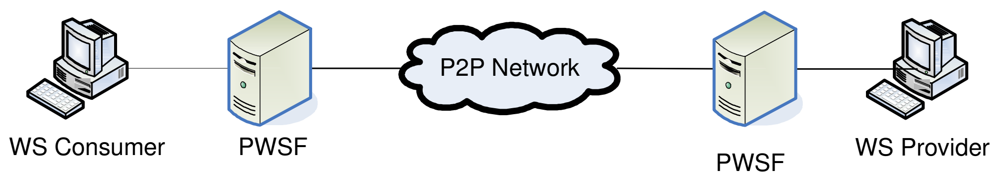

Figure 4.2: A working scenario using PWSF

PWSF is a transparent proxy intercepting the SOAP message. Figure 4.2 shows a scenario, in which PWSF plays the role of the P2P peer besides the WS provider and the WS consumer. When interacting with the P2P network, a PWSF node appears as a P2P peer and manages to build and maintain the P2P network with other PWSF nodes cooperatively. Therefore, PWSF is more like a gateway joining two networks together.

PWSF supports the Plug-in technology, by which a developer builds a software module complying with the plug-in interface and can easily plug the module into the framework to support additional functions. As shown in Figure 4.3, PWSF consists of

three layers, the proxy layer, the control layer, and the networking layer. Adjoining layers exchange information via two unidirectional message queues. Each layer consists of two isolated functional components: the framework component and the plug-in. The plug-in contains the specific logical functionality that determines how each layer should behave in a given situation. It is invoked by the framework component when a message arrives or a predefined timer is expired. Then, the plug-in performs consequent actions via the interface provided by the framework component. Next, the functionality of each layer will be described briefly.

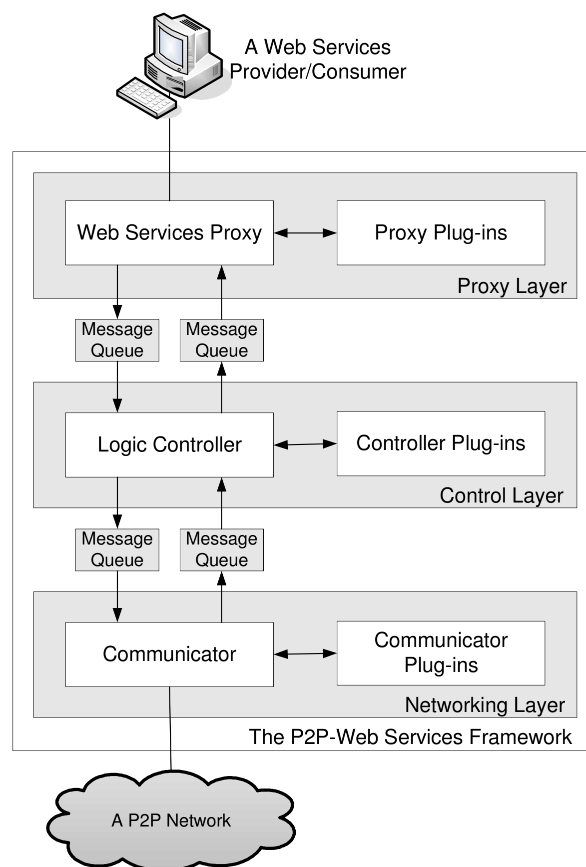

Figure 4.3: The Software Structure of PWSF

- ***Proxy Layer***

The Web Services proxy has HTTP connections with WS providers or consumers and exchanges SOAP messages with them. The plug-in at this layer manipulates the SOAP message from both the WS provider/consumer and the lower layer.

- ***Control Layer***

The logic controller determines how a service request should be fulfilled over the P2P network according to characteristics of different P2P models. Any operation in this layer does not refer to details of a specific P2P protocol. The plug-in is the key element to control the behavior. Those advanced functions, such as caching and selection, which will improve the performance of the service invocation, can be implemented in this layer. Caching enables PWSF to retrieve the state it has experienced immediately. Selection helps the WS consumer find the most suitable provider and filter out ineligible providers.

- ***Networking Layer***

This layer manages building and maintaining the underlying P2P network. The communicator implements all basic networking operations, such as building connections and sending/receiving packages. The plug-in determines how the communication between peers will be performed according to a specific P2P protocol. By switching plug-ins supporting different P2P algorithms, the framework is able to support different P2P protocols.

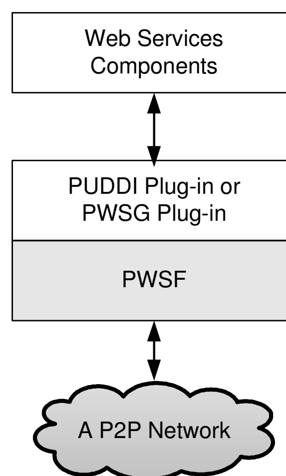

Figure 4.4: Using PWSF as the base of PUDDI and PWSG

Using PWSF as the base of PUDDI and PWSG (Figure 4.4) maximizes reusability and minimizes the effort of development. It helps developers modularize the application for easy maintenance.

4.2 The Distributed P2P UDDI (PUDDI)

A UDDI server is a centralized node in Web Services and introduces a single point of failure in the system. The information about services in UDDI may be outdated because UDDI is unable to probe the availability of the provider. The proposal of PUDDI addresses these issues in a P2P manner.

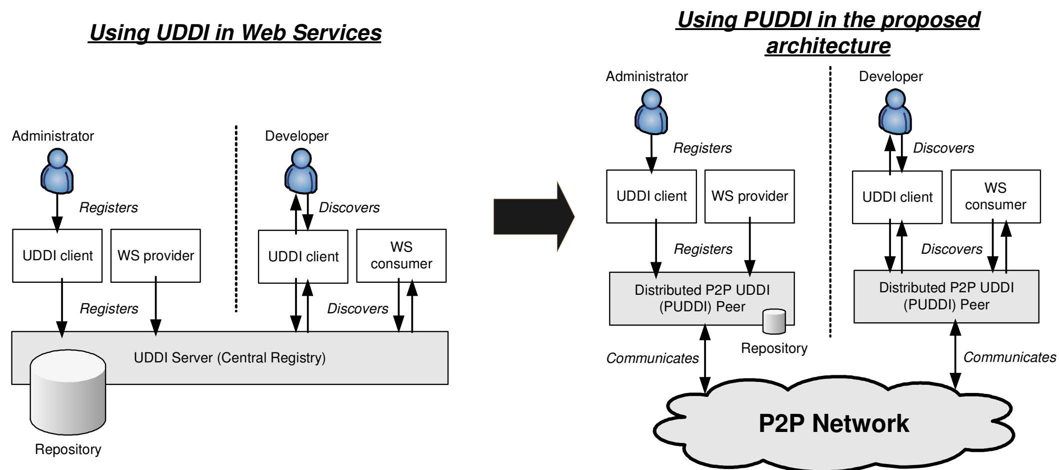

Figure 4.5: Using PUDDI in the proposed architecture to substitute UDDI

Figure 4.5 shows using PUDDI to substitute for UDDI in the proposed architecture. In Web Services, the administrator, the UDDI client, or the WS provider registers the service information to the central UDDI server. The developer, the UDDI client, or the WS consumer discovers the service information from a centralized UDDI server. In contrast, PUDDI is a decentralized system consisting of only PUDDI peers and has no central server. All peers are organized through the P2P protocol. On the provider side, the PUDDI peer accepts the register request and stores the information in the local repository. On the consumer side, the PUDDI peer accepts discovery requests and inquires other peers over the underlying P2P network. Since the PUDDI peer provides the same functionality as the UDDI server, this transformation is transparent to the UDDI client, the WS provider, and the WS consumer. They will not sense any difference between a PUDDI peer and a UDDI server.

On the provider side, the PUDDI peer should be deployed with the WS provider in the same physical machine so that its availability is consistent with that of the machine. It

probes the provider at a certain interval to determine the availability of the provider. So, there is no unavailable provider in the result of the discovery operation returned from PUDDI. When the IP address of a machine is changed, the PUDDI peer will update the corresponding binding information in the repository automatically. The administrator does not need to update the binding information explicitly.

The UDDI client and the WS consumer discover service information through the PUDDI peer installed in the same machine. A discovery request will be fulfilled over the underlying P2P network. For example, if the Gnutella protocol is applied, the discovery request will be transformed to the Query request. When a PUDDI peer receives the Query request, it begins to search the local repository. Then, it responds with a QHit message if the record in the repository matches the search keyword. PUDDI guarantees that the result of discovery is up to date.

4.3 The P2P-Web Services Gateway (PWSG)

In Web Services, a WS consumer tends to request a service from an appointed provider. When the appointed provider is not available, the service invocation fails.

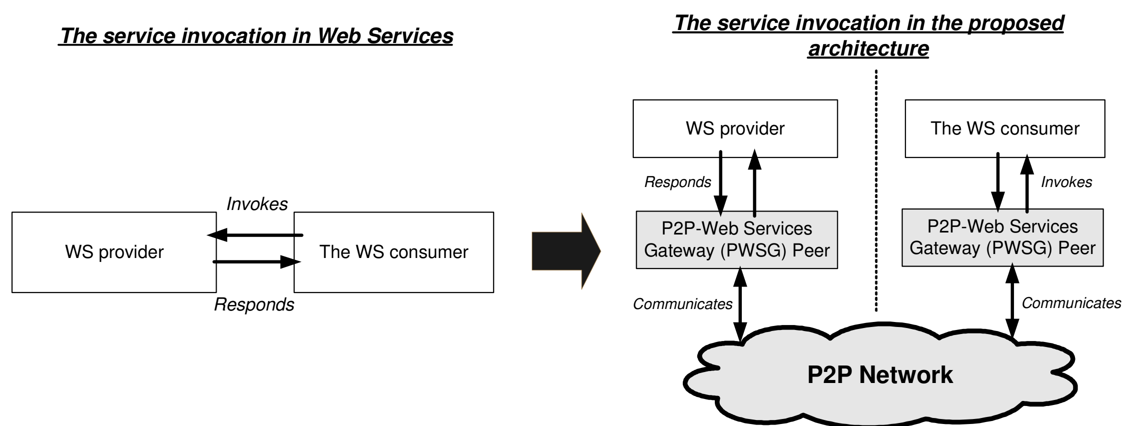

Figure 4.6: Using PWSG in the proposed architecture

PWSG manages the communication between the WS provider and the WS consumer to improve the quality of the service invocation, and yet avoid any re-designing and re-programming in existing applications. Figure 4.6 shows the transformation from the service invocation in Web Services to the one in the proposed architecture. Each WS

provider/consumer in the proposed architecture interacts with a PWSG peer installed on the same machine.

Once joining the system, the PWSG peer installed with the WS provider fetches the WSDL document from the WS provider. It stores the WSDL document in its repository for the discovery purpose.

The WS consumer treats the local PWSG peer as the WS provider and requests the service of the latter. The local PWSG peer performs the service invocation in two phases.

In the first phase, the PWSG peer (the issuer) parses the invocation request (the SOAP request) and extracts three key elements to represent the requested service. These three key elements are:

- The method name to be invoked,
- All parameters and their types, and
- The return type.

Then, the issuer sends out a discovery request with these key elements over the P2P network to find WS providers. Once a PWSG peer installed with the WS provider receives the discovery request, it will respond to the issuer if the local WSDL document fits those elements in the request.

In the second phase, the issuer receives one or more responses from the P2P network. It selects the best one from all responding PWSG peers using the QoS based selection (Section 4.5). Then, it builds a connection to the selected peer and sends it the service invocation request. The selected PWSG peer invokes the service of the attached provider according to the request and sends the result back to the issuer. Finally, the issuer returns the result to the consumer and finishes the service invocation.

4.4 Usage of PUDDI and PWSG

Although PWSG aims to fulfill any WS invocation and supports runtime discovery, PUDDI is still needed at the beginning of development as the substitute for UDDI. The *bindingTemplate* returned from the local PUDDI peer is different from that returned from the UDDI server. The location of the provider is replaced by the address to access the

local PWSG peer plus an encoded string to identify the provider. So, the developer and the WS consumer will treat the PWSG peer as the WS provider. PWSG can be easily introduced into the life cycle of the Web Service application. To illustrate the process, if the WS provider is assumed to be at:

http://192.168.0.1:1080/service1

Then, the transformed location is:

http://localhost:9000/192-168-0-1&1080&service1

Where, http://localhost:9000 is the location to access the local PWSG peer. 192-168-0-1&1080&service1 is an encoded location to access the provider. PWSG will use the encoded location to fetch the WSDL document from the provider.

To obtain the WSDL document, the developer may use the Uniform Resource Locator (URL) http://localhost:9000/192-168-0-1&1080&service1?wsdl to access the local PWSG peer. Since the location of the provider is indicated in the URL, the PWSG peer can easily fetch the WSDL document from that PWSG peer installed with the actual WS provider. Before returning the WSDL document to the developer, the PWSG peer replaces the location of the WS provider with its location (shown in Figure 4.7). This guarantees that the WS consumer developed based on the changed WSDL document will use the local PWSG peer as the provider.

```
---- WSDL returned from the WS provider ----
. . . . . .
<service name='WSProviderService'>
   <port name='WSProviderPort' binding='tns:WSProviderBinding'>
      <soap:address location='http://192.168.0.1/service1'/>
   </port>
</service>

---- WSDL returned from the local PWSG peer ----
. . . . . .
<service name='WSProviderService'>
   <port name='WSProviderPort' binding='tns:WSProviderBinding'>
      <soap:address location='http://localhost:9000/service1'/>
   </port>
</service>
```

Figure 4.7: Difference between two WSDL documents

4.5 The QoS Based Selection

After performing service discovery in the first phase of the service invocation, PWSG may find multiple service providers available and must select the most suitable. The QoS based selection used in the architecture aims to selection a provider on behalf of the consumer. It performs three operations, namely storing, collecting, and reasoning (Figure 4.8).

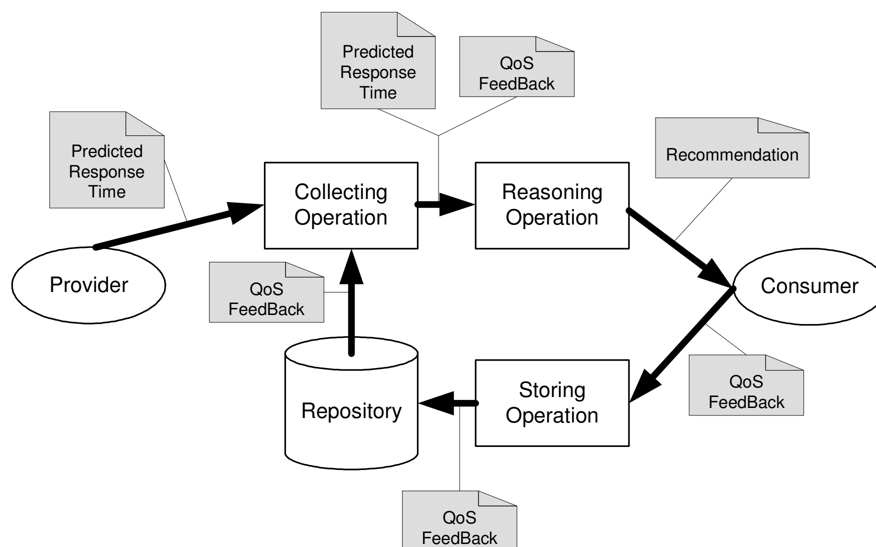

Figure 4.8: The QoS Based Selection

- Through the storing operation, a QoS feedback report from the consumer can be saved to the repository of the provider. The QoS feedback report provides a historical reference for the consumer to assess the provider. Each provider only keeps the feedback information relevant to it.
- The collecting operation retrieves all necessary data from providers for the reasoning operation.
- The reasoning operation manages to select the best service provider for the consumer according to the collected data.

In the first phase of the service invocation, the PWSG peer collects two types of information from candidate providers using the collecting operation, the QoS feedback report and the runtime context of the provider.

A QoS feedback report consists of three parameters shown in Table 4.1, in which the computation of *Predicted response time* will be explained in detail later in this section.

