Developing a Web Service Based Application for Mobile Client

Ting Wu

Pin Zheng

Supervisor & Examiner

Associate Prof. Vladimir Vlassov

KTH/ICT/ECS

Master of Science Thesis
Stockholm, Sweden 2006

ICT/ECS-2006-138

Abstract

Web service is one of the most widely discussed technologies to appear in recent years. But what is Web service? In simplest terms, Web service is an application, which can provide a defined set of functionality to achieve a specific end. It can provide application-to-application communications without human assistance or intervention. Web service give companies an unprecedented opportunity on the e-business aspect, from the airline Web shops to the patients who can access to online medical services from his cell phone, Web service are becoming a part of daily life.

By investigating Web service technology and developing an application to provide completely platform and language independent services we have achieved deep understanding of involved knowledge, a core set of standards, including XML (Extensible Markup Language), SOAP (Simple Object Access Protocol), WSDL (Web service Description Language), UDDI (Universal Description Discovery and Integration).

The main delivery of this project is a prototype of Web service which is implemented in Java language. And also a simple Midlet application to access the web service from Java enabled mobile client. We use Apache Axis as SOAP engine on the server side while a light kSOAP engine on the client side. The server side prototype is also used in the test and evaluation of the deployed Web service.

Keywords: Web Service, XML, WSDL, SOAP, UDDI, AXIS, kSOAP, J2SE, J2ME

Contents

1. Introduction....1
1.1 Background....1
1.2 Goals....3
1.3 Thesis Layout....3
2. Overview of Web service Technologies....4
2.1 Introduction....4
2.2 Web service Architecture....5
2.3 Understanding XML....6
2.3.1 Overview....6
2.3.2 XML Namespace....7
2.3.3 XML Schema....9
2.4 Understanding SOAP....11
2.4.1 SOAP Message....12
2.4.2 SOAP HTTP binding....13
2.5 Understanding WSDL....14
2.5.1 Overview....14
2.5.2 WSDL Elements....16
2.6 Understanding UDDI....21
3. Requirement Analysis....23
3.1 Identify System Scenarios....23
3.2 Identify Use Cases....24
4. System Design....26
4.1 Service Endpoint Design....26
4.1.1 Designing Service Interface....27
4.1.2 Processing Client Requests....29
4.1.3 Delegating Requests to Business Logic....30
4.1.4 Formulating Responses....31
4.2 Client Design....31
4.2.1 Locate and Access the Service....33
4.2.2 Formulate a Call....37
4.2.3 Process the Return Values....40
4.2.4 Handle Exceptions....40
5. System Implementation....43
5.1 Architecture Overview....43
5.2 Server Implementation....44
5.2.1 Technologies and Development Platform & Environment....44
5.2.2. Database Structure....46
5.2.3 Create Web service with Apache Axis....48
5.2.4 Implement Java Mail....56
5.2.5 Implement Java SMS....57

5.2.6 Implement Java Scheduling 61
5.3 Client Implementation 62
5.3.1 Technologies and Develop platform & Environment 62
5.3.2 Create MIDlet with Toolkit 65
6. Test and Evaluation 73
6.1 Test-bed Environment 73
6.2 Function Validation 73
6.3 Performance Evaluation 78
6.3.1 XML Message Size 78
6.3.2 Request Type 82
6.4 Scalability Analysis 83
6.4.1 Number of Requests 84
6.4.2 Mixture Requests 86
7. Conclusion and Future Work 89
7.1 Conclusion 89
7.2 Future Work 89
Abbreviation 91
Reference 92
Appendix WSDL Document 95

List of Figure

Figure 2-2- 1 Web service architecture 5

Figure 2-3- 1 Sample XML document 7
Figure 2-3- 2 Sample XML document with name conflict 8
Figure 2-3- 3 Improved sample XML document with namespaces 9
Figure 2-3- 4 XML schema sample......10
Figure 2-3- 5 Graphic diagram of PO schema document......11
Figure 2-3- 6 XML schema and XML document......11

Figure 2-4- 1 Simple SOAP message......12
Figure 2-4- 2 SOAP request message 12
Figure 2-4- 3 SOAP response message 13
Figure 2-4- 4 SOAP HTTP binding 14

Figure 2-5- 1 Web service interfaces to the resource 14
Figure 2-5- 2 Group messages into operations in Web service 15
Figure 2-5- 3 Binding in Web service 15
Figure 2-5- 4 Basic structure of a WSDL definition [19]......16
Figure 2-5- 5 Type element in WSDL file [19] 17
Figure 2-5- 6 Message element in WSDL file [19] 18
Figure 2-5- 7 Operation types in Web service......18
Figure 2-5- 8 PortType element in WSDL file [19] 19
Figure 2-5- 9 Binding element in WSDL file [19] 20
Figure 2-5- 10 Service element in WSDL file [19] 21

Figure 2-6- 1 Role of UDDI registry in Web service 21

Figure 3-1- 1 Identified actors of virtual shop system 23

Figure 3-2- 1 Identified use cases of customer and scheduler 24
Figure 3-2- 2 Identified use cases of administrator......25
Figure 3-2- 3 Identified use cases of shop owner......25

Figure 4-1- 1 Layered view of Web service 26
Figure 4-1- 2 XML type map to Java type 28
Figure 4-1- 3 Define service-specific exception 29
Figure 4-1- 4 Synchronous Web service 30

Figure 4-2- 1 Web service clients in Java and Non-Java platform......32
Figure 4-2- 2 Static stub mode......33

Figure 4-2- 3 Dynamic proxy mode....33
Figure 4-2- 4 Dynamic invocation interface model....34
Figure 4-2- 5 "say hello" Web service WSDL file....35
Figure 4-2- 6 Client code by using static stub....35
Figure 4-2- 7 Client code by using dynamic proxy....36
Figure 4-2- 8 Client code by using DII....37
Figure 4-2- 9 Standard mappings from WSDL to Java....38
Figure 4-2- 10 Complex mappings from WSDL to Java....39
Figure 4-2- 11 Complex data type response....39
Figure 4-2- 12 kSOAP read the complex data type....40
Figure 4-2- 13 User specified exception in WSDL....41
Figure 4-2- 14 Catch user specified exception....41
Figure 4-2- 15 kSOAP handle the exception....42

Figure 5-1- 1 System architecture overview....43

Figure 5-2- 1 SOAP message processing cycle [31]....45
Figure 5-2- 2 shopdb database design....46
Figure 5-2- 3 Package structure....48
Figure 5-2- 4 Service interface....50
Figure 5-2- 5 Java2WSDL command....50
Figure 5-2- 6 WSDL file (1/5)....51
Figure 5-2- 7 WSDL file (2/5)....51
Figure 5-2- 8 WSDL file (3/5)....52
Figure 5-2- 9 WSDL file (4/5)....52
Figure 5-2- 10 WSDL file (5/5)....53
Figure 5-2- 11 WSDL2Java command....54
Figure 5-2- 12 WSDL Mapping to Java [40]....55
Figure 5-2- 13 Web service deploy command....56
Figure 5-2- 14 JavaMail example....57
Figure 5-2- 15 Sample transmitter session [42]....58
Figure 5-2- 16 JavaSMS example....59
Figure 5-2- 17 SMPPSim start....60
Figure 5-2- 18 Tomcat console window when sending a SMS....60
Figure 5-2- 19 SMPPSim console window when sending a SMS....61
Figure 5-2- 20 Java scheduling example....62

Figure 5-3- 1 The J2ME stack [24]....63
Figure 5-3- 2 Life cycle of a MIDlet [48]....64
Figure 5-3- 3 ClientMidlet class....66
Figure 5-3- 4 Using kSOAP to formulate Web service call....67
Figure 5-3- 5 Main toolkit interface....68
Figure 5-3- 6 Creat a new MIDlet project....68
Figure 5-3- 7 Change the MIDlet project setting....68

Figure 5-3- 8 MIDlet Project created 69
Figure 5-3- 9 Test the MIDlet project (1/3) 70
Figure 5-3- 10 Test the MIDlet project (2/3) 70
Figure 5-3- 11 Test the MIDlet project (3/3) 70
Figure 5-3- 12 Run MIDlet via OTA 71

Figure 6-2- 1 Junit test case for Login Web service 75
Figure 6-2- 2 wsCaller user interface 76
Figure 6-2- 3 wsCaller test result 77

Figure 6-3- 1 Test setup 78
Figure 6-3- 2 Starting Apache JMeter 79
Figure 6-3- 3 Create Test plan and add thread group 79
Figure 6-3- 4 Create loop controller, SOAP sample and listener 80
Figure 6-3- 5 Service time with XML message size 81
Figure 6-3- 6 Service time with request type 83

Figure 6-4- 1 Thread group for number of requests 84
Figure 6-4- 2 Service time and throughput for number of requests 85
Figure 6-4- 3 Thread group for mixture requests 86
Figure 6-4- 4 Service time and throughput mixture requests 87

List of Tables

Table 5-2- 1 "userinfor" table 47
Table 5-2- 2 "shopinfor" table 47
Table 5-2- 3 "iteminfor" table 47
Table 5-2- 4 "cartinfor" table 47
Table 5-2- 5 "orderinfor" table 48
Table 5-2- 6 "wishinfor" table 48

Table 6-2- 1 "LoginService" function validation 74

Table 6-3- 1 Summary report for XML message size 81
Table 6-3- 2 Summary report for request type 83

Table 6-4- 1 Summary report for number of requests 85
Table 6-4- 2 Summary report for mixture requests 87

1. Introduction

This report is a research work towards “A Web-services based application for mobile client” within the framework of the master thesis project at the Department of Electronic, Computer and Software Systems (ECS) of the Royal Institute of Technology (KTH), Stockholm, Sweden.

This report is focusing on investigating Web service technologies and developing an application prototype to provide completely platform-and language-independent Web service to the mobile users. This includes achieving the deep understanding of involved technical knowledge, such as Simple Object Access Protocol (SOAP) [1], Web Service Description Language (WSDL) [2], Universal Description, Discovery, and Integration (UDDI) [3], Extensible Markup Language (XML) [4] and also implementing an application prototype to demonstrate the above involved Web service concepts.

1.1 Background

As Internet continues to grow, no matter the user or the support technology, there are more and more organizations and enterprises have involved in this “growth”. They don’t only satisfy with the current functionality provide by the Internet, but also demand more simplicity, flexibility and interoperability when they want to communicate with each other. So, from the late of the last century till now, various technologies come out to content the increasing needs of the organizations and enterprises. Distributed computing technology is one of the most importances. Actually Web service represents the evolution of distributed computing technology.

You may first ask: What is Web service? Is it a “service” provided by some organizations? Is it based on some standards or protocols? How can we find and use it? There are many definitions have been made about Web service however most of them have some commons based on answer to those questions.

- Web service exposes its functionalities to Web users via an open standard web protocol and this protocol is usually SOAP.
- Web service provides a standard way to describe the interface of service in order to let users talk to the service and build a client to use the service. It uses a document called Web Service Description Language (WSDL) based on XML to do this.
- Web service makes it available by registration so the potential users can find it easily. This is often done by Universal Discovery Description and Integration (UDDI).

All these three technologies (SOAP, WSDL, and UDDI) are the core and fundamental concept in Web service and all of them are based on eXtensible Markup Language (XML) [5]

After defining the Web service you may ask again: Why do we need it? Why do we need such kind of technology while the current Web technologies can provide strong and all-around functionalities? The answer is that Web service represents the evolution of past distributed computing technologies such as RPC, ORPC (DCOM, Java RMI, and CORBA) and even the modern Web application technology. However, all these technologies fail to become the ubiquitous platform for building distributed application. The key reason that Web service achieve success is that it is easier to implement and can provide greater interoperability. All the past distributed computing technologies require each participated machine to have a very complex run-time to make the underline mechanism completely transparent. Instead of focusing on providing minimalist platform, they want to implement everything. And what's even worse are they implement the functionalities in different way. They have their own communication protocol, data format, description way, discovery mechanism. Obviously, this is not good for interoperability which is the ultimate goal of Web service. Web service provides a structured way to format data, handle transactions, describe what the service does and make the service available to others. These entire make Web service provide a means for software to interoperate across programming languages, platforms, and operating system. Compare to the present Web technology, most of which use client-server architecture, Web service does not need humans to be involved. On the client side, it could be any Internet-enabled device or even a Web service to send the Web service request which could be a machine to machine communication. And that Web service can issue requests to another Web service which leads to what is called the n-tier application model.

Now you can see why Web service leaves the similar technologies behind and the next logic question may come: What can I do with Web service? What can Web service do for me? As mentioned above, interoperability is the ultimate goal, which has been a major concern across the industry over the past decades. There are two main areas where interoperability is a great challenge: Enterprise Application Integration (EAI) and Business-to-Business Integration (B2Bi).

EAI represents the challenge most enterprises face in integrating their various applications with each other. These various systems surely need to communicate and exchange information to serve the needs of the enterprise. It's always been a challenge to make this happen effectively.

B2Bi represents the business interactions between different enterprises. If one business wants to purchase supplies from another, they have to interact and exchange information—and they rarely happen to be using the same technology. Many organizations want to extend their reach to users. But users don't like being limited

into any particular platform or technology, so interoperability becomes an even bigger challenge than ever before. [6]

And Web service is coming to offer a complete platform for distributed application development that facilitates interoperability. It is the answer to these real business needs. It provides a standard way for the software within different business system to interact with greater ease.

1.2 Goals

The goal of this master project is to achieve deep understanding of Web service technology and proficiency in developing a Web service in Java2 platform as well as improving development skills in Java language.

In order to realize these goals several objectives have been set. First, make investigations to different protocols and specifications in Web service technology. This includes the core and fundamental protocols and languages: SOAP, WSDL, UDDI and XML. And also some related specifications which aim to provide more important functionalities and features of a distributed system for Web service (like security, transactions, and so on).

The second objective is to know and get familiar with the design and implement issues and limitations during developing Web service. In this master thesis, a prototype of Virtual Shop application will be implemented to demonstrate most of technical concepts involved in Web service, such as standard Web service definition, transmission of SOAP messages over HTTP, publish/discovery Web service (if needed) and etc. Also, this prototype will be the base for evaluating both performance and scalability of the implementation.

The third one is to design and implement a MIDlet application for J2ME enabled cell phone which could be the final consumer for the Web service built and deployed in the previous steps.

1.3 Thesis Layout

The thesis is structured as follows. Chapter 2 presents the survey of Web service technology, which covers XML, SOAP, WSDL, and UDDI respectively. The complete scenario analysis of the prototype for the Virtual Shop system and use cases are followed in Chapter 3. Chapter 4 describes the system analysis and design. In Chapter 5, the implement issues of the proposed design are presented and clarified. From Chapter 6, the thesis concentrates on validation and evaluation of the implement prototype and the conclusions combined with future work can be found in the last Chapter 7. The abbreviation, reference and appendix are attached at end of this thesis.

2. Overview of Web service Technologies

2.1 Introduction

Web service is rather a new technology to implement service-oriented architecture. The purpose of this technology is to provide a means for software to interoperate across programming language, platforms and developing environment.

The following are the core technologies used in Web service. These topics are covered in detail in the subsequent chapters.

- **XML** (eXtensible Markup Language): a markup language used to describe the data structure and mainly focus on what the data is. It is the base language for virtually all Web service standards.
- **SOAP** (Simple Object Access Protocol): a standard communication protocol for Web service. Due to the fact that SOAP message format is XML that make it a programming language, platform natural protocol. It allows the system to talk to each other.
- **WSDL** (Web Service Description Language): a standard mechanism to describe a Web service. A WSDL document specifies the operations a Web service provides, as well as the parameters and data types of these operations. It also provides the service interface and other access information to the service requestor.
- **UDDI** (Universal Description, Discovery, and Integration): a standard method to publish and discover Web service.

Besides SOAP, WSDL and UDDI, there have been a number of emerging standards evolved with the Web service progress. The purpose of all those new concepts is to give Web service increased functionality, reliability and security.

- **WS-Security**: As one of the specifications in Global Web service Architecture [7], it addresses how to maintain an end-to-end secure context. Besides ensuring that only authorized user can access the Web service, it also protects the SOAP messages sent and received by the appropriate parties from interruption, modification and fabrication.
- **WS-Coordination** [8]: Create a framework for supporting coordination protocols. Coordination protocols are designed to coordinate the actions of distributed applications and constrain the client to execute complex procedures in appropriate order.
- **WS-Composition**: Individual components implemented at different places and execute in different contexts. But, they need to communicate to yield desired behavior.[9]
- **WS-Resources**: Approach to declaring and implementing the association between a Web service and one or more named typed state components. When a

stateful resource is associated with a Web service and participates in the implied resource pattern, we refer to the component resulting from the composition of the Web service and the stateful resource as a WS-Resource. [10]

2.2 Web service Architecture

Web service represents a new model for software architecture. It is based upon the interactions between three roles: service ***provider***, service ***broker*** and service ***requestor***. The interactions involve ***publish***, ***find*** and ***bind*** operations, as show in Figure 2-2-1

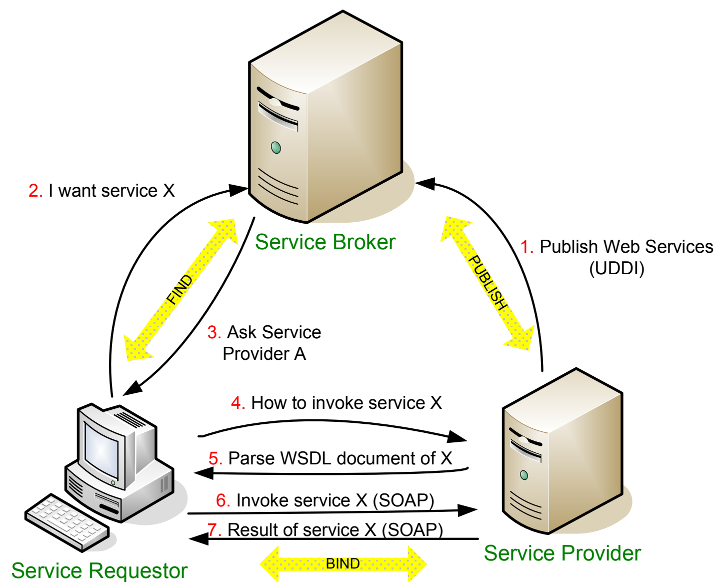


Figure 2-2- 1 Web service architecture

(1) Web service roles:

- Service Provider

  The service ***provider*** creates a Web service and possibly publishes its interface and access information to the service ***broker***. It is the owner of the service

- Service Requestor

  The service ***requestor*** looks for and invokes an interaction with a service. The service requestor can be a browser driven by a person or another program without any user interface, for example, another Web service.

- Service Broker

  Also is known as service ***register***. It is responsible to make a Web service interface and implement access information available to any service ***requestor***.

(2) Web service operations:

- Publish

  The service provider needs to ***publish*** the service description on the service broker so that the service requestor can find it. Sometime, the ***publish***

operation is optional if the service requestor knows where to find the service.

- Find
  In the ***find*** operation, the service requestor retrieves a service description directly or queries the service broker for the location of service required.
- Bind
  After finding a service, the service requestor can invoke the service by using the detailed information provided by service broker or the service provider itself. Such service invoke process is called ***bind***.

2.3 Understanding XML

2.3.1 Overview

As defined by World Wide Web Consortium(W3C) [12], eXtensible Markup Language (XML) is a simple, flexible text-based format developed from Standard Generalized Markup Language (SGML) and provide mechanisms for describing document structure using markup tags (word enclosed by '<' and '>').

Similar to HyperText Markup Language (HTML) documents, a XML documents is made up of elements, each of which consists of a start tag (such as `<table>`), content information and an end tag (such as `</table>`). However, XML is not a replacement for HTML. They designed for different purposes: while HTML was design to display data and show how data looks, XML was directed to describe data and to focus on what data is. It can be used to store, carry and exchange data between systems. Traditionally, when displaying a HTML document, the data is virtually stored inside of HTML. With XML, data can be stored in separate XML files and leave HTML to only concentrate on data layout and display. Since XML data is stored in plain text format, XML is also a cross-platform, software and hardware independent tool for exchanging information between systems, especially those which contain data in incompatible formats.

Figure2-3-1 shows an example XML document*.

(*: the code examples are referred from book "Build Web service with Java" [13] named "Purchase Order" (PO))

This first line in the document – the XML declaration – defines the XML version and the character encoding. The next line describes the root element of the document: `<po>` (indicate that this document is a "purchase order"). The root element has two attributes: `id` attribute identifies the purchase order and `submitted` attribute shows when the purchase order was made. The followed texts describe three child elements of the root (`billTo`, `shipTo` and `Order`).And finally the last line defines the end of the root element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- an Sample XML document: Purchase Order(PO)-->
  <po id="43871" submitted="2004-06-05">
     <billTo>
          <company>The Skateboard Warehouse</company>
          <street>One Warehouse Park, Building 17</street>
     </billTo>
     <shipTo>
          <company>The Skateboard Warehouse</company>
          <street>One Warehouse Park, Building 17</street>
     </shipTo>
     <order>
          <item sku="318-BP" quantity="5">
               <description>Skateboard backpack</description>
          </item>
     </order>
</po>
```

Figure 2-3- 1 Sample XML document

From the above example, we can get a brief understanding of a XML document. Here list some XML syntax rules. The whole XML 1.0 Syntax definition can be found in W3CXML 1.0 recommendation [4]

- XML tags are not predefined, instead that they are created by the author.
- There can be only one element as root element.
- All XML elements should have a closing tag.
- XML tags are case sensitive.
- XML elements are strictly nested and cannot overlap.
- Attribute values must be quoted.

2.3.2 XML Namespace

Due to the fact that all elements names in XML are not predefined and fixed, very often a name conflict will occur when two different elements use the same name. Figure2-3-2 shows a name conflict in the sample XML document. The description of `attachments:description` is named as the same as the description of `po:description`

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- an Sample XML document with name conflict   Purchase Order (PO)-->
<message from="mm@imit.kth.se" to="order@statestown.com" sent="2004-06-05">
    <text>This is my order, MM</text>
    <attachments>
        <description>PO</description>
        <item>
            <po id="44581" submitted="2004-06-05">
                <billTo id="addr-1">   ... </billTo>
                <shipTo href="addr-1"/>
                <order>
                    <item sku="316-BP" quantity="5">
                        < description>Skateboard backpack</description>
                    </item>
                </order>
            </po>
        </item>
    </attachments>
</message>
```

Figure 2-3- 2 Sample XML document with name conflict

XML namespaces in XML Recommendation is the W3C's solution to the name collisions. A namespace is a set of names in which all names are unique. It make possible to give elements and attributes unique names. For identifiers, XML namespaces must conform to the syntax for Uniform Resource Identifier (URI) references. Since there are two general forms of URI: Uniform Resource Locators (URL) and Uniform Resource Names (URN). Either type of URI may be used as a namespace identifier. Here is an example of three URLs that could be used as namespace identifiers:

```
http://www.w3c.org/2001/XMLSchema
http://statestown.com/ns/po
http://www.xcommerc.com/message
```

However, in an XML document, developers use a namespace prefix, which is just an abbreviation for the URI, to qualify the names of both elements and attributes. The syntax for a namespace declaration is:

```
xmlns : <prefix> = '<namespace identifier>'
```

And here are a few examples of namespace declaration by using namespace prefix:

```
xmlns: xsd = "http://www.w3c.org/2001/XMLSchema"
xmlns: po = "http://statestown.com/ns/po"
xmlns = "http://www.xcommerc.com/message"
```

Notice that there is no prefix in the third instance of above example. Actually, it declares a default namespace. When a default namespace declaration is used on an element, all unqualified elements name within its scope are automatically associated with the specified namespace identifier. The syntax for a default namespace declaration is:

```
xmlns = '<namespace identifier>'
```

An improved sample XML document is defied as following Figure 2-3-3.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Improved sample XML document with namespaces Purchase Order(PO)-->
<message from="mm@imit.kth.se" to="order@statestown.com" sent="2004-06-05" >
xmlns="http://www.xcommerc.com/message" xmlns:po=http://statestown.com/ns/po>
    <text>This is my order, MM</text>
    <attachments>
        <description>PO</description>
        <item>
            <po:po id="44581" submitted="2004-06-05">
                <po:billTo id="addr-1">...</po:billTo>
                <po:shipTo href="addr-1"/>
                <po:order>
                    <po:item sku="316-BP" quantity="5">
                        <po:description>Skateboard backpack</po:description>
                    </po:item>
                </po:order>
            </po:po>
        </item>
    </attachments>
</message>
```

Figure 2-3- 3 Improved sample XML document with namespaces

2.3.3 XML Schema

As introduced by previous chapter, a document which conforms to the rule of XML syntax is called "Well Formed" XML. Also, a document validated against a DTD/Schema is said to be "Valid" XML.

An XML schema describes the structure of an XML document and can support both

simple and complex user-defined data types. Once an XML documents had a reference to an XML schema, the schema-validator could check XML document's structure against the referred schema document.

Considering the previous Purchase Order (PO) sample, Figure 2-3-4 shows the PO schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Purchase Order(PO) schema example-->
<xsd:schema xmlns="http://www.skatestown.com/ns/po"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.skatestown.com/ns/po">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">Purchase Order schema</xsd:documentation>
    </xsd:annotation>
    <xsd:element name="po" type="poType"/>
    <xsd:complexType name="poType">
        <xsd:sequence>
            <xsd:element name="billTo" type="addressType"/>
            <xsd:element name="shipTo" type="addressType"/>
            <xsd:element name="order">
                <xsd:complexType>
                    <xsd:sequence>
                        <xsd:element name="item" type="itemType"/>
                    </xsd:sequence>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:positiveInteger" use="required"/>
        <xsd:attribute name="submitted" type="xsd:date" use="required"/>
    </xsd:complexType>
</xsd:schema>
```

Figure 2-3- 4 XML schema sample

A schema document starts with `xsd:schema`. The prefix `xsd` (XML Schema Definition) means that the element and data types with `xsd` as prefix come from "http://www.x3.org/2001/XMLSchema" namespace (such as `xsd:complexType`, `xsd:sequence`, `xsd:element`). Figure2-3-5 is the graphic diagram of corresponding PO schema document*.

(*: the graphic diagram is created by Altova XMLSpy2005)

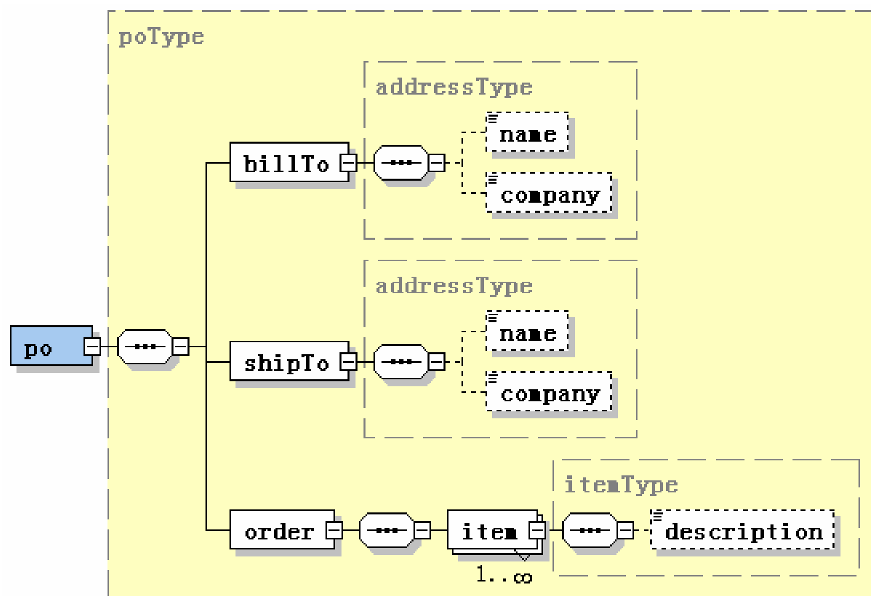


Figure 2-3- 5 Graphic diagram of PO schema document

As defined by this PO schema, there are some complex data type, such as `poType`, `addressType`, and `itemType`, and simple data type as well. The simple data type may have some restriction (`xsd:restriction`) or pattern (`xsd:pattern`).

After defining PO schema, the PO XML document may have a reference to such schema and enable validating parsers to check the PO XML document's structure against it. Figure 2-3-6 indicates how PO XML document refer to the PO schema.

```
<?xml version="1.0" encoding="UTF-8"?>
<po:po xmlns:po="http://www.skatestown.com/ns/po"
       xmlns:xsi="http://www.x3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.skatestown.com/ns/po
                          http://www.skatestown.com/schema/po.xsd"
       submitted="2004-06-10" id="12345">
...
</po:po>
```

Figure 2-3- 6 XML schema and XML document

Once the `xmlns:xsi` (XML Schema Instance namespace) is available, the `xsi:schemaLocation` can be used to indicate which namespace to use and the location of the XML schema to use for that namespace.

2.4 Understanding SOAP

If XML is the basic language for Web service, Simple Object Access Protocol (SOAP) is the grammar. SOAP enables the systems talk to one and another. According to

today's official definition from SOAP 1.2 Specification [1], SOAP is:

"A lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics."

The key points stated in this definition are 1). SOAP messages are based on XML technology 2). SOAP messages can be transmitted over a variety of underlying networking protocols. 3). SOAP is independent from platform, programming language, and development environment. (See Figure 2-4-1)

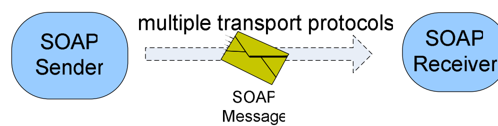


Figure 2-4- 1 Simple SOAP message

2.4.1 SOAP Message

Let us first look at a sample SOAP message, shown as Figure 2-4-2

```
POST /axis/InventoryCheck.jws HTTP/1.0
Content-Type: application/soap+xml; charset=utf-8
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:xsd=http://www.w3.org/2001/XMLSchema
      xmlns:soapenv="http://www.w3.org/2003/05/soap-envelope/"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
     <soapenv:Body>
         <doCheck soapenv:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
            <arg0 xsi:type="soapenc:string" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
                  947-TI
            </arg0>
            <arg1 xsi:type="soapenc:int" xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/">
                  1
            </arg1>
         </doCheck>
     </soapenv:Body>
</soapenv:Envelope>
```

Figure 2-4- 2 SOAP request message

The first two lines indicate that the SOAP message is transmitted over HTTP protocol.

The detail information about SOAP binding with HTTP can be found in followed Chapter 2.4.2.

The sample SOAP message is a SOAP request to check inventory availability given a product: `947-TI` and a desired product quantity: `1`. Using this sample, we will go through the different elements building a SOAP message.

- ***env:Envelope***: the root element of a SOAP message. It identifies the XML document as a SOAP message.
- ***env:Header***: the optional header element provides additional information (like authentication, payment, etc) about the SOAP message.
- ***env:Body***: contain data and instructions intended for the ultimate receiving application. In the above SOAP request message example, the SOAP body is used to invoke `doCheck` operation and transmit two required parameters. A SOAP response could look like Figure 2-4-3: SOAP response message
- ***env:Fault***: Fault element represents errors within the Body element when things go wrong.

```
HTTP/1.0 200 OK
Content-Type: application/soap+xml; charset=utf-8
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:xsd="http://www.w3.org/2001/XMLSchema"
                xmlns:soapenv=http://www.w3.org/2003/05/soap-envelope/
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <soapenv:Body>
        <doCheckResponse soapenv:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
         <rpc:result xmlns:rpc="http://www.w3.org/2003/05/soap-rpc">  </rpc:result>
            <return xsi:type="xsd:boolean">true</return>
        </doCheckResponse>
    </soapenv:Body>
</soapenv:Envelope>
```

Figure 2-4- 3 SOAP response message

2.4.2 SOAP HTTP binding

Due to the fact that SOAP messaging framework is independent from the transport protocols, SOAP messages can be transmitted over a variety of protocols, such as HTTP, SMTP and FTP etc. However, most developers use HTTP as the transport protocol for SOAP messages, because HTTP can be supported by all network browsers and servers and traffic will not blocked by the firewall or proxy server. Also the SOAP 1.2 specification codifies the SOAP HTTP binding, due to HTTP wide use.