Table 4.1: The parameters of QoS feedback

Parameter	***Type***	***Description***
Response time	int (milliseconds)	The time that the consumer has waited to get the response from the provider.
Predicted response time	int (milliseconds)	The predicted response time according to the runtime context of the provider
Time difference	int (milliseconds)	The time difference between the response time and the predicted response time. $Time\ difference = Response\ time - Predicted\ response\ time$

The runtime context information is generated by the service provider to help assess the provider more accurately. It has two parameters shown in Table 4.2.

Table 4.2: The parameters used in runtime context information

Parameter	***Type***	***Description***
Request Rate	float (requests/second)	The number of requests arriving at the provider in one second
Mean service time	int (milliseconds)	The time consumed in processing a request in the service provider

Using the open model solution technique ([18], Section 6.4.1), the response time can be given by the following equation:

$$R(\lambda) = D[1 + A(\lambda)]$$

Where, $R(\lambda)$ is the response time of a request at a give request rate λ, D is the service time that the request needs to be executed in the processor, $A(\lambda)$ is the number of requests waiting in the queue in the front of the current request.

Since each incoming request/outgoing reply goes through the PWSG peer on the provider side, $A(\lambda)$ and $R(\lambda)$ can be observed at any time by the PWSG peer. Therefore, the mean service time can be obtained by the following equation:

$$D = \frac{R(\lambda)}{1 + A(\lambda)}$$

The computation of the mean service time only needs to be executed once and then is in effect for the whole running period of the PWSG peer.

According to the QoS feedback and runtime context information, the PWSG peer can infer the best provider to invoke the requested service. To describe the reasoning operation more clearly, the notation of the parameters is presented in Table 4.3.

Table 4.3: The notation of parameters

λ_i	*Arrival rate* of the i-th provider pair
S_i	*Mean service time* of the i-th provider peer
D_i^n	*Mean time difference* reported by the n-th consumer peer regarding the i-th provider peer
D_i	*Mean time difference* reported by the current consumer peer regarding the i-th provider peer
N_i	The number of QoS feedback reports regarding the i-th provider peer
T_i	The predicted response time of the i-th provider peer
PRT_i	The predicted response time (including a network lag) for the i-th provider peer
A_i	The rank of the i-th provider peer

Using the open model solution technique ([18], Section 6.4.1) and Little's Law ([18], Section 3.3), the response time can be given by:

$$R(\lambda) = \frac{D}{1 - \lambda D}$$

Where, $R(\lambda)$ is the response time of a request at a give request rate λ, D is the service time. Then, the response time T_i in the QoS selection can be defined as:

$$T_i = \frac{S_i}{1 - \lambda_i S_i}$$

Where, T_i does not take other lags (e.g. the network and the firewall lags) into account. QoS feedback contains the information about the mean external lag time.

If the current consumer peer has its own QoS feedback data on the provider peer (previous experiences), PRT_i can be defined as

$$PRT_i = T_i + D_i$$

If the current consumer peer does not have its own QoS feedback data, PRT_i can be defined as:

$$PRT_i = T_i + \frac{1}{N_i} \sum_{n=1}^{N_i} D_i^n$$

Then, the rank of the i-th provider peer, i.e. A_i, can be defined as:

$$A_i = {}^{1}\!/_{PRT_i}$$

Eventually, the provider peer with the highest value of A_i will be selected by the QoS based selection.

4.6 Deployment

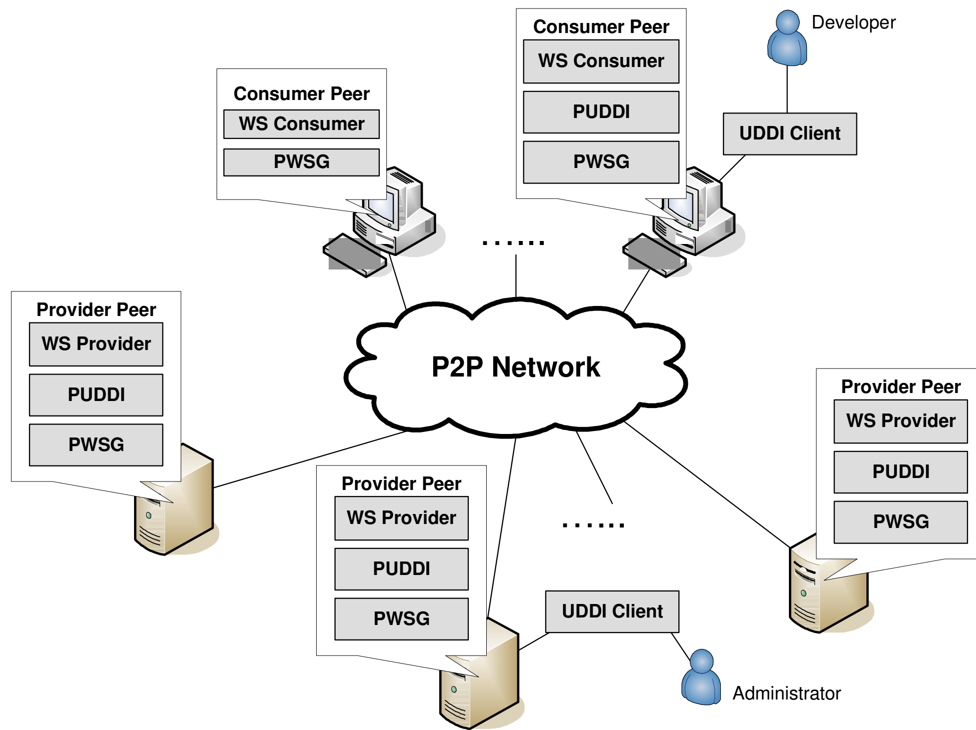

Figure 4.9: The deployment scenario of the proposed architecture

At the deployment stage, each WS provider/consumer must have a PWSG peer (Figure 4.9) installed on the same local machine. They will only interact with the local PWSG peer in the future. To do so, the WS consumer needs to switch its preset provider address

to the local PWSG peer. If the WS consumer is developed based on the WSDL document obtained from the proposed architecture, there is no change required for the consumer. For the WS provider, its address should be set in the configuration file of the PWSG peer. Then, the PWSG peer will send any request to it.

The setup of PUDDI is similar. The WS consumer/UDDI client needs to switch its preset UDDI address to the local PUDDI peer. On the provider side, the WS provider/UDDI client also needs to switch its preset provider address to the local PUDDI peer.

Once the system is deployed, the WS provider, the WS consumer, and the UDDI client work seamlessly with the architecture and are not aware of the organization of the P2P network in their whole life cycle.

CHAPTER 5

EXPERIMENT 1: OVERHEAD AND PERFORMANCE OF PUDDI AND PWSG

The P2P-Web Services architecture uses PUDDI and PWSG to perform Web Service discovery and invocations in a dynamic networking environment. The P2P characteristics (e.g., request propagation and Ping-Pong mechanism) of these two components introduce overheads to the system. According to the Gnutella protocol, the more peers the system has, the more time the discovery operation takes. It can be inferred that the performance of PUDDI/PWSG is related to the number of peers in the system. The experiments in this chapter aim to investigate how the proposed architecture performs in different sizes of networks in the following aspects.

- ***CPU and network bandwidth consumption***

  These help determine quantitatively how much the architecture uses computational resources, and whether or not it is a burden to the deployed environment.

- ***Throughput and response time***

  These help determine the processing capacity and speed of the architecture in a given hardware/software environment.

This part of experimentation will be conducted in a reference system, which is a working implementation of the architecture and consists of prototype peers running on multiple machines. The scalability of the reference system is limited to the available hardware resources.

5.1 Experimental Setup

The reference system is set up using two kinds of networking topologies, a two-peer topology (Figure 5.1) and a four-peer topology (Figure 5.3). PUDDI and PWSG are

examined using both topologies. Each consumer/provider peer in both networking topologies is a physical machine that has a PUDDI/PWSG peer running on it. Each PUDDI/PWSG peer has full connectivity which means it has a direct connection with all other PUDDI/PWSG peers. The topology shown in Figure 5.2 is simplified from the two-peer topology and has only one PUDDI peer. It is dedicated to examining the PUDDI Publication API because this experiment only involves operations between a UDDI client and a PUDDI peer.

The service requestor, the UDDI client for PUDDI or the WS consumer for PWSG, runs on a dedicated machine to generate requests at various request rates (from 1req/sec up to 80req/sec). It logs replies from the attached PUDDI/PWSG peer, and generates a performance report according to its log.

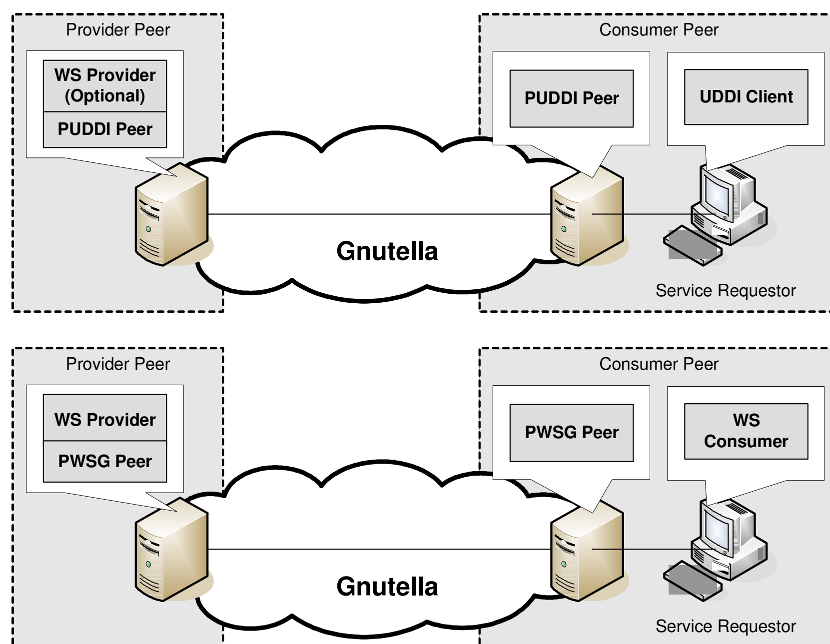

Figure 5.1: Two-peer networking topology

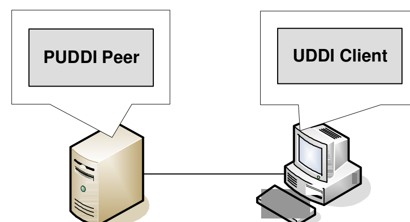

Figure 5.2: Topology used for examining the PUDDI Publication API

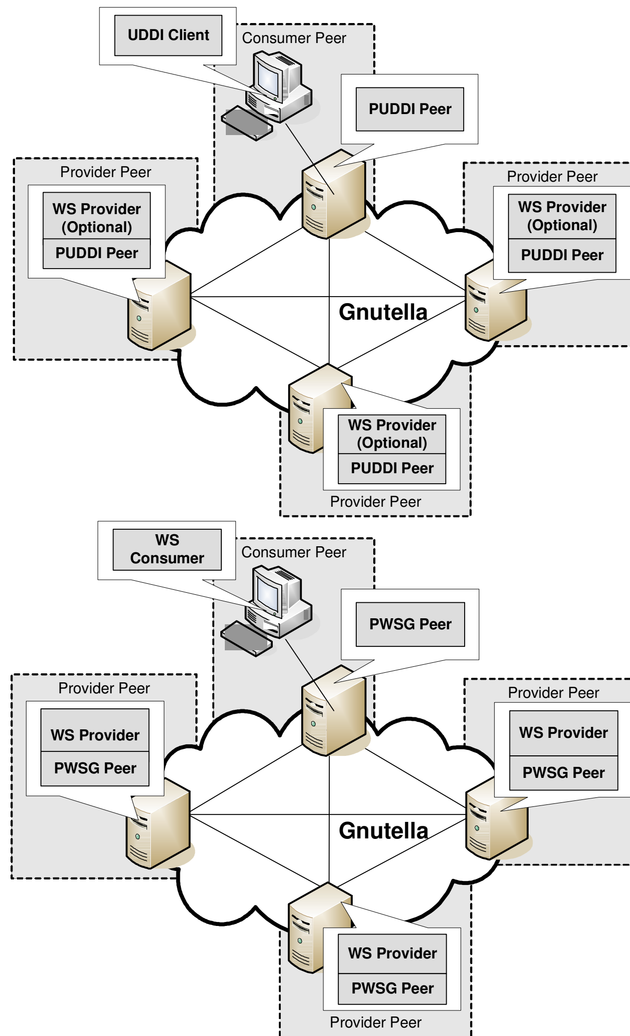

Figure 5.3: Four-peer networking topology

Table 5.1 summarizes the hardware configuration adopted in experiments.

Table 5.1: Hardware configuration of experimental machines

<table>
<tr><th>Role</th><th>CPU</th><th>Memory</th><th>Network Interface</th></tr>
<tr><td>Service Requestor</td><td>3.2G Pentium 4</td><td>2G</td><td rowspan="3">100Mb Ethernet</td></tr>
<tr><td>Service Provider</td><td colspan="2">Each service provider runs on the same machine with the attached PUDDI/PWSG peer</td></tr>
<tr><td>PUDDI/PWSG Peer</td><td>600Mhz Pentium III</td><td>512M</td></tr>
</table>

The Web Service provided by each WS Provider in Figure 5.1, 5.2, and 5.3 only implements one light-weight “echo” method that takes an integer as the input and returns

the same integer. Since the experiments in this chapter aim to investigate PUDDI/PWSG, the low resource consumption of this service provider enables it to be deployed with PUDDI/PWSG on the same machine to simplify the setup of all experiments. The provider's impact on the system performance is minimal and can be ignored.

The PUDDI peer provides the UDDI client standard UDDI services, which are, in the current implementation of PUDDI, the UDDI Inquiry and Publication APIs.

Table 5.2 summarizes all Web Services used in experiments.

Table 5.2: Web Service provided by each peer

	PUDDI	***PWSG***
Consumer Peer	The PUDDI peer provides the UDDI Inquiry API (including *FindBusiness*, *FindService*, and *FindBinding*), which is invoked by the UDDI client	PWSG peer on the WS consumer side does not provide any Web Service
Provider Peer	The PUDDI peer provides the UDDI Publication API (including *SaveBusiness*, *SaveService*, and *SaveBinding*), which is invoked by the WS provider (or the UDDI client)	Each WS Provider provides a light weight echo service. To simplify the experiment, all WS providers provide the same service (identical WSDL definition).

In the experiments, all workloads are generated by consumer peers, specifically, the UDDI client and the WS consumer, at various rates. The UDDI client issues publication requests to the PUDDI peer to examine the PUDDI Publication API. It issues inquiry requests to the PUDDI peer to examine the PUDDI Inquiry API. The WS consumer issues service requests to the PWSG peer to examine the service invocation.

5.2 CPU Usage of PUDDI

Measuring CPU usage of PUDDI is process based, which presents the percentage of time that the PUDDI process occupies the CPU in the whole running period. The higher the CPU usage, the less available the CPU is for other processes. It is useful for users to determine whether or not a PUDDI peer and other Web Services applications can be deployed together on the same physical machine.

Figure 5.4 shows the CPU usage of the PUDDI Publication API, in which three essential functions, i.e., *SaveBusiness*, *SaveService*, and *SaveBinding*, are measured.