Naturally, SOAP request/response model will map to the HTTP request/response

model. Figure 2-4-4 illustrates how SOAP binds with HTTP. [16]

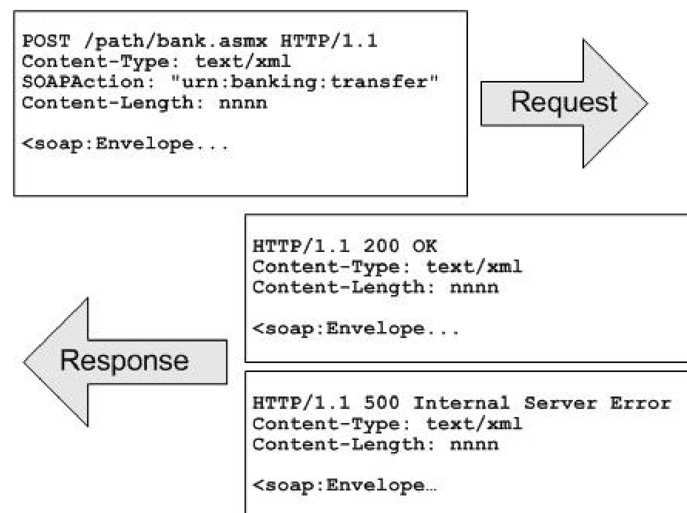


Figure 2-4- 4 SOAP HTTP binding

The ***Content-Type*** header for the SOAP request/response defines the MIME type for the message and it must be set to `text/xml` (`application/soap+xml` in SOAP 1.2). The ***Content-Length*** header specifies the number of bytes in the body of the request/response. The ***SOAPAction*** header is used to specify the URI that identifies the intent of the message. 200 status code in the HTTP response indicates that no error occurred while 500 status code expresses that the body contains a SOAP fault.

2.5 Understanding WSDL

2.5.1 Overview

Web Service Description Language (WSDL) is a XML based document to describe the complete details of application communication. Make the Web service approachable by using WSDL definition to generate code which knows exactly how to interact with the Web service described. It also hides the underline mechanism in sending and receiving SOAP message over different protocols. It makes the resource accessible by transmitting XML message over standard protocols like, HTTP and SMTP. However, the Web service can be seen like a piece of code which implements the XML interface to a resource as showed in Figure 2-5-1[19]

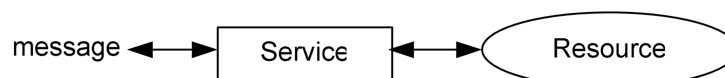


Figure 2-5- 1 Web service interfaces to the resource

In order to make it possible for any consumer with XML support to communicate with Web service applications, consumers must know the precise XML interface

along with other specific message details.

Consumers do not only need to know what message should be sent and received but also need to know the relations between these messages. That means consumers should know these messages as a group which include what message will be sent as requests and what message will be received as responses. This message exchange is also called as an operation which is what the consumers care most since it is the key point when interacting with a Web service application. And related operations can be grouped into interface which is also what the consumers must be aware of when they writing the codes to invoke the Web service (see Figure 2-5-2).

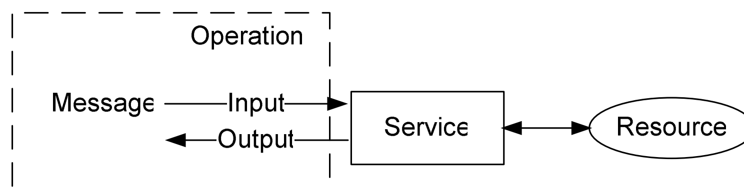


Figure 2-5- 2 Group messages into operations in Web service

After knowing what kind of message should be sent to invoke the Web service and received as a response, consumers must also know how to send these messages to service. What communication protocol and specific mechanics should be used? The binding tells all the details happening inside the communication wire by describing how to use an interface with a particular communication protocol and specifying the style of service (`document` or `RPC`) and encoding mechanism (`literal` or `encoded`) to determine the way the abstract messages are encoded on the wire.[19]

An interface of a service can be bind with multiple communication protocols but for each binding there should be a unique address identified by a URI (Web service endpoint) to make the binding accessible (see Figure 2-5-3)

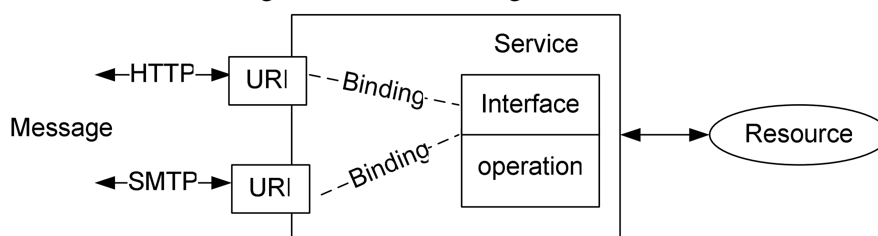


Figure 2-5- 3 Binding in Web service

Consumers have to know all the above details before they can interact with a Web service. WSDL describe all these details by using XML grammar to "*Provide a way to group message into operations and operations into interfaces and also a way to define bindings for each interface and protocol combination along with the endpoint address for each one. A complete WSDL definition contains all of the information necessary to invoke a Web service*" [19]

The WSDL1.1 [2] realizes the goal of "describing" Web service by using XML schema to define the "element" and "attribute" information inside the file. A WSDL

file defines the Web service as an integration of "service's endpoint" (where the actually implementation application runs). There are five main elements in XML schema used by WSDL1.1: `types`, `message`, `portType`, `binding`, `service` and `port`. And all these elements are under a root `definition` element. All of these elements come from `http://schemas.xmlsoap.org/wsdl/` namespace. Figure 2-5-4 is a basic structure of a WSDL definition.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- WSDL definition structure -->
<definitions name="MathService"
        targetNamespace="http://example.org/math/" xmlns="http://schemas.xmlsoap.org/wsdl/">
     <types> ...</types>
     <message> ...</message>
     <portType> ...</portType>
     <binding> ...</binding>
     <service> ...</service>
     <port> ...</port>
</definitions>
```

Figure 2-5- 4 Basic structure of a WSDL definition [19]

The `targetNamespace` is for what you name in your WSDL definition. And the first three elements (`types`, `message`, and `portType`) are abstract interface descriptions which you can interface with in your application code. They define the data format, the communication message and the way to exchange message. The last three elements (`binding`, `service` and `port`) describe the concrete details of how the abstract interface maps to message on the communication wire and where is the access point of the Web service.

2.5.2 WSDL Elements

The necessary elements in a WSDL file include:

- **element <types>**

`type` element is used to define the type of data exchanged between the client and server when a Web service is invoked. The default schema language for these definitions is XML Schema. However, WSDL1.1 allows using any schema language for this type definition but it is much better for interoperability when using XML Schema.

One can use any XML Schema construct within the schema element, such as simple type definitions, complex type definitions, and element definitions. The following WSDL fragment contains an XML Schema definition that defines one element of type `MathInput (AddRequest)` and one element of type `MathOutput (AddResponse)` (see

Figure 2-5-5)

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:y="http://example.org/math/"
             xmlns:ns="http://example.org/math/types/"
             targetNamespace="http://example.org/math/">
    <types>
        <xs:schema targetNamespace="http://example.org/math/types/"
                   xmlns="http://example.org/math/types/">
            <xs:complexType name="MathInput">
                <xs:sequence>
                    <xs:element name="x" type="xs:double"/>
                    <xs:element name="y" type="xs:double"/>
                </xs:sequence>
            </xs:complexType>
            <xs:complexType name="MathOutput">
                <xs:sequence>
                    <xs:element name="result" type="xs:double"/>
                </xs:sequence>
            </xs:complexType>
            <xs:element name="AddRequest" type="MathInput"/>
            <xs:element name="AddResponse" type="MathOutput"/>
        </xs:schema>
    </types>
  ...
</definitions>
```

Figure 2-5- 5 Type element in WSDL file [19]

After defining XML Schema types, the next step is to define the logical messages that will form the operations.

- ## element <message>

This element defines an abstract concept of Web service details which can serve as the input, output or error information of an operation. There are one or more parts in the message and each part is associated with either one element (when the style of service is `document`) or one type (when the style of service is `RPC`) (see Figure 2-5-6)

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:y="http://example.org/math/"
             xmlns:ns="http://example.org/math/types/"
```

```
            targetNamespace="http://example.org/math/">
    <message name="AddMessage">
        <part name="parameter" element="ns:AddRequest"/>
    </message>
    <message name="AddResponseMessage">
        <part name="parameter" element="ns:AddResponse"/>
    </message>
  ...
</definitions>
```

Figure 2-5- 6 Message element in WSDL file [19]

As mentioned at the beginning, type and message definition in WSDL file are considered to be abstract definitions which means you don't know how they will appear in the concrete message format until you have applied the binding to them.

- **element <portType>**

The `portType` (change to `interface` in WSDL2.0) element in WSDL defines a group of operations which are corresponding to the messages that defined in the `message` element including input and output message that will be transferred. It defines what the Web service actually does in an abstract way. We can see operation as an interface, a contract about how the service-requestor and the service-provider interact with each other to perform an action. In WSDL1.1 there are four main types of operation, which shown in Figure 2-5-7.

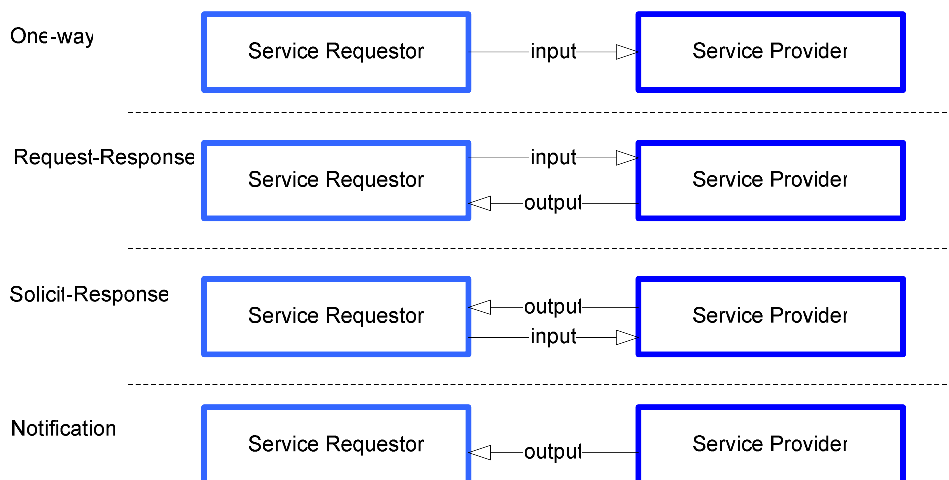


Figure 2-5- 7 Operation types in Web service

- ***One-Way***: The endpoint receives a message.
- ***Request-Response***: The endpoint receives a message and sends a correlated message.
- ***Solicit-Response***: The endpoint sends a message and receives a correlated message.
- ***Notification***: The endpoint sends a message.

The `input`, `output`, and `fault` elements used in an operation must refer to a message definition by name. Figure 2-5-8 is an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:y="http://example.org/math/"
             xmlns:ns="http://example.org/math/types/"
             targetNamespace="http://example.org/math/">
  ...
  <portType name="MathInterface">
        <operation name="Add">
             <input message="x:AddMessage"/>
             <output message="y:AddResponseMessage"/>
        </operation>
  </portType>
   ...
</definitions>
```

Figure 2-5- 8 PortType element in WSDL file [19]

● element <binding>

`Binding` element describes the details of associating a particular `portType` with a given protocol. This is implemented by extensibility elements which are defined outside of WSDL namespace as well as a WSDL operation element for each operation in the `portType` it's describing. The WSDL specification defines three kinds of bindings so there are three sets of extensibility elements for specifying binding information: SOAP, HTTP GET/POST, and MIME. And also you can define how to react when there is an error happen during an operation. Figure 2-5-9 is an example shows a SOAP/HTTP binding:

The `soap:binding` element indicates that this is a SOAP 1.1 binding. It also indicates the default style of service (possible values include `document` and `RPC`) along with the required transport protocol (HTTP in this case). The `soap:operation` element defines the `SOAPAction` HTTP header value for each operation. And the `soap:body` element defines how the message parts appear inside of the SOAP Body element (possible values include `literal` or `encoded`).

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:y="http://example.org/math/"
             xmlns:ns="http://example.org/math/types/"
             targetNamespace="http://example.org/math/">
  ...
  <binding name="MathSoapHttpBinding" type="y:MathInterface">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="Add">
             <soap:operation soapAction="http://example.org/math/#Add"/>
             <input>
                  <soap:body use="literal"/>
             </input>
             <output>
                  <soap:body use="literal"/>
             </output>
        </operation>
       ...
  </ binding >
  ...
</definitions>
```

Figure 2-5- 9 Binding element in WSDL file [19]

● element <service>

The `port` element is the sub-element of service. `service` element may contain one or more `port` elements. A `port` element exposes a unique address (URL) of a Web service which supports a particular binding. Each port represents an access point in order to invoke Web service, which is the service's endpoint in the service binding.
Figure 2-5-10 is an example defines a service called `MathService` that exposes the `MathSoapHttpBinding` at the `http://localhost/math/math.asmx` URL:

Each port has a unique name associated with a `binding`. One can use extensibility element within port to define the address details specific to the binding. [19]

Because WSDL is a machine-readable language, developers can use some tools to generate code that know exactly how to interact with Web service as WSDL has described. These generated codes hide the underline mechanism of sending and receiving SOAP message over different protocols. During our thesis project we use a tool named *WSDL2Java* from Apache Axis. It can generate the class files for consuming the service and also for implementing the service.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:y="http://example.org/math/"
             xmlns:ns="http://example.org/math/types/"
             targetNamespace="http://example.org/math/">
  ...
  <service name="MathService">
        <port name="MathEndpoint" binding="y: MathSoapHttpBinding">
             <soap:address location="http://localhost/math/math.asmx"/>
        </port>
    </service>
</definitions>
```

Figure 2-5- 10 Service element in WSDL file [19]

2.6 Understanding UDDI

The Universal Description, Discovery and Integration (UDDI) specification defines a framework for registering, deregistering and discovering Web service. It provides different services for different user, such as for the Web service provider, it enables the provider to register/deregister their services, and for the service requestor, UDDI provide a way to query services. Once a service is found, the UDDI registry plays no more roles between the requestor and provider. Figure 2-6-1 shows the different role play for UDDI registry

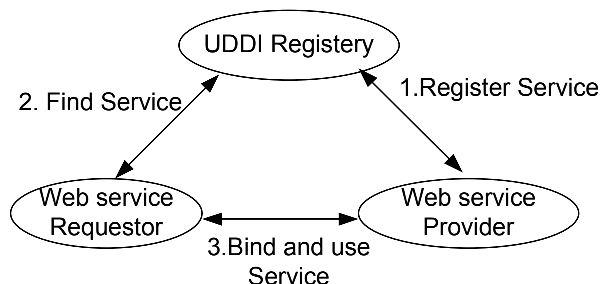


Figure 2-6- 1 Role of UDDI registry in Web service

To be able to register Web service with UDDI registry, a provider must furnish business, service, binding and technical information about the services. This information is kept in a common format that contains the following three parts:

- ***White pages***: contain general business information such as name, description.
- ***Yellow pages***: contain classification information about the types and location of the service the entry offers
- ***Green pages***: list the service, binding, and service-specific technical

information to invoke the offered services.

Besides, it also defines a set of data structures and API specification for programmatically registering and finding. Two categories of API are classified to enable access UDDI services from application, which are,

- UDDI ***Inquiry*** APIs: enable find registry entries and provide detail information
- UDDI ***Publisher*** APIs: enable add, modify and delete registry entries

For more information about UDDI, please refer to UDDI specification 3.0.1. [3]. How to use UDDI to publish Web service and let the user find from UDDI registry is out of the scope of this thesis project.

3. Requirement Analysis

3.1 Identify System Scenarios

As stated at the beginning, in order to realize the goal of this project we will design and implement a prototype of "Virtual Shop" system which is a typical Web service application. Through the development process, we illustrate how to design and implement a typical Web service application. Also, we will explain the motivational factors and limitations that need to be considered, and make these issues concrete by showing how we came to the decisions we eventually made as we architected the virtual shop system.

Figure 3-1-1 shows that there are 4 actors in the whole virtual shop scenarios: `Customer`, `ShopOwner`, `Administrator` and `Scheduler`. In this project, all the client side including customer, shop owner and administrator will use handsets as the consumer to the virtual shop Web service.

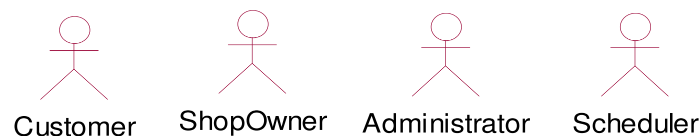


Figure 3-1- 1 Identified actors of virtual shop system

The `Customer` is actually a user of the virtual shop system. After successfully logged in with user name and password, the customer can browse any shop at his desire and buy various items that are maintained in a shopping cart. The shopping cart details can be viewed and items can be removed. The customer can also send shop creation request to administrator to ask for approve as the owner of his own shop. The customer also manages a wish list. If the customer can't find desired item in the entire virtual shop he can leave his desired item as a wish list to the shop. The wish list will then be stored in the data base with a pending status. Once that particular item is available, the customer will receive an offer list sent by shop through either a SMS or E-mail. The customer profile can be modified to alter the customer's personal information like phone number, address, etc.

The `ShopOwner` has the privileges for all activities of the requested shop, such as setting it up, add or remove items from the shop, hand the pending order and ship orders. He also has the right to discontinue the shop, which sends out a discontinue request to the administrator.

The `Administrator` is responsible for handing all shop creation requests. These requests can either be approved or rejected. Also, he can view all the available shop

information and search for specific shop by category.

As the matching mechanism, the `scheduler` is used to match customer's wish list with the items that are current available in the shop. Once the wish list is matched, the offer list is send to customer by either E-mail or SMS.

3.2 Identify Use Cases

When inspecting the system scenarios above, we realize that they cover a broad range of functionality initiated by many actors. We attempt to split them into self-contained and independent use cases initiated by single actor.

Figure 3-2-1 below illustrates the low-level use cases identified for both `Customer` and `Scheduler`.

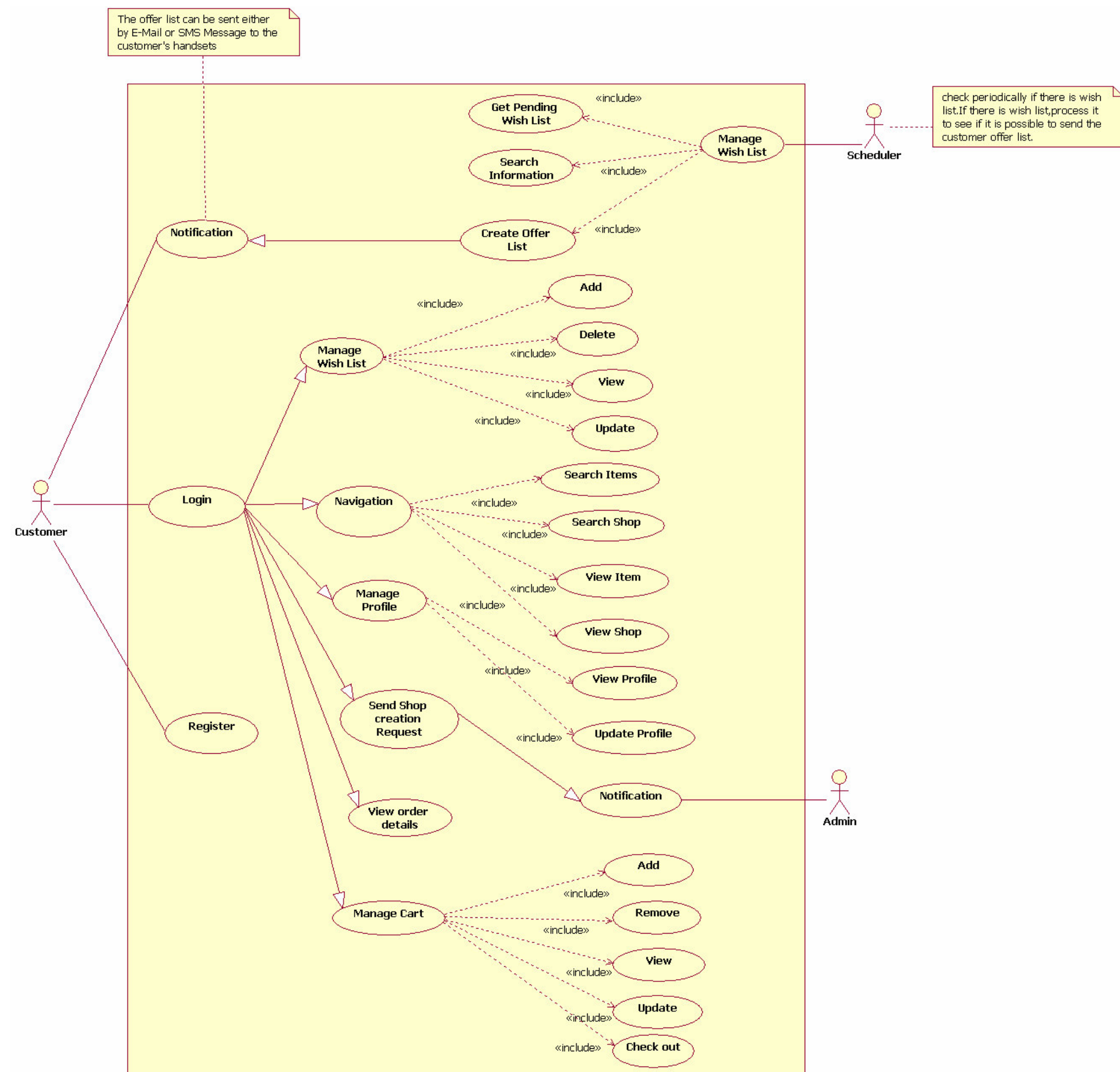


Figure 3-2- 1 Identified use cases of customer and scheduler

The Figure 3-2-2 illustrates the use case of `Administrator`. As the same case for customer, the `Administrator` needs to login the system as an administrator. He will be notified the shop creation requests sent by customer and view detail information of any shop at his desire.

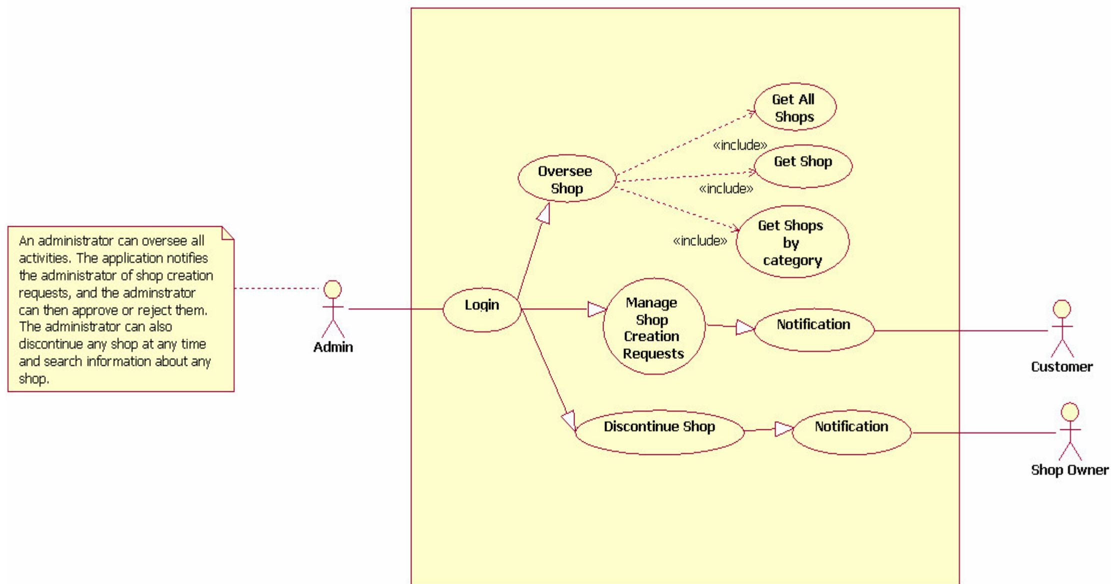


Figure 3-2- 2 Identified use cases of administrator

When the shop administrator approves a shop creation request, the requester is notified of the approval and from thereon the requestor assumes the role of a shop owner. Figure 3-2-3 reflects all the use cases initiated by `ShopOwner`.

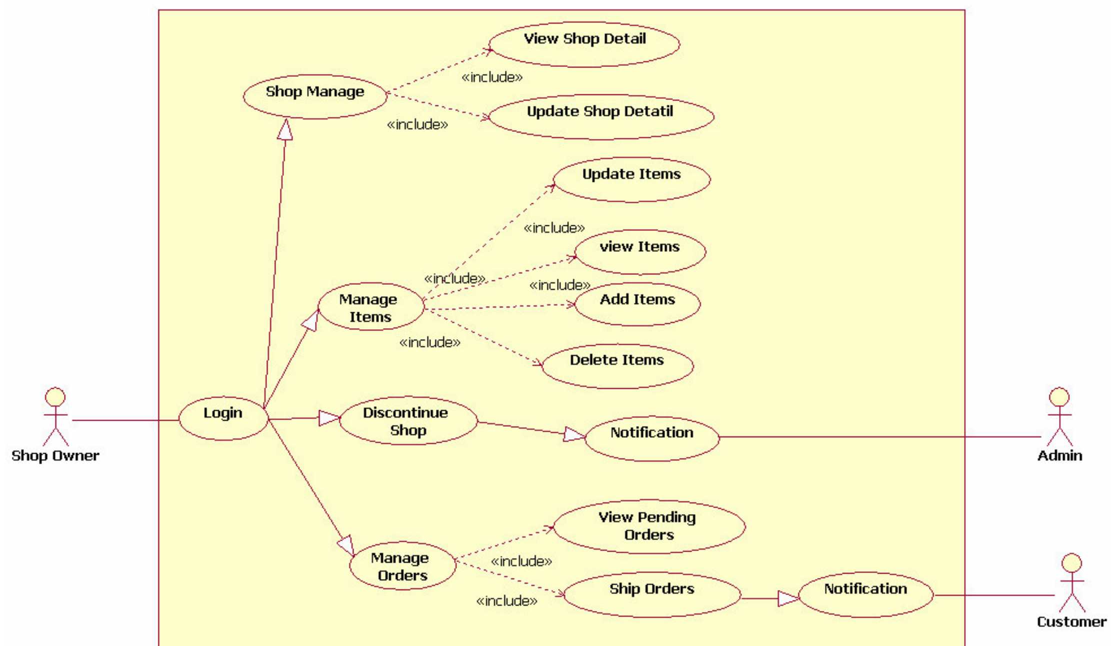


Figure 3-2- 3 Identified use cases of shop owner

4. System Design

Web service interacts with clients to receive client's requests and return responses. In between the request and the response, a Web service applies business logic of the application and fulfills a client's request. Therefore, an efficient Web service design starts from the deep understanding of services to be provided. That is how to expose an interface that client can use to make requests to the service, how clients compose the requests with correct data types and parameters, how to delegate the request to business logic to process the requests and finally how to formulate and send the response back to the clients. Certainly, Web service is also not immune from errors. How to throw out or handle exception from the normal execution needs to be considered as well.

4.1 Service Endpoint Design

Before exploring deep into service endpoint design issues, let's group a service implementation into two layers: a service interface layer and a business logical layer (see Figure 4-1-1).

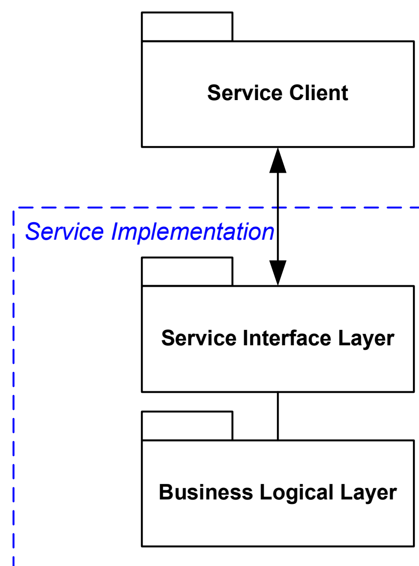


Figure 4-1- 1 Layered view of Web service

The service interface layer consists of the endpoint interface that the service exposes to clients and through which it receives client requests. It also includes any required preprocessing, for example, the convention between different data type, before the requests delegate to the lower business logical layer. On the other hand, after the business logical layer completes, service interface layer is still responsible for the preprocessing of the response received from lower and send proper formed response back to the clients.

The business logical layer holds all business logic used to process client requests. Usually, the design of Web service capabilities for an application is separate from designing the business logic of the application. Since the thesis is focusing on the studies of Web service, especially its nature of interoperability, how to design an effective business logical layer is out of scope.

The following four sections cover the major responsibilities taken by service interface layer. Firstly, a good design of the interface that the services present to the client is the start point because it is the only way through which the client can access. Besides, how to process the client request, how to delegate the processed requests to appropriate business logic, and finally how to formulate the correct response to the client are also the important issues that need to be handled. This section offers the general guideline for a Web service design.

4.1.1 Designing Service Interface

Before we deep into any design issues about service interface, let's introduce the JAX-RPC concept. JAX-RPC stands for Java API for XML-based RPC; it enables Java technology developers to develop SOAP based interoperable and portable Web service. Also, it provides a programming model to simplify the process of building Web service and map the XML types with Java types to hide the details for XML processing. In the JAX-RPC model, define a Web service may start from either Java code or a WSDL document. It is also possible to start with both a WSDL and a Java class, and define a Web service via customizations to either. [22]

- ***Java-to-WSDL***: building Java proxies and skeletons and data type from WSDL documents.
- ***WSDL-to-Java***: building WSDL from Java interface classes.

With the ***Java-to-WSDL*** approach, developers start with Java interfaces and generate a WSDL document without knowing much detail about WSDL. But the major drawback of the approach is that the developer loss the better control of WSDL document. Any change in WSDL might result in the developer going back to the Java interface and even require rewriting the service's clients. The changes, and the corresponding instability, greatly affect the interoperability of the Web service itself and limit the main advantages to use Web service.

On the other hand, the ***WSDL-to-Java*** approach leaves developers a more efficient way to expose a stable Web service interface. Certainly, a good knowledge of how to write a properly WSDL document, how to define the operations and corresponding signature and data format, how the service is accessible and where a service is located is required.

Compared both advantages and disadvantages, we use both approaches to design and implement our virtual shop interface. We first start with *Java-to-WSDL* to generate the basic WSDL file which servers as a template and *WSDL-to-Java* is later used to finalize all service design details to gain back the design flexibility and build more stable Web service interface.

After deciding on which service interface development approach, we also need to consider the parameter type for Web service operations, as each service interface exposes a set of operations to clients. As we introduced previously, in the Web service world, all the method calls and their parameters are sent as a SOAP message between the client and the service. This XML-formatted SOAP message is actually built from the parameters by the service requester and when received at the service end, they are converted back to their original proper types or objects. Fortunately, JAX-RPC gives us a better user-friendly mapping system. It enables the run-time system to map each XML type defied in WSDL to its corresponding Java types. Most simple XML data types are mapped directly to Java types, like Java primitive types: Boolean, byte, short, int and standard Java classes: String, Date, Calendar. Also JAX-RPC support complex types, like array and user-defined class. Figure4-1-2 shows the mapping samples.

XML type	Java type
<xs:element name="shopID" type="xsd:int" />	int shopID
<complexType name = "ArrayOfString"> <complexContent> <restriction base="soapenc:Array"> <attribute ref="soapend:ArrayType" wsdlArrayType="xsd:string[]"> </restriction> </complexContent> </complexType>	String[]
<xs:complexType name="shopType"> <xs:sequence> <xs:element name="shopID" type="xs:int"/> <xs:element name="shopName" type="xs:string"/> </xs:sequence> </xs:complexType>	public class ShopType implements java.io.Serializable { private int shopID; private String shopName; public ShopType() {} public int getShopID() {return shopID;} public void setShopID(int shopID) {this.shopID = shopID; } public String getShopName() {return shopName;} public void setShopName(String shopName) {this.shopName = shopName;}

Figure 4-1- 2 XML type map to Java type

Beside the interface type and parameter type, another important task in the Web service world is learning how to handle exceptions. How a Web service application responds to the error condition while processing a client request or the incorrect user input? Therefore, a completed mechanism to properly catch any exceptions thrown by an error and propagate these exceptions is highly required. According to JAX-RPC specification, a SOAP fault is mapped either to *javax.xml.rpc.soap.SOAPFaultException*, a *java.rmi.RemoteException,* or a service-specific exception class.