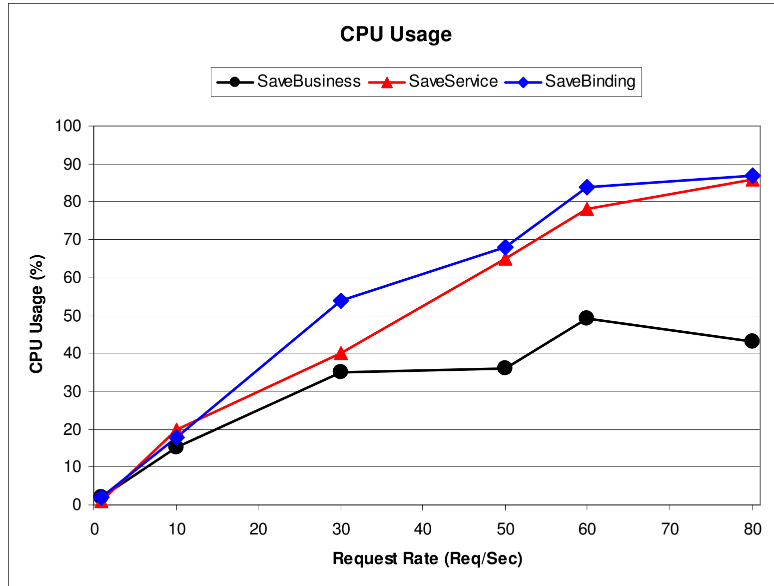

Figure 5.4: CPU usage of the PUDDI Publication API

The *SaveBusiness* operation consumes less CPU than the other two operations to process a request, because its internal data manipulation is simpler than other two.

Figure 5.5 and 5.6 present the CPU usage of the PUDDI Inquiry API using the two-peer topology and the four-peer topology respectively.

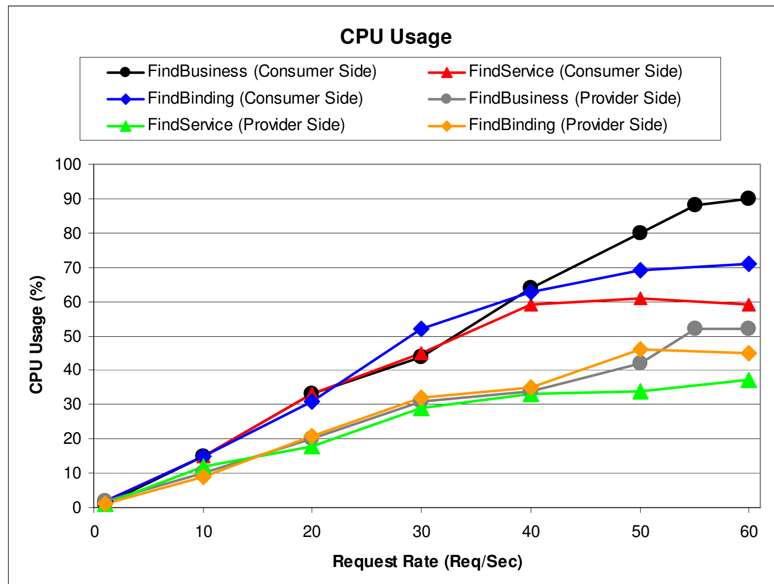

Figure 5.5: CPU usage of the PUDDI Inquiry API using two-peer topology

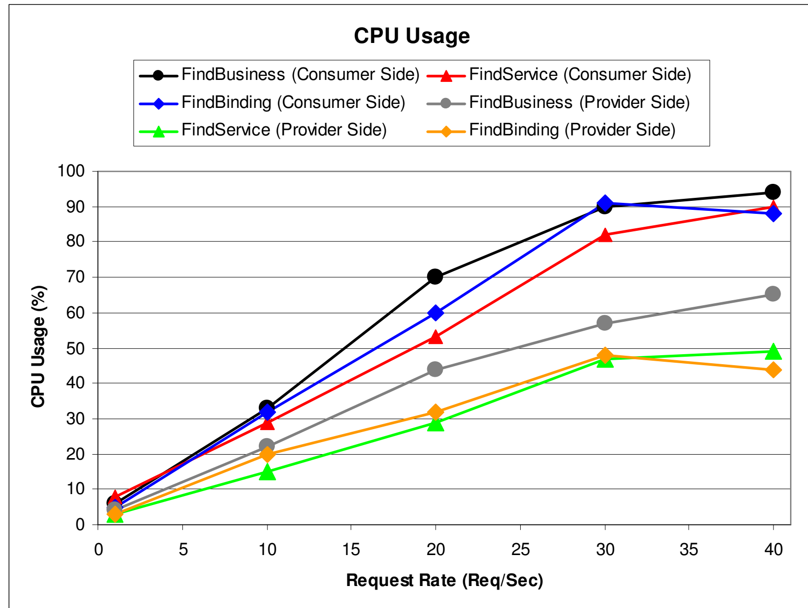

Figure 5.6: CPU usage of the PUDDI Inquiry API using four-peer topology

In Figure 5.5, three operations on the consumer side show large differences at high request rates (> 50req/sec). These differences are mainly caused by the complexity of their data manipulation. On the provider side, these differences are less distinct. Given a request rate, PUDDI in the four-peer topology consumes more CPU resources than the two-peer topology. This is because the four-peer PUDDI system has to process more packages than the two-peer PUDDI system. The increase in the number of packages in the four-peer system will be examined in Section 5.4.

Comparing CPU usage of the consumer peer and the provider peer (Figure 5.5 and Figure 5.6) shows a significant difference. The consumer peer has a higher CPU usage than the provider peer. This is because, for each request, the PUDDI peer on the consumer side has to interact with both other PUDDI peers and the UDDI client, while the PUDDI peer on the provider side only interacts with other peers.

5.3 CPU Usage of PWSG

Measuring the CPU usage of PWSG is also process based. It is as useful as measuring CPU usage of PUDDI. Figure 5.7 presents the CPU usage of PWSG in the two-peer network, while Figure 5.8 presents the CPU usage in the four-peer network.

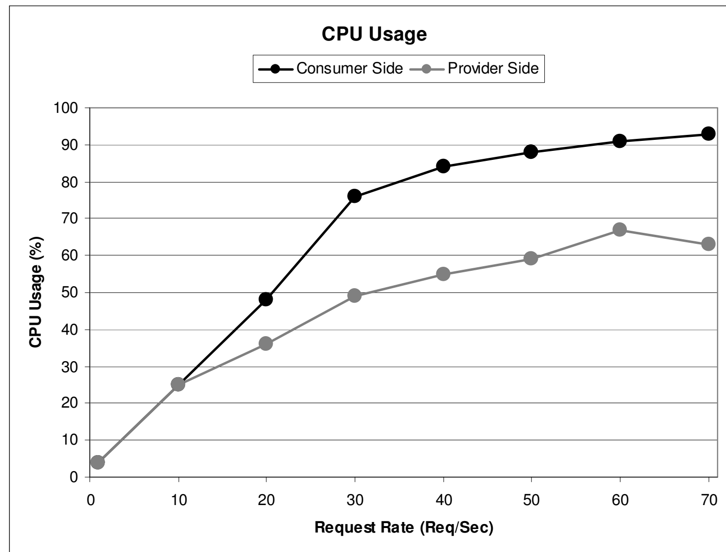

Figure 5.7: CPU usage of PWSG using two-peer topology

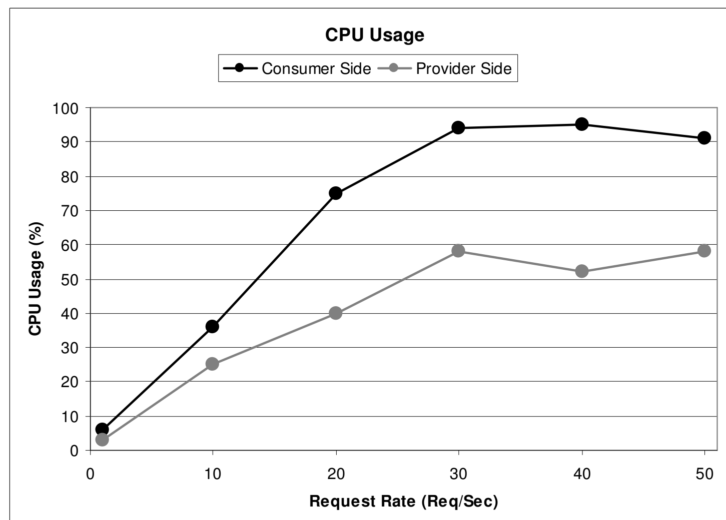

Figure 5.8: CPU usage of PWSG using four-peer topology

In both topologies, the consumer peer has a higher CPU usage than the provider peer because the consumer peer has to process more data packages. At the same request rate, peers in the four-peer topology have higher CPU usage than those in the two-peer topology because the four-peer network has more data packages transferred.

In Figure 5.8, there is a drop in the CPU usage on the provider side when the request rate is 40req/sec. This may be caused by the interference from the garbage collection feature of the development platform (Microsoft C# and .Net framework) and the virtue memory management of OS (Windows XP).

5.4 Bandwidth Consumption of PUDDI

The main drawback of the Gnutella protocol is its high bandwidth consumption. Since PUDDI and PWSG use the Gnutella protocol to manage all peers, it is necessary to examine their bandwidth consumption to determine in what magnitude they consume the bandwidth.

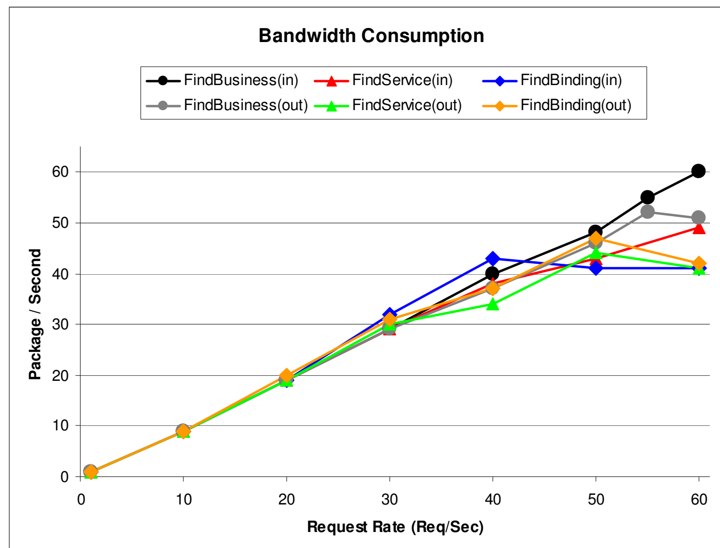

Figure 5.9: PUDDI bandwidth consumption on the consumer side using two-peer topology

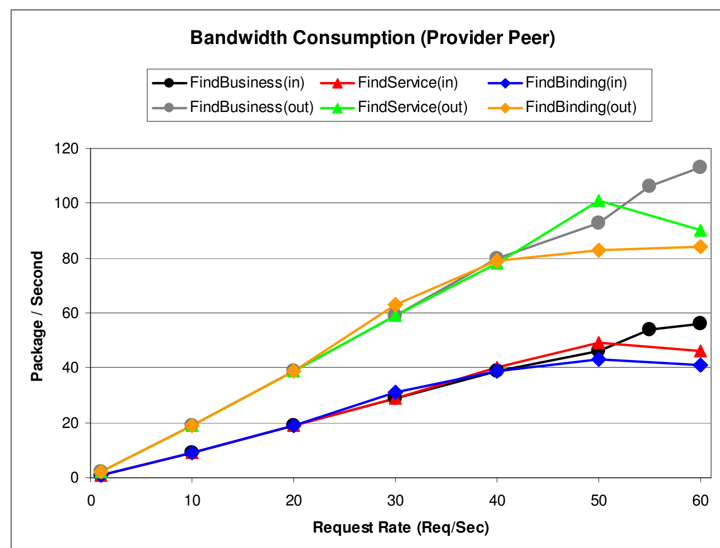

Figure 5.10: PUDDI bandwidth consumption on the provider side using two-peer topology

Figure 5.9 and 5.11 present the PUDDI bandwidth consumption on the consumer side, while Figure 5.10 and 5.12 present the bandwidth consumption on the provider side.

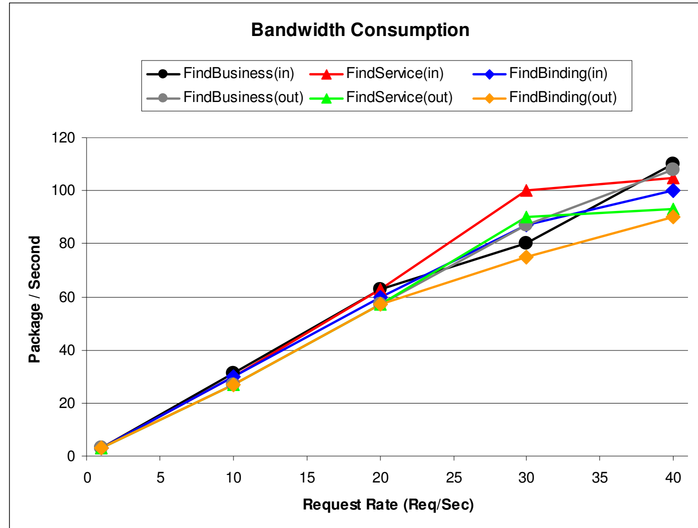

Figure 5.11: PUDDI bandwidth consumption on the consumer side using four-peer topology

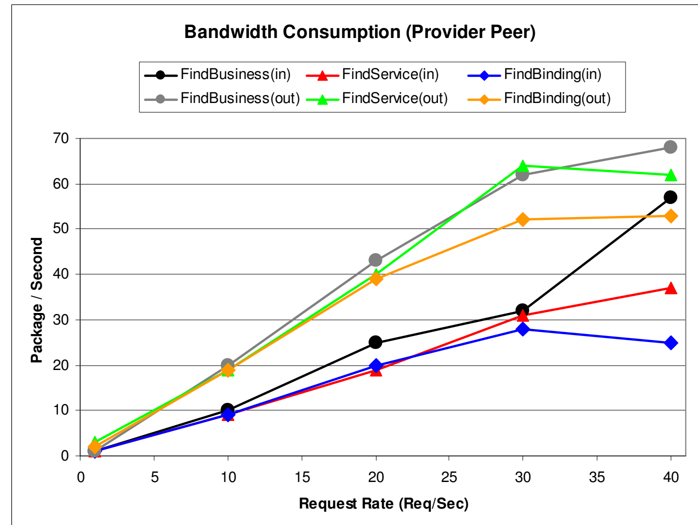

Figure 5.12: PUDDI bandwidth consumption on the provider side using four-peer topology

In the two-peer network, the number of incoming packages/outgoing packages on the consumer side equals the number of requests because there is no redundant package transferred. On the provider side, the number of outgoing packages is two times higher than that of incoming packages because the provider peer also propagates each incoming request to its neighbor, the only consumer peer in the network.

Figure 5.11 shows that the increase in the number of incoming/outgoing packages on the consumer side is proportional to the increase in the number of peers. The higher the number of peers, the higher the bandwidth consumption is. On the provider side (Figure 5.12), the number of incoming packages equals the request rate because any redundant package will be deleted by the peer. The number of outgoing packages is proportional to the number of peers because the peer propagates incoming requests to its neighbors.