The *RemoteException* may happen from communications or runtime difficulties, like a network connection is down. The service-specific exception is usually designed to fit the application scenario. Let's examine it in the context of the virtual shop Web service example. When the client requests shop information for a nonexistent shop, the Web service should throw a user-defined exception, such as `ShopException` to the client that initiated the request. In the WSDL, the *wsdl:fault* element specifies the error messages that may be output as a result of a remote operation. Figure 4-1-3 shows a typical implementation of a service-specific exception.

XML definition

```
<wsdl:operation name="getShopDetail" parameterOrder="shopID">
  <wsdl:input name="getShopDetailRequest" message="impl:getShopDetailRequest"/>
  <wsdl:output name="getShopDetailResponse" message="impl:getShopDetailResponse"/>
  <wsdl:fault name="ShopException" message="impl:ShopException"/>
</wsdl:operation>
```

Java code

```
public ShopType getShopDetail(int shopID) throws RemoteException,ShopException;
public class ShopException  extends Exception
    private String shopErr;
    public ShopException() {
    }
    public ShopException (String shopErr) {
       this.shopErr = shopErr;
    }
    public String getShopErr() {
       return shopErr;
    }
    public void setShopErr(String shopErr) {
       this.shopErr = shopErr;
```

Figure 4-1- 3 Define service-specific exception

4.1.2 Processing Client Requests

Generally speaking, when the client requests, which are in the form of SOAP messages, arrive at server side, the server maps the received XML document to the

method call defied by the Web service interface. But before delegating the incoming requests to the Web service method call, some preprocessing work need to be done, like parameter validation, transformation, transformation of the incoming objects to the domain-specific objects, and etc.

If we have to parser or unparser the data in SOAP format by the programmer selves, Web service programming will be more difficult and unpractical. Luckily there are many Web service tool kits on the market also in the open source area which greatly simplifies Web service development. One such tool kit which is used in the thesis is Apache Axis [28]. Axis is the third generation of Apache SOAP implementation. Axis is working as a SOAP engine as well as a code generator and WSDL processing tool. By using Axis, the customer and service provider don't have to worry about the intricacies of SOAP message handing. Axis will convert the SOAP message and delegate the domain-specific objects to the Web service business logic. For more detail about Apache Axis, refer to section 5.2.1 Development platform

4.1.3 Delegating Requests to Business Logic

After the necessary request preprocessing, now we need to design how to delegate the request to the business logical layer. By the nature of HTTP transport, all the services in the virtual shop system is designed by synchronous manner, that is the invoking client blocks until the request is processed completely and the response is received. Let's take the `getShop` information service as an example. (See Figure 4-1-4)

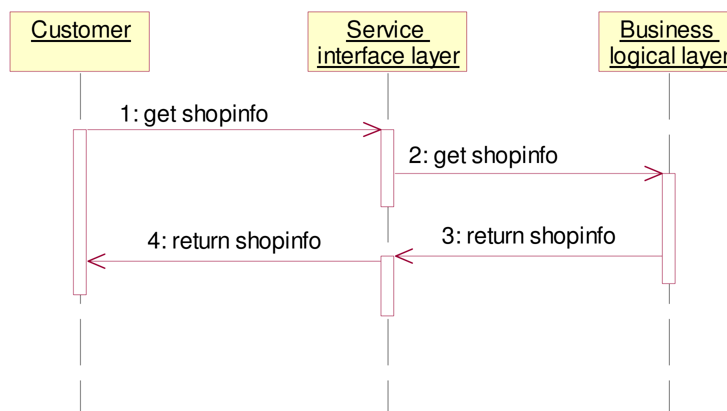


Figure 4-1- 4 Synchronous Web service

A client invokes the `getShop` service with the `shopID` and waits for the detailed shop information back. The look-up and return of the information can be done in a relatively short time, during which the client can be expected to block and wait. Typically, the synchronous Web service is useful when the client program required the data returned from the request immediately.

On the other hand, there are asynchronous services, which mean that after sending the request, the client doesn't need to wait for the response, instead, then client may continue with other processing. The service may send a response back to the client at a later time. Typically, asynchronous Web service is good choices when client send information that doesn't require a response. Consider the travel agency service from [22] that needs to collect data from many sources and match based on the results. The service performs such steps as user account verification, credit card authorization, accommodations checking, and itinerary building, and so on. Since the service needs a series of often time-consuming steps in the workflow, the client cannot afford to pause and wait for the steps to complete.

4.1.4 Formulating Responses

After we delegate the request to the business logical layer and the business logical complete it's processing, we are going to form the correct response to the request.

As we see in the previous chapter, we are using the SOAP engine Apache Axis to convert the SOAP message and delegate the domain-specific objects to the Web service business logic. After processing, Axis acts the same role, which gives the result from the business logic to its SOAP processor and parses the result into a SOAP response message.

4.2 Client Design

After the Web service are built and deployed on the server, it is time to write client program to access the services. The design of Web service client is largely different from the classic Client -Server design model due to the fact that to the client, a Web service is like a black box: the client doesn't have to talk to the server according to the pre-defined communication protocol, instead, the client only primarily cares about the functionality the service provides. Usually, the client gets to know the service functionality by WSDL file from the provider. Some developers even regard that deploying a Web service is all about publishing a WSDL file, which contains the definition of the Web service, the message that can be sent to the service and so forth. Therefore, the communication between service and client is platform, operating system, and programming language independent not only because the client has no concern on the service's platform, but also the client's implementation language is completely independent of the service. This thesis focuses on the Web service clients based on the J2ME platform.

Various clients running on different types of platforms can all access the Web service. We can design and develop a full functionalities client using J2EE technologies, or a rich Java application by standard J2SE, or even a light-weight application client which running on a J2ME-enabled phone connected to the Internet. Figure 4-2-1

shows how these different types of clients might access the Web service.

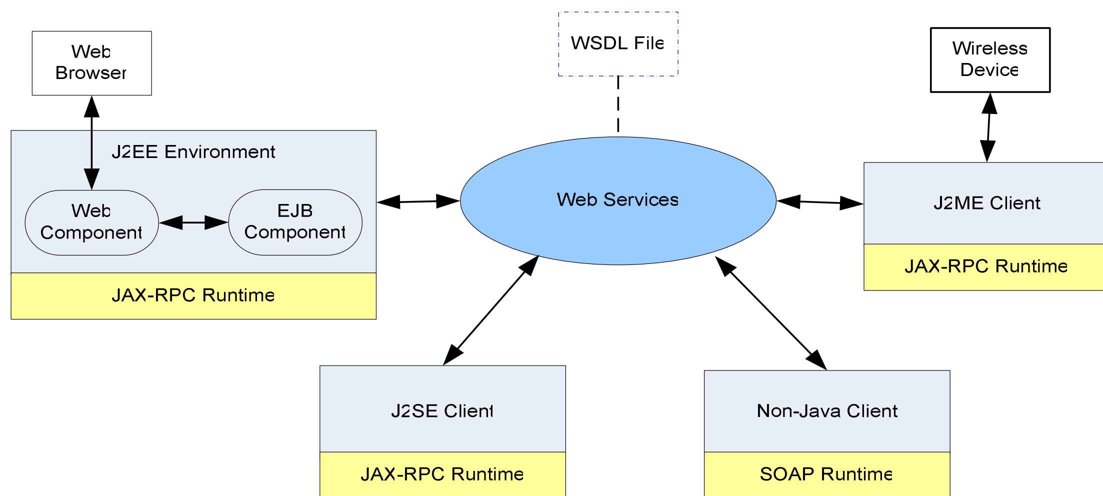


Figure 4-2- 1 Web service clients in Java and Non-Java platform

One of the goals in the thesis is to design and develop a J2ME client to enable the communication between J2ME-enabled mobile phone and the service. Therefore, this section will mainly address the steps taken in the J2ME client design and development.

Obviously, unlike the standard Java application running on the desktop clients and server, a J2ME client application, which is loaded on a small wireless device, needs additional consideration in its design.

First of all, due to the facts of limited amount of memory, GUI capabilities, process power, together with network connection, bandwidth limitation and so forth, design a good J2ME client is becoming a issue that concern not only basic design consideration for the wire device, but also other aspects, like how to limit the data exchange rate, reduce the amount of data transferred, simplify the user GUI design, take the data type exchanged that require less pre- and post-processing and etc.

Secondly, the traditional SOAP engine such as Apache Axis is far too large and resource-intensive to work on small device. It is not possible to expect the mobile device to work with MB sized package which is originally designed for desktop clients and servers. Therefore, the J2ME client typically uses only a small set of the JAX-RPC API. One of such subset used in the thesis is kSOAP [23], an open source project from Enhydra. It is a lightweight SOAP implementation especially suitable for J2ME program. Unlike the common SOAP packages which contain hundreds of classes, kSOAP is combined into a single jar file which takes up less than 42K and makes it become a lightweight SOAP implementation to enable SOAP and XML application to run within a low-memory virtual machine on a micro device.

The following sections will summary most of steps in developing a Web service client.

In brief, it is about how to locate the service, how to formulate a call to the specified service, and process and return data. Also, the client needs the capability to handle the exceptions occurring during the communication with the server.

4.2.1 Locate and Access the Service

There are three principal modes for a client to locate and access a Web service: static stub, dynamic proxy, and dynamic invocation interface (DII) [22]

- **Static Stub** – The static stub classes are generated from the JAX-RPC runtime toolkit, e.g. Apache Axis, wscompile, to enable the client to communicate with the service. Contrast to the remote "skeleton" object in the server, the stub is a local object that acts as a proxy for the service endpoint. Because this stub is created before runtime (by the IDE), it is called a static stub. The stub takes the main responsibilities to convert a request from the client to a SOAP message and send it to the service. It also converts the response from the service endpoint to a proper format understandable by the client. Figure 4-2-2 shows how the client use static stub produced by JAX-PRC toolkit to communicate with the server

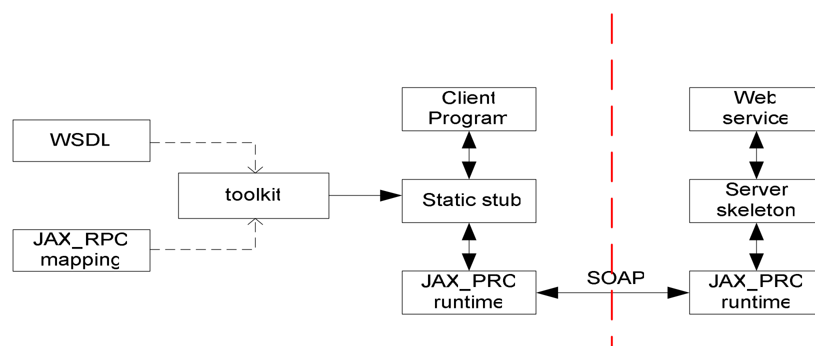


Figure 4-2- 2 Static stub mode

- **Dynamic Proxy** –As the same as the static stub, dynamic proxy provides a method for the client to access the service but in a more dynamic fashion. Dynamic proxy client supports a service endpoint interface dynamically at runtime without requiring any code generation of a stub class during development. The client gets information about the service by looking up its WSDL document and invokes the service directly.

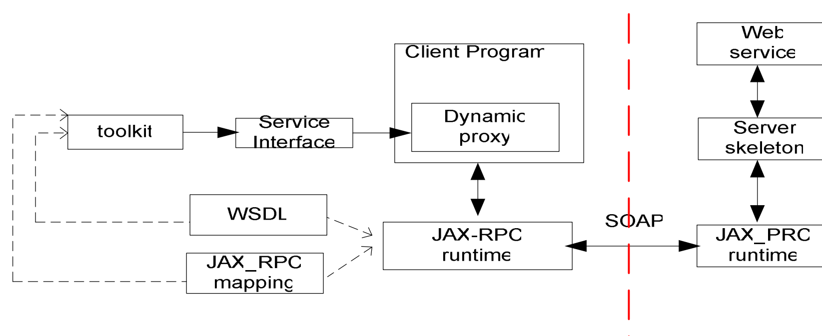


Figure 4-2- 3 Dynamic proxy mode

- **Dynamic Invocation Interface (DII) –** a good reason to use dynamic invocation is to support asynchronous communications, which allow the client application sends the request and free to do other work. It is used at compile time when a client does not have the knowledge about the remote service at the time the client application was written. Compare to the static proxy and dynamic proxy, DII doesn't need prior definition of a service endpoint interface, even the signature of the remote procedure or the name of the service is unknown. All the client knows is about a WSDL file. Obviously the DII client will be much complicated than the other two types of clients.

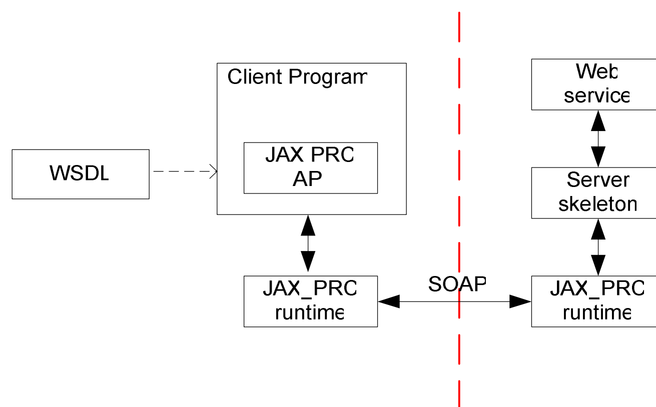


Figure 4-2- 4 Dynamic invocation interface model

In summary, in the static stub based method, both service interface and service implementation (static stub) are created at compile time; As to the dynamic proxy, only the service interface is created at compile time while implementation (dynamic proxy) is generated at runtime; Lastly, the DII method produces both service interface and implement at the runtime.

Another good example to explain the differences between above 3 types of communication models is to compare the client implementation code. Let us say that there is a "say hello" Web service, which basically receives the user name, and print out the "Hello" with user name string. And we also are able to obtain its WSDL file (as show in Figure 4-2-5):

By means of static stub, the client first uses the JAX-RPC runtime toolkit (ex. Axis) to generate the client side static stub, and then easily program the client code to invoke the service. Figure 4-2-6 is a sample client program with using the static stub created by Axis. In Axis, a client program would not instantiate a stub directly. It would instead instantiate a server locator (`HelloServiceLocator`) and calls a get method which returns a stub (`Hello`)

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
                   . . .
  <wsdl:message name="sayHelloRequest">
        <wsdl:part name="name" type="xsd:string"/>
  </wsdl:message>
  <wsdl:message name="sayHelloResponse">
        <wsdl:part name="sayHelloReturn" type="xsd:string"/>
  </wsdl:message>
  <wsdl:portType name="Hello">
        <wsdl:operation name="sayHello" parameterOrder="name">
          <wsdl:input name="sayHelloRequest" message="impl:sayHelloRequest"/>
          <wsdl:output name="sayHelloResponse" message="impl:sayHelloResponse"/>
        </wsdl:operation>
   </wsdl:portType>
   <wsdl:binding> . . .  </wsdl:binding>
   <wsdl:service name="HelloService">
        <wsdl:port name="Hello" binding="impl:HelloSoapBinding">
          <wsdlsoap:address location="http://localhost:8080/axis/services/Hello"/>
        </wsdl:port>
    </wsdl:service>
</wsdl:definitions>
```

Figure 4-2- 5 "say hello" Web service WSDL file

```java
Import Hello_pkg.Hello;
import Hello_pkg.HelloServiceLocator;
public class helloClient {
     public static void main(String[] args) {
       HelloServiceLocator lo= new HelloServiceLocator();
       try{
          Hello svc = lo.getHello();
          System.out.println(svc.sayHello("myWorld"));
       }catch (javax.xml.rpc.ServiceException se){
                  se.printStackTrace();
                  return;
       }catch (java.rmi.RemoteException re){
                  re.printStackTrace();
                  return;    }
     } }
```

Figure 4-2- 6 Client code by using static stub

On the contrary, a dynamic proxy class can be use to create client invocation without requiring pre-generation of the implementation-specific proxy class, which is the case in the static stub communication model. It calls a remote procedure through a

dynamic proxy, a class that is created during runtime. Figure 4-2-7 shows how the client code might use a dynamic proxy instead of a static stub to access a service. First, it creates a `Service` object named `ots` from the `createService` method by the `ServiceFactory` object. Secondly, the client code creates a proxy (`dyproxy`) with a type of the service endpoint interface (`Hello_pkg.Hello`), which is generated by Axis tool *WSDL2Java*.

```
import javax.xml.rpc.ServiceFactory;
import java.net.URL;
import javax.xml.rpc.Service;
import javax.xml.namespace.QName;
public class helloClientDy {
    public static void main(String[] args) {
        try{
        ServiceFactory sf = ServiceFactory.newInstance();
        String wsdlURI =
            "http://localhost:80/axis/services/Hello?WSDL";
        URL wsdlURL = new URL(wsdlURI);
        Service ots = sf.createService(wsdlURL,new QName("urn:Hello", "HelloService"));
        Hello_pkg.Hello dyproxy = (Hello_pkg.Hello)ots.getPort(new QName("urn:Hello",
"Hello"), Hello_pkg.Hello.class);
        System.out.println(dyproxy.sayHello("myWorld"));
        }catch (Exception e){
            e.printStackTrace();
            return;
        }   }}
```

Figure 4-2- 7 Client code by using dynamic proxy

As come to the DII communication model, the client code is more complex than others by the reason that the code solely depends on the WSDL file. The DII client does not require runtime classes generated by any toolkit, like Axis, or wscompile. Instead, the programmer needs to specify the operation name, operation parameters and return type. All the information is obtained from the WSDL file. Figure 4-2-8 illustrates a piece of client DII code.

In the thesis project, the kSOAP does not provide any support to generate static stub and service interface. The developer has to use the DII method to invoke the service on the server side. The good aspect to use DII is that the developers have complete control to client programmer; however the programming complexity is also raised. Typically, the client needs to create Call object first and set the operation and parameters during runtime.

```
import javax.xml.rpc.Call;
import javax.xml.rpc.Service;
import javax.xml.rpc.JAXRPCException;
import javax.xml.namespace.QName;
import javax.xml.rpc.ServiceFactory;
import javax.xml.rpc.ParameterMode;
public class helloClientDII {
   private static String BODY_NAMESPACE_VALUE =  "urn:Hello";
   private static String ENCODING_STYLE_PROPERTY =
       "javax.xml.rpc.encodingstyle.namespace.uri";
   private static String NS_XSD = "http://www.w3.org/2001/XMLSchema";
   private static String URI_ENCODING = "http://schemas.xmlsoap.org/soap/encoding/";
   public static void main(String[] args) {
      try {
         ServiceFactory factory = ServiceFactory.newInstance();
         Service service = factory.createService( new QName("HelloService"));
         Call call = service.createCall(new QName("Hello"));
         call.setTargetEndpointAddress("http://localhost:80/axis/services/Hello");
         call.setProperty(Call.SOAPACTION_USE_PROPERTY, new Boolean(true));
         call.setProperty(Call.SOAPACTION_URI_PROPERTY, "");
         call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);
         QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");
         call.setReturnType(QNAME_TYPE_STRING);
         call.setOperationName(new QName(BODY_NAMESPACE_VALUE,"sayHello"));
         call.addParameter("String_1", QNAME_TYPE_STRING, ParameterMode.IN);
         String[] params = { "myWorld!" };
         String result = (String)call.invoke(params);
         System.out.println(result);
      } catch (Exception ex) {
         ex.printStackTrace();
      }   } }
```

Figure 4-2- 8 Client code by using DII

4.2.2 Formulate a Call

Once a service is located, the client needs to formulate a call to invoke the service. If this is the case, the methods `_setProperty` and `_getProperty` defined by the `javax.xml.rpc.Call` are particularly useful. Figure 4-2-12 client code by using DII illustrates setting the properties on the `Call` interface.

Besides, the mapping between SOAP-defined types used by the service and the Java-defined types used by the client application is also an important issue when

formulate a service call. As we all understand, when a client invokes a service, the JAX-RPC runtime maps java type parameters to the corresponding SOAP representations and sends a SOAP message via HTTP request to the service. Once the service responds to the request, the JAX-RPC runtime need to map the SOAP type return values back to Java objects.

When stubs are used, take Apache Axis for example, the tool *WSDL2Java* maps parameters, exception and return values type from xml type into the generated classes. Complex types defined within a WSDL are represented by individual Java classes. Figure 4-2-9 shows the basic data types mapping from XML type to Java object. While Figure 4-2-10 show how the complex type in the WSDL type section is mapped to a new Java class. In addition, the *WSDL2Java* tool also handles the serialization and deserialization of the Java object to XML. With the benefit of WSDL to Java tool, the developer's work is extremely simplified and more standardized.

SOAP Type	Java Type
xsd:base64Binary	byte[]
xsd:Boolean	Boolean
xsd:byte	Byte
xsd:dateTime	java.util.Calendar
xsd:decimal	java.math.BigDecimal
xsd:double	Double
xsd:float	Float
xsd:hexBinary	byte[]
xsd:int	Int
xsd:integer	java.math.BigInteger
xsd:long	Long
xsd:QName	javax.xml.namespace.QName
xsd:short	Short
xsd:string	java.lang.String

Figure 4-2- 9 Standard mappings from WSDL to Java

XML complex type

```
<xs:complexType name="shopType">
         <xs:sequence>
              <xs:element name="shopID" type="xs:int"/>
              <xs:element name="shopName" type="xs:string"/>
              . . .
         </xs:sequence>
</xs:complexType>
```

generated Java class

```
public class ShopType  implements java.io.Serializable {
   private int shopID;
   private java.lang.String shopName;
   . . .
   public int getShopID()
   {  return shopID;}
   public void setShopID(int shopID)
   {   this.shopID = shopID;}
    . . .
}
```

Figure 4-2- 10 Complex mappings from WSDL to Java

As to another SOAP engine - kSOAP, which is specialized for microdevice, the support of data mapping is very limited. For the simple standard types, `xsd:string`, `xsd:long`, `xsd:int`, and `xsd:boolean`., kSOAP is able to take care of them by its own framework.

However, as to the customer-defined complex data types, the developers have to implement the mapping logic themselves with the help of existing kSOAP APIs. That is, if the SOAP element is a standard primitive type, kSOAP converts to a Java object of a matching type. However, if the SOAP element is a complex type, it converts to a `KvmSerializable` object. `KvmSerializable` is an interface; the kSOAP package provides the interface's convenience implementation: `SoapObject`. We have to set the element's property from `setProperty`() before sending the SOAP request and retrieve the element from the incoming SOAP packet through `getProperty()` method.

For example, to retrieve user profile information from the virtual shop system, the client invokes `getProfile`() service. After receiving the SOAP response (see Figure 4-2-11), which is in the user-defined complex data type format, the client code needs `getProperty()`method to readout the data.

```
<soapenv:Envelope
   xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema"
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
   xmlns:ns1="urn:Services:CS>
    <soapenv:Body>
      <userType xsi:type=”ns1:UserType”>
        <userID xsi:type="xsd:string">iw_pc</userID>
        <name xsi:type="xsd:string">pin</name>
        <password xsi:type="xsd:string">iw_pc</password>
        <email xsi:type="xsd:string">iw03_pzh@it.kth.se</email>
      </userType>
   </soapenv:Body>
</soapenv:Envelope>
```

Figure 4-2- 11 Complex data type response

Figure 4-2-12 demonstrate how to read the embedded data from the SOAP message body by using kSOAP APIs and the object `SoapObject`. It can be seen straightforward that the code just traces down the hierarchy tree using element names and the `SoapObject.getProperty()` method.

```
// Get the parsed structure
SoapObject userprofile = (SoapObject) envelope.getResult();
// Retrieve the values as first layer soapobjcet
String oneuser = (SoapObject) userprofile.getProperty ("userType ");
// Retrieve the values as appropriate Java objects
String userID = (String) oneuser.getProperty ("userID");
String name = (String) oneuser.getProperty ("name");
String password = (String) oneuser.getProperty ("password ");
```

Figure 4-2- 12 kSOAP read the complex data type

4.2.3 Process the Return Values

As we see above, once the client invoke the call and the service responds to the request, the JAX-RPC runtime needs to map the SOAP type return values back to Java objects. In the most cases, the client might like to display the result in a Web page using a HTML browser. The J2EE platform provides a rich set of component technologies to support such demands, such as JavaServer Pages (JSP) technology. However, as we are only going to build a demo J2ME application as Web service consumer in the thesis project, for the demonstration purpose, displaying the return values as Java object to the client application is already good enough.

4.2.4 Handle Exceptions

There are two types of exceptions for client applications that access Web service: system exceptions and user-specified exceptions, which are specific for each service [22].

System exceptions usually happen beyond the control of client application, such as invoking the service call using incorrect parameters, network error, or service inaccessibility. Some most common system exceptions are,

- `java.rmi.RemoteException`: the exceptions that may occur during the execution of a remote method call, such as, network failures or remote server unavailability or unreachablility. Service endpoint interface’s methods must throw `java.rmi.RemoteException` exception.
- `javax.xml.rpc.JAXRPCException`: the error thrown from the core JAX-RPC APIs to indicate an exception related to the JAX-RPC runtime mechanism. It often happens when using stubs to `_getProperty` and `_setProperty` for invalid,

inappropriate parameter values.

User-specified exceptions occur when a Web service call results in the service returning a fault. For example, to search a non-existing data, to add duplicated/invalid/incomplete data, etc. They are described in the service's WSDL file and are referred to as `wsdl:fault` elements. Figure 4-2-13 shows the `wsdl:fault` definition in a WSDL file to indicate that when a client may pass to the service an shop identifier that doesn't mach shop records kept by the server, the client may receive an `ShopException` exception.

```
<wsdl:operation name="getShopDetail" parameterOrder="shopID">
     <wsdl:input name="getShopDetailRequest" message="impl:getShopDetailRequest"/>
     <wsdl:outputname="getShopDetailResponse" message="impl:getShopDetailResponse"/>
     <wsdl:fault name="ShopException" message="impl:ShopException"/>
</wsdl:operation>
```

Figure 4-2- 13 User specified exception in WSDL

For a normal Web service client, which ruuning on J2SE or J2EE platform, we can use Apache Axis *WSDL2Java* tool to map faults to Java objects. It generates the necessary parameter mappings for the exception classes and generates the necessary classes for the mapping. All the exception classes extend `org.apache.axis.AxisFault`. Usually, it is the client application responsibilities to catch these checked exceptions and recover the program from the error state. Figure 4-2-14 is a piece of client code to handle cases where a matched shop ID is not found.

```
try {
     shopType shop = service.getShopDetail (shopId);
   } catch (ShopException shex) {
     JOptionPane.showMessageDialog(gui,
          "Shop Not found with shop ID " + shopId, "Error",
          JOptionPane.ERROR_MESSAGE);
   }
```

Figure 4-2- 14 Catch user specified exception

Come back to J2ME client with kSOAP as engine, unfortunally, the kSOAP does not have any mechanism to map the user defined exception to Java class. But we still can use org.ksoap.SoapFault class to catch SOAP faulty during the procedure.

```
import org.ksoap2.*;
try{
     SoapObject rpc = new SoapObject("urn:Services:LS","login");
     rpc.addProperty("userID",userID);
     rpc.addProperty( "pwd",password);
     SoapSerializationEnvelope envelope =
```

```
            new SoapSerializationEnvelope(SoapEnvelope.VER11);
    envelope.bodyOut = rpc;
    HttpTransport ht = new HttpTransport("http://localhost:8080/axis/services/Login");
    ht.call("urn:Services:LS#login", envelope);
    . . .
}catch (SoapFault sf){
    faultString = "Code: " + sf.faultcode + "\nString: " +sf.faultstring;
    mvLoginform. statusItem. setText(“Fault:” + faultString)
}
```

Figure 4-2- 15 kSOAP handle the exception

5. System Implementation

5.1 Architecture Overview

The system architecture is a typical three-tier style which organizes the system into three layers (see Figure 5-1-1):

- ***Interface Layer*** includes all the objects that deal with the end users, including windows, forms, Web pages and so on.
- ***Application Logic Layer*** realizes the application business logic, all the business logic design issues have been addressed
- ***Storage Layer*** realizes the storage, retrieval and query of persistent objects.

The one which we implemented in the business logic layer is using Tomcat Application Server with an Axis backbone. The Apache Tomcat processes the client requests and the axis backbone of Tomcat strips off the XML content of the request and reads the client request. The server is also treated as a services container, which contain all the deployed Web services and have the ability to interact with the storage layer, which is a MySQL database. According to the scenario analysis, the deployed services include: *LoginService*, *AdminService*, *OwnerService*, *CustomerService*, *MailService*, and *SMSService*. Also, a stand alone scheduler program is developed to check the database periodically and notify the customer by *MailService* or *SMSService* about the satisfaction of the wish list. Lastly, in order to demon the SMS function, a simulated SMSC (Short Message Service Center) is used to simulate the process of sending SMS from the *SMSService*.

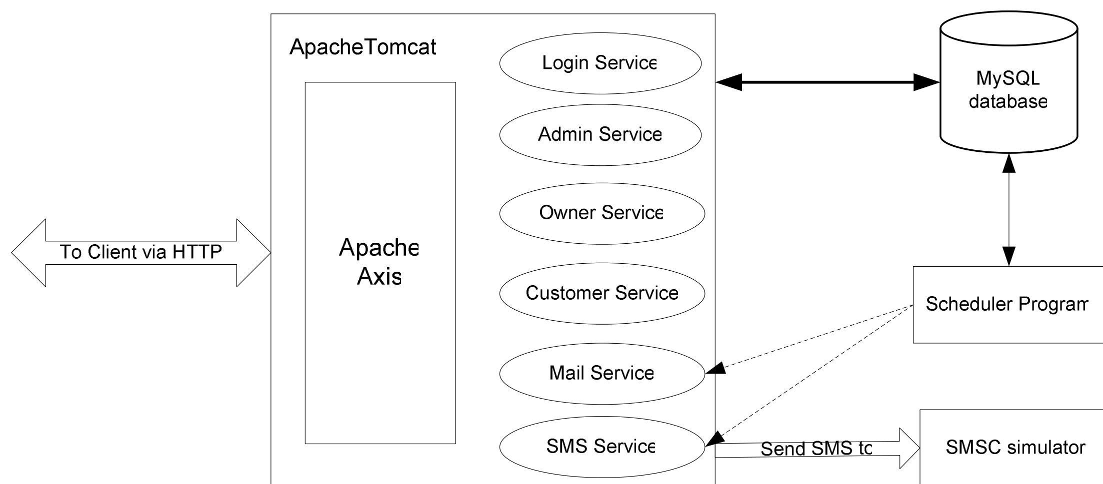


Figure 5-1- 1 System architecture overview

5.2 Server Implementation

5.2.1 Technologies and Development Platform & Environment

Several software tools are used to ease the development of the prototype. The target operation system is Microsoft Windows XP. Java is chosen as implementation language, the version and edition is Java 2 Platform, Standard Edition (J2SE), version1.4.2. The Java Integrated Development Environment (IDE) to develop the services is Eclipse 3.0 [26]. Besides, we are using the MySQL database server 4.0 [27] to manage the server's data in an easy, reliable and fast way. Apache Axis 1.1[28] is used as an implementation of SOAP submission to W3C. As a SOAP engine, Axis provides a mechanism for sending and receiving SOAP message packets to/from Web service clients and server. As to the application server, Tomcat 5.0.28 serves a reliable application server when deploy the Web service on it. In the following part of this section, the deep understanding of the above software tools is introduced.

(1) J2SE (V1.4.2)
Java Platform, with Standard Edition provides the developers a widely spread programming environment. It includes classes that support the development of Java applications, Java Web Service, and is the foundation for Java platform.

There are usually two principal products in the Java SE family: Java SE Development Kit (JDK) and Java SE Runtime Environment (JRE)

- ***JRE***: the Java Runtime Environment. It is an implementation of the Java Vritual Machine (JVM) which actually executes Java programs. For normal end users, install JRE is enough to run java based applet or application.
- ***JDK***: the Java Development Kit. Besides everything that in the JRE, it also has a bundle of software that is used to develop Java based software, plus necessary development tools, like command-line compiler and debugger. Typically, each JDK contains one (or more) JRE's along with the various development tools like the Java source compilers, bundling and deployment tools, debuggers, development libraries, etc. As a Java developer, JDK is the must component to be installed before doing any programming job.

(2) Tomcat Application Server (V5.0.28)
According to the official description of Apache Tomcat,

"Apache Tomcat is the servlet container that is used in the official Reference Implementation for the Java Servlet and JavaServer Pages technologies. The Java Servlet and JavaServer Pages specifications are developed by Sun under the Java Community Process."

Apache Tomcat powers numerous large-scale, mission-critical web applications across a diverse range of industries and organizations. Tomcat supplies the developers a platform to support the deployment and invocation of Web service developed in Java language. From a system point of view, Tomcat' is to receive the HTTP request from client and deliver it to the Web application, which might be a deployed a JSP or Servlet. The application interprets the HTTP request, performs the appropriate application business logic and generates a response, The Tomcat framework then, transport the response back to the client as a HTTP response.

(3) Apache Axis (V1.1)
As introduced previously, SOAP messages are usually exchanged via HTTP between systems to enable their communication over Internet. However, due to the fact that SOAP messages are XML format, which mean they are independent from programming language, there is a need for a SOAP engine to convert programming language objects, e.g. Java objects, and SOAP messages.

Apache Axis, the third generation open-source SOAP engine released by Apache Software Foundation (ASF) [30], aids both the client of Web service and their providers to accomplish their tasks without worrying about the complexity of SOAP messages handing. The only issue that service provider need to focus on is left to implement the business logic.

As shown in Figure 5-2-1, SOAP message processing cycle. When a client invokes a remote operation on the server side, the client's SOAP engine converts the method invocation into a SOAP message, which is transmitted through a transport, such as HTTP or SMTP, to the provider's SOAP processor, which parses the message into a method invocation. The provider then executes the appropriate logic and gives the result to its SOAP processor. The SOAP processor parses the information into a SOAP response message and transmitted it to the consumer. In turn, customer's SOAP processor parses the response message into a result object that it returns to the invoking entity.

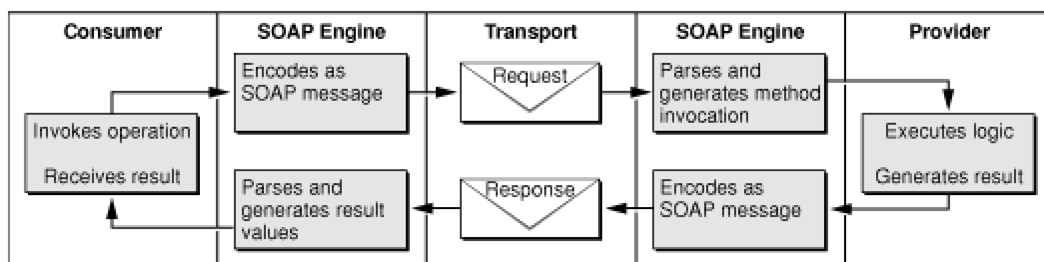


Figure 5-2- 1 SOAP message processing cycle [31]

Obviously the SOAP processing engine is Axis's primary feature, but Axis is not just only an engine. It is also a simple stand-alone server, which can easily plugs into several well-know application servers such as Apache's Tomcat. Axis also has

extensive support for the Web Service Description Language (WSDL). The *WSDL2Java* tool generates Java classes from WSDL file and allows the consumer to easily build client stubs to access remote services.

(4) MySQL Database Server (V4.0) and MySQL Connector/J (V3.1.7)
In order to manage large number of data efficiently, MySQL, the most popular open source SQL database, is used for all data storage. MySQL is a SQL based relational database management system that runs under a broad array of operating systems. It provides users with a platform to add, access, and process data in a fast, reliable and easy way.

Besides MySQL database server itself, to allow Java code to access the database, *MySQL Connector/J* is needed to be setup. As a sub-project of MySQL, it provides connectivity for client applications developed in the Java programming language via a JDBC driver [36].

(5) Eclipse (V3.0.2)
Eclipse is used as IDE in the thesis project. It is an independent open eco-system around royalty-free technology and a universal platform for tools integration. Eclipse provides a plug-in based framework that makes it easier to create, integrate and utilize software tools, saving time and money. The Eclipse Platform is written in the Java language and comes with extensive plug-in construction toolkits and examples. [26]

5.2.2. Database Structure

The database `shopdb` contains tables for `user`, `shop`, `item`, `order`, `cart` and `wish` list information (see Figure 5-2-2).

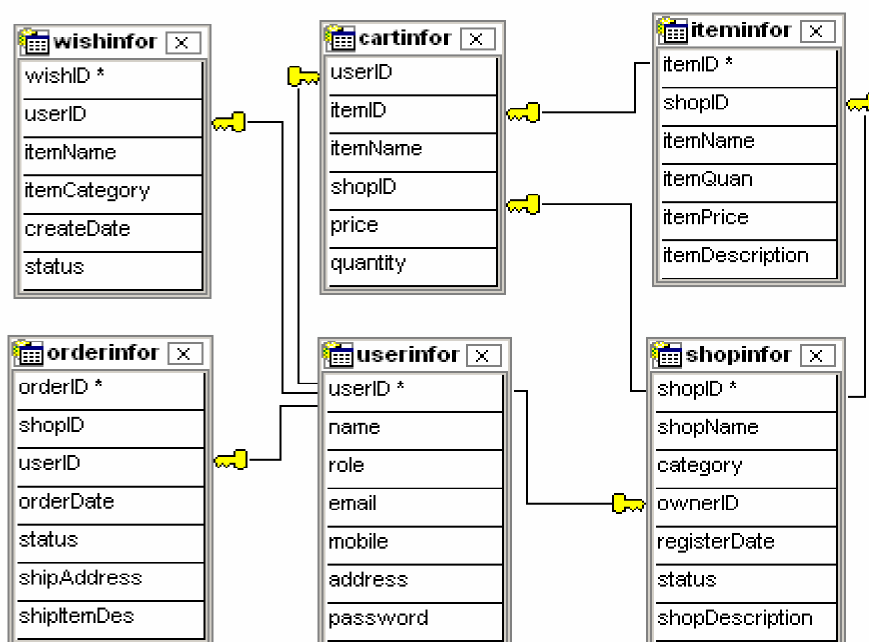


Figure 5-2- 2 shopdb database design

The userinfor table holds the userID (primary key), name, password, role (enumeration type 'Admin','Owner','Customer') and other contact information. Below in Table 5-2-1 is a list of existing users in the database

Table 5-2- 1 "userinfor" table

userID	name	role	email	mobile	address	passwod
iw_admin	Titi	Admin	yaliy05@yahoo.com	07600000	Kista	admin
iw_amazon	Peter	Customer	iw03_twu@it.kth.se	07600001	solna	amazon
iw_pc	Pin	Owner	iw03_pzh@it.kth.se	07600002	Renming	iw_pc

The shopinfor table shows all the information when create a shop, such as shopID (primary key), shopName, category, which is one of 'Book', 'CD', 'Toy', 'Electronics', owenID that is also a foreign key to the userID field in the userinfor table. The filed status can be 'Pending', 'Approved', 'Rejected', or 'Discontinued'. The optional field shopDescription is used to describe the shop (see Table 5-2-2):

Table 5-2- 2 "shopinfor" table

shopID	shopName	category	ownerID	registerDate	status	shopDescription
1	pc shop	Electronics	iw_pc	2005-4-5	Approved	
2	My Amazon	Book	iw_amazon	2005-4-1	Discontinued	Various book
3	vero modo	Book	monday	2005-5-24	Approved	makeup

The iteminfor table saves the detail information about items. Similar to shopinfor table, the field itemID is the primary key and the field shopID refers to the shopID in the shopinfor table (see Table 5-2-3)

Table 5-2- 3 "iteminfor" table

itemID	shopID	itemName	itemQuan	itemPrice	itemDescription
1	1	eMachines3255	50	1,000.000	InterCelereon Processor 330, 256Mb internminne
2	1	Toshiba Laptop	38	1,000.000	InterCeleron Processor M370
4	1	Apple MP3-spelare	10	1,000.000	iPod Mini 4Gb MP3-spelare
5	1	HP DigitalKamera	105	899.000	4.1megapixel,3x optisk zoom

The cartinfor table represents the product detail in a customer's shopping cart. The field userID, itemID and shopID refer to corresponding fields in userinfor, iteminfor, and shopinfor respectively (see Table 5-2-4)

Table 5-2- 4 "cartinfor" table

userID	itemID	itemName	shopID	price	quantity
monday	1	eMachines3255	1	1,000.000	1
monday	2	Toshiba Laptop	1	1,000.000	2

The orderinfor table is the entries for orderID, the identity of an order, userID of the customer, orderDate, using the date type in SQL, the status of the order (either

'Pending' or 'Shipped'), shipping Address and the items description. The format in the filed shipItemDes lies with "itemID, quantity, price". If more than one items need to be shipped, enter "#" at the end of each item (see Table 5-2-5)

Table 5-2- 5 "orderinfor" table

orderID	shopID	userID	orderDate	status	shipAddress	shipItemDes
03185757	1	monday	2005-5-24	Pending	Kista Allevag44a2	2,2,1000.0
24183737	1	monday	2005-5-24	Pending	Kista Allevag44a1	1,2,1000.0# 2,2,1000.0# 3,2,1000.0# 4,2,1000.0
25225010	2	iw_amazon	2005-4-25	Shipped	RenMin South Road	6,3,425# 7,2,567

The last table is wishinfor table. It records all the items that one like to include on his wish list. Server will search the iteminfor table periodically to check if any wish item is available and send the customer the offer by emails or SMS later then. The field status indicate whether the wish item is still waiting for a response('Pending'), canceled by the customer ('Canceled') or has already got a result ('Offered')(see Table 5-2-6)

Table 5-2- 6 "wishinfor" table

wishID	userID	itemName	itemCategory	createDate	status
1	monday	tele2.comviq card	Toy	2005-5-24	Offered
2	monday	Nivea Hand	Book	2005-5-27	Pending
3	monday	interconnections	Book	2005-5-28	Canceled

5.2.3 Create Web service with Apache Axis

According to different roles and function played in the virtual shop system, the Java package structure for the project is like Figure 5-2-3, that all the classes are under the package named services, and there are eight packages included:

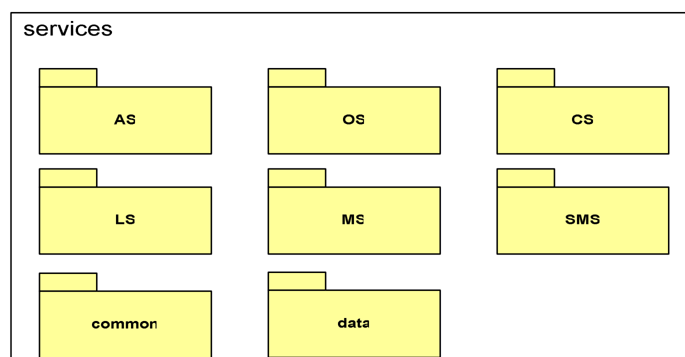


Figure 5-2- 3 Package structure

- ***AS*** (Administrator Services)
  Implement all the functions on behalf of an administrator, such as get shop information by keywords: shop name, category or status; manage the shop creation request and send notification to the requestor. The notification is either email provide by MS service or a short message by SMS service.
- ***OS*** (Owner Services)
  When requestor is notified of shop creation approval and from thereon the requestor assumes the role of a shop owner. As shop owner, the supposed operations include `getShopDetails`, `updateShopDetails`, `disShopRequest`, `addItem`, `updateItem`, `deleteItem`, `getItemDetail`, `getItems`, `getPendingOrders`, and `shipOrder`
- ***CS*** (Customer Services)
  The CS is service providing to end users of the virtual shop application. The customer can invoke various operations to enjoy different services. For example, to manage customer's shopping cart, five operations are available: `addToCart`, `removeFromCart`, `getCart`, `updateCart` and `checkout`.
- ***LS*** (Login Services)
  The users are required to authenticate themselves before using any services provided by the application. LS service is the first step to access the application and users will be prompted to input username and password.
- ***MS*** (Mail Services)
  This service is implemented as ordinary stateless Web service that uses the Java Mail API to send notification as email to customer or administrator. Email addresses are stored in the `userinfor` table in the database
- ***SMS*** (Short Message Service)
  This service is deployed as a SMS-powered Web service to allow other Web service written in Java language to connect to a Short Message Service Centre (SMSC) over a TCP/IP network. In this thesis, it is implemented as an ordinary Java class that use the Java SMPP API [38] to send short message to SMPPSim [39], an open source Java SMSC simulator.
- ***common***
  The `common` package includes some common classes needed in all the services. They are `paraConst` file and java exception files, such as `CartException`, `UserException`
- ***data***
  All the data type files generated from a WSDL type and also are named after the WSDL type. They are the mapping from XML to Java types.

In this section, we will concentrate on discussing the procedure needed to build a Web service using Apache Axis, take the Administrator Services for example. Usually, the procedure is divided into 5 steps.

(1). Define the service interface in Java

Define what services will be provided to the outer world, and what operations are

going to be available. The AS allows users to perform two operations: `getShops` and `manageShopRequest`. Write and compile the Java interface. The sample code is shown as following (Figure 5-2-4):

```
package services.AS;
  public interface AdminService {
  public services.data.ShopType[] getShops(String in0, String in1)
  public void manageShopRequest(int shopID, String status)
}
```

Figure 5-2- 4 Service interface

(2). Java2WSDL: Build WSDL from given service interface

As introduced before, Apache Axis provides emitter tools to generate Java classes from WSDL and in turn, create a WSDL file from the Java interface. The tool for the latter purpose is called *Java2WSDL*. It provides a better option for the Web service developer, since it is more user-friendly than directly coding in WSDL.
Figure 5-2-5 shows how to use this tool from the command line.

```
%java org.apache.axis.wsdl.Java2WSDL
       -o c:\AdminS.wsdl
       -l "http://localhost:8080/axis/services/services/Admin"
       -n "urn:Services:MS"
       -p "services.MS"="urn: services.MS"
           Services.MS.AdminService
```

Figure 5-2- 5 Java2WSDL command

```
Where:

    -o :name and location of the output WSDL file(c:\AdminS.wsdl)
    -l :location of the service
    (http://localhost:8080/axis/services/services/Admin)
    -n :target namespace for WSDL(urn:Services:MS)
    -p :mapping from the package to a namespace
    (services.MS"="urn: services.MS)
    Fully qualified class itself (Services.MS.AdminService)
```

After the command runs, a new file `AdminS.wsdl` was created for us. When we look inside the file, the generated WSDL file contains first of all, the root element, which is <`definitions`> tag, as shows as part of Figure 5-2-6 WSDL file

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
    xmlns:impl="urn:Services:AS"
    xmlns:intf="urn:Services:AS"
     xmlns:apachesoap=http://xml.apache.org/xml-soap
     xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
```

```
     xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:tns2="urn:Services:data"
    xmlns:tns3="urn:Services:common"
     xmlns:wsdl=http://schemas.xmlsoap.org/wsdl/
     xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="urn:Services:AS">
</definitions>
   . . .
```

Figure 5-2- 6 WSDL file (1/5)

The root element is used to declare all the namespaces we are going to use. The target namespace is `urn:Services:AS`. This means that all the `PortType` and `Message` defined in this WSDL file belong to this namespace.

Secondly, the `<wsdl:types>` tag, which defined the data type used in this WSDL file

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions . . .>
<wsdl:types>
  <xsd:schema>
    <xsd:import namespace="urn:Services:data" schemaLocation="dataType.xsd"/>
    <xsd:import namespace="urn:Services:common" schemaLocation="common.xsd"/>
  </xsd:schema>
  <xsd:schema targetNamespace="urn:Services:AS">
    <xsd:complexType name="ArrayOf_shopType">
     <xsd:complexContent>
      <xsd:restriction base="soapenc:Array">
        <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="tns2:shopType[]"/>
      </xsd:restriction>
     </xsd:complexContent>
   </xsd:complexType>
  </xsd:schema>
</wsdl:types>
. . .
```

Figure 5-2- 7 WSDL file (2/5)

The `<xsd:import>` element import schema components from other schema documents. The `<xsd:complex>` element is the XML representation of a complex type, which in this WSDL file illustrates the use of array of `shopType`.

Next, using the <`wsdl:message`> tag:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions . . .>
<wsdl:types. . . >
```

```
<wsdl:message name="getShopsRequest">
     <wsdl:part name="in0" type="xsd:string"/>
     <wsdl:part name="in1" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="getShopsResponse">
     <wsdl:part name="getShopsReturn" type="impl:ArrayOf_shopType"/>
</wsdl:message>
<wsdl:message name="manageShopRequestRequest">
     <wsdl:part name="shopID" type="xsd:int"/>
     <wsdl:part name="status" type="xsd:string"/>
</wsdl:message>
<wsdl:message name="manageShopRequestResponse">
</wsdl:message>
. . .
```

Figure 5-2- 8 WSDL file (3/5)

The `<wsdl:message>` element are composed of `<wsdl:part>`. The message `getShopsRequest` has two parts, which indicates that two parameters are needed for invoking `getShops` operation.

Also, `<wsdl:portType>` tag is included. Inside the `<wsdl:portType>`, we have `<wsdl:operation>` tags for two operations exposed in AS: `getShops` and `manageShopRequest`

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions . . .>
<wsdl:types. . . >
<wsdl:message. . . >
<wsdl:portType name="AdminService">
   <wsdl:operation name="getShops" parameterOrder="in0 in1">
         <wsdl:input name="getShopsRequest" message="impl:getShopsRequest"/>
         <wsdl:output name="getShopsResponse" message="impl:getShopsResponse"/>
         <wsdl:fault name="ShopException" message="impl:ShopException"/>
   </wsdl:operation>
   <wsdl:operation name="manageShopRequest" parameterOrder="shopID status">
        <wsdl:input   name="manageShopRequestRequest"
                     message="impl:manageShopRequestRequest"/>
         <wsdl:output name="manageShopRequestResponse"
                     message="impl:manageShopRequestResponse"/>
         <wsdl:fault name="ShopException" message="impl:ShopException"/>
     </wsdl:operation>
</wsdl:portType>
. . .
```

Figure 5-2- 9 WSDL file (4/5)

Finally, the `<wsdl:binding>` and `<wsdl:services>` elements are added at the end of the WSDL file.

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions . . .>
<wsdl:types. . . >
<wsdl:message. . . >
<wsdl:portType. . . >
<wsdl:binding name="AdminSoapBinding" type="impl:AdminService">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
         <wsdl:operation name="getShops">
              <wsdlsoap:operation/>
              <wsdl:input>
                   <wsdlsoap:body use="encoded"
                   encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
                   namespace="urn:Services:AS"/>
              </wsdl:input>
              <wsdl:output>
                   <wsdlsoap:body use="encoded"
                   encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
                   namespace="urn:Services:AS"/>
              </wsdl:output>
              <wsdl:fault name="ShopException">
                   <wsdlsoap:fault name="shopErr" use="encoded"
                   encodingStyle=http://schemas.xmlsoap.org/soap/encoding/
                   namespace="urn:Services:AS"/>
              </wsdl:fault>
         </wsdl:operation>
     . . .
</wsdl:binding>
<wsdl:service name="AdminServiceService">
         <wsdl:port name="Admin" binding="impl:AdminSoapBinding">
              <wsdlsoap:address location="http://localhost:8080/axis/services/Admin"/>
         </wsdl:port>
</wsdl:service>
```

Figure 5-2- 10 WSDL file (5/5)

(3). WSDL2Java – Generate server-side skeleton and client-side stub code

The next step is to use the tool *WSDL2Java*. It takes in a WSDL file and generates a suite of Java class files to automatically interface with the specified services. All the SOAP specific details are completely hidden while the services implementation is comparatively simple.

The full command for the AS example is shown as Figure 5-2-11

```
%java org.apache.axis.wsdl.WSDL2Java
        --server-side
        --skeletonDeploy true
        -N"urn:Services:AS"="services.AS"
         AdminS.wsdl
```

Figure 5-2- 11 WSDL2Java command

Where:

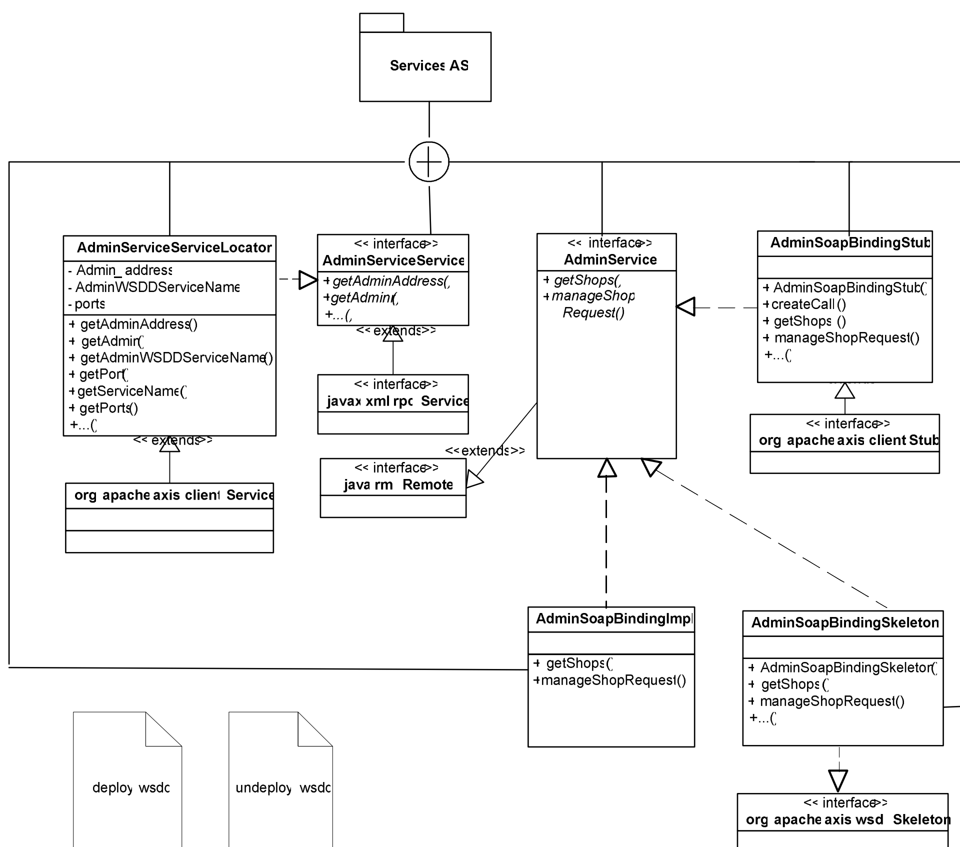
--server-side: emit server-side bindings for web service

--skeletonDeploy: deploy skeleton <true> or implementation <false> in deploy.wsdd

-N: mapping of namespace to package (*urn:Services:AS"="services.AS*)

Name of WSDL file (*AdminS.wsdl*)

After executing the command, a suit of code has been generated. They will reside in the directory "`services/AS`". The following Figure shows the generated class suite.



Where:

- **AdminSoapBindingImpl.java**: This is the implementation code for AS service. This is the one file we will need to edit
- **AdminService.java**: New interface file that contains the appropriate java.rmi.Remote usages.
- **AdminServiceService.java**: Service interface of the Web service. The

ServiceLocator implements this interface.

- **AdminServiceServiceLocator.java:** Helper factory for retrieving a handle to the service.
- **AdminSoapBindingSkeleton.java:** Server-side skeleton code.
- **AdminSoapBindingStub.java:** Client-side stub code that encapsulates client access.
- **deploy.wsdd:** Deployment descriptor that we pass to the Axis system to deploy these Web service.
- **undeploy.wsdd:** Deployment descriptor that will undeploy the Web service from the Axis system.[40]

The mapping relationship between a WSDL file to the Java class is shown below,

WSDL clause	Java class(es) generated
For each entry in the type section	A java class A holder if this type is used as an inout/out parameter
For each *portType*	A java interface *(AdminService.java)*
For each *binding*	A stub class *(AdminSoapBindingStub.java)* A skeleton class *(AdminSoapBindingSkeleton.java)* An implementation template class *(AdminSoapBindingImpl.java)*
For each *service*	A service interface (AdminServiceService.java) A service implementation (the locator) *(AdminServiceServiceLocator.java)*
For all services	One deploy file *(deploy.wsdd)* One undeploy file *(undeploy.wsdd)*

Figure 5-2- 12 WSDL Mapping to Java [40]

(4). Complete the code in Impl file

Implement the business logic in the output file AdminSoapBindingImpl. Add the codes into the methods that it created. In the example AS, two methods should to be realized: getShops and manageShopRequest. Both of them probably need to interact with the MySQL database and take the input parameters and turn out the result.

(5). Deploy the service

The last step is to deploy the service to Tomcat. Copy all the class files generated by compiling the above .java file to %CATALINA_HOME%\webapps\axis\WEB-INF\classes directory. Apache Axis has an Admin client command line tool that can be used to

deploy/undeploy the services. And we pass the deployment descriptor `deploy.wsdd` file to this program. Execute the following command (see Figure 5-2-13):

```
% java org.apache.axis.client.AdminClient deploy.wsdd
% Processing file deploy.wsdd
% <Admin>Done processing</Admin>
```

Figure 5-2- 13 Web service deploy command

Now the AS is alive and running on the server. It is waiting for the invocation from the clients.

5.2.4 Implement Java Mail

This section will discuss the implementation of MS (MailService) provided by virtual shop application. In order to send email to users by Web service developed by Java language, the JavaMail API [33] is used. It is an optional package for reading, composing and sending electronic messages. Firstly, download the JavaMail 1.3.2 Release package and follow the instruction to finish the installation. After the environment is ready, developing the JavaMail based application is straightforward. The implementation source code should be written to the file "`MailSoapBindingImpl.java`" as previous explanation. Figure 5-2-14 shows the sending mail operation. Sending an e-mail message involves setting the properties of the mail server, getting a session, creating and filling a message, and sending it.

```
try{
     // Gets the System properties
     Properties props = System.getProperties();
     // Puts the SMTP server name to properties object
     props.put("mail.smtp.host", "smtp.kth.se");
     // Get the default Session object based on the properties
     Session session = Session.getDefaultInstance(props, null);
     session.setDebug(false); // Disable the debug mode?
     // Create a MimeMessage from the session
     MimeMessage msg = new MimeMessage(session);
      // Set the "From" address
      msg.setFrom(new InternetAddress(from));
      // Check if it is a valid email-id
      if (to.indexOf("@") != -1) {
         // Setting the "To" addresses
          msg.setRecipients(Message.RecipientType.TO,
              InternetAddress.parse(to, false));
      }
      // Check if it is a valid email-id
```

```
      if (cc.indexOf("@") != -1) {
        // Setting the "Cc" addresses
         msg.setRecipients(Message.RecipientType.CC,
            InternetAddress.parse(cc, false));
      }
      // Sets the Subject
      msg.setSubject(subject);
      // Create and set the content of the message
      MimeBodyPart mbp = new MimeBodyPart();
      mbp.setContent(message,"text/html");
      // Create the Multipart and its parts to it
      Multipart mp = new MimeMultipart();
      mp.addBodyPart(mbp);
      // Add the Multipart to the message
      msg.setContent(mp);
      // Set the Date: header
      msg.setSentDate(new Date());
      // Use a Transport to send the message
      Transport.send(msg);
      }catch (MessagingException e){}
}
```

Figure 5-2- 14 JavaMail example

Compile and copy the class file to `%CATALINA_HOME%\webapps\axis\WEB-INF\classes`. Deploy the service as introduced in Chapter 5.2.3 and if successful, the MS is ready to use.

5.2.5 Implement Java SMS

The Short Message Service (SMS) is the technology that allows text messages to be received and send over mobile devices. The message size is limited to 255 characters, in standard. Messages can be delivered immediately when the called phone is turned on, or otherwise, like e-mail, they can also be reviewed or stored before submitting. Later, with technical development, mobile message is evolving beyond text by taking a development path to Enhanced Messaging Service (EMS) and Multimedia Messaging Service (MMS). Since the notification service only need text message, there is no need to look into the EMS and MMS technology in the thesis.

SMS messages are transferred using SMPP protocol. SMPP stands for Short Message Peer to Peer protocol. It is a communication protocol designed for transfer of short messages between short message centre and a mobile device or SMS application, also called External Short Message Entity (ESME). When an SMS message is sent from ESME, it will first reach a Short Message Service Center (SMSC). SMSC takes the

responsibilities to forward the message to the destination. If the SMS recipient is unavailable, the SMSC will store the message until the recipient come back alive. As defined in SMPP specification (v5.0), a typical session for SMS transmitter is shown as Figure 5-2-15

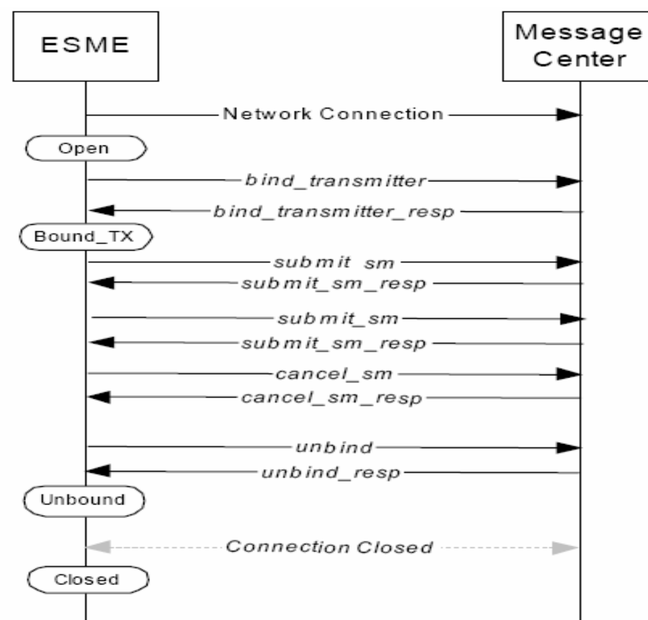


Figure 5-2- 15 Sample transmitter session [42]

In order to send a message from ESME, firstly, a network connection is required to set up between ESME and SMSC. Then, the connected ESME issue a bind request as a transmitter (by using a `bind_transmitter` message) and receive a `bind_tranmitter_resp` message from the SMSC authorizing its bind request. After that, ESME is enabled to send messages to SMSC (by using `submit_sm` message) and the SMSC forward the messages to the receiver. The ESME can exchange messages with the message center as long as it has a valid, bound session to it. When ESME wants to terminate the session, it should unbind the session with unbind and `unbind_resp` messages and close the network connection.

In the thesis, Java SMPP library [38] is used to make our SMS Web service speaks with SMSC through SMPP protocol. Figure 5-2-16 is the sample code for transmission a short message.

```
try{
    // Connect to SMSC
       ie.omk.smpp.Connection myConnection =
           new ie.omk.smpp.Connection("localhost",2775);
    // bind to SMSC
       BindResp resp = myConnection.bind(
           ie.omk.smpp.Connection.TRANSMITTER, "smppclient", "password", null);
        if (resp.getCommandStatus() != 0){
             System.out.println("SMSC bind failed.");
           }
```

```
   // submit message
      SubmitSM sm = (SubmitSM)myConnection.newInstance(SMPPPacket.SUBMIT_SM);
      sm.setDestination(new Address(0, 0, to));
      sm.setMessageText(smsMsg);
      SubmitSMResp smr = (SubmitSMResp)myConnection.sendRequest(sm);
    // unbind
       UnbindResp ubr = myConnection.unbind();
       if (ubr.getCommandStatus() == 0) {
          System.out.println("Successfully unbound from the SMSC");
           } else {
              System.out.println("There was an error unbinding.");}
}catch (MessagingException e){}
```

Figure 5-2- 16 JavaSMS example

To start, we establish a connection to the SMSC. For this purpose, we use `ie.omk.smpp.Connnection` class. Next, we should bind the connection with the connection type (eg,"`TRANSMITTER`") and credential provided by the message center. The credential is composed of a system identification (eg,"`smppclient`") and a password to authenticate to the SMSC (eg,"`password`"). If bind is successful, we will create a message that will be delivered to the SMSC by calling the `newInstance` method with the appropriate SMPP package type (`SMPPPacket.SUBMIT_SM` in this case). Then, specify the destination phone number by `setDestination` method and fill in the message context by `setMessageText`. Use `sendRequest` method to submit the message. Finally, ESME issues an `unbind` request to the SMSC requesting termination of the SMPP session and the latter, in turn, then respond with an `unbind_resp` acknowledging the request to end the session.