5.5 Bandwidth consumption of PWSG

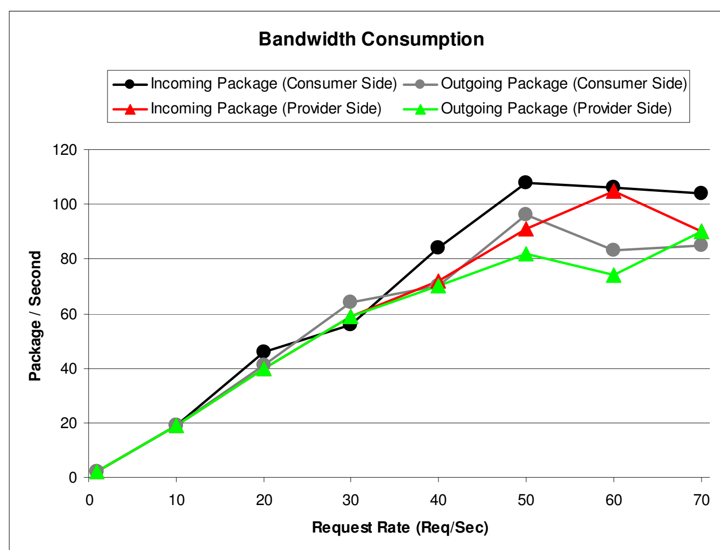

Figure 5.13: PWSG bandwidth consumption using two-peer topology

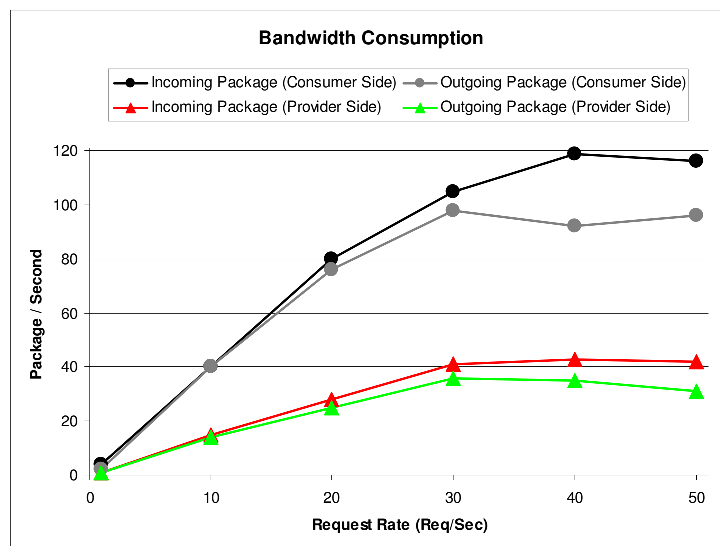

Figure 5.14: PWSG bandwidth consumption using four-peer topology

In the two-peer topology (Figure 5.13), both the consumer peer and the provider peer use the same amount of bandwidth. In the four-peer topology (Figure 5.14), the consumer peer requires much higher bandwidth than the provider peer because it interacts with all providers to process each request. The provider peer in the four-peer network consumes slightly less bandwidth than it does in the two-peer network, because, in the four-peer network, all invocation requests are balanced among three provider peers. In the two-peer network, all invocations are processed by the only one provider.

5.6 Throughput and Response Time of PUDDI

This section and the following section will examine the performance of PUDDI and PWSG in terms of throughput and response time. Throughput represents the number of processed requests by the system per second at a given request rate. The maximal capacity of the system is represented by the maximal throughput. Response time is the time the system consumes to process a request at a given request rate.

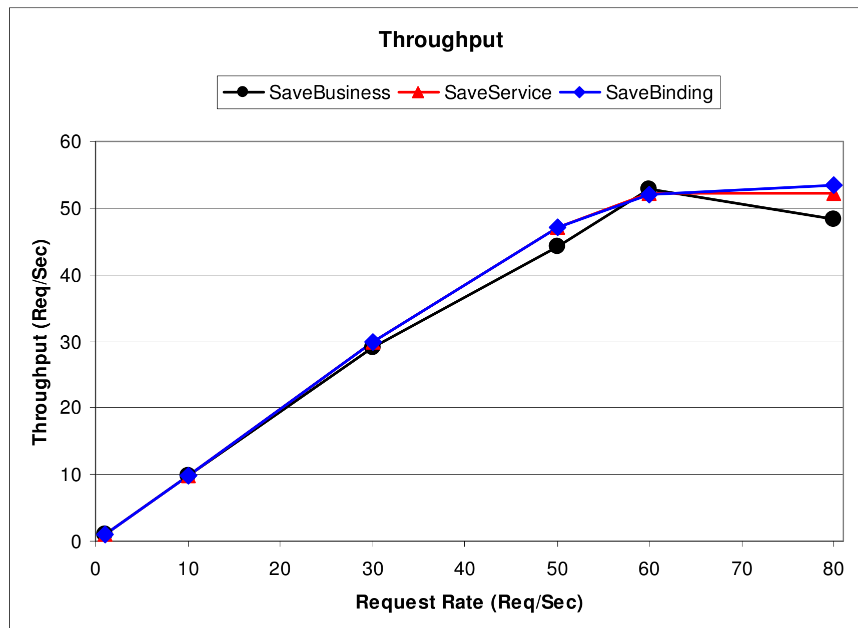

Figure 5.15: Throughput of the PUDDI Publication API

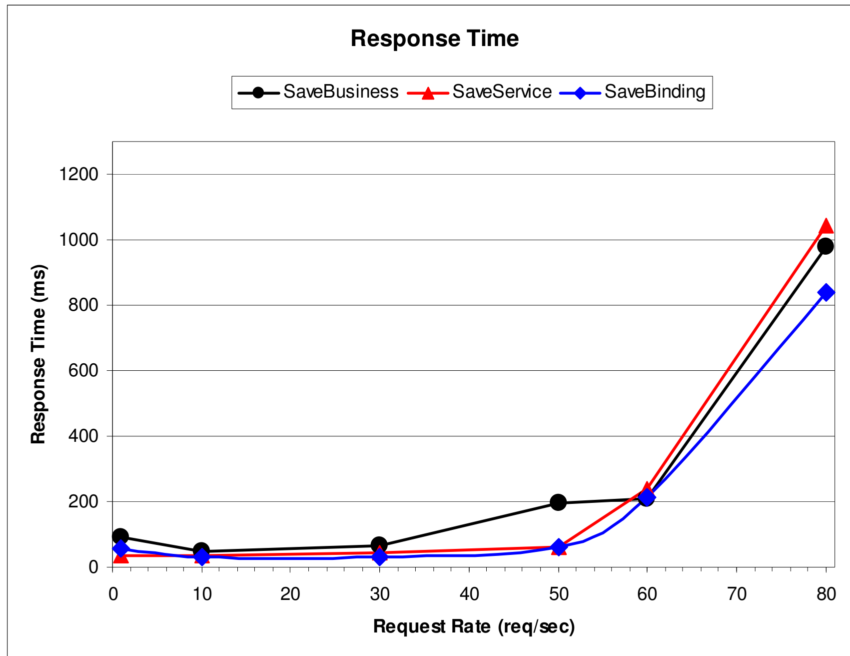

Figure 5.16: Response time of the PUDDI Publication API

The methods measured in the experiment have the same throughput and response time curves. They reach their maximal throughput at the request rate 60req/sec. When the request rate is greater than 60req/sec, the performance deteriorates rapidly. The response time goes up fast and the throughput stops going up and even declines.

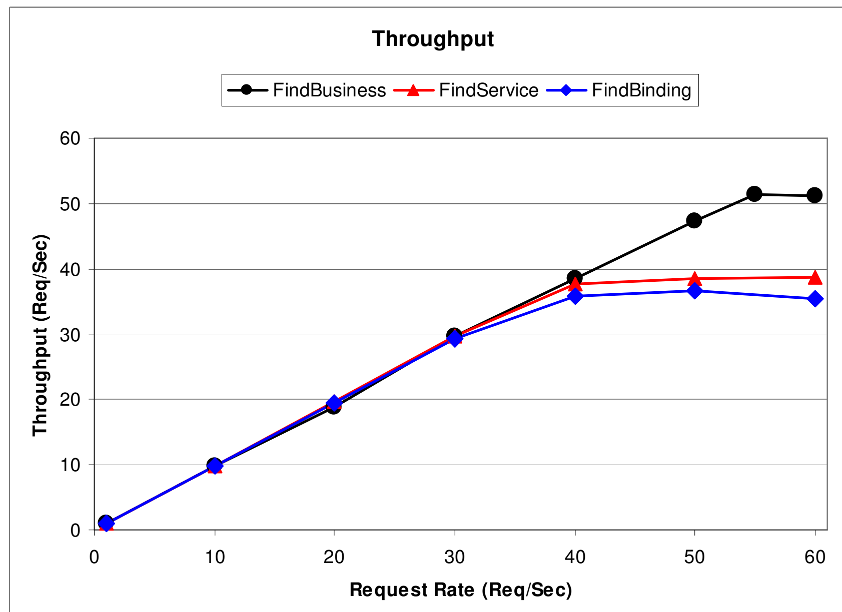

Figure 5.17: Throughput of the PUDDI Inquiry API using two-peer topology

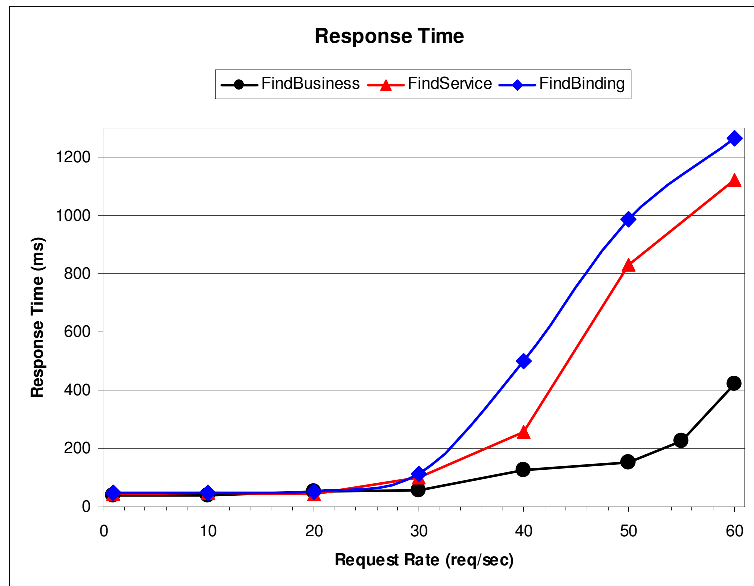

Figure 5.18: Response time of the PUDDI Inquiry API using two-peer topology

Figure 5.17 and 5.18 show that the performance of *FindBusiness* is significantly better than the other two. This is because *FindService* and *FindBinding* have to do more data manipulation internally which lengthens response time and lowers throughput.

In the four-peer network, the performance (Figure 5.19 and 5.20) is lower because more packages are transferred over the network and come at a cost. The differences in data manipulation among three methods are not visible in Figure 5.19 and 5.20 because the performance bottleneck is mainly caused by the high number of packages.

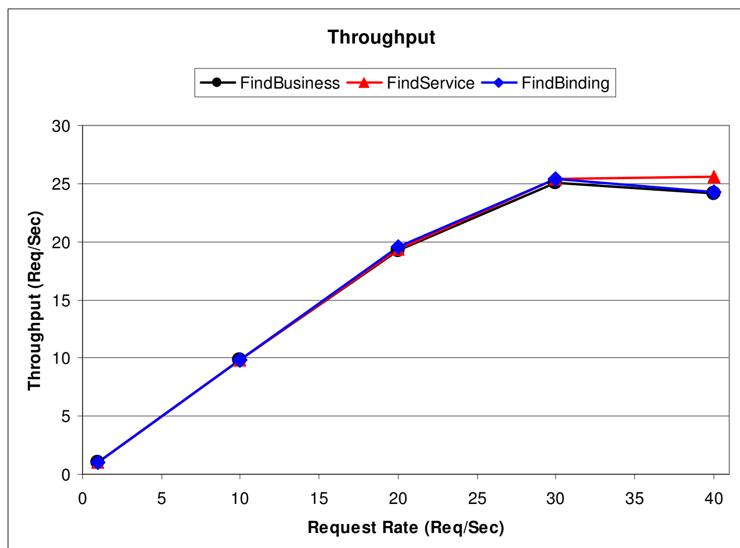

Figure 5.19: Throughput of the PUDDI Inquiry API using four-peer topology

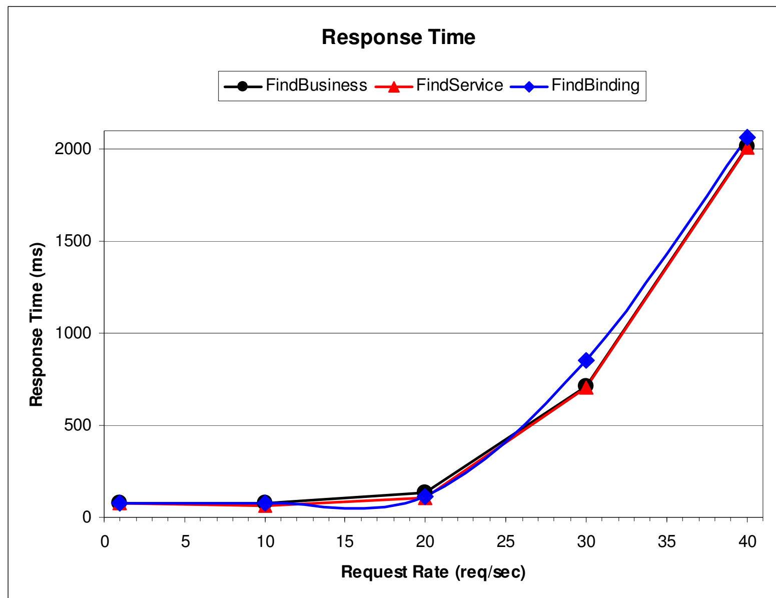

Figure 5.20: Response time of the PUDDI Inquiry API using four-peer topology

5.7 Throughput and Response Time of PWSG

In Figure 5.21 and 5.22, the performance of the four-peer PWSG system is lower than that of the two-peer PWSG system because more packages are transferred over the four-peer network than the two-peer network.

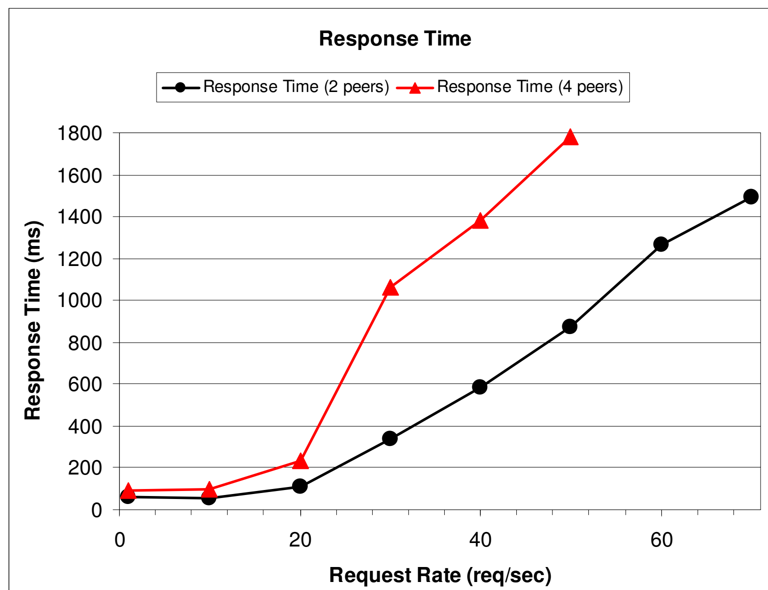

Figure 5.21: Response time of PWSG

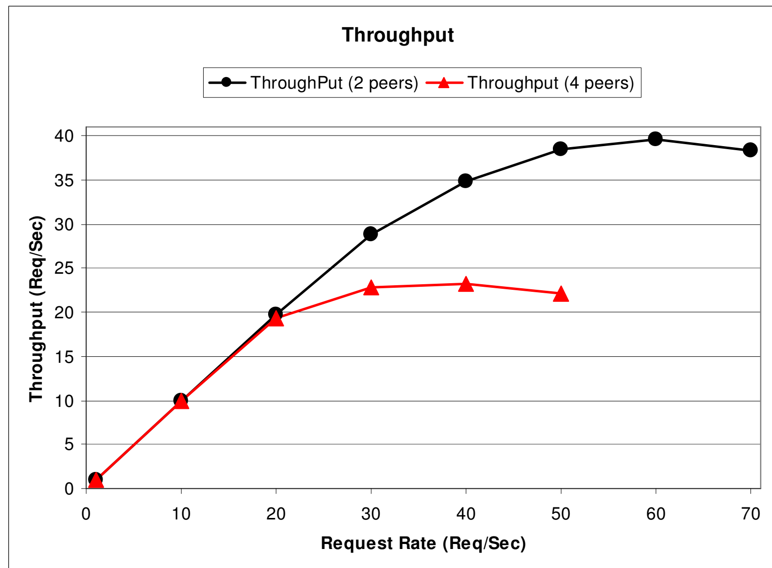

Figure 5.22: Throughput of PWSG

5.7 Conclusions

The use of PUDDI and PWSG adds an overhead to the target machine in terms of CPU usage and bandwidth consumption. This overhead varies according to two factors: the request rate and the number of peers. The higher the request rate is, the higher the overhead is. The more peers the system has, the higher the overhead is. The overhead on the consumer side is higher than that on the provider side because the consumer peer needs to process more messages than the provider peer does for each service invocation.