Compile and copy the class file to `%CATALINA_HOME%\webapps\axis\WEB-INF\classes`. Deploy the service on to the Tomcat application server, the SMS Web service is ready on the server

Normally, the process of obtaining access to a real SMSC is time-consuming and some network operators run stringent acceptance tests before they will allow your application to interact with the production SMSCs. Therefore, SMPPSim [39], an open source SMPP SMSC simulator, is set up and adopted for the initial testing purpose.

SMPPSim is a testing utility which mimics the behaviors of an SMPP based SMSC. Installation is easy. Unzip the download package smppapi-0.3.4.zip into a location on the hard drive and change to the SMPPSim home directory and run the script `startsmppsim.bat`. You should get something like Figure 5-2-17 on the console window. By default, the SMPPSim will open a network connection on port 2775 and wait to be bound by ESME.

```
ion
2005.09.11 01:13:20 593 INFO    12 StandardConnectionHandler waiting for connect
ion
2005.09.11 01:13:20 593 INFO    13 StandardConnectionHandler waiting for connect
ion
2005.09.11 01:13:20 593 INFO    14 StandardConnectionHandler waiting for connect
ion
2005.09.11 01:13:20 633 INFO    15 Starting InboundQueue service....
2005.09.11 01:13:20 633 INFO    15 InboundQueue: empty  - waiting
2005.09.11 01:13:20 633 INFO    16 StandardConnectionHandler waiting for connect
ion
2005.09.11 01:13:20 633 INFO    17 StandardConnectionHandler waiting for connect
ion
2005.09.11 01:13:20 633 INFO    18 StandardConnectionHandler waiting for connect
ion
2005.09.11 01:13:20 633 INFO    19 StandardConnectionHandler waiting for connect
ion
2005.09.11 01:13:20 633 INFO    20 StandardConnectionHandler waiting for connect
ion
2005.09.11 01:13:20 633 INFO    21 StandardConnectionHandler waiting for connect
ion
2005.09.11 01:13:20 633 INFO    22 Starting Lifecycle Service (OutboundQueue)
2005.09.11 01:13:20 633 INFO    22 Lifecycle Service: OutboundQueue is empty  -
waiting
```

Figure 5-2- 17 SMPPSim start

Let us assume that the shop administrator wants to sent an approve notification to the user with phone number 07600003 and says "Congratulations! Your shop create request have been proved!" with SMS Web service. Run SMPPSim, and start the program, the console window of both Tomcat and SMPPSim should have something shown as Figure 5-2-18 and Figure 5-2-19. If both Tomcat and SMPPSim display success status, the SMS notification is submitted successfully.

```
Tomcat

....get toNumber:...07600003
...Connecting to SMSC
- Could not find API properties to load
...Binding to the SMSC
- Binding to the SMSC as type 1
- Opening network link.
- Opening TCP socket to localhost/127.0.0.1:2775
- Setting state 1
- Setting state 2
- Disabling optional parameter support as no sc_interface_version parameter was
received
...Bind successful!submitting a message.
- Unbinding from the SMSC
- Setting state 3
- Successfully unbound
- Setting state 0
Successfully unbound from the SMSC
```

Figure 5-2- 18 Tomcat console window when sending a SMS

```
2005.09.11 01:30:17 566 INFO    11
2005.09.11 01:30:17 846 INFO    11 : Standard SUBMIT_SM:
2005.09.11 01:30:17 856 INFO    11 Hex dump (100) bytes:
2005.09.11 01:30:17 866 INFO    11 00000064:00000004:00000000:00000002:
2005.09.11 01:30:17 866 INFO    11 00000000:00003037:36303030:30330000:
2005.09.11 01:30:17 866 INFO    11 00000000:00000000:3B436F6E:67726174:
2005.09.11 01:30:17 866 INFO    11 756C6174:696F6E73:2120596F:75722073:
2005.09.11 01:30:17 866 INFO    11 686F7020:63726561:74652072:65717565:
2005.09.11 01:30:17 866 INFO    11 73742068:61766520:6265656E:2070726F:
2005.09.11 01:30:17 866 INFO    11 76656421:
2005.09.11 01:30:17 896 INFO    11 cmd_len=100,cmd_id=4,cmd_status=0,seq_no=2,se
rvice_type=,source_addr_ton=0
2005.09.11 01:30:17 896 INFO    11 source_addr_npi=0,source_addr=,dest_addr_ton=
0,dest_addr_npi=0
2005.09.11 01:30:17 896 INFO    11 dest_addr=07600003,esm_class=0,protocol_ID=0,
priority_flag=0
2005.09.11 01:30:17 896 INFO    11 schedule_delivery_time=,validity_period=,regi
stered_delivery_flag=0
2005.09.11 01:30:17 906 INFO    11 replace_if_present_flag=0,data_coding=0,sm_de
fault_msg_id=0,sm_length=59
2005.09.11 01:30:17 906 INFO    11 short_message=Congratulations! Your shop crea
te request have been proved!
2005.09.11 01:30:17 906 INFO    11
2005.09.11 01:30:17 916 INFO    11 :SUBMIT_SM_RESP (ESME_RINVSRCADR):
2005.09.11 01:30:17 916 INFO    11 Hex dump (16) bytes:
2005.09.11 01:30:17 926 INFO    11 00000010:80000004:0000000A:00000002:
2005.09.11 01:30:17 926 INFO    11
2005.09.11 01:30:17 936 INFO    11 cmd_len=16,cmd_id=-2147483644,cmd_status=10,s
eq_no=2,message_id=0
```

Figure 5-2- 19 SMPPSim console window when sending a SMS

5.2.6 Implement Java Scheduling

As the customer sends a wish request to the server, the server should provide certain mechanism to check the item information in the `shopdb` database at exactly specified time to mach the wish request to the available product. If any matching has been found, the mechanism will manage to send wish offer notification to the end user. The scheduler checking behavior is accomplished by Java Timer API. As of J2SE 1.4.x, Java contains the `java.util.Timer` and `java.util.TimerTask` classes that can be used for this purpose. It is designed as a standalone Java application.

As shown in Figure 5-2-20, Java scheduling example. First of all, we should implement a class that will do the scheduled task. Here is called `WishGenerator`. It extends from `java.util.TimerTask`, which implements `java.lang.Runnable`. We override `run()` method with our own implementation code to meet the business logic. Then, we schedule this object's execution using `schedule()` method of `java.util.Timer` class. The `schedule()` method specifies the date of the first execution and the period of subsequent executions in milliseconds. (eg, scheduler run from now and repeat every 10 minutes)

```
public class WishGenerator extends TimerTask {
   public void run() {
      get the pending wishes
      if (exist pending wishes)
```

```
        {
           search the database to find the matching items
           if (found) send notification to customer
         } . . .
    }. . .
}
public class MainScheduler {
   public static void main(String[] args) {
      Timer timer=new Timer();
      Calendar date = Calendar.getInstance();
      // scheduler run from now and repeat every 10 minutes
      timer.schedule(
        new WishGenerator(),
        date.getTime(),
        1000 * 60 *10
      );
     }
}
```

Figure 5-2- 20 Java scheduling example

5.3 Client Implementation

The target mobile client in the thesis project is a J2ME enabled mobile device. It is a fat client which has more computing power and memory resource compare to traditional wireless access device. The client will do all parse work on its own. When the client wants to access the Web service, it will construct SOAP request and send it to the server. When the XML based SOAP response returns, the mobile client is capable to parse the data from SOAP response and display the result to the user. The main delivery from the client side is a MIDlet application.

5.3.1 Technologies and Develop platform & Environment

Before we go deep into the develop platform and enviromnet, we will give an quick introduction to Java 2 Platform, Micro Edition (J2ME) technlogy and provide some brief information about *MIDlet* program.

What is J2ME? Briefly, *J2ME combines a resource constrained JVM and a set of Java APIs for developing applications for mobile devices*.[24] Contrast to a traditional Java Virtual Machine (*JVM*), the *JVM* running on micro device (actually called *KVM* in the case) is very limited and supports only a small number of traditional Java classes. Usually, the device manufacturers will install and prepackage the device with this *KVM* and associated APIs. The developers only takes care the limited functional APIs and program applications targeting the device.

J2ME can be divided into three layers, as shown in Figure 5-3-1:

- ***Configuration***: contains *KVM* and some class libraries. The most popular configuration that Sun provides is Connected Limited Device Configuration (*CLDC*). *CLDC* is for device with very limited configuration, like 16/32bit microprocessor and 160kb-512kb available memory. For the more feature-rich device with at least 2 MB of memory available, the configuration is correspondingly Connected Device Configuration (*CDC*).
- ***Profile***: Profile also contains a set of API which is defined for a specific series of device and it is implemented above certain configuration layer. The developers just develop the applications within some specific profile. For the connected limited device like: PDA and cell phone, their profile layer is called Mobile Information Device Profile (*MIDP*). *MIDP* consist of a series of APIs that allow the developers to create not only the customized GUI shown on a mobile device but also full-scale business applications using external and internal data sources.
- ***Optional packages***: an optional set of APIs that may or may not contained in the application, like media API and 3D API.[24]

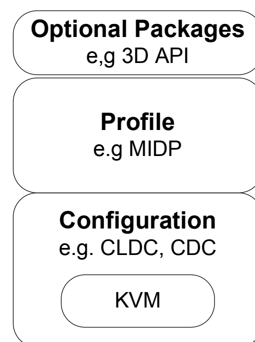


Figure 5-3- 1 The J2ME stack [24]

The most popurlar profile and configuration on the recent market are *MIDP* and *CLDC*. And the MIDlet, as the name suggests, is a small J2ME application written for MIDP.

"All applications for the MID Profile must be derived from a special class, MIDlet. The MIDlet can be compared to J2SE applets, which is a small program running on mobile device. A MIDlet can exist in four different states: loaded, active, paused, and destroyed." [47]

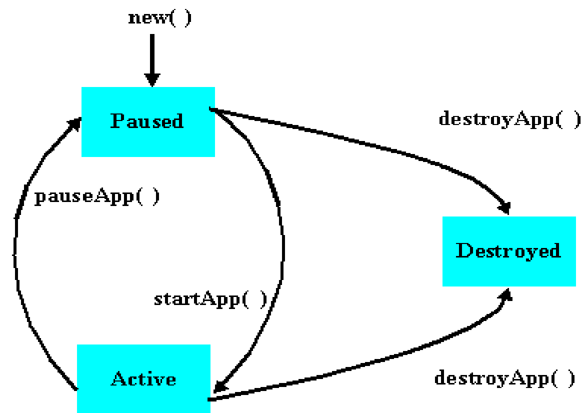


Figure 5-3- 2 Life cycle of a MIDlet [48]

Figure 5-3-2 gives an overview of the MIDlet lifecycle. The implementation of MIDlet is manipulated by Application Management software (AMS) which is also called Java Application Manager (JAM) and actually responsible for the whole function mechanism of MIDlet. JAM can destroy or turn the MIDlet to paused status and activate it again based on these three MIDlet states.

Every time when JAM try to create a new instance of MIDlet, it will call the constructor of MIDlet first and let it be in the paused status and then make MIDlet enter into Active status by calling the method `MIDlet.startApp()`. After this JAM can either make MIDlet back to paused status again by calling `MIDlet.pauseAapp()` or destroy the MIDlet by calling the `MIDlet.destroyApp()`.From a programmer's point of view, the state of MIDlet can be changed by calling methods: `resumeRequest`, `notifyPaused` and `notifyDestroyed`.

Come back to our thesis project, the objective for the client side is to design and implement a simple MIDlet application, which user can download and install on the J2ME enabled mobile to invoke the implenmented virtual shop services. The develop platform is Windows XP, and Java is chosen as implementation language. The client MIDlet application is developed completely based on the resource of open source, which includes following softwares tools and libraries:

(1). Sony Ericsson SDK 2.2.4 for the Java ME platform [52]
It is a customized version of Sun Microsystems' J2ME Wireless Toolkit version 2.2 which includes a device customization for the Sony Ericsson MIDP2 mobile phones. The reason why we choose Sony Ericsson is we have a Sony Ericsson handset K700 on which we can test our MIDlet in a real environment. And also the Sony Ericsson J2ME SDK can be integrated with IDE, like, Eclipse. The SDK supports Universal Emulator Interface (UEI) and can be integrated with Eclipse that also supports UEI. The required software includes Java SDK1.4.1 or higher and DirectX8.1 or later.

(2). kSOAP (V2.0.1)
kSOAP is a lightweight parser designed specially for use with MIDP. The most

advantage of kSOAP is its relative simplicity. Instaed of having hunders of class file like other SOAP engines, kSOAP is packaged into single JAR file and used for SOAP application running within a KVM. Two objects play verry important roles in the entire packet: `org.ksoap.SoapObject`.and `org.ksoap.transport`.The first object make up the body of a soap envelope by using methods `getProperty()` and `setProperty()`.The transport object facilitate SOAP calls over HTTP using the J2ME generic connection framework. In addition, kSOAP also make it easy to capture fault data. It maps all the SOAP faults to an exception object `org.ksoap.SoapFault`.

5.3.2 Create MIDlet with Toolkit

In the thesis, we are going to use the J2ME wireless toolkit embedded in the Sony Ericsson SDK package to develop our client application. Generally speaking, there are a few steps to develop a MIDlet for the mobile device. These steps are: designing, coding, compiling, preverfication, packaging testing and deployment. [24] By using the wireless toolkit, a few above steps are abstracted to easy the whole development procedure.

(1). Design

We organize the client program into two layers:

- ***Interface Layer***: the interactive layer with the user. This layer gets access to the display by obtaining an instance of `Display` class and my call `setCurrent()` to give the user the first screen. The display object is usually inheriting form the `Displayable` object which will fill the whole screen of the device, like a `Form`, `List` or `Alert`.
- ***Application Logic Layer***: realizes the application business logic. In the thesis project, the main objective for this layer is to locate the Web service which deployed in Section 5.2, formulate the call via kSOAP engine and process the return data.

(2). Code

Like creating an applet by extending the `java.applet.Applet` class, each MIDlet must extend the abstract class `javax.microedition.midlet.MIDlet`. And the MIDlet must overwrite three methods of this abstract class: `startApp()`, to signal the MIDlet that it has entered the `Active` state; `pauseApp()`, to signal the MIDlet to enter the `Paused` state; `destroyApp(boolean unconditional)`,to signal the MIDlet to terminate and enter the `Destroyed` state. Figure 5-3-3 is the `ClientMidlet` class:

```
package myClient;

import javax.microedition.midlet.*;
```

```
import javax.microedition.lcdui.*;
import myClient.client.Screen.*;
public class ClientMidlet extends MIDlet implements CommandListener{
   public Display display;
   public List mainScreen;
   public Login_Form loginForm;
   public ClientMidlet() {
      mainScreen = new List("ShopClient",List.IMPLICIT);
      mainScreen.addCommand(new Command("Launch",Command.OK,0));
      mainScreen.addCommand(new Command("Exit",Command.EXIT,0));
      mainScreen.setCommandListener(this);
      display=Display.getDisplay(this);
   }
   protected void startApp() throws MIDletStateChangeException {
      display.setCurrent(mainScreen);
   }
   protected void pauseApp() {
   }
   protected void destroyApp(boolean arg0) throws MIDletStateChangeException {
      this.notifyDestroyed();
   }
   public void commandAction(Command c, Displayable d) {
    try{
      if (c.getCommandType()==Command.EXIT){
             destroyApp(true);
      }
        else if(c.getCommandType()==Command.OK)
           display.setCurrent(createLoginForm());
    }catch (Exception e){}
    }
   private Screen createLoginForm(){
      loginForm = new Login_Form("LoginService",this);
      return loginForm;
   }  }
```

Figure 5-3- 3 ClientMidlet class

In this example, ClientMidelet's constructor create the element that is necessary to display a main screen list which has "Launch" and "Exit" command choice. And the startApp method does the the actual task of displaying this element. The `commandAction` method indicates that either command event "Launch" or "Exit" has occurred on the main screen. If the user press "Launch", then a loginform which is created by `createLoginForm` method will be returned and fill the mobile device.

Regarding to the business logic layer, we are going to use kSOAP to formulate Web service call and process the return value. Take `LoginS` Web service which we

implemented in Section 5.2 for example, the service authenticates the user and return the user's role back. The service WSDL file can be found from the Appendix of the thesis, which includes the description of the service interface.

```
import org.ksoap.*;
import org.ksoap.transport.*;
. . .
 public void run() {
       try{
          SoapObject rpc = new SoapObject("urn:Services:LS","login");
          rpc.addProperty("userID",userID);
          rpc.addProperty( "pwd",password);
          HttpTransport ht = new HttpTransport(http://192.16.126.235/axis/services/Login,
                                                    "urn:Services:LS#login");
          role = ht.call(rpc);
          if (role.equalsIgnoreCase("admin"))
          . . .
          if (role.equalsIgnoreCase("owner"))
          . . .
          else {
             mvLoginform.roleItem.setText(" invalid userID/password,please input again");
             mvLoginform.tf_pwd.setString(null);
          }
       }catch (SoapFault sf ){
              faultstr = "Code: " + sf.faultcode + "\nString: " +sf.faultstring;
              mvLoginform.roleItem.setText(faultstr);}}
```

Figure 5-3- 4 Using kSOAP to formulate Web service call

In Figure 5-3-4, we first import all the necessary kSOAP libraries by inserting the Line1-2 statements. To prepare our client to use a SOAP server, we create a new `SoapObject`, passing the constructor the namespace URI for the SOAP call and then the method name being called. We obtain the URI and method name information from the service WSDL file. The `SoapObject` also needs to be prepared with the method parameters by using `addProperty()` method. Next, we create e new `HttpTransport` object, which provide the necessary functionality to invoke the service. The `HttpTransport` object takes the constructor of the service endpoint URL and the SOAPAction URI as input. To execute the service, call it using the `call()` method, passing call the `SoapObject` that will invoke the service. The `call()` method return the value whatever it returns from the SOAP service. In our case, it the user's role, which might be one of `Admin`, `Owner`, `Customer` or `null`. To catch the exception, we use `SoapFault` class. The class will catch any SOAP fault exception the `call()` method throws.

(3). Compile and Preverify

With the code in place, we need to know how to compile it with the toolkit so that it is

ready for mobile devices. Star J2ME wireless toolkit, and create a new project for your client MIDlet. Input project name “THclient” and MIDlet class name with the package name “myclient.ClientMidlet”

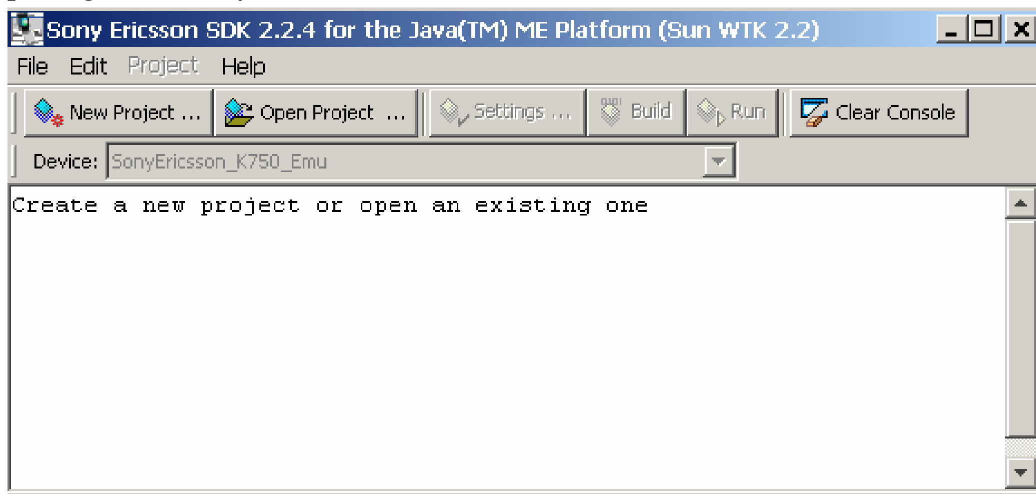


Figure 5-3- 5 Main toolkit interface

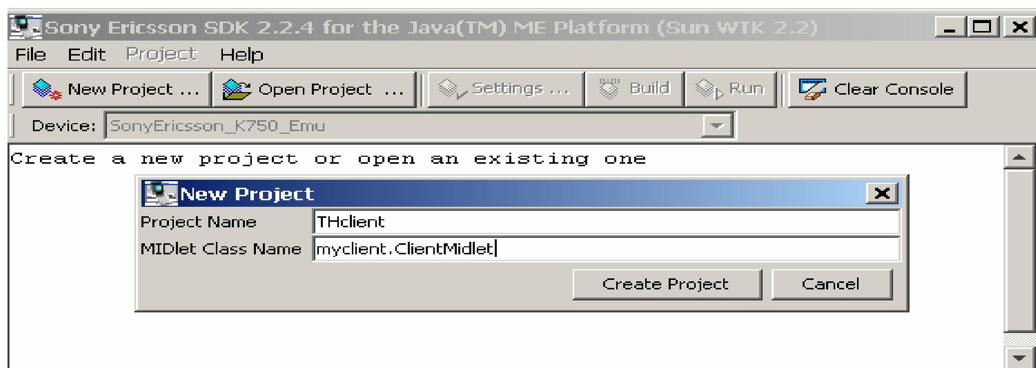


Figure 5-3- 6 Creat a new MIDlet project

The next window that prompt out will allow the user change the project settings that control the target platform of the MIDlet. Here we use Sony Ericsson K700 as target platform, MIDP2.0 as profile and CLDC1.1 as configuration. Press OK, the project is created successfully. It also prints out where the relevant project files are saved. (`src` folder, `lib` folder, `resource` folder)

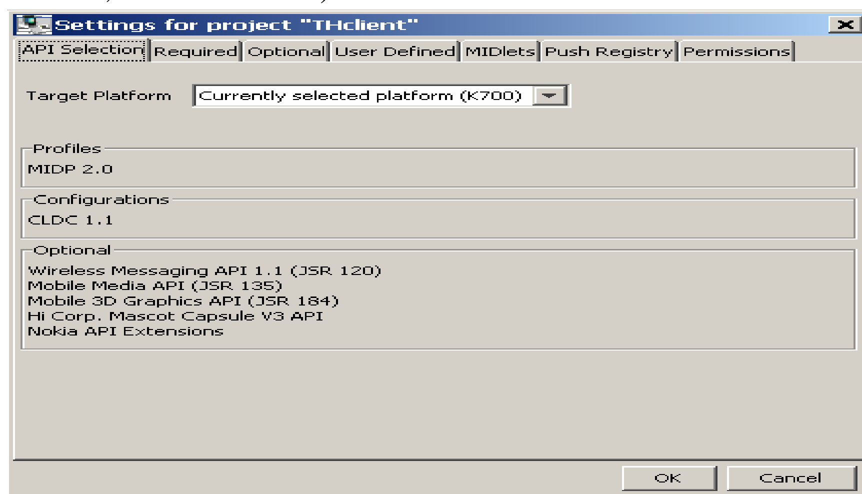


Figure 5-3- 7 Change the MIDlet project setting

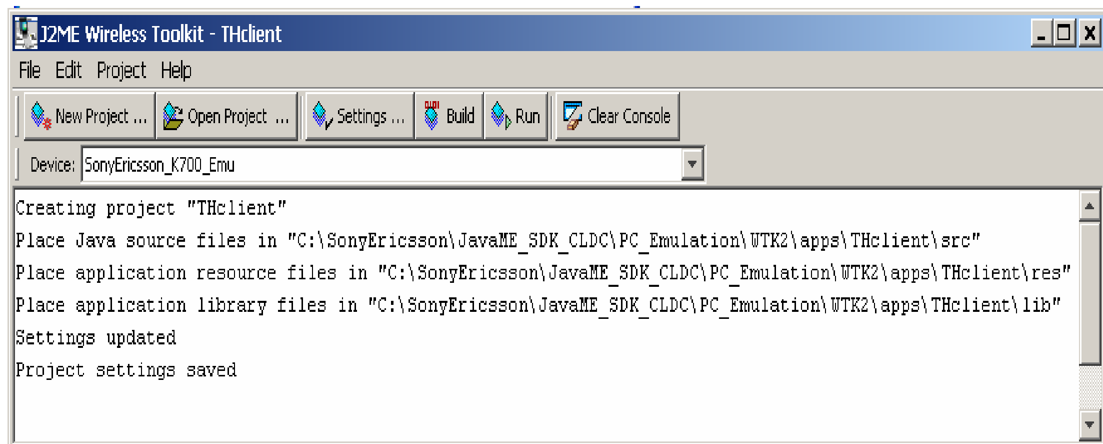


Figure 5-3- 8 MIDlet Project created

Copy the source code crated in step2 to the project `src` folder. Also, copy the kSOPA library `ksoap2-j2me-full.jar` to the project `lib` folder. Hit the Build menu button, the toolkit will start to compile and preverify the MIDlet. It is interesting to look at the other project folder. The bin folder contains the JAD file "THclient.jad" and Manifest files; the class folder has all the compiled class files.

(4). Package

The toolkit is also able generate the project JAR file by selecting the Project menu item and select create Package submenu. By doing this, the JAR file will be saved in the bin folder with updated Manifest information in the JAD file

(5). Test

As part of the wireless toolkit, we use an emulator device that mimics the functionality of a real mobile device via PC to test our implemented MIDlet. To run the emulator, hit the Run menu, an emulating k700 mobile pop up on the screen as shown in Figure 5-3-9.

Click on the "Launch" button, a new login form pop up as shown in Figure 5-3-10. Fill in the username and password, and press OK. If the user login as the shop owner, the customer interface will pop up. From the list, the shop owner can manage shop, manage item and manage orders. (See Figure 5-3-11)

Figure 5-3- 9 Test the MIDlet project (1/3)

Figure 5-3- 10 Test the MIDlet project (2/3)

Figure 5-3- 11 Test the MIDlet project (3/3)

(6). Deploy

After testing, we only verified that our MIDlet was running OK on the simulated environment. But how is about on the real device? How to deploy the MIDlet directly on it? For the J2ME enabled phone, there are two options available. The first is via a network connection between the computer and the handset, like, via USB cable, via Bluetooth wireless connection. Second, we can also deploy the MIDlet by using Internet connection, provided the device is able to access the Internet using it internal Web browser. The basic idea is to put the MIDlet JAD file on the Internet, and let the mobile device find, download, install and run it. Usually, the developer will create a HTML file which has a link to the MIDlet JAD file, and the JAD file provides a link to the associate JAR file via the `MIDlet-Jar-URL: THclint.jar` attribute. Upload the newly created HTML file, the JAD file, the original JAR file to the web server and make it available to the mobile device browser, so anyone with a mobile device that can browse the Internet should be able to point to the deployed MIDlet. [24]

The wireless toolkit also provide a method to test the deployment procedure, click the Run via OTA (Over the Air) menu, and the toolkit will simulate the emulator running the MIDlet via the Internet, and also create the HTML file in the project `bin` folder. Figure 5-3-12 show the procedure to run via OTA.

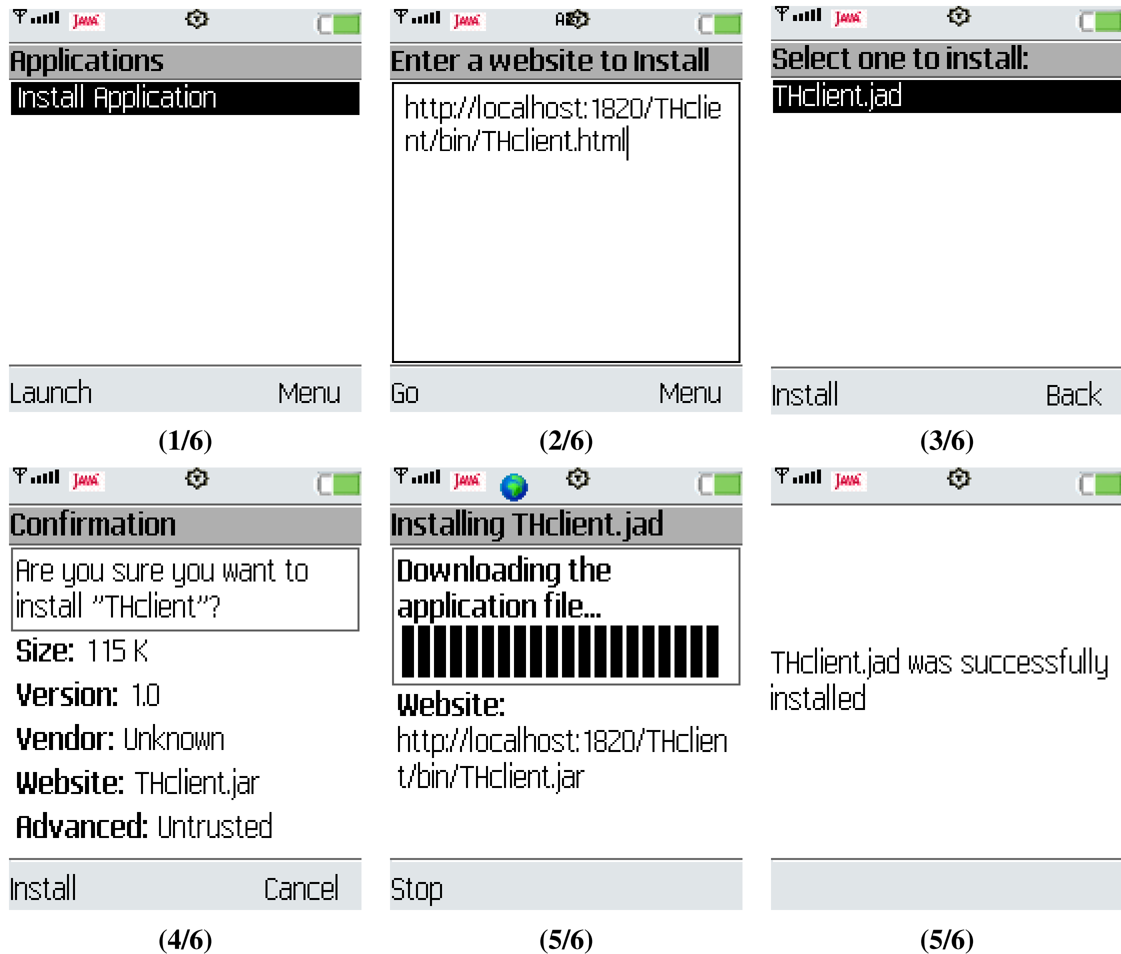


Figure 5-3- 12 Run MIDlet via OTA

So far, we have completed all the steps required to create and deploy a MIDlet using wireless toolkit. These steps have helped us to understand the whole development flow of a creation a J2ME application. Due to the time limitation, we don not program the client code for all the deployed virtual shop Web service. For the demonstration purpose, we only implement the MIDlet which is capable to send simple Web service requests, like login request. To cover all the server side services is left to the future work.

6. Test and Evaluation

In this chapter, the prototype of virtual shop Web service application deployed on the server side will be tested and evaluated based on functionality, performance, and scalability. The function validation is to test whether the implementation fulfils the proposed design and the very importance is to see if it realizes detailed scenarios of system use cases (described in section 3.2). The performance evaluation concentrates on the server performance based on time measurement. That is the calculation of service response time regarding to different type of requests and the XML message size. After that, the server scalability is also analyzed to verify if the server can stand heavy load client requests, such as simultaneous requests and mixture type of requests.

6.1 Test-bed Environment

The target test server is a Compaq Evo N800c with 512 MB in RAM. The operation system was Microsoft Windows XP Professional Service Pack2.

To run the basic functionality validation tests, the following tools were used: Java 2 Platform, Standard Edition (J2SE) version1.4.2 [25], Apache Axis, JUnit 3.8[44] and wsCaller 1.0[45] and Apache JMeter [46].

To test the system performance and scalability, Apache JMeter [46] is used to measure the time consuming according to XML message size, different service request type, number of service request and mixture request. Apache JMeter is a tool that can be used to test application, typically Web applications, to load test functional behavior and measure performance. A typical JMeter test starts from creating a test plan, which might consist of one or more loops, thread groups, samples and listeners. The loop simulates sequential requests to the server with a preset delay. A thread group is designed to simulate a concurrent load. A sample might be a HTTP request, FTP request, or Web Service (SOAP) request. A listener is a component that shows the results of samples. The result can be shown in a tree, table, graphs or simple write to a file [49]. In the following chapters, we are going to create different test plans in the JMeter to perform both performance and scalability tests.

6.2 Function Validation

In order to test whether all kinds of deployed services work satisfactory, unit testing is adopted by developers to evaluate whether the services perform the required functions and return the correct results and data. Unit testing is white box testing which makes use of xUnit testing framework. Based on the programming language, there are a

number of unit testing frameworks, such as *Junit* in Java, *Pyunit* in Python, *VbUnit* in Visual Basic, and etc. Theoretically, unit testing is to help the developer for better system design, and it should be produced at the same time as the source code. In the Test-Driven Development (TDD) technique, which repeatedly first write a test case and then implement only the code necessary to pass the test, the unit test is the first piece of code written by developer. However, in the thesis, we also use the JUnit testing framework to validate the functions.

First download Junit 3.8.1 installation package from site [44]. Unzip and add the `junit.jar` file to *CLASSPATH*. To create the Junit test framework, there are a few steps need to follow:

- Create subclass which inherits *TestCase* base class
- Define individual test case
- Add several test cases to a *Test Suite* and run the suit
- Check for the response from each test case and report failures.

Let say we are going to validate the "*LoginService*". The service authenticates user according to username and password. For the valid user, return the user role, otherwise, return null. To pass the function validation, the real output should be the same as the expected output.

Table 6-2- 1 "LoginService" function validation

Method name	Input (userID, pwd)		Expected Output	Real Output
login	"iw_admin"	"admin"	"Admin"	"Admin"
	"iw_amazon"	"amzon"	"Customer"	"Customer"
	"nonexisting"	"nonexisting"	null	null

First we obtain the service WSLD document, use Apache Axis tool *WSDL2Java* to generate the client stub code. Then to validate such service, we write the Junit test case as shows in Figure 6-2-1.

```
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;
import LS.Services.*;

public class LoginServiceTest extends TestCase {
    LoginService ls;
    protected void setUp() {
        try{
            LoginServiceServiceLocator llo = new LoginServiceServiceLocator();
            this.ls = llo.getLogin();
        }catch (Exception e){e.printStackTrace();}
```

```
    }
   public static Test suite() {
         return new TestSuite(LoginServiceTest.class);
    }
    public void testLoginExistingCustomer() {
         try{
              String role= this.ls.login("iw_amazon","amazon");
              assertEquals(role, "Customer");
         }catch (Exception e){e.printStackTrace();}
    }
    public void testLoginExistingAdmin() {
         try{
              String role= this.ls.login("iw_admin","admin");
              assertEquals(role, "Admin");
         }catch (Exception e){e.printStackTrace();}
    }
    public void testLoginNonexisting() {
         try{
              String role= this.ls.login("nonexisting","nonexisting");
              assertEquals(role, null);
         }catch (Exception e){e.printStackTrace();}
    }
    public static void main (String[] args) {
         junit.textui.TestRunner.run(suite());
    }
}
```

Figure 6-2- 1 Junit test case for Login Web service

Using the Junit test framework, we can set up all test cases for other deployed Web service.

Another more general way to validate the function of a Web service is to use a tool called *wsCaller*. *wsCaller* is a simple Web server client program which written in Java. Download wsCaller-1.0 from site [45] and run it. The initial user interface is shown in Figure 6-2-2

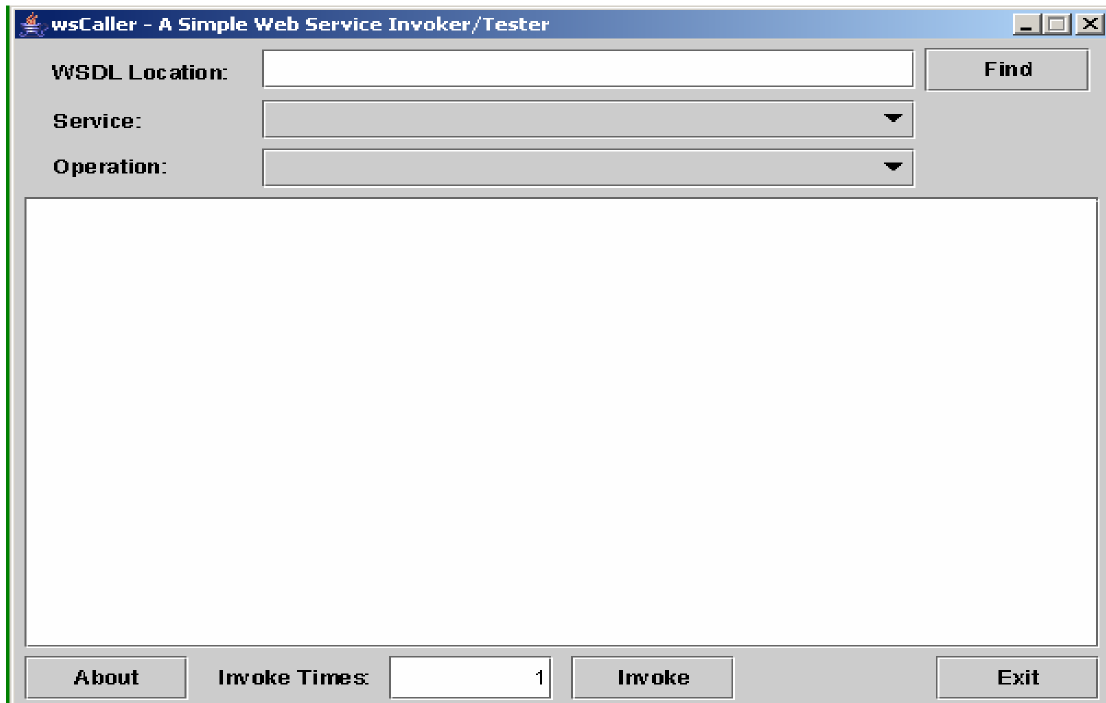
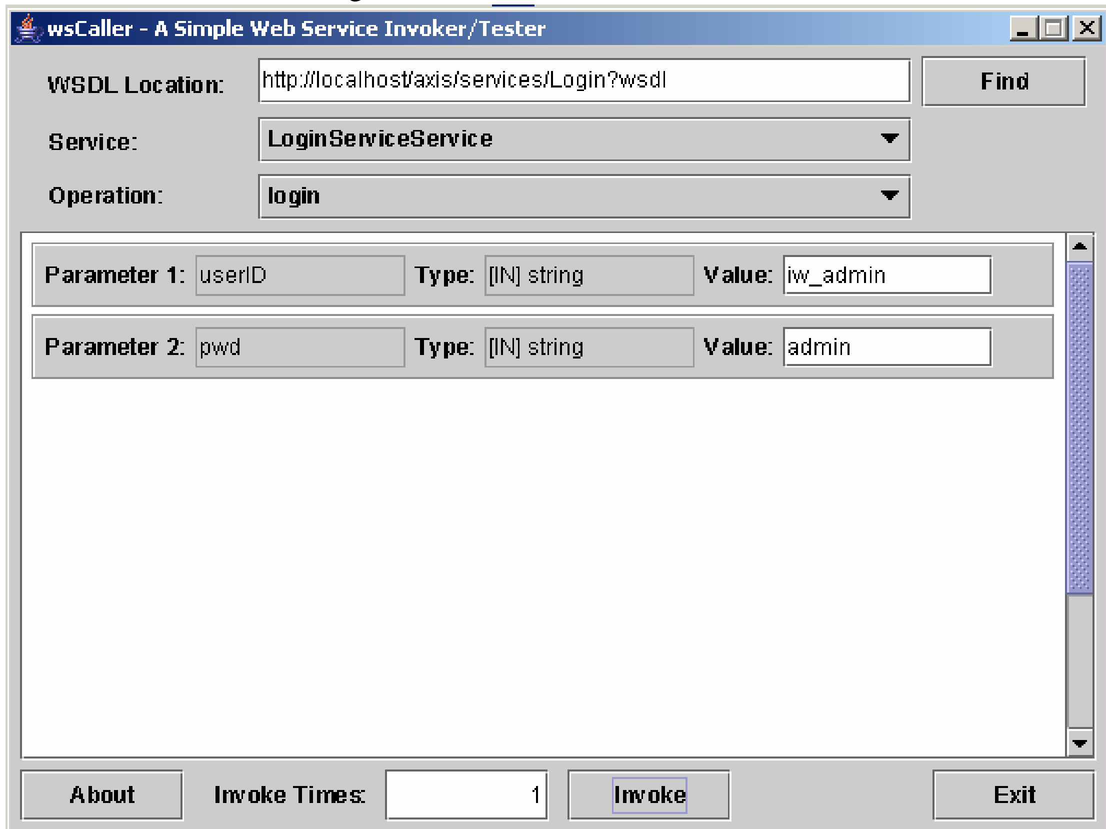


Figure 6-2- 2 wsCaller user interface

Input the location of WSDL document, and find the target service and operation. Input correct parameters and press "Invoke" button. The result dialog will prompt out to show the test result. See Figure 6-2-3



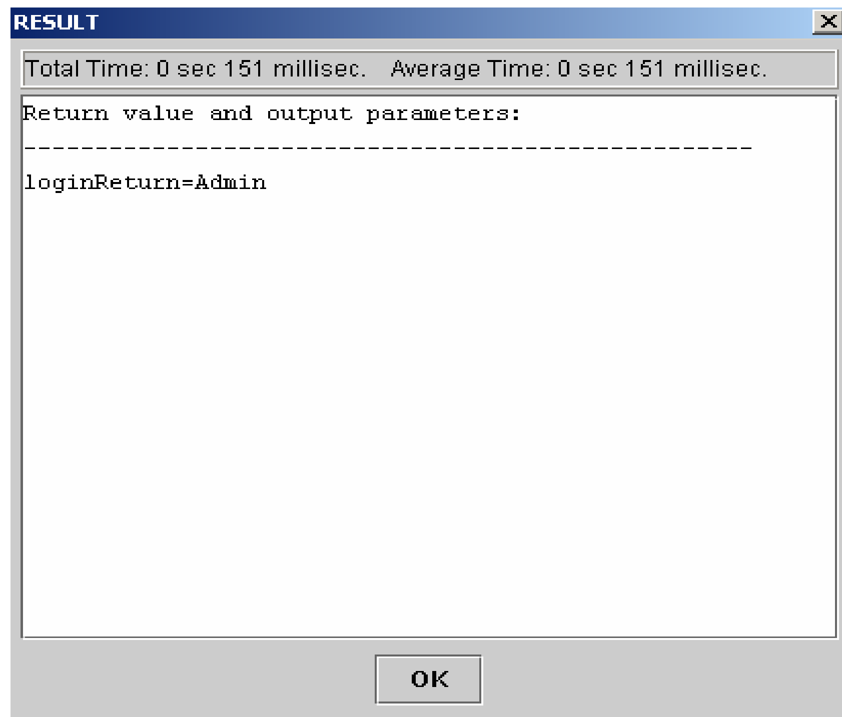


Figure 6-2- 3 wsCaller test result

Without the support of user-defined complex data type is the known limitation of *wsCaller*. We can only test some simple Web services which have standard XML data type as both input and output. For the Web service with complex data type, we have to write our Junit test code.

In addition to use Junit and wsCaller to validate single service funcationality, it is also necessary to define the business work flow using Apache JMeter to verify the realization of detailed use case scenarios. Take the shop customer for example: a typical scenario might be follows: after logining with correct username and password, the customer looks througth his profile and do the profile update. After that, he walks throught the virtual shop and search for specific shop item. If nothing found, he adds wish list and leaves the shop. A number of services are identified in this specific scenario: login -> getProfile -> updateProfile -> searchItem -> addWish. Fortunately, JMeter provide a method to define the customer work flow and help us filful the testing of use case scenario. Create a test plan, add a thread group, spacify the number of running threads (=1), a ramp-up priod (=1) and total loop account (=1). Under the thread group, add identified Web service samples with correct order, and fills in necessary information, like, the link to the WSDL filed, the method name, server IP, port number, path and Soap/XML-RPC Data. Lastly, add a lister "summary report" to the end of the test plan to show the test result. After running the defined test plan, we can easly see that all the tests pass and no error occures. By defining scenior specific test plan, we are able to verify if the implemented system realize the detailed scenarios of system use cases.

6.3 Performance Evaluation

This section will concentrate on performance based on the time measurement. We will first look at the connection between the service time and different XML message size. Then we will try to measure the service time spent on different request type.

Here the ***service time*** donates the time interval between the moment that the test client requests the service, e.g. clicking on start test button and the moment the test client has received and processed the response. Since our test client is placed in the same network as the test server via a 100Mb switch (as Figure 6-3-1), we can assume that the transmission time between the client and server is small enough to be ignored. Therefore, the service time is actually the time from which server receive the SOAP request, interpreter to Java object, perform the internal business logic, and parser the Java object to SOAP response.

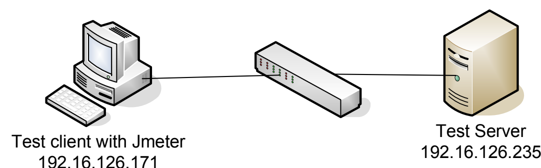


Figure 6-3- 1 Test setup

6.3.1 XML Message Size

Usually, The SOAP XML message size is different depending on the request. To find out how the XML message size effects the service time, we choose 8 SOAP requests which respond 8 different message size from our implemented virtual shop system described in Section5.2.

The test requests are

- `login`: a small login request, which response a simple string
- `search1Item`: a more detailed ("heavy") information query request to get one item (including itmeID, name, quality, price, description and etc.)
- `searchNItem`: a even more heavier request to obtain information about N items information

To setup the expected test scenario, we create JMeter test plan according to the following steps:

1. Start Apache JMeter

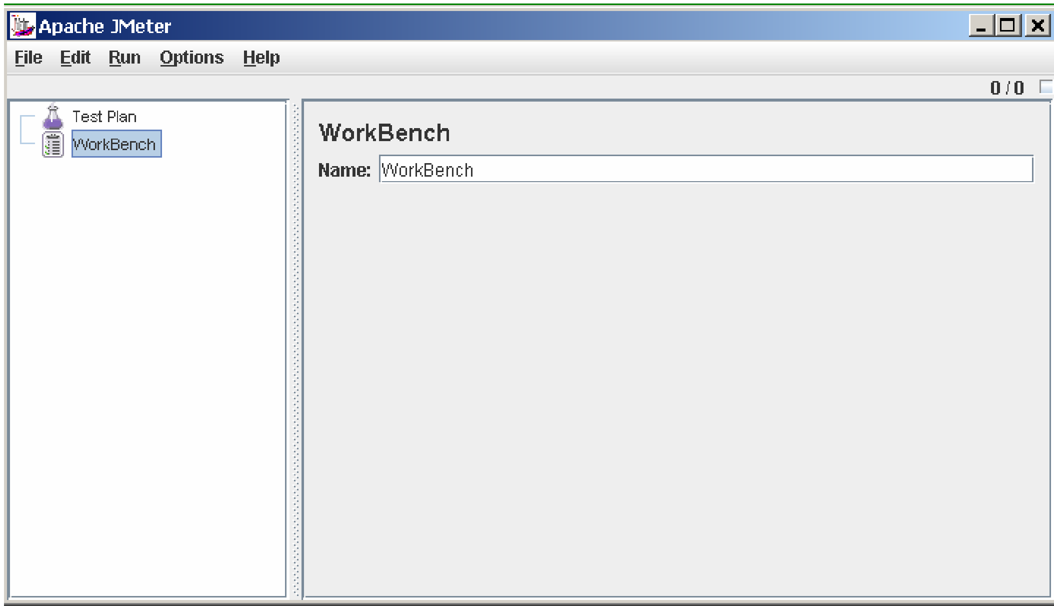


Figure 6-3- 2 Starting Apache JMeter

2. Create test plan, add a thread group, specify the number of running threads (=1), a ramp-up period (=1) and total loop count (=1). Each thread simulates a user and the ramp-up period specifies the time to create all the threads. The loop count defines the running time for a thread.

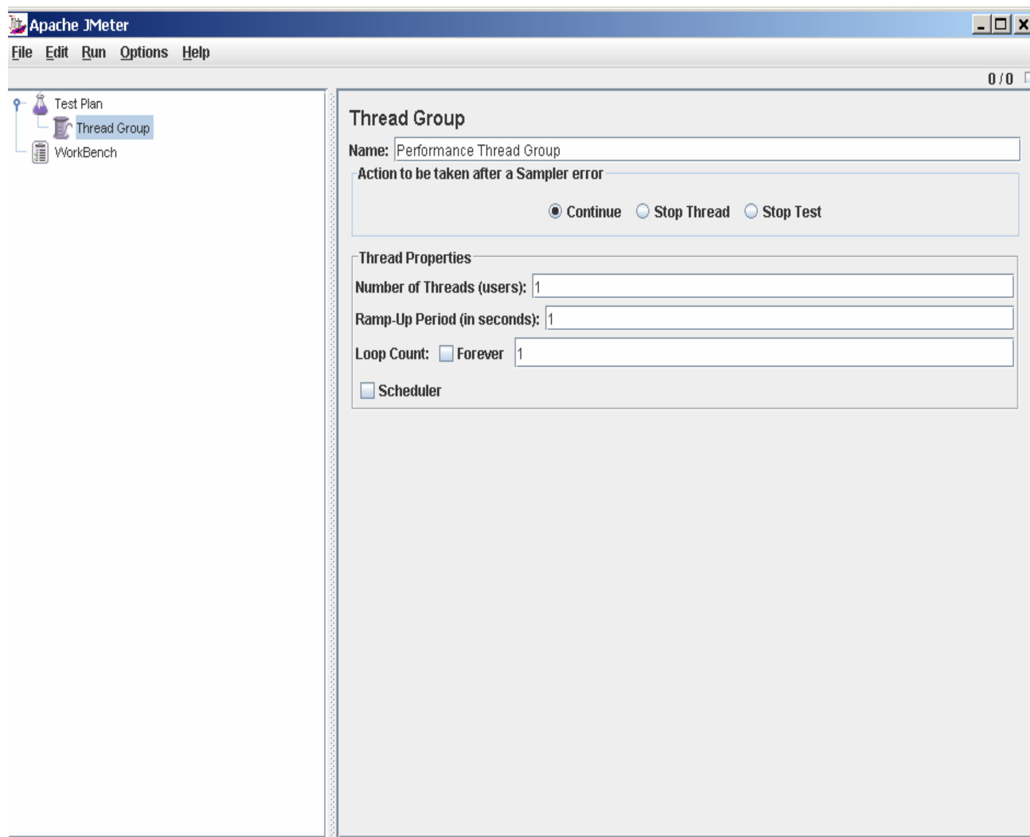


Figure 6-3- 3 Create Test plan and add thread group

3. Under the thread group, add loop controller, set the loop controller count to 10 so we can get the average statistic data for every 10 execution. Add a sample "*WebService(SOAP) Request*": `login` and a listener "*View Results in Table*". In the "*WebService(SOAP) Request*" window, fill in the link to the WSDL filed, the method name, server IP, port number, path and Soap/XML-RPC Data. Repeat above steps and create other SOAP sample requests: `search1Item`, `search2Item` ... `search7Item`. Finally, add a listener "*Aggregated graph*" to the end of the test plan to see the aggregation report after all the test cases are done.

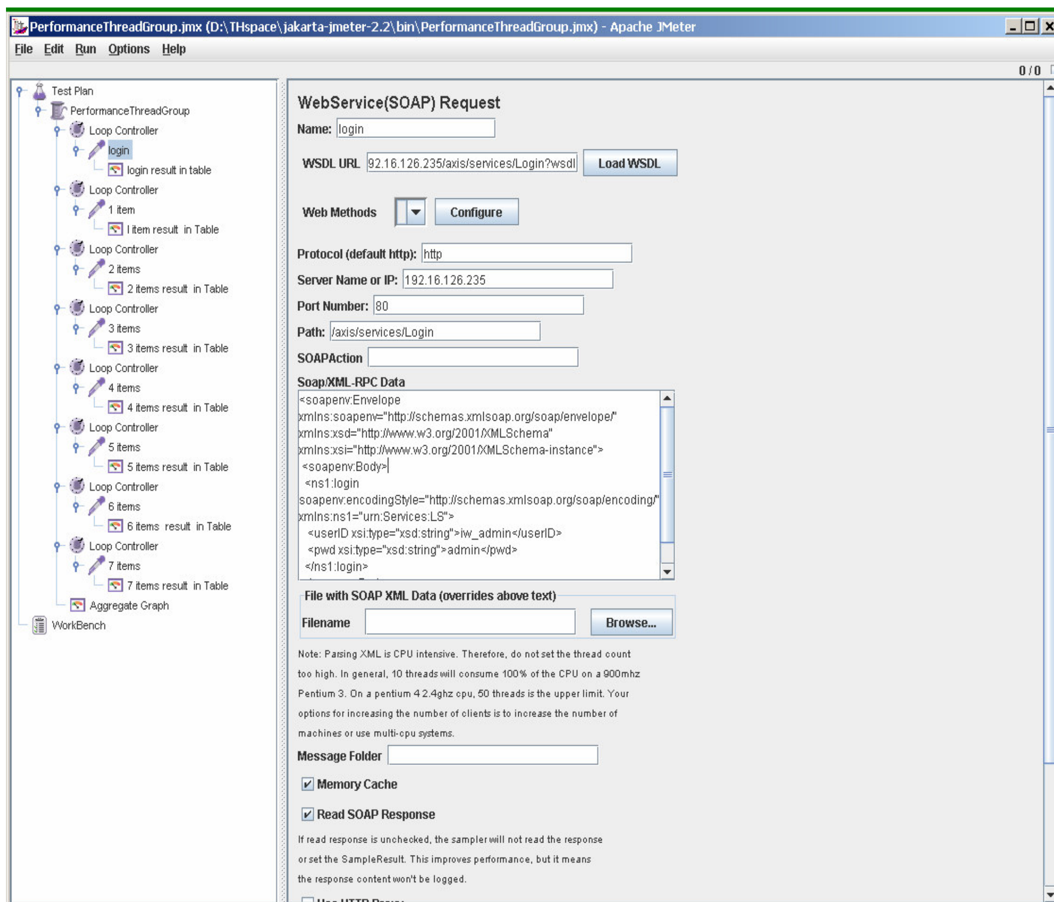


Figure 6-3- 4 Create loop controller, SOAP sample and listener

4. Run the load test by choosing *Run* from the menu and start.

5. After all the test cases are done, we can look at the summary report. It totals the response information and provides request count, min, max, average, error rate, approximate throughput (request/second) and Kilobytes per second throughput. (See Table 6-3-1 and Figure 6-3-5)

Table 6-3- 1 Summary report for XML message size

Summary Report

Name: Summary Report

Write All Data to a File

Filename | Browse... | Log Errors Only | Configure

Label	# Samples	Average	Min	Max	Error %	Throughput	KB/sec	Avg. Bytes
login	10	190	125	204	0.00%	5.2/sec	2.32	0.45
1 item	10	198	187	219	0.00%	5.0/sec	6.85	1.37
2 items	10	200	188	218	0.00%	5.0/sec	10.73	2.15
3 items	10	198	188	203	0.00%	5.0/sec	14.64	2.93
4 items	10	196	187	203	0.00%	5.0/sec	18.39	3.68
5 items	10	2098	2094	2109	0.00%	28.5/min	2.12	4.45
6 items	10	2092	2000	2110	0.00%	28.7/min	2.49	5.22
7 items	10	2100	2093	2110	0.00%	28.5/min	2.84	5.97
TOTAL	80	909	125	2110	0.00%	1.1/sec	3.59	3.28

Where:

- `Label` - The label of the sample.
- `# Samples` - The number of samples for the URL
- `Average` – (ms) The average time of a set of results
- `Min` – (ms)The lowest time for the samples of the given URL
- `Max` – (ms) The longest time for the samples of the given URL
- `Error` % - Percent of requests with errors
- `Throughput` - Throughput measured in requests per second/minute/hour
- `Kb`/`sec` - The throughput measured in Kilobytes per second
- `Avg. Bytes` - average size of the sample response in bytes.[50]

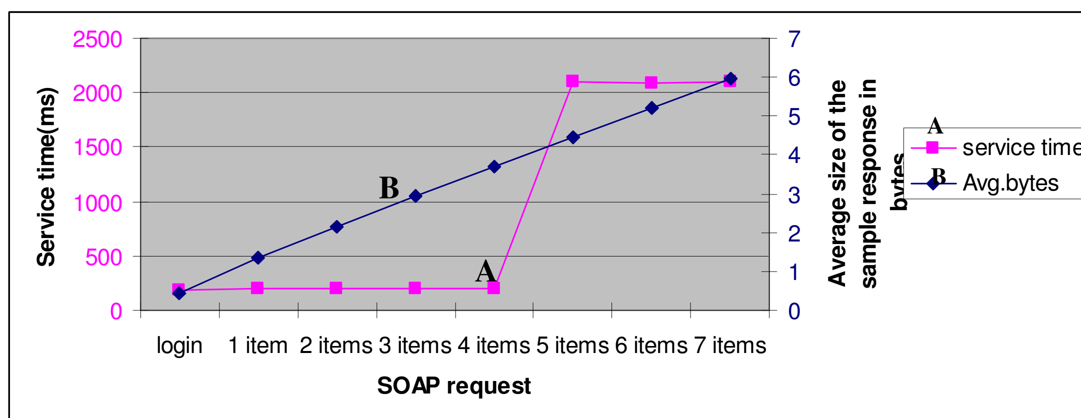


Figure 6-3- 5 Service time with XML message size

In summary, as Table 6-3-1 shows, there is no significant difference between service times for the XML message with average bytes from 0.45 to 3.68. While for the request with SOAP response message size bigger than 4.45, the service time increase dramatically. Consider the fact that the test client and the test server are locating in the same Ethernet network, the transmission time and network delay can be ignored. Therefore, the reason that the service time depends on the message size after certain point is that SOAP overheads. SOAP overheads include: extracting the SOAP

envelope; parsing the contained XML information; SOAP requires type information in every SOAP message; binary data gets expanded (by an average of 5-fold) when included in XML, and also requires encoding/decoding. With the growth of the XML message size, more effect are taken to preprocess and post process a client request, which ends with much more service time.

However, our conclusion that the service time increases with the addition of XML message size is actually based on a precondition. We assume that when the server receives a SOAP request, it spends much more time on SOAP message handling than the other operations, like database queries. With the big size of XML message, it is obvious that the time spending on both handling SOAP message and operating database is longer than a small size of XML message, however, if the increasing rate of former is higher than later, our conclusion could be more convincing and scientific. In order to gather such statistic information and investigate the time spending on different part of program, a code profiling is highly recommended. Code profiling is an investigation of a program's behavior using information gathered at the run time, it is used to determine which part of program need to be optimized for speed or memory usage. It also allow the developer to set the breakpoint at runtime to get a detailed listing of number of times a particular function has executed and the amount of time spent within each function. Using a profiler tool to analysis the server side implementation is planed in the future work.

6.3.2 Request Type

Another interesting experiment about the performance of Web service is the impact of different request type on service time. Here we define the ***request type*** based on the complexity of application's internal business logic or functionality. For example, if we classify that all the database related operations, such as add a record, delete a record, update a record and etc, we will find out that they all have a similar service response time. That is the server responds to this kind of service with a relatively even time.

We still use JMeter to make SOAP Web service requests to our virtual shop system. To setup expected test scenario, a bunch of testing elements are created under the JMeter test tree: a test plan, a thread loop, a group of simple controller, SOAP Web service request sample and the result listener. For the thread loop, we set the number of threads equal to 1, and the loop count equal to 10. So each test case runs 10 times to calculate an average value. The selected request types are following:

- `login`: authenticate the valid users. A typical database SEARCH operation.
- `updateProfile`: change the user's profile. A typical dateable UPDATE operation.
- `addtoCart`: add a shopping item to the cart. A database INSERT operation.

- `removeFromCart`: delete a shopping item from the cart. A database DELETE operation

Run the load test, we get very detailed test result from the summary report, as Table 6-3-2 and Figure 6-3-6

Table 6-3- 2 Summary report for request type

Summary Report

Name: Summary Report

Write All Data to a File

Filename [ ] Browse... Log Errors Only Configure

Label	# Samples	Average	Min	Max	Error %	Throughput	KB/sec	Avg. Bytes
login	10	179	110	188	0.00%	1.4/sec	0.58	0.42
updateProfile	10	206	203	219	0.00%	1.3/sec	0.51	0.38
addtoCart	10	209	203	219	0.00%	1.3/sec	0.50	0.37
removFromCart	10	187	172	203	0.00%	1.4/sec	0.51	0.38
TOTAL	40	195	110	219	0.00%	5.0/sec	1.96	0.39

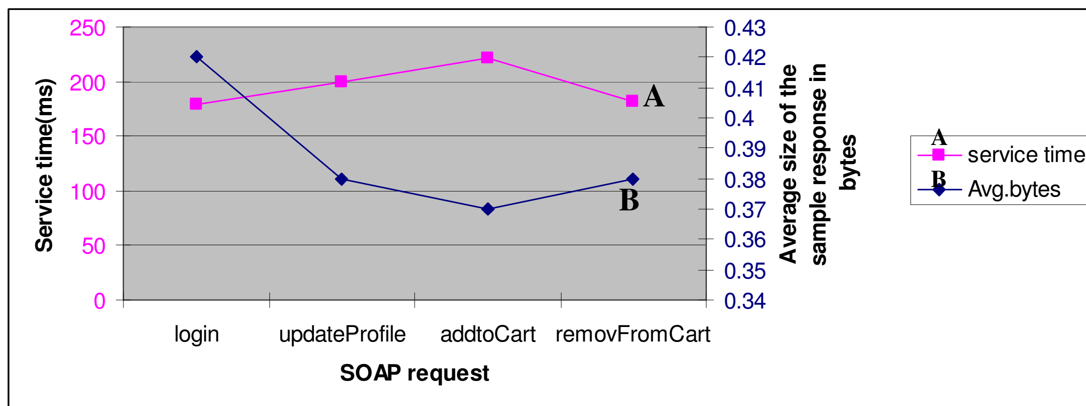


Figure 6-3- 6 Service time with request type

It can be seen from the curve chart that with the precondition that the average size of the sample response in bytes keeps similar, there is no significant difference in the amount of service time for the testing request type. The service time remains constant at about 200 milliseconds.

6.4 Scalability Analysis

In this section, we present two different sets of experiments. In the first subsection, we evaluate the scalability of Web service according to number of requests. This experiments show that how the server scalable is with an increasing number of simultaneous clients for single service. In the second subsection, we demonstrate how the server responds to the simultaneous clients with mixture of services.

6.4.1 Number of Requests

We will now try to find out how the server scalability is according the number of simultaneous requests for one service. First, let us look at the time it takes for 10 users to invoke the `getProfile` service.

Start JMeter, create a thread group element, and the `getProfile` SOAP request sample. The thread group element controls the number of threads JMeter will use to execute the test. The controls for a thread group allow us to:

- Set the number of threads: simulate the number of concurrent users
- Set the ramp-up period
- Set the number of times to execute the test

Each thread will execute the test plan in its entirety and completely independently of other test threads. Multiple threads are used to simulate concurrent connections to your server application.

The ramp-up period tells JMeter how long to take to "ramp-up" to the full number of threads chosen. If 10 threads are used, and the ramp-up period is 100 seconds, then JMeter will take 100 seconds to get all 10 threads up and running. Each thread will start 10 (100/10) seconds after the previous thread was begun. If there are 30 threads and a ramp-up period of 120 seconds, then each successive thread will be delayed by 4 seconds. [50]

We begin with 10 threads, ramp-up period equal to 10, and let the test run 10 times. See Figure 6-4-1.