The number of PUDDI/PWSG peers is a major factor that determines the performance of the whole system. Given a certain request rate, PUDDI/PWSG has higher throughput and lower response time in a two-peer network than it in a four-peer network. Therefore, the more peers the system has, the lower the performance is. This result is consistent with the observation result of the overhead of PUDDI/PWSG.

CHAPTER 6

CALIBRATING THE SIMULATION SYSTEM

The size of the reference system used in the above experiments is not large enough to examine the proposed architecture thoroughly. The parameters of the P2P protocol, e.g., TTL and the number of neighbors, do not show their characteristics in a small system. To observe the nature of these parameters, a large scale system, e.g., a system with 1000 peers, is required.

Furthermore, the reference system is not flexible enough for examining the QoS selection which requires changing some system settings, e.g., the network connection speed, CPU processing capacity, and the number of peers. These settings are difficult to manipulate as needed on existing hardware.

A simulation system will be used in the rest of experiments to achieve the requirements mentioned above. The simulation system is an application that simulates the logical working mechanism of the reference system on a highly abstract level. It uses a number of system parameters collected from the reference system to control each action internally. When using the same topology, if the result observed from the simulation system is similar to the one observed from the reference system, one can say that the simulation system is accurate and correct enough to observe the nature of the architecture. Configuring these parameters in the simulation system to represent behaviours of the reference system is called calibration in this thesis.

Two sets of experiments are conducted on the simulation system:

- ***Examining the effects of P2P parameters, i.e., TTL and the number of neighbors***

  The result of this experiment helps users properly configure P2P parameters of the PWSG system to achieve different system performance.

- ***Comparing the QoS selection with other selection methods in terms of improving overall performance***

  The purpose of this experiment is to determine whether or not the QoS selection will improve the system performance.

In this chapter, a calibration process will be conducted on the simulation system. First, system parameters will be measured on the reference system. Then, these parameters will be set up in the simulation system. Finally, the simulation system will be examined using the same configuration as the reference system. If the result obtained from the simulation system is similar to that from the reference system, the calibration is finished.

6.1 Design of the Simulation System

The simulation system consists of virtual peers, a task scheduler, and a message bus (Figure 6.1). The virtual peer is an object representing the functionality of the consumer/provider peer in the reference system. It has three layers:

- Virtual networking layer, which sends/receives packages to/from the message bus just like operating over a real network,
- Protocol layer, which implements the Gnutella protocol, and
- Functional layer, which implements service discovery and service invocations. The QoS selection works in this layer.

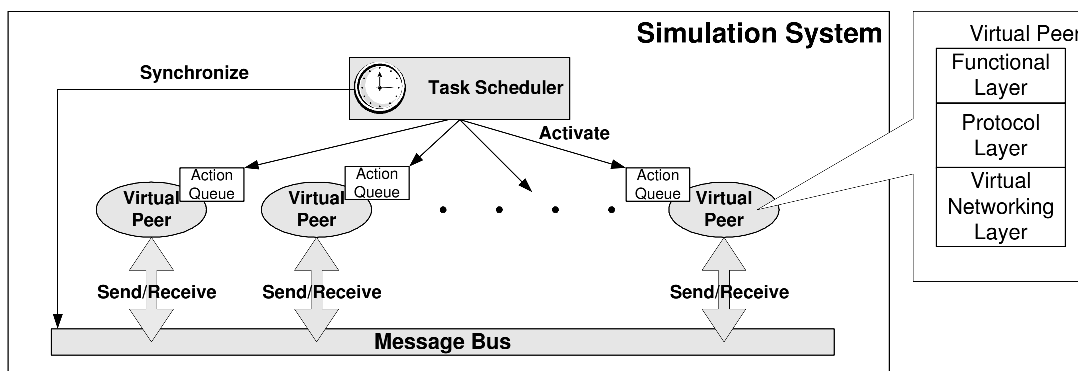

Figure 6.1: The simulation system

The virtual peer is categorized as the provider peer and the consumer peer according to the role it plays in the service invocation. The consumer peer requires services, while the provider peer provides services.

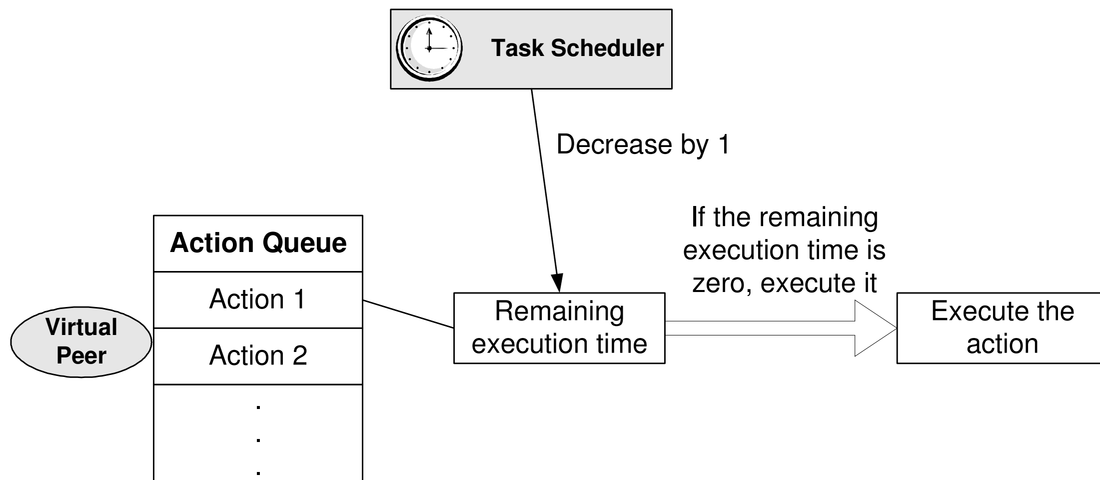

Figure 6.2: Execution of the action

The task scheduler activates each virtual peer sequentially according to an internal timer. When a virtual peer is activated, the task scheduler will examine all actions in the action queue of the peer (Figure 6.2). Each action has a field indicating its remaining execution time that will be decreased by 1 each time when the virtual peer is activated. If the remaining execution time is zero, the action will be removed from the queue and executed. Actions are created by functional layers according to specific events or messages. After all actions of the current peer are manipulated, the task scheduler will activate the next peer. When all peers are activated once, a run is finished. The internal timer moves one step (configurable) forward. Then, the task scheduler starts the next run.

The message bus is a software module used by all virtual peers to exchange messages. It receives the message from the virtual peer and notifies the destination peer of the message arrival. Then the destination peer can receive the message from the bus in the next activation period. To simulate different network transferring lags, the message bus is able to delay message transferring for a certain time length (configurable). This delay is synchronized with the internal timer in the task scheduler.

6.2 Choice of System Parameters

Since performance is a major concern in experiments conducted on the simulation system, it is necessary for the simulation system to represent the execution time (response time) as well as the effect of each operation.

The response time can be represented by the following equation ([18] Section 6.4.1, Open model solution technique):

$$R(\lambda) = D[1 + A(\lambda)]$$

Where, $R(\lambda)$ is the response time of a request at a give request rate λ, D is the service time that the request needs to be executed in the processor, $A(\lambda)$ is the number of requests waiting in the queue in front of the current request. Based on this equation, the response time of a request can be intuitively understood as its service time plus the waiting time used to process requests in front of it.

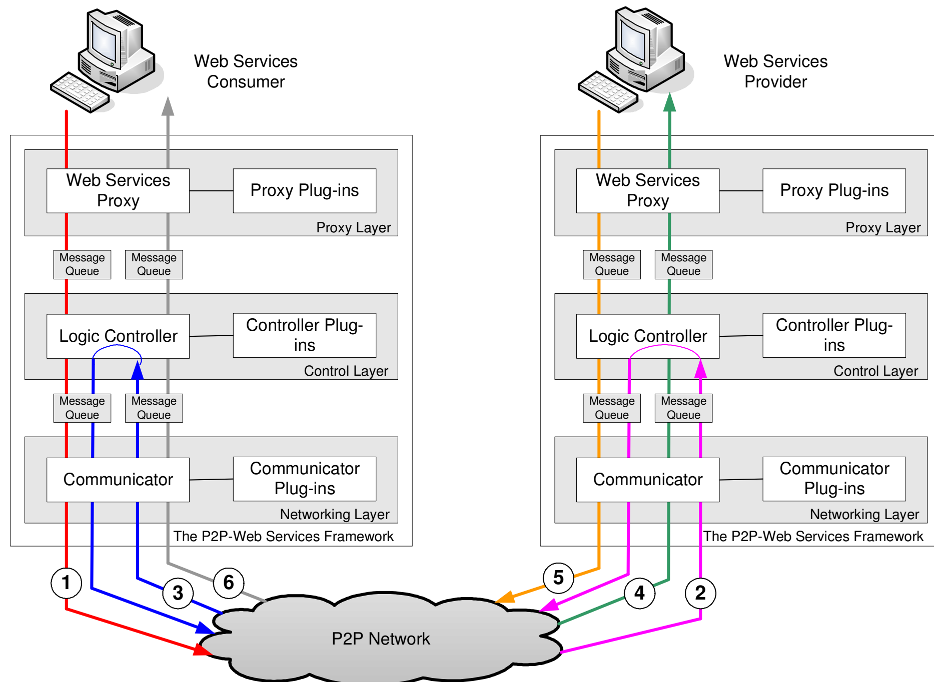

Figure 6.3: Decomposing a request process

According to the above analysis, processing a request in the reference system can be decomposed to six sub-operations, which are shown in Figure 6.3. These sub-operations are:

① Sending a discovery request from a consumer peer to a provider peer,

② Responding the discovery request,

③ Selecting the provider peer and sending it the invocation request,

④ Receiving the invocation request and performing the invocation,

⑤ Returning the invocation result to the consumer peer, and

⑥ Receiving the result and finalizing the service invocation.

These sub-operations help determine the system parameters used in the simulation system. In the simulation system, there are also six corresponding actions executed to process each request. The execution of these actions, which represent those sub-operations on the reference system, is controlled by six sets of service time. The service time of each sub-operation can be easily collected from the reference system.

6.3 Calibration

The parameters determined above do not include the latency caused by the network. Since there are some other delays existing in the reference system, the network latency can be used as a means to tweak the simulation system. The response time given by the equation in Section 6.2 is not an instant value but a long-term, statistic value. Therefore, measuring the accuracy of the simulation system is also based on statistical results.

The calibration is conducted using three kinds of networking topologies, a two-peer network shown in Figure 6.4, a three-peer topology shown in Figure 6.6, and a four-peer topology shown in Figure 6.8. Figure 6.5, 6.7, and 6.9 present the performance curves measured in the reference system and the simulation system using two-peer, three-peer, and four-peer topologies.

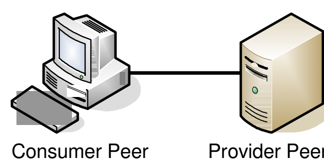

Figure 6.4: Two-peer topology

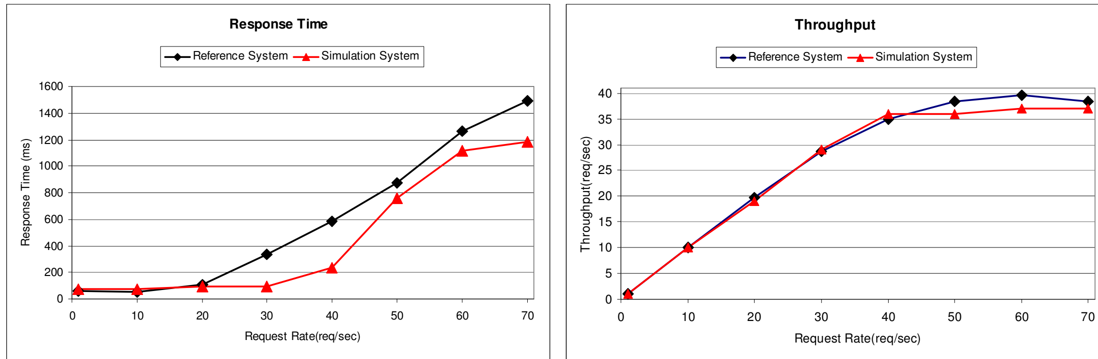

Figure 6.5: Performance comparison using the two-peer topology

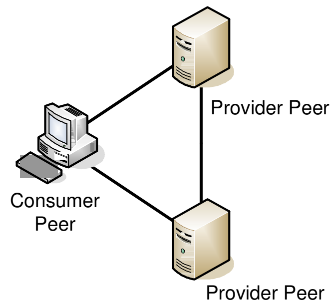

Figure 6.6: Three-peer topology

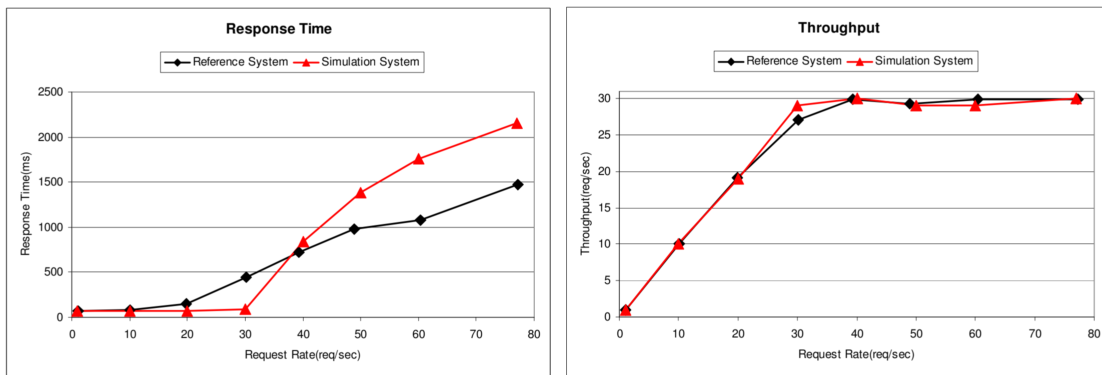

Figure 6.7: Performance comparison using the three-peer topology

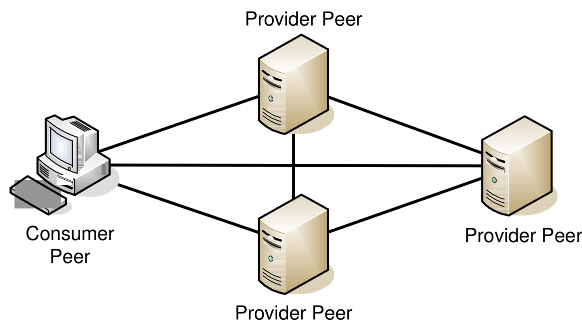

Figure 6.8: Four-peer topology

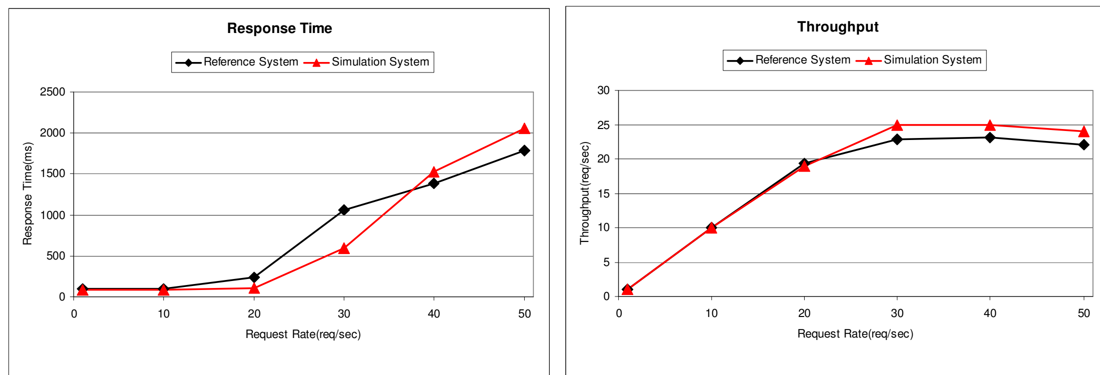

Figure 6.9: Performance comparison using the four-peer topology

6.4 Conclusions

Based on the result of calibration, one can draw the following conclusions.

- The throughput measured in the simulation system matches the throughput measured in the reference system when the system is not saturated (the request rate is less than the maximal throughput). It is also close to the reference data when the system is saturated (the request rate is equal to or greater than the maximal throughput).
- The response time measured in the simulation system matches the response time measured in the reference system when the request rate is less than 10req/sec. It shows a deviation from the reference data when the request rate is greater than 10req/sec.
- Throughput and request time measured in both systems show the same tendency.

Although the simulation system does not behave exactly as the reference system in some situations, its approximation is accurate enough for the experiments conducted in it for the following reasons:

- The experiment examining the effects of the P2P parameters is working in an extreme low request rate situation. Each request is only issued after the previous request is completed. There are no concurrent requests issued. In this situation, the simulation system precisely represents the behaviour of the reference system.

- The experiment examining the QoS selection will compare the result with the other two selection methods in the same situation. The deviation of the simulation system in representing the behaviour of the reference system is eliminated in the comparison. The correctness of the performance tendency of the simulation system plays a major role in guaranteeing the correctness of the result.

CHAPTER 7

EXPERIMENT 2: EFFECTS OF P2P PARAMETERS

The P2P-Web Services architecture has a number of parameters related to the Gnutella protocol that can be changed in the configuration file. The most important parameters are: TTL, the number of neighbors, and the waiting time for discovery replies. When deploying the system, a user needs to configure these parameters properly. Incorrect values may have negative effects on the system performance and efficiency. Research about the effects of these P2P parameters is scarce. The papers and technical reports mentioning the effects, e.g., [33] and [20], draw their conclusions only based on mathematic or logical analysis. A simulation based study about these P2P parameters will be an inevitable supplement to this research field. The experiment in this chapter focuses on the following aspects.

- ***Effects of different TTLs on the number of reachable peers, the discovery duration, and the number of packages transferred.***
- ***Effects of different numbers of neighbors on the number of reachable peers, the discovery duration, and the number of packages transferred.***

This experiment will help reveal the effects of P2P parameters in different situations, and optimize the configuration of the P2P-Web Services architecture.

7.1 Effects of TTL

Generally, the value of TTL should be chosen according to the number of peers in the system. The more peers in the system, the greater TTL should be. Ritter [33] argued that a high TTL increases the number of reachable peers geometrically but incurs high bandwidth consumption. He gave a table in his report showing the relationship between TTL and the number of reachable peers, and TTL and bandwidth consumption.

In the P2P-Web Services architecture, the discovery duration is also a crucial factor for users because it determines the response time of a service invocation. The more

reachable peers, the longer the discovery duration. The discovery duration has not been studied by researchers due to limitations of the methodologies they chose. The simulation system used in the experiments provides a means to investigate this crucial factor.

The experiment is conducted in the simulation system with 2000 and 10000 provider peers separately. There is only one consumer peer in the system that issues requests and records each response. Each peer builds connections to other peers according to the Ping-Pong algorithm of Gnutella that finds out new peers using the ping request and chooses neighbors randomly from all known peers.

To simplify service discovery, all provider peers provide the same service. The consumer peer sequentially issues requests with a TTL value chosen from 1 to 15. The new request is issued only after the previous request is finished and there is not package propagated over the network. For each TTL, the consumer peer will record the number of reachable peers, the waiting time for all responses, the number of requests (Query) propagated, and the number of responses (QHit) propagated.

Table 7.1 summarizes the configuration of the experiment.

Table 7.1: Experimental configuration

#Provider Peer	2000 and 10000 separately
#Consumer Peer	1
TTL	From 1 to 15
#Neighbors	5
Networking Latency	5ms for any connection between two peers

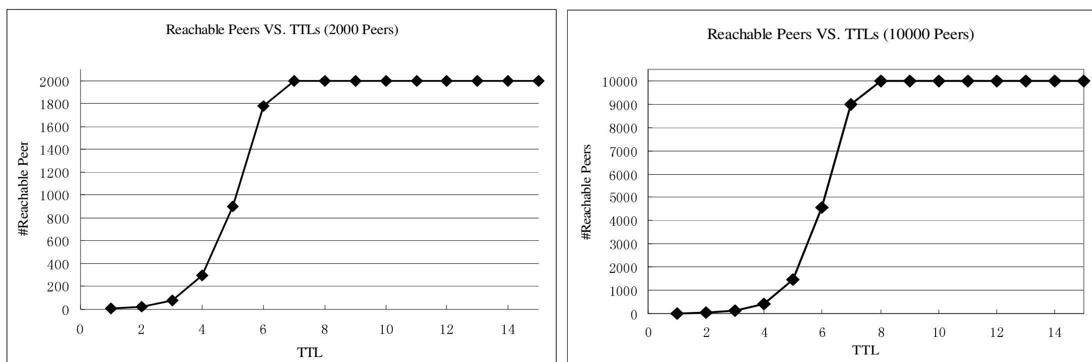

Figure 7.1: Number of reachable peers V.S. TTLs

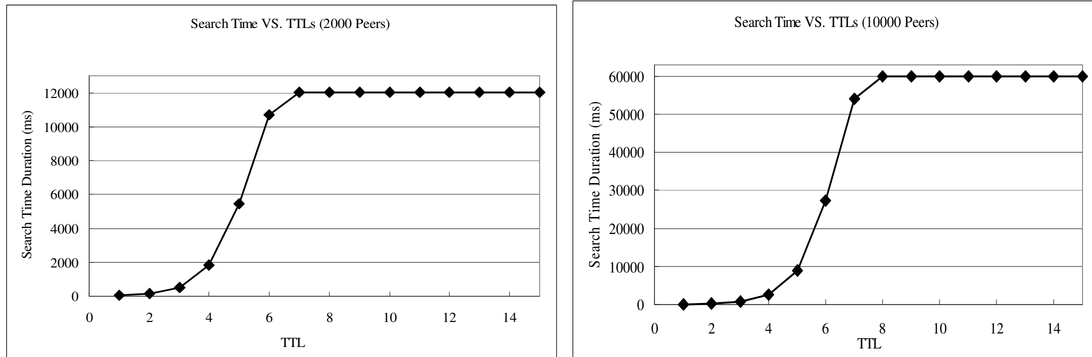

Figure 7.2: Search time duration V.S. TTLs

Figure 7.1 presents that for the same network topology a slight increase in the value of TTL will significantly increase the number of reachable peers. For instance, when TTL is 4, the number of reachable peers is about 300. When TTL is 6, about 1800 peers can be reached. For the same TTL, the number of reachable peers changes in different network topologies. The more peers the system has, the more peers a search request can reach. This is because the small network has more connection overlaps than the large one. A connection overlap means that several close peers have some common neighbors. The package propagated in a connection overlap is unable to reach any new peer outside the overlap. So, the connection overlap makes the discovery less efficient.

Figure 7.2 presents that the search duration is only proportional to the reachable peers. The time curve exactly matches the curve of reachable peers in Figure 7.1. Searching 2000 peers takes approximate 12000 ms and searching 10000 peers takes approximate 60000 ms.

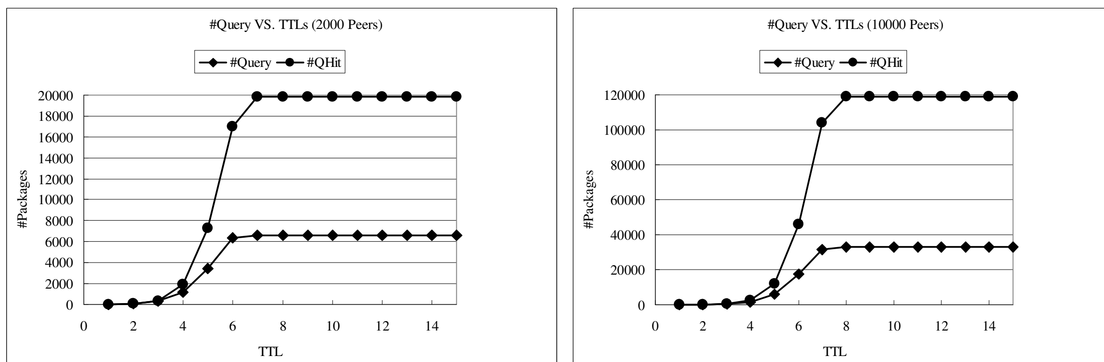

Figure 7.3: Number of packages V.S. TTLs

The number of packages (Query and QHit) transferred in the system (Figure 7.3) shows a similar curve as the reachable peers. The number of QHit packages is far greater than that of Query packages. This is because that along a Query propagation route each peer will create a QHit package that will be transferred back to the consumer peer along the Query route.

7.2 Effects of the Number of Neighbors

Besides TTL, the number of neighbors also has an effect on the reachable peers. The greater the number of neighbors is, the more peers a request can reach. Although the effect is intuitive, there is no data available to show the effect precisely. It is still unclear whether or not the number of neighbors has the same effects as TTL on the number of reachable peers, search duration, and number of packages transferred. This experiment aims to examine the effects using different numbers of neighbors.

The experimental setup is the same as the experiment examining TTL using 2000 peers except that the value of TTL is fixed to 5 and the number of neighbors is not fixed. Table 7.2 summarizes the configuration of the experiment.

Table 7.2: Experimental configuration

#Provider Peer	2000
#Consumer Peer	1
TTL	5
#Neighbors	From 1 to 15
Networking Latency	5ms for any connection between two peers

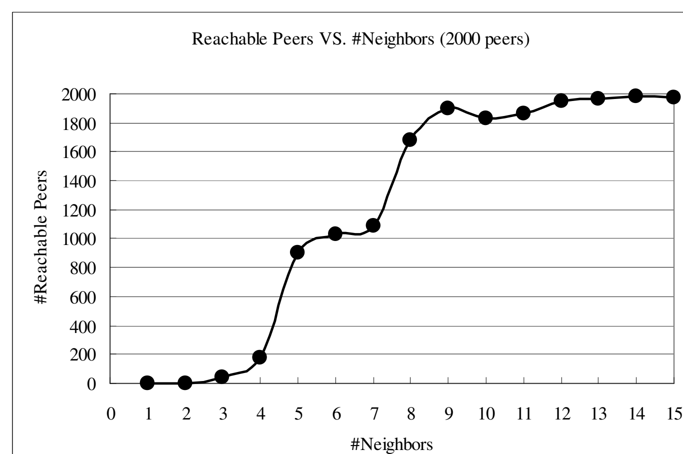

Figure 7.4: Reachable peers V.S. #Neighbors

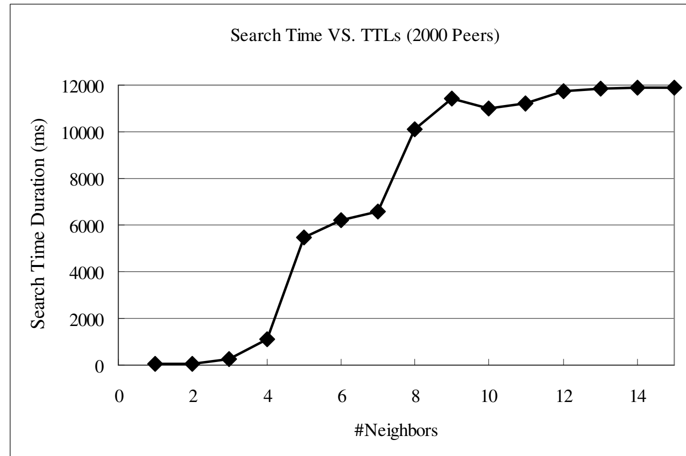

Figure 7.5: Search time V.S. #Neighbors

Increasing the number of neighbors leads to a fluctuating curve on the number of reachable peers (Figure 7.4). Correspondingly, the search duration also shows the same fluctuating curve when the number of neighbors changes (Figure 7.5). Some parts of the curve in Figure 7.4, e.g., from 5 to7 on the X axis, are flatter than others. This means that the increase of the number of neighbors may not effectively expand the reachable area over the network, because the increase in the number of neighbors involves increasing overlaps as well. When the increase in the number of neighbors is great enough, the reachable area can be expanded substantially.

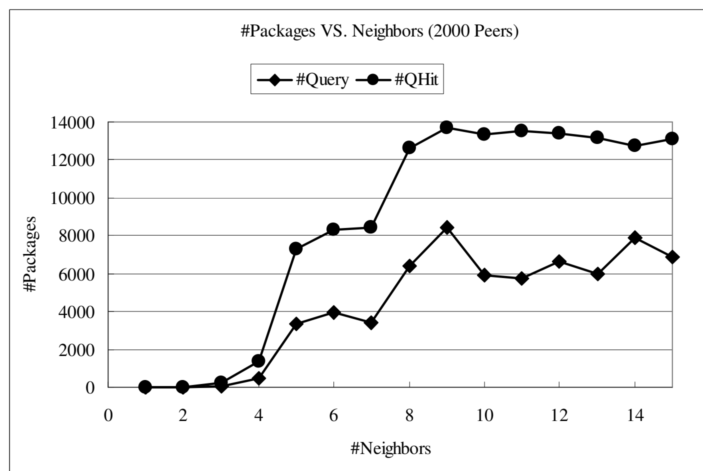

Figure 7.6: #Packages V.S. #Neighbors

Figure 7.6 presents that increasing the number of neighbors can reduce the number of QHit packages transferred over the network. This is because the high number of

neighbors shortens the average route length from the consumer peer to every provider peer.

7.3 Conclusions

According to the experimental result, one can observe that the search time and the number of packages transferred are directly proportional to the reachable peers instead of TTL and the number of neighbors. When the number of neighbors is fixed, a slight increase in TTL will significantly increase the number of reachable peers. By contrast, when TTL is fixed, a slight change in the number of neighbors may not affect the number of reachable peers significantly.

When the search efficiency is the major concern in deployment, the user can set a large TTL in the system. When the search time is the major concern, the user can set the search waiting time of PWSG to the value with which PWSG is able to reach desirable number of peers. When the bandwidth consumption is the major concern, the number of neighbors should be given a fairly large value that should be less than the limit of OS (e.g., by default, 10 connections on Windows XP Professional).

CHAPTER 8

EXPERIMENT 3: EXAMINING QOS SELECTION

In this experiment, the focus is the key issue of the research, the QoS based selection. Since the QoS selection aims to improve the system performance, its effects about response time and throughput will be examined in various situations in the simulation system.

To make the experimental result intuitive, another two common selection methods, i.e., random and feedback based selection, will be examined in the same situations to serve as references. The performance of these three selection methods will be compared to show the improvement of the QoS selection.

8.1 Experimental Setup

In most real systems, the number of service providers (servers) is far less than that of service consumers (clients) because the cost of a provider is higher than that of a consumer. The QoS selection is designed to work in this situation to help each consumer find the most available of all providers. The simulation system configured in this experiment has less provider peers (10) and more consumer peers (100 and 200) than the system in Experiment 2 that, in contrast, has 2000/10000 provider peers and only 1 consumer peer. This configuration is chosen for the following reasons:

- According to the response time curve shown in Experiment 1, the steepest part of the curve is achieved when the request rate is greater than 50% of the maximal throughput. Any slight change in the request rate in this range will significantly change the response time and present a distinct difference in the experimental result. Based on the service time set in the simulation system, the ratio of providers to consumers is at about 1:10 to achieve desirable request rates.
- The number of requests issued by each consumer should be large enough for the QoS selection to work effectively (warming time). In the experiment, each

consumer will issue 100 requests. Based on preliminary tests on the experiment machine (Pentium 4 3.2G, 2G Ram), the basic service time is set to 50ms to achieve a reasonable experiment time (10 hours for a run). The maximal throughput under this service time is about 20req/sec that is also consistent with the observed result in Experiment 1. Then, the numbers of providers and consumers can be set to 10 and 100 respectively.