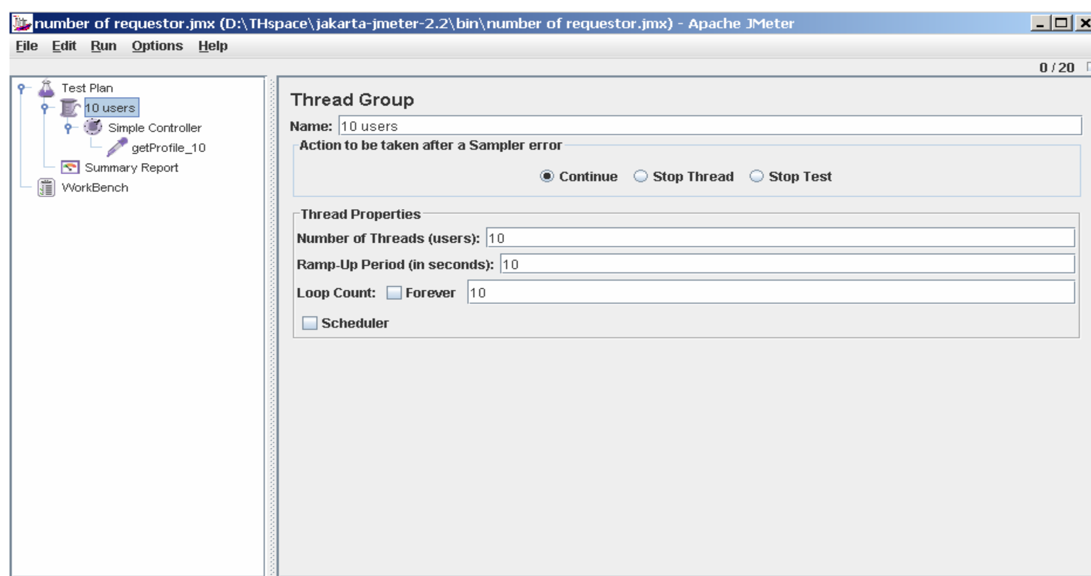


Figure 6-4- 1 Thread group for number of requests

Run the load test by clicking "*Run*" menu. After the first 10 users' test is done, we repeat the same procedure, but reconfigure the thread number to 20, 30, 40, 50, until 110 to simulate the increase of number of requests. Let each test run 10 times. Table 6-4-1 and Figure 6-4-2 present the experimental result.

Table 6-4- 1 Summary report for number of requests

Summary Report

Name: Summary Report

Write All Data to a File

Filename | Browse... | Log Errors Only | Configure

Label	# Samples	Average	Min	Max	Error %	Throughput	KB/sec	Avg. Bytes
getProfile_10	100	198	125	500	0.00%	9.1/sec	12.03	1.32
getProfile_20	200	191	109	219	0.00%	17.5/sec	23.07	1.32
getProfile_30	300	196	141	235	0.00%	25.6/sec	33.77	1.32
getProfile_40	400	202	140	531	0.00%	34.1/sec	44.91	1.32
getProfile_50	500	201	125	500	0.00%	42.6/sec	56.06	1.32
getProfile_60	600	206	109	500	0.00%	50.9/sec	67.09	1.32
getProfile_70	700	212	156	516	0.00%	59.1/sec	77.86	1.32
getProfile_80	800	234	156	578	0.00%	67.1/sec	88.40	1.32
getProfile_90	900	316	140	750	0.00%	71.7/sec	94.50	1.32
getProfile_100	1000	443	125	937	0.00%	71.7/sec	94.42	1.32
getProfile_110	1100	552	140	1204	0.00%	72.1/sec	95.02	1.32

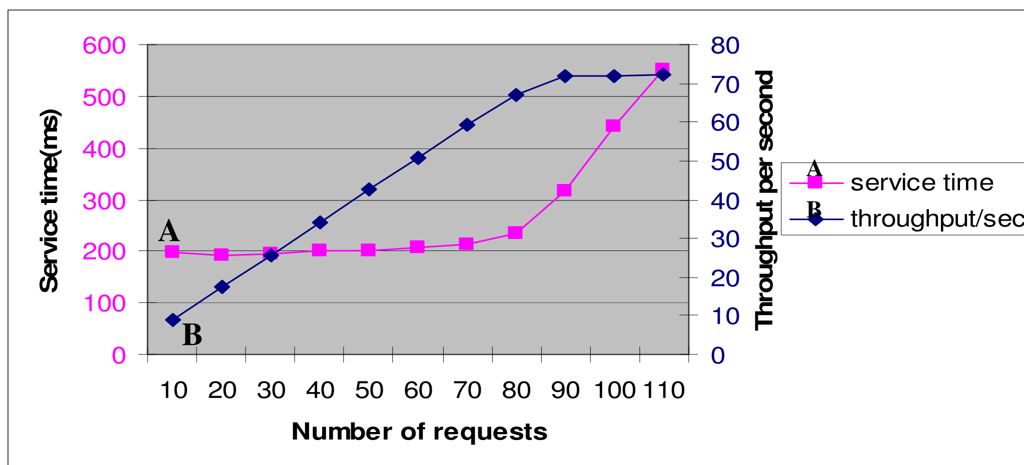


Figure 6-4- 2 Service time and throughput for number of requests

In the Figure 6-4-2, the curve B stands for the server's throughput, which measures in requests per second, while the curve A stands for the average service time for a specified number of requests. From the graph, we can see clearly that when the number of request is in the range of 10 to 80, the service time keeps constantly and meanwhile the corresponding server throughput is on the rise from about 9% in 10 requests to the peak of approximately 71% for 90 simultaneous client requests. However, from this time onwards, the server throughput value stays constant around 71%, while the service time experience a sharp increase from 200 ms to almost 500 ms

In brief, the increase in response time with the addition of threads is obvious. Increasing this number from 10 to 70 does not make a big impact. However, when the number of thread close to or exceeding the server constrains, server seems remains its max throughput capability and no longer capable of responding the client promptly.

6.4.2 Mixture Requests

The mixture requests simulate the real life scenario that multiple clients simultaneously ask for rather than one single service but different services. So this mixture requests experiment demonstrates how the server reacts when receiving multiple client requests for different service.

Fortunately, JMeter can still help us to perform the mixture requests testing. Start JMeter, instead of adding one thread group under the test plan tree, we add 5 thread groups. Each thread group sends Web service requests to one service deployed on the server and totally there are 5 different types of requests. They are: `login`, `updateProfile`, `getProfile`, `searchItem`, `getShop` requests.

For each thread group, we starts with 2 threads (users), with ramp-up period equal to 10 and let the test run 10 loops. See Figure 6-4-3.

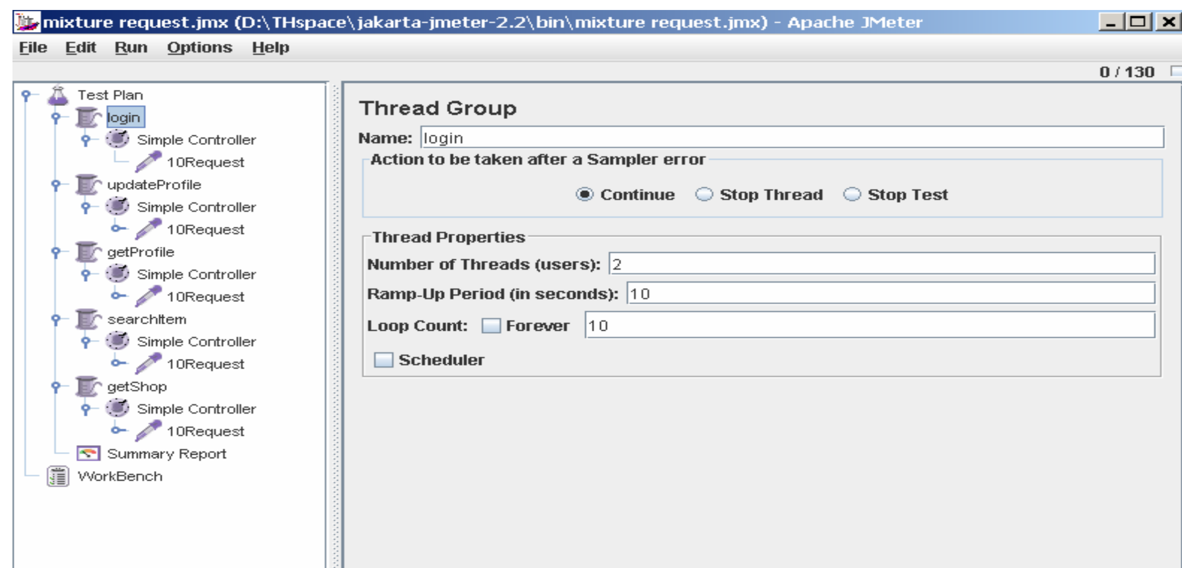


Figure 6-4- 3 Thread group for mixture requests

As we define 5 thread groups, there are actually 10 threads in total (2 threads× 5 services), which are evenly spread into 5 services. For each thread group, let it run 10 times (loop count) to calculate an average service time. Therefore, this test round will create 100 samples (2 threads/group × 5 groups × 10 loops/thread).

After the first test round is done, we increase the thread number per each group from 2 to 4, to send totally 200 samples (4 threads/group × 5 groups × 10 loops/thread). With the addition of threads number, the summary report is displayed as Table 6-4-3

and Figure 6-4-4.

It can be seen obviously that the trend of both service time and the server throughput for the number of mixture service requests is pretty much the same as to the single service request. The server seems to be able to handle up to 90 clients simultaneously and server all their requests in prompt time. When we run 100 or more clients, the server is reaching the max capability, the dramatic service time increase take place.

Table 6-4- 2 Summary report for mixture requests

Summary Report

Name: Summary Report

Write All Data to a File

Filename [ ] Browse... [ ] Log Errors Only Configure

Label	# Samples	Average	Min	Max	Error %	Throughput	KB/sec	Avg. Bytes
10Request	100	193	125	250	0.00%	14.3/sec	11.27	0.79
20Request	200	206	156	563	0.00%	20.9/sec	16.49	0.79
30Request	300	202	109	422	0.00%	29.0/sec	22.79	0.79
40Request	400	206	125	547	0.00%	37.0/sec	29.12	0.79
50Request	500	216	125	547	0.00%	45.1/sec	35.47	0.79
60Request	600	218	125	3359	0.00%	53.5/sec	42.10	0.79
70Request	700	224	125	594	0.00%	60.4/sec	47.52	0.79
80Request	800	229	125	547	0.00%	68.5/sec	53.88	0.79
90Request	900	280	109	735	0.00%	74.4/sec	58.57	0.79
100Request	1000	370	109	859	0.00%	76.5/sec	60.19	0.79
110Request	1100	394	109	1094	0.00%	81.7/sec	64.28	0.79
120Request	1200	495	110	1234	0.00%	80.3/sec	63.17	0.79
130Request	1300	561	109	1391	0.00%	82.3/sec	64.77	0.79

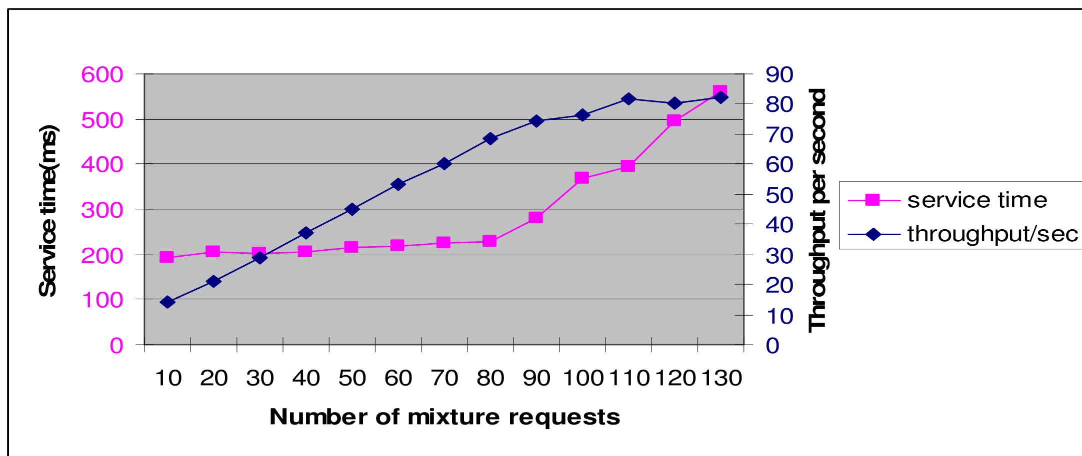


Figure 6-4- 4 Service time and throughput mixture requests

Actually in the real life, it is impossible to have equal number of requests for each different type of service. Therefor a more practical test case should be to have different portion of each service and then mix them together. For example, the total number of requests can keep constant in each test round while `searchItem` service has 50% of the total requests, `login` has 30%, `updateProfile` and `getProfile` have 10% and `getShop` requests has 10%.

As we can see from the scability analysis result the service time will increase rapidly after the server reaches its maximum handling capability. And for our application server this capability is too small when comparing to the real life environment. The limits of the application server we put in the scalability test is around to handle 90 requests at the same time. Obviously this number is too small to put this application into real life test environment where you can get thousands of requests at the same time. This is encountered as a scalabilty problem as the service grows in the real commercial case. However there are many ways we can do to improve the scability performance of the server side in the future when we want to deploy our web service application in a large scale.

- Deploy the web service on several servers along with their own database servers.
- Set up load balance between client and application server which can distribute service requests evenly to different application and database servers based on the type of requests or the the response time of each server.

7. Conclusion and Future Work

7.1 Conclusion

By spending a lot of time in investigating and studying the relevant Web Service standards and protocols we get to know what Web service is, where it will orient in the future and the meaning of utilizing it. Nevertheless, not only focusing on theoretical knowledge we also develop a Web service based application and a client on a cell phone as the final consumer, through which we get much deeper understanding of Web service and how the Web service is being used: how we can implement and deploy it. Based on the fact that all the coding work is using J2SE and J2ME technology, we also improve our Java developing skills a lot.

Regarding to the implementation part, we choose to take a quite common topic “virtual shop” to put our theoretical knowledge of Web service, which is got from protocol specifications, standard documents and technical articles written by the pioneers in this field, into practice.

At the server side we succeed in realizing all the use cases identified in the design phase, deploying them on the server and validating each function by writing both unit test case and individual Web service client. For the client on the cell phone, we only implement part of the identified use cases and increase our experience in dealing with Web service on J2ME platform.

Further more, by investigating and using a third party tool *Jmeter* [46] we have done a thoroughly performance evaluation and scalability analysis to our deployed Web services, which could be a very practical and useful example to the similar test work.

Finally, we hope our effort and experience gained from this thesis project can also help the people who want to develop and deploy Web service in some way and provide them with a basic guideline when coming to Web service development.

7.2 Future Work

First, more investigations and implementations can be done with respect to:

- WS-Security, which consist of a family of specifications and introduce how we can enforce integrity and confidentiality on Web Services messaging.
- WS-Resource, which *is a family of specifications that enable and standardize interfaces for Web Services to give the appearance of statefulness*. [51] It stipulates how the Web Services can access and manipulate the remote data resources in a standard way.

Second, more live tests can be done for the cell phone client based on a real network environment. Access the Web services from the cell phone by different wireless access network, such as, 802.11b wireless LAN, Bluetooth, as well as GPRS networks, and test the performance for each of them.

Third, add more functions to the simple MIDlet application on the cell phone which got limited functionalities in this thesis project because of luck of user-defined data type support from kSOAP2.

Fourth, for the server side application architecture, the database is manipulated from the service code directly. Considering the expandability and repeatability of the code, it's better to implement an adaptor layer between the service business logical and database.

Fifth, due to the facts of limited amount of memory, GUI capabilities, process power, together with network connection, bandwidth limitation and so forth, design a good J2ME client is becoming a issue that concern not only basic design consideration for the wire device, but also other aspects, like how to limit the data exchange rate, reduce the amount of data transferred, simplify the user GUI design, take the data type exchanged that require less pre- and post-processing and etc.

Lastly, to optimize the server side implementation, a code profiling is highly recommended. A profiling records the frequency and time spends on each line of the code, as the application is running. It also help the developer to improve the code quality based on the summary shows that the slowest code and methods.

Abbreviation

API	Application Programming Interface
B2Bi	Business-to-Business Integration
CDC	Connected Device Configuration
CLDC	Connected, Limited Device Configuration
COM	Component Object Model
CORBA	Common Object Request Broker Architecture
DCOM	Distributed COM
DII	Dynamic Invocation Interface
DTD	Document Type Definition
EAI	Enterprise Application Integration
ESME	External Short Messaging Entity
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IDE	Integrated Development Environment
J2ME	Java2 Platform, Micro Edition
J2SE	Java 2 Standard Edition
JAX-RPC	Java APIs for XML-Based Remote Procedure Call
JDBC	Java Database Connectivity
JDK	Java Development Kit
JRE	Java Runtime Environment
JSP	JavaServer Pages
JVM	The Java Virtual Machine
KVM	The K Virtual Machine
MIDP	Mobile Information Device Profile
ORPC	Object Remote Procedure Call
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SGML	Standard Generalized Markup Language
SMPP	Short Message Peer-to-Peer protocol
SMS	Short Message Service
SMSC	Short Message Service Center
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
UDDI	Universal Discovery Description and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
URN	Uniform Resource Names
W3C	The World Wide Web Consortium
WSDL	Web Service Description Language
XML	eXtensible Markup Language

Reference

1. M.Gudgin et.al June 24.2003, Simple Object Access Protocol (SOAP) 1.2.
   [Online] Available: http://www.w3.org/TR/soap/
2. E.Christensen et.al March15.2001, Web service Description Language (WSDL) 1.1.
   [Online] Available: http://www.w3.org/TR/wsdl
3. B.Atkinson et.al October14.2003, Universal Description, Discovery and Integration (UDDI) 3.0.1.
   [Online] Available: http://uddi.org/pubs/uddi_v3.htm
4. T.Bray et.al February04.2004, Extensible Markup Language (XML) 1.0 (Third Edition).
   [Online] Available: http://www.w3.org/TR/REC-xml/
5. Roger Wolter.2001, XML Web Service Basics-An overview of the value of XML Web service for developers, with introductions to SOAP, WSDL, and UDDI, Microsoft Corporation
   [Online] Available: http://msdn2.microsoft.com/en-us/library/ms996507.aspx
6. Aaron Skonnard.October.2002, The Birth of Web service, *MSDN Magazine* Microsoft Corporation
   [Online] Available: http://msdn.microsoft.com/msdnmag/issues/02/10/XMLFiles/
7. Microsoft Corporation 2001, Global XML Web service Architecture White Paper
   [Online]Available: http://www.gotdotnet.com/team/XMLwebservices/gxa_overview.aspx
8. L.F.Cabrera, et.al Nov.2004, Web service Coordination (WS- Coordination)
   [Online] Available: http://dev2dev.bea.com/pub/a/2004/03/ws-coordination.html
9. Satya Sanket Sahoo July.2004, Web service Composition (An AI-based Semantic Approach), Research Seminar (CSCI 8990). Department of Computer Science University of Georgia
   [Online]Available:
   http://lsdis.cs.uga.edu/~satya/Presentation/Web%20Services%20Composition.ppt
10. Ian Foster, Jeffrey Frey et.al May.2004, Modeling Stateful Resources with Web service White Paper
    [Online]Available:
    http://www-128.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf
11. Infravio, Inc. July.2003, Web Services: Next generation application architecture. White paper
    [Online]Available:
    http://www.infravio.com/forms/resource_signup.php?download=infravio_whitepaper.pdf
12. World Wide Web Consortium
    [URL]: http://www.w3.org
13. S. Graham, et.al June.2004, *Buiding Web service with Java: Making sense of XML, SOAP, WSDL and UDDI. 2nd Edition*. ISBN: 0672326418, Publisher: Sams.

14. W3 Schools
[URL]: http://www.w3schools.com/default.asp
15. Microsoft Web service Developer Center
[URL]: http://msdn.microsoft.com/webservices/
16. A. Skonnard March.2003, Understanding SOAP,
[Online]Available:
http://msdn.microsoft.com/webservices/understanding/webservicebasics/default.aspx?pull=/library/en-us//dnsoap/html/understandsoap.asp#understandsoap_topic4
17. U.Wahli et.al February09.2004, *WebSphere Version 5.1 Application Developer 5.1.1 Web service Handbook*.
[Online] Available: http://www.redbooks.ibm.com/abstracts/sg246891.html
18. H.Kreger May.2001, Web service Conceptual Architecture (WSCA1.0).
[Online]Available:
http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf
19. A. Skonnard October.2003, Understanding WSDL.
[Online]Available:
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwebsrv/html/understandwsdl.asp
20. Web Service Interoperability Organization (WS-I)
[URL]: http://www.ws-i.org
21. Sami I.Sarhan June.2003, XML Transformer Web Service XML-SQL Web Application. Master Paper Computer Science Department Florida Sate University
[Online]Available: http://www.cs.fsu.edu/research/reports/TR-030603.pdf
22. Inderjeet Singh et.al June.2004. *Designing Web service with the J2EE 1.4 platform JAX-PRC, SOAP and XML Technologies*. ISBN-13-978-0-321-20521-6, Published by Prentice Hall.
23. kSOAP Project
[URL]: http://ksoap.objectweb.org/ [URL]: http://ksoap2.sourceforge.net/
24. Vikram Goyal February.2005, J2ME Tutorial, Part 1: Creating MIDlets
[URL]: http://today.java.net/pub/a/today/2005/02/09/j2me1.html
25. Sun Microsystems. Java 2 Platform, Standard Edition (J2SE).
[URL] Available: http://java.sun.com/j2se/1.4.2/index.jsp
26. Eclipse Project
[URL] Available: http://www.eclipse.org/projects/index.html
27. MySQL AB, MySQL Database Server
[URL] Available: http://dev.mysql.com/downloads/mysql/4.0.html
28. Apache, Web service – Axis
[URL] Available: http://ws.apache.org/axis/
29. Apache Jakarta Project
[URL] Available: http://jakarta.apache.org/tomcat/
30. Apache Software Foundation
[URL] Available: http://www.apache.org/
31. Apple Computer, Inc Aug.2005, WebObjects Web service Programming Guide
[URL]:

http://developer.apple.com/documentation/WebObjects/Web_Services/About/chapter_1_section_1.html
32. JavaBeans Activation Framework (JAF)
[URL] Available: http://java.sun.com/products/javabeans/jaf/index.jsp
33. J2EE JavaMail
[URL] Available: http://java.sun.com/products/javamail/index.jsp
34. Apache XML Security
[URL] Available: http://xml.apache.org/security/
35. SQLManager.net
[URL] Available: http://sqlmanager.net/en/
36. MySQL Connector/J36
[URL] Available: http://dev.mysql.com/doc/connector/j/en/index.html
37. Dion Almaer May.2002, Creating Web service with Apache Axis
[URL]: http://www.onjava.com/pub/a/onjava/2002/06/05/axis.html
38. Java SMPP API Homepage
[URL] Available: http://smppapi.sourceforge.net/index.php
39. The Mobile Landscape
[URL] Available: http://www.mobilelandscape.co.uk/
40. Axis User's Guide version 1.2
[Online] Available: http://ws.apache.org/axis/java/user-guide.html
41. Dejan Bosanac Sep.2004, SMS-Powered Applications.
[URL]: http://www.onjava.com/pub/a/onjava/2004/06/09/sms.html
42. Short Message Feb.2003, Peer-to-Peer Protocol Specification V5.0.
[URL]: http://smsforum.net/doc/download.php?id=smppv50
43. Dejan Bosanac Oct.2004, Job Scheduling in Java.
[URL]: http://www.onjava.com/pub/a/onjava/2004/03/10/quartz.html
44. Erich Gamma and Kent Beck, JUnit 4.1
[URL]: http://www.junit.org/index.htm
45. wsCaller, a general Web service client and test tool
[URL]: http://www.contextfree.net/wangyg/c/wsCaller/wsCaller.html
46. Apache Jmeter
[URL]: http://jakarta.apache.org/jmeter/index.html
47. Michael Kroll and Stefan Haustein, June.2002. *J2ME Application Development*. ISBN:0-672-323095-9, published by Pearson Education
48. J2ME(CLDC/MIDP) Introduction
[URL]: http://www.zdnet.com.cn/developer/code/story/0,3800066897,39147380,00.htm
49. Dmitri Nevedrov August.2006, Using JMeter to Performance Test Web Services
[URL]: http://dev2dev.bea.com/pub/a/2006/08/jmeter-performance-testing.html
50. Apache Jmeter User's manual
[URL]: http://jakarta.apache.org/jmeter/usermanual/index.html
51. WS-Resource definition
[URL]: http://en.wikipedia.org/wiki/WS-Resource
52. Sony Ericsson SDK 2.2.0 for the Java(TM) ME platform
[URL]: http://developer.sonyericsson.com/site/global/home/p_home.jsp

Appendix WSDL Document

The appendix contains the WSDL file of core Web service deployed at server side, which includes:

- Admin Web Service: describe the service interface for the administrator of the virtual shop.
- Customer Web Service: describe the service interface for the customer.
- Owner Web Service: describe the service interface for the owner.
- Login Web Service: describe the service interface of user authentication.
- Mail Web Service: describe the service interface of sending e-mail notification of virtual shop to end users.
- common XML Schema File: contain the user defined exception data type description.
- dataType XML Schema File: contain all the user defined data type description.

1. Admin Web Service WSDL File

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions xmlns:impl="urn:Services:AS" xmlns:intf="urn:Services:AS"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns2="urn:Services:data"
    xmlns:tns3="urn:Services:common" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="urn:Services:AS">
- <wsdl:types>
- <xsd:schema>
  <xsd:import namespace="urn:Services:data" schemaLocation="dataType.xsd" />
  <xsd:import namespace="urn:Services:common" schemaLocation="common.xsd" />
  </xsd:schema>
- <xsd:schema targetNamespace="urn:Services:AS">
- <xsd:complexType name="ArrayOf_shopType">
- <xsd:complexContent>
- <xsd:restriction base="soapenc:Array">
  <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="tns2:shopType[]" />
  </xsd:restriction>
  </xsd:complexContent>
  </xsd:complexType>
  </xsd:schema>
  </wsdl:types>
- <wsdl:message name="getShopsRequest">
  <wsdl:part name="in0" type="xsd:string" />
  <wsdl:part name="in1" type="xsd:string" />
  </wsdl:message>
```

```
- <wsdl:message name="getShopsResponse">
   <wsdl:part name="getShopsReturn" type="impl:ArrayOf_shopType" />
  </wsdl:message>
- <wsdl:message name="manageShopRequestRequest">
   <wsdl:part name="shopID" type="xsd:int" />
   <wsdl:part name="status" type="xsd:string" />
  </wsdl:message>
   <wsdl:message name="manageShopRequestResponse" />
- <wsdl:message name="ShopException">
   <wsdl:part name="shopErr" type="tns3:ShopException" />
  </wsdl:message>
- <wsdl:portType name="AdminService">
- <wsdl:operation name="getShops" parameterOrder="in0 in1">
   <wsdl:input name="getShopsRequest" message="impl:getShopsRequest" />
   <wsdl:output name="getShopsResponse" message="impl:getShopsResponse" />
   <wsdl:fault name="ShopException" message="impl:ShopException" />
  </wsdl:operation>
- <wsdl:operation name="manageShopRequest" parameterOrder="shopID status">
   <wsdl:input name="manageShopRequestRequest" message="impl:manageShopRequestRequest"
     />
   <wsdl:output name="manageShopRequestResponse"
     message="impl:manageShopRequestResponse" />
   <wsdl:fault name="ShopException" message="impl:ShopException" />
  </wsdl:operation>
  </wsdl:portType>
- <wsdl:binding name="AdminSoapBinding" type="impl:AdminService">
   <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="getShops">
   <wsdlsoap:operation />
- <wsdl:input>
   <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:AS" />
  </wsdl:input>
- <wsdl:output>
   <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:AS" />
  </wsdl:output>
- <wsdl:fault name="ShopException">
   <wsdlsoap:fault name="shopErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:AS"
     />
  </wsdl:fault>
  </wsdl:operation>
- <wsdl:operation name="manageShopRequest">
```