The experiment has 4 settings representing the local-area network (LAN), the wide-area network (WAN), WAN with mixed servers, and heavy load WAN with mixed servers.

The local-area network setting (Setting 1)

The LAN environment features a fast network connection for all peers. All provider peers have the same basic service time (50ms). The setting is summarized in Table 8.1.

Table 8.1: Local-area network setting

#Providers	10 (All providers provide the identical service)
#Consumer	100
Service Time	50ms
Network Connection Latency	5ms
TTL	5
#Neighbors	5

The wide-area network setting (Setting 2)

The WAN environment features mixed network connection speed. A consumer may have different connection qualities with different providers. To simplify the experiment, there are only two kinds of connections used in the system. The setting is summarized in Table 8.2.

Table 8.2: Wide-area network setting

#Providers	10 (All providers provide the identical service)
#Consumer	100
Service Time	50ms
Network Connection Latency	5ms for the consumer-provider pair whose ID are both even or odd, Otherwise, 50ms for the rest
TTL	5
#Neighbors	5

WAN with mixed servers (Setting 3)

This setting represents a scenario close to the real system. It has servers with different processing abilities as well as different connection qualities. To simplify the experiment, there are only two kinds of processing abilities used for servers. Table 8.3 summarizes this setting's parameters.

Table 8.3: WAN with mixed servers setting

#Providers	10 (All providers provide the identical service)
#Consumer	100
Service Time	Mixed servers: 50ms for the server with an even id 500ms for the server with an odd id
Network Connection Latency	5ms for the consumer-provider pair whose ID are both even or odd, Otherwise, 50ms for the rest
TTL	5
#Neighbors	5

Heavy load WAN with mixed servers (Setting 4)

This setting is modified based on Setting 3. The only difference is that this setting has 200 consumers. The providers in the system have a higher request rate than above three settings. Table 8.4 summarizes this setting's parameters.

Table 8.4: Heavy load WAN with mixed servers setting

#Providers	10 (All providers provide the identical service)
#Consumer	200
Service Time	Mixed servers: 50ms for the server with an even id 500ms for the server with an odd id
Network Connection Latency	5ms for the consumer-provider pair whose ID are both even or odd, Otherwise, 50ms for the rest
TTL	5
#Neighbors	5

The workload is generated by each consumer peer individually. Each consumer peer generates 100 service invocation requests in each setting. It issues a request immediately after it receives the response for the previous request. It generates a feedback report that

will be kept in the provider peer as a reference for the following invocations. The feedback report keeps the information about predicted and actual response time.

For each kind of setting, three selection methods, i.e., QoS selection, feedback based selection, and random selection, will be examined. Random selection selects a provider using a random function. Feedback based selection selects a provider based on the average response time S_b. It always selects the provider with the shortest response time. S_b is given as below:

$$S_b = \begin{cases} \sum_{1}^{n} \frac{R_i^n}{n}, \ \textit{if the provider already has feedback reports} \\ \sum_{i=1}^{m} \sum_{n=1}^{n} \frac{R_i^n}{n} \Big/ m, \ \textit{otherwise} \end{cases}$$

Where, R_i^n denotes the response time recorded in the n-th feedback report regarding the i-th provider, m denotes the number of providers.

8.2 Experimental Results

The experimental results are presented in terms of the mean response time, throughput, and standard deviation. The mean response time is the average response time of all requests. Throughput represents how many requests all providers process in one second. The standard deviation presents a statistical difference indicating how much the response time of each request deviates from the mean response time.

Figure 8.1 presents the experimental result of Setting 1. In Setting 1, the response time of random selection shows less fluctuation (lowest standard deviation) as all requests are evenly loaded on providers (balanced load). QoS selection is very effective in arranging requests for providers (lowest mean service time). Also, its response time is the same as random selection. Feedback selection does not work well in this setting.

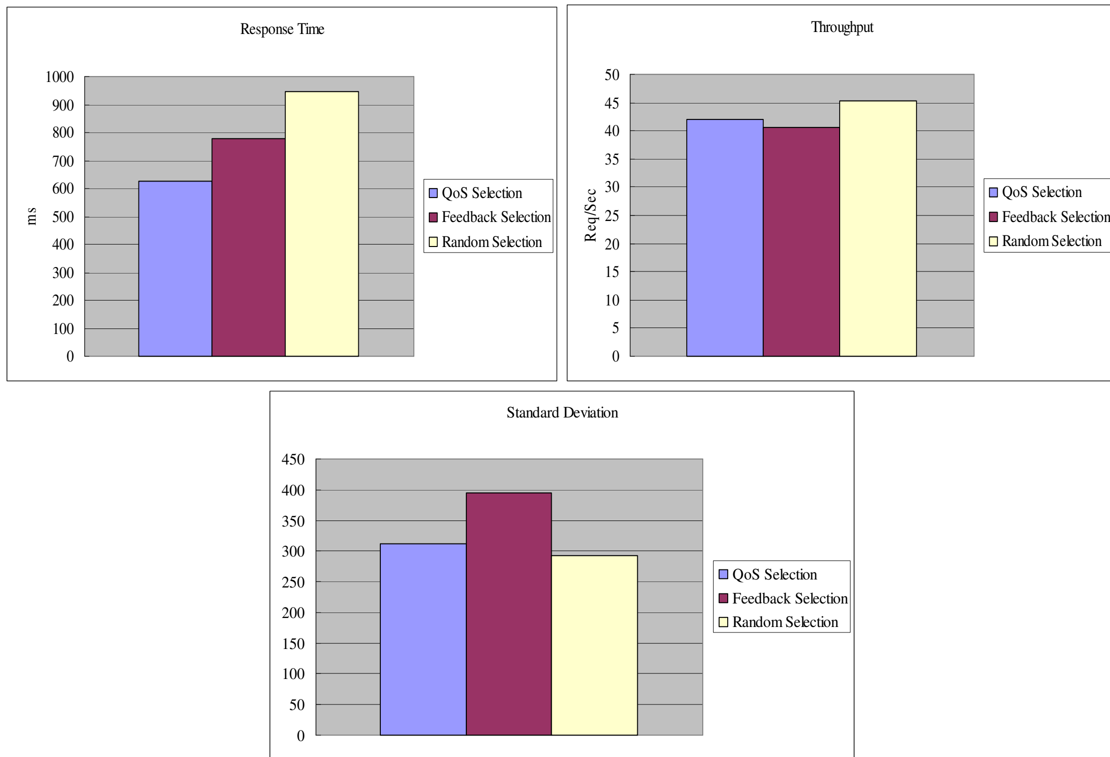

Figure 8.1: Experimental result of Setting 1

Figure 8.2 presents the experimental result of Setting 2.

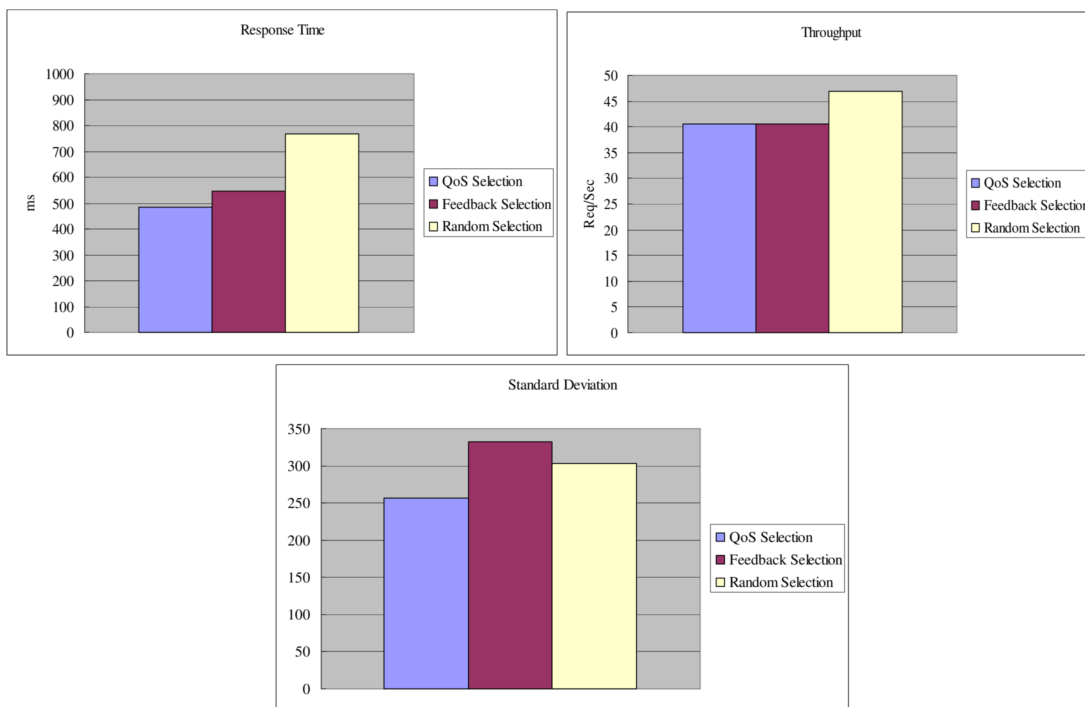

Figure 8.2: Experimental result of Setting 2

In the experiment of Setting 2, feedback selection is getting better than it is in Setting 1. Random selection deteriorates due to its high response time and standard deviation. QoS selection still shows its advantages in response time and standard deviation.

Figure 8.3 presents the experimental result of Setting 3.

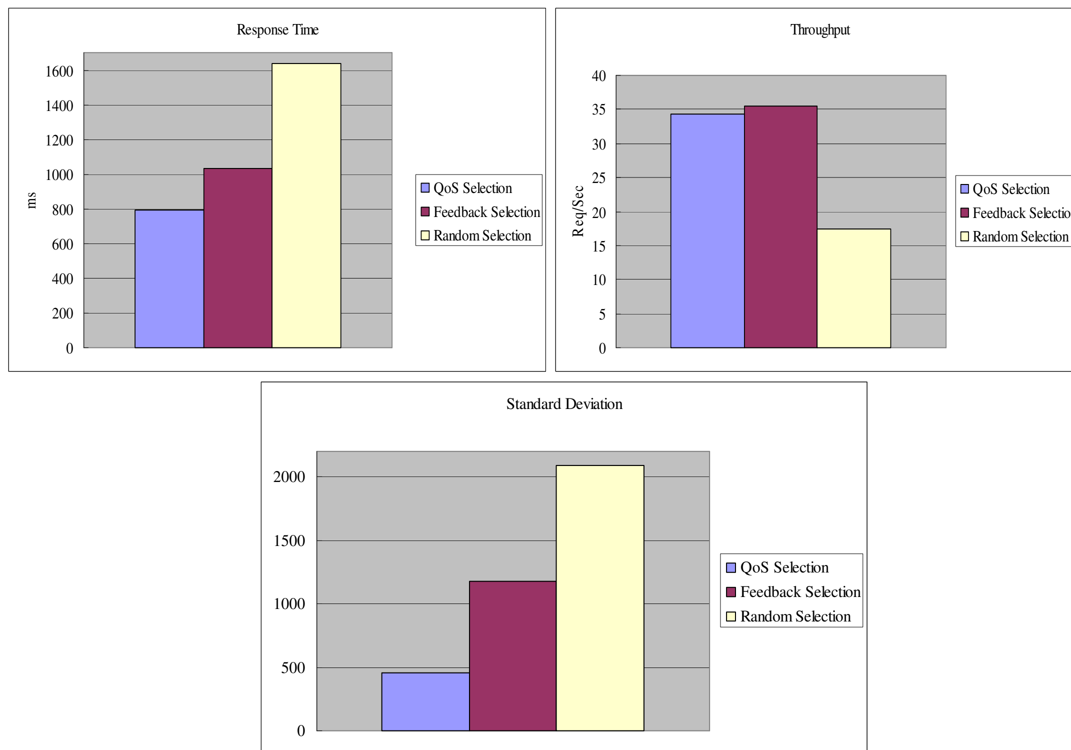

Figure 8.3: Experimental result of Setting 3

In the experiment of Setting 3, the system configuration is getting more complicated due to the introduction of provider differences. Since QoS selection takes the remaining load of each provider into account, it can arrange requests for providers optimally. This advantage results in the lowest response time and standard deviation. Also, the throughput of QoS selection is very close to the highest one. Random selection achieves the worst performance.

Figure 8.4 presents the experimental result of Setting 4.

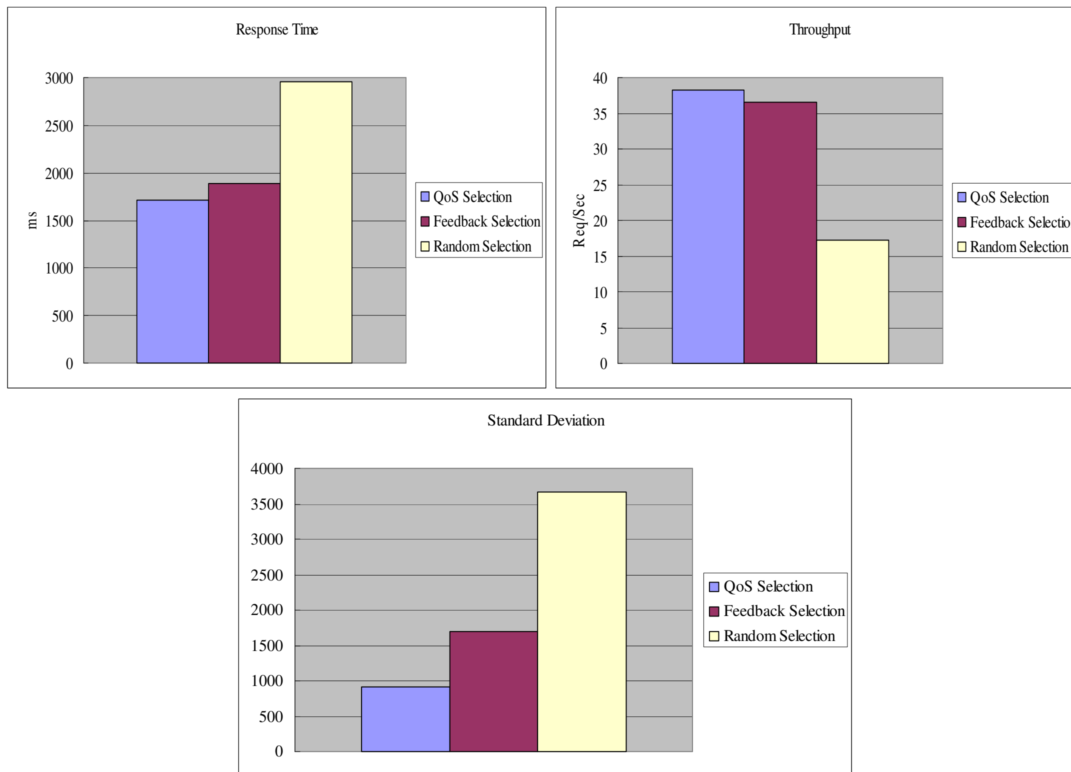

Figure 8.4: Experimental result of Setting 4

In the experiment of Setting 4, QoS selection outperforms other two methods in terms of response time, throughput, and standard deviation. The performance of feedback selection is close to that of QoS selection. Random selection is still the worst.

8.3 Conclusions

QoS selection is able to achieve a lower and less fluctuating response time than feedback selection and random selection in all situations. To each individual consumer peer, QoS selection provides it a better choice than other two selection methods because of the guarantee of the lowest response time and deviation. The response time curves presented in Experiment 1 (measuring performance of PWSG) indicates that high throughput leads to high response time. This conclusion is applicable in the experiment result of examining QoS selection. Therefore, QoS selection is unable to achieve highest throughput in some cases when the response time is relatively low. Overall, the low standard deviation of QoS selection proves its stability.

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

SOA is a software model that "represents software functionality as discoverable services on the network" [9]. It consists of the service provider, the service consumer, and the central registry. To maximize the scalability of the system, these components are loosely coupled by message based interactions and standardized service definition schemas. Even so, the communication between components is still server-centric. A client component connects to a pre-determined server component without taking the status of the server component into account. This server-centric approach excludes SOA from a dynamic networking environment, in which all participants can enter and depart at any time without notice (e.g., wireless network).

The P2P-Web Services architecture proposed in this thesis eliminates this constraint and enables the use of SOA in dynamic networking environments. Specifically, it integrates P2P concepts into Web Services to support service discovery and service invocations in a dynamic networking environment. The architecture consists of PUDDI and PWSG, supporting the life cycle of Web Services components, from development to deployment, which is lacking in other proposed systems.

PUDDI is a distributed UDDI system used as a substitute for the standard UDDI server. Each Web Services component, i.e., the WS consumer, the WS provider, or the UDDI client, interacts with a PUDDI peer paired locally with it and treats this PUDDI peer as a UDDI server. The PUDDI peer receives the register request from the WS provider or the UDDI client and keeps the information in the local repository for service discovery. When a PUDDI peer receives a service discovery request from the WS consumer or the UDDI client, it will inquire other PUDDI peers for the service over the P2P network. The discovery result will be returned to the WS consumer or the UDDI client.

PWSG is a software gateway connecting to Web Service components to perform service invocations in a P2P manner. Each WS consumer/WS provider interacts with its local PWSG peer. The WS consumer treats the PWSG peer as a service provider, while the WS provider treats the PWSG peer as a service consumer. When a PWSG peer receives a service invocation request from the WS consumer, it will perform the service invocation in two phases. In the first phase, this PWSG peer will inquire other PWSG peers for the service invoked by the request over the P2P network. The provider responding to the inquiry will be considered as a candidate. In the second phase, this PWSG peer will select one provider from all candidates and send it the invocation request. The result of the invocation will be returned to the WS consumer.

The P2P-Web Services architecture uses PWSF as the basis for PUDDI and PWSG. PWSF is a transparent proxy joining the P2P network and Web Service components. It uses three layers, the proxy layer, the control layer, and the networking layer, to manage functions hierarchically. PUDDI and PWSG are developed as two sets of plug-ins residing in these three layers. This design pattern maximizes reusability and minimizes the effort of development.

A reference system and a simulation system have been developed to investigate the P2P-Web Services architecture. The reference system is a working implementation of the architecture and consists of prototype peers running on multiple machines. Its scalability is limited to the available hardware resources. The simulation system is used to observe the nature of system parameters for a large scale system. It is an application simulating the logical working mechanism of the reference system on a highly abstract level. The simulation system uses a number of system parameters collected from the reference system to control each action internally. When using the same topology, if the result observed in the simulation system is similar to the one observed in the reference system, one can say that the simulation system is accurate enough to be used in the experiments.

The use of PUDDI and PWSG adds an overhead to the Web Services system. The experiment examining overhead and performance of PUDDI and PWSG measured the cost quantitatively in terms of CPU usage and bandwidth consumption. Also, it measured

the performance of PUDDI and PWSG in terms of response time and throughput. This experiment showed:

- The higher the request rate is, the higher the overhead is. The CPU usage and bandwidth consumption have a linear relationship with the request rate when the system is not nearly saturated (CPU usage < 80%). They show a flat curve gradually approaching their maximal values when the system is nearly saturated.
- The overhead in the consumer peer is higher than that in the provider peer because the consumer peer processes more network packages.
- The more provider peers the system has, the higher the overhead is, and the lower the performance is.

The experiment examining the effects of P2P parameters revealed that a slight change in TTL may significantly affect the number of reachable peers, the search duration, and bandwidth consumption. In a 2000-peer network, when TTL is changed from 5 to 6, the number of reachable peers raises from about 900 to 1780. A slight change in the number of neighbors may not affect the number of reachable peers all the way. However, increasing the number of neighbors will help reduce bandwidth consumption. In a 2000-peer network, to reach the same number of peers, a high number of neighbors (#neighbors = 15, TTL=5) may reduce the number of packages by 26% when compared with a low number of neighbors (#neighbors = 5, TTL=7).

PWSG performs a service discovery operation for each service invocation. There may be more than one provider available for the consumer. To determine the most suitable provider for the consumer, the QoS based selection is used in PWSG. It collects the information from providers regarding their current remaining capacity and historical feedback reports. The experiment examining QoS selection compared QoS based selection with random selection and feedback based selection in terms of response time and overall throughput. It showed that QoS based selection can achieve an 11%-51% lower response time than the other two methods in all situations. The standard deviation of its response time is also low. This means that the QoS based selection is stable.

The P2P-Web Services architecture has achieved the purpose of integrating P2P concepts into Web Services to enable Web Services to work effectively in dynamic networking environments. By using the Gnutella Query/QHit mechanism to fulfill each discovery/invocation request, the consumer peer needs to propagate a Query request over the P2P network and wait for replies from provider peers. This mechanism introduces an overhead into the P2P-Web Services architecture. In a small business environment that usually contains 1-2 replica servers (mirror servers), this overhead is relatively small and has a minor impact on the system performance. When the number of peers in the system is greater than four, this overhead will significantly impact the system performance. Using a distributed hash table P2P protocol can effectively shorten the search time and achieve a much lower overhead than the Gnutella protocol. This solution will be presented below as a part of the future work.

The QoS based selection of the P2P-Web Services architecture ensures that a service invocation can be executed in a shorter time than the traditional QoS selection. The traditional QoS selection only uses the historical invocation information to infer the QoS of the provider. The QoS selection proposed in this thesis not only uses the historical invocation information, but uses the current workload of the provider. It is able to infer the QoS of the provider more accurately. No matter what algorithm the traditional QoS selection uses, it lacks the context information about the provider. Comparing the new QoS selection with the traditional QoS selection reveals that the new QoS selection achieves the similar throughput as the traditional QoS selection, and outperforms the traditional QoS selection in terms of response time and standard deviation.

The P2P-Web Services architecture can be further improved in the following aspects:

- ***Supporting DHT P2P protocols***

  Using DHT P2P protocols is an effective means for the architecture to reduce bandwidth consumption and search waiting time. DHT does not support arbitrary keyword search because its searching pattern requires an exact match on the predefined keyword. When DHT is used in the P2P-Web Services architecture, this weakness can be addressed using domain-specific ontologies to standardize the service keywords (e.g., a service description and a method name). A

standardized keyword has an unambiguous meaning in the system and can be hashed for the search purposes.

- ***Using Semantic Web Services to enhance service discovery***

  The service discovery currently used in the P2P-Web Services architecture is keyword based. It requests an exact match in the method name and parameters. This limitation lowers the hit rate because there is no criterion for defining a service. Developers can define services in any way they like. Using semantic descriptions, a WS provider is able to publish its service in a strict, machine-understandable way. Then, service discovery will be functionality based.

- ***Response time/throughput based QoS selection***

  The QoS selection that has been implemented aims to minimize the mean response time. It will always select the provider that has the least pressure to achieve short response time. Although the mean response time is short, the system overall throughput may not be optimal. For some systems, the overall throughput is the major concern. It is useful for QoS based selection to support maximizing throughput. The user can switch the selection function to either maximize throughput or minimize response time.

REFERENCES

[1] Banaei-Kashani, F., Chen, C., and Shahabi, C. "WSPDS: Web Services Peer-to-Peer Discovery Service". In Intl. Symposium on Web Services and Applications, 7 pages. 2004.

[2] Bellwood, T. "UDDI Version 2 Specifications". http://uddi.org/pubs/ProgrammersAPI-V2.04-Published-20020719.pdf, 93 pages. 2002.

[3] Bellwood, T., Capell, S., Clement, L., Colgrave, J., Dovey, M.J., Feygin, D., Hately, A., Kochman, R., Macias, P., Novotny, M., Paolucci, M., Riegen, C.V., Rogers, T., Sycara, K., Wenzel, P., and Wu, Z. "UDDI Version 3.0.2". http://www.oasis-open.org/committees/uddi-spec/doc/spec/v3/uddi-v3.0.2-20041019.pdf, 420 pages. 2004.

[4] Booth, D. and Liu, C.K. "Web Services Description Language (WSDL) Version 2.0 Part 0: Primer". http://www.w3.org/TR/2005/WD-wsdl20-primer-20050803/wsdl20-primer.pdf, 84 pages. 2005

[5] Boubez, T., Hondo, M., Kurt, C., Rodriguez, J., and Rogers, D. "UDDI Programmer's API 1.0". http://www.uddi.org/pubs/ProgrammersAPI-V1.01-Published-20020628.pdf, 68 pages. 2002

[6] Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H.F., Thatte, S., and Winer, D. "Simple Object Access Protocol (SOAP) 1.1". http://www.w3.org/TR/2000/NOTE-SOAP-20000508, 1 page. 2000

[7] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., and Yergeau, F. "Extensible markup language (xml) 1.0 (third edition)". http://www.w3.org/TR/REC-xml, 1 page. 2004.

[8] Castro, M., Druschel, P., Kermarrec, A., and Rowstron, A. "One ring to rule them all: Service discovery and binding in structured peer-to-peer overlay networks". In Proceedings of the SIGOPS European Workshop, Saint-Emilion, 6 pages. France. 2002

[9] Chatarji, J. "Introduction to Service Oriented Architecture (SOA)". http://www.devshed.com/c/a/Web-Services/Introduction-to-Service-Oriented-Architecture-SOA, 5 pages. 2004.

[10] Chawathe, Y., Ratnasamy, S., Breslau, L., Lanham, N., and Shenker, S. "Making gnutella-like P2P systems scalable". In Proceedings of the ACM SIGCOMM '03 Conference, pages 407-418. New York, USA. 2003.

[11] Christensen, E., Curbera, F., Meredith, G., and Weerawarana, S. "Web services description language (wsdl) 1.1". http://www.w3.org/TR/wsdl, 1 page. 2001.

[12] Day, J. and Deters, R. "Selecting the best web service". In Proceedings of the IBM Centers for Advanced Study Conference (CASCON '04), pages 293-307. Ontario, Canada. 2004.

[13] Hagge, D. "Dynamic Discovery and Invocation of Web services". http://www-128.ibm.com/developerworks/library/ws-udax.html?n-ws-10312, 1 page. 2001.

[14] Haller, A., Cimpian, E., Mocan, A., Oren, E., and Bussler, C. "WSMX - a semantic service-oriented architecture". In Proceedings of the International Conference on Web Services (ICWS 2005), 8 pages. Orlando, Florida, USA. 2005.

[15] Gong, L. "JXTA: A network programming environment". IEEE Internet Computing, Volume 5, pages 88-95. 2001.

[16] jUDDI. Web Services Project @ Apache. http://ws.apache.org/juddi - Accessed on September, 2005.

[17] Laoveerakul, S., Laongwaree, K., and Tongsima, S. "Decentralized UDDI based on P2P". http://www.hpcc.nectec.or.th/C4/grid/UDDI.pdf, 4 pages. 2002.

[18] Lazowska, E.D., Zahorjan, J., Graham, G.S., and Sevcik, K.C. "Quantitative System Performance: Computer System Analysis Using Queueing Network Models". Prentice Hall. 1984.

[19] Liu, Y., Ngu, A.H., and Zeng, L. "QoS computation and policing in dynamic web service selection". In Proceedings of the 13th international World Wide Web Conference on Alternate Track Papers. WWW Alt. '04. ACM Press, pages 66-73. New York, USA. 2004.

[20] Lv, Q., Cao, P., Cohen, E., Li, K., and Shenker, S. "Search and replication in unstructured peer-to-peer networks". In *Proceedings of the 16th international Conference on Supercomputing*. ICS '02. ACM Press, pages 84-95. New York, USA. 2002

[21] Manola, F. and Miller, E. "RDF (Resource Description Framework) primer". http://www.w3.org/TR/rdf-primer, 1 page. 2004.

[22] Maximilien, E.M. and Singh, M.P. "Reputation and endorsement for web services". SIGecom Exch. Volume 3.1, pages 24-31. 2001.

[23] Maximilien, E.M. and Singh, M.P. "Conceptual Model of Web Service Reputation". SIG-MOD Record, ACM Special Interest Group on Management of Data. Volume 31, number 4, pages 36-41. 2002.

[24] Maximilien, E.M. and Singh, M.P. "Toward autonomic web services trust and selection". In Proceedings of the 2nd international Conference on Service Oriented Computing. ICSOC '04. ACM Press, pages 212-221. New York, USA. 2004.

[25] Microsoft Corporation. "DCOM Technical Overview". http://msdn.microsoft.com/-library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomtec.asp, 1 page. 1996.

[26] Milojicic, D.S., Kalogeraki, V., Lukose, R., Nagaraja, K., Pruyne, J., Richard, B., Rollins, S., and Xu, Z. "Peer-to-peer computing". Tech. Rep. HPL-2002-57, Hewlett Packard Laboratories, 52 pages. 2002.

[27] Mitra, N. "Soap version 1.2 part 0: Primer". http://www.w3.org/TR/soap12-part0, 1 page. 2003.

[28] "New to SOA and Web services", http://www-128.ibm.com/developerworks/webservices/newto/websvc.html, 1 page. Accessed on November, 2005.

[29] Object Management Group. "Common Object Request Broker Architecture (CORBA)". http://www.corba.org - Accessed on July, 2005.

[30] Orchard, D. "Achieving Loose Coupling". http://dev2dev.bea.com/pub/a/2004/02/-orchard.html, 1 page. 2004.

[31] Padovitz, A., Krishnaswamy, S., and Loke, S.W. "Towards Efficient Selection of Web Services". Workshop on Web Services and Agent-based Engineering (WSABE), at 2nd International Conference on Autonomous Agents and Multi-agent Systems (AAMAS), 9 pages. Melbourne, Australia. 2003.

[32] Papazoglou, M.P., Kramer, B.J., and Yang, J. "Leveraging Web-services and Peer-to-Peer Networks". In Proceedings of the 15th Conference on Advanced Information Systems Engineering (CAiSE '03), pages 485-501. 2003.

[33] Ran, S. "A model for web services discovery with QoS". ACM SIGecom Exchanges, Vol. 4, No. 1, pages 1-10. 2003.

[34] Ritter, J. "Why Gnutella Can't Scale. No, Really". http://www.darkridge.com/~jpr5/-doc/gnutella.html. 1 page. 2001.

[35] Rowstron, A. and Druschel, P. "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems". IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), pages 329-350. Heidelberg, Germany. 2001.

[36] Samtani, G. & Sadhwani, D. "Web Services and Peer to Peer Computing". http://www.webservicesarchitect.com/content/articles/samtani05.asp, 1 page. 2002.

[37] Schlosser, M., Sintek, M., Decker, S., and Nejdl, W. "A Scalable and Ontology-Based P2P Infrastructure for Semantic Web Services". Second International Conference on Peer-to-Peer Computing (P2P'02), pages 104-111. Linkoping, Sweden. 2002.

[38] Schneider, J. "Convergence of Peer and Web Services". O'Reilly's Open Source Convention, 1 page. 2002.

[39] Smith, R.G. "The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver". IEEE Transactions on Computers, C-29(12), pages 1104-1113. 1980.

[40] Toma, I., Sapkota, B., Scicluna, J., Gomez, J.M., Roman, D., and Fensel, D. "A P2P Discovery mechanism for Web Service Execution Environment". In Proc. of the 2nd Int'l WSMO Implementation Workshop (WIW 2005), 10 pages. Innsbruck, Austria. 2005.

[41] Tyson, J. "How the Old Napster Worked". Marshall Brain's HowStuffWorks. http://www.howstuffworks.com/napster1.htm, 4 pages. Accessed on January, 2006.

[42] UDDI4J open-source project, http://sourceforge.net/projects/uddi4j - Accessed on August, 2005.

[43] UDDI Browser, http://www.uddibrowser.org - Accessed on August, 2005.

[44] Verma, K., Sivashanmugam, K., Sheth, A., Patil, A., Oundhakar, S., and Miller, J. "METEOR-S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services". Journal of Information Technology and Management, 24 pages. 2004.

[45] Vu, L., Hauswirth, M., and Aberer, K. "Towards P2P-based Semantic Web Service Discovery with QoS Support". Technical report, 15 pages. 2005.