```
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:AS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:AS" />
    </wsdl:output>
- <wsdl:fault name="ShopException">
  <wsdlsoap:fault name="shopErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:AS"
     />
    </wsdl:fault>
    </wsdl:operation>
    </wsdl:binding>
- <wsdl:service name="AdminServiceService">
- <wsdl:port name="Admin" binding="impl:AdminSoapBinding">
  <wsdlsoap:address location="http://localhost:8080/axis/services/Admin" />
    </wsdl:port>
    </wsdl:service>
    </wsdl:definitions>
```

2. Customer Web service WSDL file

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions xmlns:impl="urn:Services:CS" xmlns:intf="urn:Services:CS"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns2="urn:Services:data"
    xmlns:tns3="urn:Services:common" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="urn:Services:CS">
- <wsdl:types>
- <xsd:schema>
  <xsd:import namespace="urn:Services:data" schemaLocation="dataType.xsd" />
  <xsd:import namespace="urn:Services:common" schemaLocation="common.xsd" />
    </xsd:schema>
- <schema xmlns="http://www.w3.org/2001/XMLSchema" targetNamespace="urn:Services:CS">
  <import namespace="http://schemas.xmlsoap.org/soap/encoding/" />
- <complexType name="ArrayOf_tns2_itemType">
- <complexContent>
- <restriction base="soapenc:Array">
  <attribute ref="soapenc:arrayType" wsdl:arrayType="tns2:itemType[]" />
    </restriction>
    </complexContent>
    </complexType>
- <complexType name="ArrayOf_xsd_string">
- <complexContent>
- <restriction base="soapenc:Array">
  <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:string[]" />
    </restriction>
    </complexContent>
    </complexType>
- <complexType name="ArrayOf_xsd_int">
- <complexContent>
- <restriction base="soapenc:Array">
  <attribute ref="soapenc:arrayType" wsdl:arrayType="xsd:int[]" />
    </restriction>
    </complexContent>
    </complexType>
- <complexType name="ArrayOf_tns2_shopType">
- <complexContent>
- <restriction base="soapenc:Array">
  <attribute ref="soapenc:arrayType" wsdl:arrayType="tns2:shopType[]" />
    </restriction>
    </complexContent>
    </complexType>
```

```
– <complexType name="ArrayOf_tns2_wishType">
– <complexContent>
– <restriction base="soapenc:Array">
  <attribute ref="soapenc:arrayType" wsdl:arrayType="tns2:wishType[]" />
    </restriction>
    </complexContent>
    </complexType>
    </schema>
    </wsdl:types>
– <wsdl:message name="addToCartRequest">
  <wsdl:part name="userID" type="xsd:string" />
  <wsdl:part name="itemID" type="xsd:int" />
  <wsdl:part name="qty" type="xsd:int" />
    </wsdl:message>
  <wsdl:message name="addToCartResponse" />
– <wsdl:message name="removeFromCartRequest">
  <wsdl:part name="userID" type="xsd:string" />
  <wsdl:part name="itemID" type="xsd:int" />
    </wsdl:message>
  <wsdl:message name="removeFromCartResponse" />
– <wsdl:message name="updateCartRequest">
  <wsdl:part name="userID" type="xsd:string" />
  <wsdl:part name="newItemIDs" type="impl:ArrayOf_xsd_int" />
  <wsdl:part name="newQties" type="impl:ArrayOf_xsd_int" />
    </wsdl:message>
  <wsdl:message name="updateCartResponse" />
– <wsdl:message name="getCartRequest">
  <wsdl:part name="userID" type="xsd:string" />
    </wsdl:message>
– <wsdl:message name="getCartResponse">
  <wsdl:part name="getCartReturn" type="impl:ArrayOf_tns2_itemType" />
    </wsdl:message>
– <wsdl:message name="checkoutRequest">
  <wsdl:part name="userID" type="xsd:string" />
    </wsdl:message>
  <wsdl:message name="checkoutResponse" />
– <wsdl:message name="getProfileRequest">
  <wsdl:part name="userID" type="xsd:string" />
    </wsdl:message>
– <wsdl:message name="getProfileResponse">
  <wsdl:part name="getProfileReturn" type="tns2:userType" />
    </wsdl:message>
– <wsdl:message name="updateProfileRequest">
  <wsdl:part name="profile" type="tns2:userType" />
```

```
   </wsdl:message>
  <wsdl:message name="updateProfileResponse" />
– <wsdl:message name="registerRequest">
  <wsdl:part name="profile" type="tns2:userType" />
   </wsdl:message>
  <wsdl:message name="registerResponse" />
– <wsdl:message name="shopCreatRequestRequest">
  <wsdl:part name="shop" type="tns2:shopType" />
   </wsdl:message>
  <wsdl:message name="shopCreatRequestResponse" />
– <wsdl:message name="searchItemRequest">
  <wsdl:part name="keyword" type="xsd:string" />
   </wsdl:message>
– <wsdl:message name="searchItemResponse">
  <wsdl:part name="searchItemReturn" type="impl:ArrayOf_tns2_itemType" />
   </wsdl:message>
– <wsdl:message name="getItemRequest">
  <wsdl:part name="itemID" type="xsd:int" />
   </wsdl:message>
– <wsdl:message name="getItemResponse">
  <wsdl:part name="getItemReturn" type="tns2:itemType" />
   </wsdl:message>
– <wsdl:message name="getItemsinShopRequest">
  <wsdl:part name="shopID" type="xsd:int" />
   </wsdl:message>
– <wsdl:message name="getItemsinShopResponse">
  <wsdl:part name="getItemsinShopReturn" type="impl:ArrayOf_tns2_itemType" />
   </wsdl:message>
– <wsdl:message name="getShopRequest">
  <wsdl:part name="shopID" type="xsd:int" />
   </wsdl:message>
– <wsdl:message name="getShopResponse">
  <wsdl:part name="getShopReturn" type="tns2:shopType" />
   </wsdl:message>
– <wsdl:message name="getShopsinCategoryRequest">
  <wsdl:part name="category" type="xsd:string" />
   </wsdl:message>
– <wsdl:message name="getShopsinCategoryResponse">
  <wsdl:part name="getShopsinCategoryReturn" type="impl:ArrayOf_tns2_shopType" />
   </wsdl:message>
– <wsdl:message name="addWishRequest">
  <wsdl:part name="userID" type="xsd:string" />
  <wsdl:part name="itemName" type="xsd:string" />
  <wsdl:part name="category" type="xsd:string" />
```

```
    </wsdl:message>
  <wsdl:message name="addWishResponse" />
- <wsdl:message name="removeWishRequest">
  <wsdl:part name="wishID" type="xsd:int" />
    </wsdl:message>
  <wsdl:message name="removeWishResponse" />
- <wsdl:message name="getwishesRequest">
  <wsdl:part name="userID" type="xsd:string" />
    </wsdl:message>
- <wsdl:message name="getwishesResponse">
  <wsdl:part name="getwishesReturn" type="impl:ArrayOf_tns2_wishType" />
    </wsdl:message>
- <wsdl:message name="UserException">
  <wsdl:part name="userErr" type="tns3:UserException" />
    </wsdl:message>
- <wsdl:message name="ItemException">
  <wsdl:part name="itemErr" type="tns3:ItemException" />
    </wsdl:message>
- <wsdl:message name="WishException">
  <wsdl:part name="wishErr" type="tns3:WishException" />
    </wsdl:message>
- <wsdl:message name="ShopException">
  <wsdl:part name="shopErr" type="tns3:ShopException" />
    </wsdl:message>
- <wsdl:message name="CartException">
  <wsdl:part name="cartErr" type="tns3:CartException" />
    </wsdl:message>
- <wsdl:portType name="CustomerService">
- <wsdl:operation name="register" parameterOrder="profile">
  <wsdl:input name="registerRequest" message="impl:registerRequest" />
  <wsdl:output name="registerResponse" message="impl:registerResponse" />
  <wsdl:fault name="UserException" message="impl:UserException" />
    </wsdl:operation>
- <wsdl:operation name="addToCart" parameterOrder="userID itemID qty">
  <wsdl:input name="addToCartRequest" message="impl:addToCartRequest" />
  <wsdl:output name="addToCartResponse" message="impl:addToCartResponse" />
  <wsdl:fault name="CartException" message="impl:CartException" />
    </wsdl:operation>
- <wsdl:operation name="removeFromCart" parameterOrder="userID itemID">
  <wsdl:input name="removeFromCartRequest" message="impl:removeFromCartRequest" />
  <wsdl:output name="removeFromCartResponse" message="impl:removeFromCartResponse" />
  <wsdl:fault name="CartException" message="impl:CartException" />
    </wsdl:operation>
- <wsdl:operation name="getCart" parameterOrder="userID">
```

```
<wsdl:input name="getCartRequest" message="impl:getCartRequest" />
<wsdl:output name="getCartResponse" message="impl:getCartResponse" />
  </wsdl:operation>
- <wsdl:operation name="updateCart" parameterOrder="userID newItemIDs newQties">
<wsdl:input name="updateCartRequest" message="impl:updateCartRequest" />
<wsdl:output name="updateCartResponse" message="impl:updateCartResponse" />
<wsdl:fault name="CartException" message="impl:CartException" />
  </wsdl:operation>
- <wsdl:operation name="checkout" parameterOrder="userID">
<wsdl:input name="checkoutRequest" message="impl:checkoutRequest" />
<wsdl:output name="checkoutResponse" message="impl:checkoutResponse" />
<wsdl:fault name="CartException" message="impl:CartException" />
  </wsdl:operation>
- <wsdl:operation name="getProfile" parameterOrder="userID">
<wsdl:input name="getProfileRequest" message="impl:getProfileRequest" />
<wsdl:output name="getProfileResponse" message="impl:getProfileResponse" />
<wsdl:fault name="UserException" message="impl:UserException" />
  </wsdl:operation>
- <wsdl:operation name="updateProfile" parameterOrder="profile">
<wsdl:input name="updateProfileRequest" message="impl:updateProfileRequest" />
<wsdl:output name="updateProfileResponse" message="impl:updateProfileResponse" />
<wsdl:fault name="UserException" message="impl:UserException" />
  </wsdl:operation>
- <wsdl:operation name="shopCreatRequest" parameterOrder="shop">
<wsdl:input name="shopCreatRequestRequest" message="impl:shopCreatRequestRequest" />
<wsdl:output name="shopCreatRequestResponse" message="impl:shopCreatRequestResponse" />
<wsdl:fault name="ShopException" message="impl:ShopException" />
  </wsdl:operation>
- <wsdl:operation name="searchItem" parameterOrder="keyword">
<wsdl:input name="searchItemRequest" message="impl:searchItemRequest" />
<wsdl:output name="searchItemResponse" message="impl:searchItemResponse" />
<wsdl:fault name="ItemException" message="impl:ItemException" />
  </wsdl:operation>
- <wsdl:operation name="getItemsinShop" parameterOrder="shopID">
<wsdl:input name="getItemsinShopRequest" message="impl:getItemsinShopRequest" />
<wsdl:output name="getItemsinShopResponse" message="impl:getItemsinShopResponse" />
<wsdl:fault name="ItemException" message="impl:ItemException" />
  </wsdl:operation>
- <wsdl:operation name="getItem" parameterOrder="itemID">
<wsdl:input name="getItemRequest" message="impl:getItemRequest" />
<wsdl:output name="getItemResponse" message="impl:getItemResponse" />
<wsdl:fault name="ItemException" message="impl:ItemException" />
  </wsdl:operation>
- <wsdl:operation name="getShop" parameterOrder="shopID">
```

```
  <wsdl:input name="getShopRequest" message="impl:getShopRequest" />
  <wsdl:output name="getShopResponse" message="impl:getShopResponse" />
  <wsdl:fault name="ShopException" message="impl:ShopException" />
    </wsdl:operation>
- <wsdl:operation name="getShopsinCategory" parameterOrder="category">
  <wsdl:input name="getShopsinCategoryRequest" message="impl:getShopsinCategoryRequest" />
  <wsdl:output name="getShopsinCategoryResponse"
     message="impl:getShopsinCategoryResponse" />
  <wsdl:fault name="ShopException" message="impl:ShopException" />
    </wsdl:operation>
- <wsdl:operation name="addWish" parameterOrder="userID itemName category">
  <wsdl:input name="addWishRequest" message="impl:addWishRequest" />
  <wsdl:output name="addWishResponse" message="impl:addWishResponse" />
    </wsdl:operation>
- <wsdl:operation name="removeWish" parameterOrder="wishID">
  <wsdl:input name="removeWishRequest" message="impl:removeWishRequest" />
  <wsdl:output name="removeWishResponse" message="impl:removeWishResponse" />
  <wsdl:fault name="WishException" message="impl:WishException" />
    </wsdl:operation>
- <wsdl:operation name="getwishes" parameterOrder="userID">
  <wsdl:input name="getwishesRequest" message="impl:getwishesRequest" />
  <wsdl:output name="getwishesResponse" message="impl:getwishesResponse" />
    </wsdl:operation>
    </wsdl:portType>
- <wsdl:binding name="CustomerSoapBinding" type="impl:CustomerService">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="register">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
    </wsdl:output>
- <wsdl:fault name="UserException">
  <wsdlsoap:fault name="userErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
     />
    </wsdl:fault>
    </wsdl:operation>
- <wsdl:operation name="addToCart">
  <wsdlsoap:operation />
```

```
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:output>
- <wsdl:fault name="CartException">
  <wsdlsoap:fault name="cartErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
     />
  </wsdl:fault>
  </wsdl:operation>
- <wsdl:operation name="removeFromCart">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:output>
- <wsdl:fault name="CartException">
  <wsdlsoap:fault name="cartErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
     />
  </wsdl:fault>
  </wsdl:operation>
- <wsdl:operation name="getCart">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:output>
  </wsdl:operation>
- <wsdl:operation name="updateCart">
  <wsdlsoap:operation />
- <wsdl:input>
```

```
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:output>
- <wsdl:fault name="CartException">
  <wsdlsoap:fault name="cartErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
     />
  </wsdl:fault>
  </wsdl:operation>
- <wsdl:operation name="checkout">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:output>
- <wsdl:fault name="CartException">
  <wsdlsoap:fault name="cartErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
     />
  </wsdl:fault>
  </wsdl:operation>
- <wsdl:operation name="getProfile">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:output>
- <wsdl:fault name="UserException">
  <wsdlsoap:fault name="userErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
     />
  </wsdl:fault>
```

```
</wsdl:operation>
- <wsdl:operation name="updateProfile">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:output>
- <wsdl:fault name="UserException">
  <wsdlsoap:fault name="userErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
     />
  </wsdl:fault>
  </wsdl:operation>
- <wsdl:operation name="shopCreatRequest">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:output>
- <wsdl:fault name="ShopException">
  <wsdlsoap:fault name="shopErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
     />
  </wsdl:fault>
  </wsdl:operation>
- <wsdl:operation name="searchItem">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:CS" />
  </wsdl:output>
- <wsdl:fault name="ItemException">
```

```
  <wsdlsoap:fault name="itemErr" use="encoded"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
      />
    </wsdl:fault>
    </wsdl:operation>
- <wsdl:operation name="getItemsinShop">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:Services:CS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:Services:CS" />
    </wsdl:output>
- <wsdl:fault name="ItemException">
  <wsdlsoap:fault name="itemErr" use="encoded"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
      />
    </wsdl:fault>
    </wsdl:operation>
- <wsdl:operation name="getItem">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:Services:CS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:Services:CS" />
    </wsdl:output>
- <wsdl:fault name="ItemException">
  <wsdlsoap:fault name="itemErr" use="encoded"
      encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
      />
    </wsdl:fault>
    </wsdl:operation>
- <wsdl:operation name="getShop">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
      namespace="urn:Services:CS" />
    </wsdl:input>
- <wsdl:output>
```

```
<wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:CS" />
   </wsdl:output>
- <wsdl:fault name="ShopException">
  <wsdlsoap:fault name="shopErr" use="encoded"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
    />
   </wsdl:fault>
   </wsdl:operation>
- <wsdl:operation name="getShopsinCategory">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:CS" />
   </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:CS" />
   </wsdl:output>
- <wsdl:fault name="ShopException">
  <wsdlsoap:fault name="shopErr" use="encoded"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
    />
   </wsdl:fault>
   </wsdl:operation>
- <wsdl:operation name="addWish">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:CS" />
   </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:CS" />
   </wsdl:output>
   </wsdl:operation>
- <wsdl:operation name="removeWish">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:CS" />
   </wsdl:input>
- <wsdl:output>
```

```
<wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:CS" />
  </wsdl:output>
- <wsdl:fault name="WishException">
  <wsdlsoap:fault name="wishErr" use="encoded"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:CS"
    />
  </wsdl:fault>
  </wsdl:operation>
- <wsdl:operation name="getwishes">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:CS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:CS" />
  </wsdl:output>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:service name="CustomerServiceService">
- <wsdl:port name="Customer" binding="impl:CustomerSoapBinding">
  <wsdlsoap:address location="http://localhost:8080/axis/services/Customer" />
  </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

3. Owner Web Service WSDL file

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions xmlns:impl="urn:Services:OS" xmlns:intf="urn:Services:OS"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:tns2="urn:Services:data"
    xmlns:tns3="urn:Services:common" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="urn:Services:OS">
- <wsdl:types>
- <xsd:schema>
  <xsd:import namespace="urn:Services:data" schemaLocation="dataType.xsd" />
  <xsd:import namespace="urn:Services:common" schemaLocation="common.xsd" />
  </xsd:schema>
- <xsd:schema targetNamespace="urn:Services:OS">
- <xsd:complexType name="ArrayOf_itemType">
- <xsd:complexContent>
- <xsd:restriction base="soapenc:Array">
  <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="tns2:itemType[]" />
  </xsd:restriction>
  </xsd:complexContent>
  </xsd:complexType>
- <xsd:complexType name="ArrayOf_orderType">
- <xsd:complexContent>
- <xsd:restriction base="soapenc:Array">
  <xsd:attribute ref="soapenc:arrayType" wsdl:arrayType="tns2:orderType[]" />
  </xsd:restriction>
  </xsd:complexContent>
  </xsd:complexType>
  </xsd:schema>
  </wsdl:types>
- <wsdl:message name="setShopForUserRequest">
  <wsdl:part name="userID" type="xsd:string" />
  </wsdl:message>
- <wsdl:message name="setShopForUserResponse">
  <wsdl:part name="shopID" type="xsd:int" />
  </wsdl:message>
- <wsdl:message name="getShopDetailRequest">
  <wsdl:part name="shopID" type="xsd:int" />
  </wsdl:message>
- <wsdl:message name="getShopDetailResponse">
  <wsdl:part name="getShopDetailReturn" type="tns2:shopType" />
  </wsdl:message>
- <wsdl:message name="updateShopDetailRequest">
```

```
<wsdl:part name="shop" type="tns2:shopType" />
  </wsdl:message>
<wsdl:message name="updateShopDetailResponse" />
- <wsdl:message name="disShopRequestRequest">
<wsdl:part name="shopID" type="xsd:int" />
  </wsdl:message>
<wsdl:message name="disShopRequestResponse" />
- <wsdl:message name="addItemRequest">
<wsdl:part name="item" type="tns2:itemType" />
<wsdl:part name="shopID" type="xsd:int" />
  </wsdl:message>
<wsdl:message name="addItemResponse" />
- <wsdl:message name="updateItemRequest">
<wsdl:part name="item" type="tns2:itemType" />
  </wsdl:message>
<wsdl:message name="updateItemResponse" />
- <wsdl:message name="deleteItemRequest">
<wsdl:part name="itemID" type="xsd:int" />
<wsdl:part name="shopID" type="xsd:int" />
  </wsdl:message>
<wsdl:message name="deleteItemResponse" />
- <wsdl:message name="getItemDetailRequest">
<wsdl:part name="itemID" type="xsd:int" />
  </wsdl:message>
- <wsdl:message name="getItemDetailResponse">
<wsdl:part name="getItemDetailReturn" type="tns2:itemType" />
  </wsdl:message>
- <wsdl:message name="getItemsRequest">
<wsdl:part name="keyword" type="xsd:string" />
<wsdl:part name="shopID" type="xsd:int" />
  </wsdl:message>
- <wsdl:message name="getItemsResponse">
<wsdl:part name="getItemsReturn" type="impl:ArrayOf_itemType" />
  </wsdl:message>
- <wsdl:message name="getPendingOrdersRequest">
<wsdl:part name="shopID" type="xsd:int" />
  </wsdl:message>
- <wsdl:message name="getPendingOrdersResponse">
<wsdl:part name="getPendingOrdersReturn" type="impl:ArrayOf_orderType" />
  </wsdl:message>
- <wsdl:message name="shipOrderRequest">
<wsdl:part name="orderID" type="xsd:string" />
  </wsdl:message>
<wsdl:message name="shipOrderResponse" />
```

```
- <wsdl:message name="ShopException">
  <wsdl:part name="shopErr" type="tns3:ShopException" />
  </wsdl:message>
- <wsdl:message name="ItemException">
  <wsdl:part name="itemErr" type="tns3:ItemException" />
  </wsdl:message>
- <wsdl:message name="OrderException">
  <wsdl:part name="orderErr" type="tns3:OrderException" />
  </wsdl:message>
- <wsdl:portType name="OwnerService">
- <wsdl:operation name="setShopForUser" parameterOrder="userID">
  <wsdl:input name="setShopForUserRequest" message="impl:setShopForUserRequest" />
  <wsdl:output name="setShopForUserResponse" message="impl:setShopForUserResponse" />
  <wsdl:fault name="ShopException" message="impl:ShopException" />
  </wsdl:operation>
- <wsdl:operation name="getShopDetail" parameterOrder="shopID">
  <wsdl:input name="getShopDetailRequest" message="impl:getShopDetailRequest" />
  <wsdl:output name="getShopDetailResponse" message="impl:getShopDetailResponse" />
  <wsdl:fault name="ShopException" message="impl:ShopException" />
  </wsdl:operation>
- <wsdl:operation name="updateShopDetail" parameterOrder="shop">
  <wsdl:input name="updateShopDetailRequest" message="impl:updateShopDetailRequest" />
  <wsdl:output name="updateShopDetailResponse" message="impl:updateShopDetailResponse" />
  <wsdl:fault name="ShopException" message="impl:ShopException" />
  </wsdl:operation>
- <wsdl:operation name="disShopRequest" parameterOrder="shopID">
  <wsdl:input name="disShopRequestRequest" message="impl:disShopRequestRequest" />
  <wsdl:output name="disShopRequestResponse" message="impl:disShopRequestResponse" />
  <wsdl:fault name="ShopException" message="impl:ShopException" />
  </wsdl:operation>
- <wsdl:operation name="addItem" parameterOrder="item shopID">
  <wsdl:input name="addItemRequest" message="impl:addItemRequest" />
  <wsdl:output name="addItemResponse" message="impl:addItemResponse" />
  <wsdl:fault name="ItemException" message="impl:ItemException" />
  </wsdl:operation>
- <wsdl:operation name="updateItem" parameterOrder="item">
  <wsdl:input name="updateItemRequest" message="impl:updateItemRequest" />
  <wsdl:output name="updateItemResponse" message="impl:updateItemResponse" />
  <wsdl:fault name="ItemException" message="impl:ItemException" />
  </wsdl:operation>
- <wsdl:operation name="deleteItem" parameterOrder="itemID shopID">
  <wsdl:input name="deleteItemRequest" message="impl:deleteItemRequest" />
  <wsdl:output name="deleteItemResponse" message="impl:deleteItemResponse" />
  <wsdl:fault name="ItemException" message="impl:ItemException" />
```

```
  </wsdl:operation>
– <wsdl:operation name="getItemDetail" parameterOrder="itemID">
  <wsdl:input name="getItemDetailRequest" message="impl:getItemDetailRequest" />
  <wsdl:output name="getItemDetailResponse" message="impl:getItemDetailResponse" />
  <wsdl:fault name="ItemException" message="impl:ItemException" />
  </wsdl:operation>
– <wsdl:operation name="getItems" parameterOrder="keyword shopID">
  <wsdl:input name="getItemsRequest" message="impl:getItemsRequest" />
  <wsdl:output name="getItemsResponse" message="impl:getItemsResponse" />
  <wsdl:fault name="ItemException" message="impl:ItemException" />
  </wsdl:operation>
– <wsdl:operation name="getPendingOrders" parameterOrder="shopID">
  <wsdl:input name="getPendingOrdersRequest" message="impl:getPendingOrdersRequest" />
  <wsdl:output name="getPendingOrdersResponse" message="impl:getPendingOrdersResponse" />
  <wsdl:fault name="OrderException" message="impl:OrderException" />
  </wsdl:operation>
– <wsdl:operation name="shipOrder" parameterOrder="orderID">
  <wsdl:input name="shipOrderRequest" message="impl:shipOrderRequest" />
  <wsdl:output name="shipOrderResponse" message="impl:shipOrderResponse" />
  <wsdl:fault name="OrderException" message="impl:OrderException" />
  </wsdl:operation>
  </wsdl:portType>
– <wsdl:binding name="OwnerSoapBinding" type="impl:OwnerService">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
– <wsdl:operation name="setShopForUser">
  <wsdlsoap:operation />
– <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:OS" />
  </wsdl:input>
– <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
  </wsdl:output>
– <wsdl:fault name="ShopException">
  <wsdlsoap:fault name="shopErr" use="encoded"
    encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
    />
  </wsdl:fault>
  </wsdl:operation>
– <wsdl:operation name="getShopDetail">
  <wsdlsoap:operation />
– <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
  </wsdl:input>
```

```
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:output>
- <wsdl:fault name="ShopException">
  <wsdlsoap:fault name="shopErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
     />
    </wsdl:fault>
    </wsdl:operation>
- <wsdl:operation name="updateShopDetail">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
    </wsdl:output>
- <wsdl:fault name="ShopException">
  <wsdlsoap:fault name="shopErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
     />
    </wsdl:fault>
    </wsdl:operation>
- <wsdl:operation name="addItem">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:output>
- <wsdl:fault name="ItemException">
  <wsdlsoap:fault name="itemErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
     />
    </wsdl:fault>
    </wsdl:operation>
- <wsdl:operation name="updateItem">
  <wsdlsoap:operation />
- <wsdl:input>
```

```
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
      </wsdl:input>
  - <wsdl:output>
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       namespace="urn:Services:OS" />
      </wsdl:output>
  - <wsdl:fault name="ItemException">
    <wsdlsoap:fault name="itemErr" use="encoded"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
       />
      </wsdl:fault>
      </wsdl:operation>
  - <wsdl:operation name="deleteItem">
    <wsdlsoap:operation />
  - <wsdl:input>
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       namespace="urn:Services:OS" />
      </wsdl:input>
  - <wsdl:output>
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       namespace="urn:Services:OS" />
      </wsdl:output>
  - <wsdl:fault name="ItemException">
    <wsdlsoap:fault name="itemErr" use="encoded"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
       />
      </wsdl:fault>
      </wsdl:operation>
  - <wsdl:operation name="getItemDetail">
    <wsdlsoap:operation />
  - <wsdl:input>
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       namespace="urn:Services:OS" />
      </wsdl:input>
  - <wsdl:output>
    <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
       namespace="urn:Services:OS" />
      </wsdl:output>
  - <wsdl:fault name="ItemException">
    <wsdlsoap:fault name="itemErr" use="encoded"
       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
       />
      </wsdl:fault>
      </wsdl:operation>
```

```
- <wsdl:operation name="getItems">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:output>
- <wsdl:fault name="ItemException">
  <wsdlsoap:fault name="itemErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
     />
    </wsdl:fault>
    </wsdl:operation>
- <wsdl:operation name="disShopRequest">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:output>
- <wsdl:fault name="ShopException">
  <wsdlsoap:fault name="shopErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
     />
    </wsdl:fault>
    </wsdl:operation>
- <wsdl:operation name="getPendingOrders">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:output>
- <wsdl:fault name="OrderException">
```

```
  <wsdlsoap:fault name="orderErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
     />
    </wsdl:fault>
    </wsdl:operation>
- <wsdl:operation name="shipOrder">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
     namespace="urn:Services:OS" />
    </wsdl:output>
- <wsdl:fault name="OrderException">
  <wsdlsoap:fault name="orderErr" use="encoded"
     encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:OS"
     />
    </wsdl:fault>
    </wsdl:operation>
    </wsdl:binding>
- <wsdl:service name="OwnerServiceService">
- <wsdl:port name="Owner" binding="impl:OwnerSoapBinding">
  <wsdlsoap:address location="http://localhost:8080/axis/services/Owner" />
    </wsdl:port>
    </wsdl:service>
    </wsdl:definitions>
```

4. Login Web Service WSDL file

```
<?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions xmlns:impl="urn:Services:LS" xmlns:intf="urn:Services:LS"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="urn:Services:LS">
- <wsdl:message name="loginRequest">
  <wsdl:part name="userID" type="xsd:string" />
  <wsdl:part name="pwd" type="xsd:string" />
    </wsdl:message>
- <wsdl:message name="loginResponse">
  <wsdl:part name="loginReturn" type="xsd:string" />
    </wsdl:message>
- <wsdl:portType name="LoginService">
- <wsdl:operation name="login" parameterOrder="userID pwd">
  <wsdl:input name="loginRequest" message="impl:loginRequest" />
  <wsdl:output name="loginResponse" message="impl:loginResponse" />
    </wsdl:operation>
    </wsdl:portType>
- <wsdl:binding name="LoginSoapBinding" type="impl:LoginService">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="login">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:LS" />
    </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:LS" />
    </wsdl:output>
    </wsdl:operation>
    </wsdl:binding>
- <wsdl:service name="LoginServiceService">
- <wsdl:port name="Login" binding="impl:LoginSoapBinding">
  <wsdlsoap:address location="http://localhost:8080/axis/services/Login" />
    </wsdl:port>
    </wsdl:service>
    </wsdl:definitions>
```

5. Mail Web Service WSDL File

```
  <?xml version="1.0" encoding="UTF-8" ?>
- <wsdl:definitions xmlns:impl="urn:Services:MS" xmlns:intf="urn:Services:MS"
    xmlns:apachesoap="http://xml.apache.org/xml-soap"
    xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns="http://schemas.xmlsoap.org/wsdl/" targetNamespace="urn:Services:MS">
- <wsdl:message name="sendMailRequest">
  <wsdl:part name="from" type="xsd:string" />
  <wsdl:part name="to" type="xsd:string" />
  <wsdl:part name="cc" type="xsd:string" />
  <wsdl:part name="subject" type="xsd:string" />
  <wsdl:part name="message" type="xsd:string" />
  </wsdl:message>
- <wsdl:message name="sendMailResponse">
  <wsdl:part name="sendMailReturn" type="xsd:string" />
  </wsdl:message>
- <wsdl:message name="sendFailException">
  <wsdl:part name="mailErr" type="xsd:string" />
  </wsdl:message>
- <wsdl:portType name="mailService">
- <wsdl:operation name="sendMail" parameterOrder="from to cc subject message">
  <wsdl:input name="sendMailRequest" message="impl:sendMailRequest" />
  <wsdl:output name="sendMailResponse" message="impl:sendMailResponse" />
  <wsdl:fault name="sendFailException" message="impl:sendFailException" />
  </wsdl:operation>
  </wsdl:portType>
- <wsdl:binding name="MailSoapBinding" type="impl:mailService">
  <wsdlsoap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="sendMail">
  <wsdlsoap:operation />
- <wsdl:input>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:MS" />
  </wsdl:input>
- <wsdl:output>
  <wsdlsoap:body use="encoded" encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
    namespace="urn:Services:MS" />
  </wsdl:output>
- <wsdl:fault name="sendFailException">
```

```
<wsdlsoap:fault name="mailErr" use="encoded"
  encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" namespace="urn:Services:MS"
  />
  </wsdl:fault>
  </wsdl:operation>
  </wsdl:binding>
- <wsdl:service name="mailServiceService">
- <wsdl:port name="Mail" binding="impl:MailSoapBinding">
  <wsdlsoap:address location="http://localhost:8080/axis/services/Mail" />
  </wsdl:port>
  </wsdl:service>
  </wsdl:definitions>
```

6. common XML Schema File

```
<?xml version="1.0" encoding="UTF-8" ?>
- <xs:schema xmlns="urn:Services:common" xmlns:xs="http://www.w3.org/2001/XMLSchema"
   targetNamespace="urn:Services:common" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
- <xs:complexType name="ShopException">
- <xs:sequence>
  <xs:element name="shopErr" type="xs:string" nillable="true" />
  </xs:sequence>
  </xs:complexType>
- <xs:complexType name="ItemException">
- <xs:sequence>
  <xs:element name="itemErr" type="xs:string" nillable="true" />
  </xs:sequence>
  </xs:complexType>
- <xs:complexType name="OrderException">
- <xs:sequence>
  <xs:element name="orderErr" type="xs:string" nillable="true" />
  </xs:sequence>
  </xs:complexType>
- <xs:complexType name="CartException">
- <xs:sequence>
  <xs:element name="cartErr" type="xs:string" nillable="true" />
  </xs:sequence>
  </xs:complexType>
- <xs:complexType name="UserException">
- <xs:sequence>
  <xs:element name="userErr" type="xs:string" nillable="true" />
  </xs:sequence>
  </xs:complexType>
- <xs:complexType name="WishException">
- <xs:sequence>
  <xs:element name="wishErr" type="xs:string" nillable="true" />
  </xs:sequence>
  </xs:complexType>
  </xs:schema>
```

7. dataType XML Schema File

```
<?xml version="1.0" encoding="UTF-8" ?>
- <xs:schema xmlns="urn:Services:data" xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:tns="urn:Services:data"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" targetNamespace="urn:Services:data"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
- <xs:simpleType name="categoryEnum">
- <xs:restriction base="xs:string">
  <xs:enumeration value="Book" />
  <xs:enumeration value="CD" />
  <xs:enumeration value="Toy" />
  <xs:enumeration value="Electronics" />
    </xs:restriction>
    </xs:simpleType>
- <xs:simpleType name="statusEnum">
- <xs:restriction base="xs:string">
  <xs:enumeration value="Pending" />
  <xs:enumeration value="Approved" />
  <xs:enumeration value="Rejected" />
  <xs:enumeration value="Discontinued" />
    </xs:restriction>
    </xs:simpleType>
- <xs:complexType name="shopType">
- <xs:sequence>
  <xs:element name="shopID" type="xs:int" />
  <xs:element name="shopName" type="xs:string" />
  <xs:element name="category" type="categoryEnum" />
  <xs:element name="ownerID" type="xs:string" />
  <xs:element name="regDate" type="xs:date" minOccurs="0" />
  <xs:element name="status" type="statusEnum" />
  <xs:element name="shopDes" type="xs:string" minOccurs="0" />
    </xs:sequence>
    </xs:complexType>
- <xs:complexType name="itemType">
- <xs:sequence>
  <xs:element name="itemID" type="xs:int" />
  <xs:element name="shopID" type="xs:int" />
  <xs:element name="itemName" type="xs:string" />
  <xs:element name="itemQuan" type="xs:int" />
  <xs:element name="itemPrice" type="xs:double" />
  <xs:element name="itemDes" type="xs:string" minOccurs="0" />
    </xs:sequence>
    </xs:complexType>
```

```
- <xs:complexType name="orderType">
- <xs:sequence>
  <xs:element name="orderID" type="xs:string" />
  <xs:element name="userID" type="xs:string" />
  <xs:element name="shopID" type="xs:int" />
  <xs:element name="orderDate" type="xs:date" minOccurs="0" />
- <xs:element name="status">
- <xs:simpleType>
  <xs:restriction base="xs:string" />
    </xs:simpleType>
    </xs:element>
  <xs:element name="orderAddress" type="xs:string" minOccurs="0" />
  <xs:element name="itemDetails" type="itemDetail" maxOccurs="unbounded" />
    </xs:sequence>
    </xs:complexType>
- <xs:complexType name="itemDetail">
- <xs:sequence>
  <xs:element name="itemID" type="xs:int" />
  <xs:element name="quantity" type="xs:int" />
  <xs:element name="price" type="xs:double" />
    </xs:sequence>
    </xs:complexType>
- <xs:complexType name="userType">
- <xs:sequence>
  <xs:element name="userID" type="xs:string" />
  <xs:element name="name" type="xs:string" minOccurs="0" />
- <xs:element name="role">
- <xs:simpleType>
- <xs:restriction base="xs:string">
  <xs:enumeration value="Admin" />
  <xs:enumeration value="Owner" />
  <xs:enumeration value="Customer" />
    </xs:restriction>
    </xs:simpleType>
    </xs:element>
  <xs:element name="password" type="xs:string" />
  <xs:element name="email" type="xs:string" />
  <xs:element name="mobile" type="xs:string" />
  <xs:element name="address" type="xs:string" />
    </xs:sequence>
    </xs:complexType>
- <xs:complexType name="wishType">
- <xs:sequence>
  <xs:element name="wishID" type="xs:int" />
```

```
<xs:element name="userID" type="xs:string" />
<xs:element name="itemName" type="xs:string" />
<xs:element name="itemCategory" type="categoryEnum" />
<xs:element name="createDate" type="xs:date" minOccurs="0" />
- <xs:element name="status">
- <xs:simpleType>
- <xs:restriction base="xs:string">
<xs:enumeration value="Pending" />
<xs:enumeration value="Cancled" />
<xs:enumeration value="Offered" />
  </xs:restriction>
  </xs:simpleType>
  </xs:element>
  </xs:sequence>
  </xs:complexType>
  </xs:schema>
```