

國立臺灣大學管理學院資訊管理研究所

碩士論文

Department of Information Management

College of Management

National Taiwan University

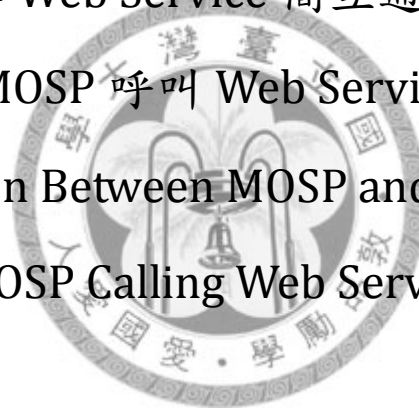
Master Thesis

MOSP 與 Web Service 間互通性實作：

MOSP 呼叫 Web Service

Interoperation Between MOSP and Web Service:

MOSP Calling Web Service



郭家禎

Chia-Chen, Kuo

指導教授：莊裕澤 博士

Advisor: Yuh-Jzer, Joung, Ph.D.

中華民國 97 年 7 月

July, 2008

THESIS ABSTRACT

GRADUATE INSTITUTE OF INFORMATION MANAGEMENT
NATIONAL TAIWAN UNIVERSITY

Student: Chia-Chen, Kuo

Month/Year: July, 2008

Advisor: Yuh-Jzer, Joung

Interoperation between MOSP and Web Service: MOSP Calling Web Service

Nowadays, companies are moving their main operations to web for better automation, efficient business processes and global visibility. We need an integrated, robust solution for leveraging the existing applications, rapidly adapt to the unique needs and continually evolve as requirements change over time. Web Service, with loosely coupled and dynamically bound components, is the present evolution of this new category of services.

Although Web Service is very popular and in general use, which solves many problems, there are still some insufficiency. For example, it is a stateless service system, which does not record the state of each client using it and can only provide services with simpler interaction.

A brand-new solution, MeshObject Service Protocol (MOSP) provides another choice now. MOSP uses the concept of object-oriented, which enables users to obtain an object instance of the service provided by a peer by binding to its MOSP URL. MOSP can provide stateful services with such way. Besides, MOSP contains the concept of inheritance as well, which enables MOSP services to be reused more freely and easily, and therefore reduces the cost and time to develop applications.

In this thesis, we proposed a gateway system which enables MOSP clients to call Web Service in order to promote MOSP and to make Web Service still available in MOSP environment.

Keywords: Web Service, MOSP, MeshObject Service Protocol, Gateway, WSDL, SOAP.

Contents

Chapter 1	Introduction	1
1.1	Background	1
1.2	Motivation.....	10
1.2.1	Challenges	11
1.3	Research Goal.....	14
Chapter 2	Related Work.....	15
2.1	Web Service	15
2.1.1	Extensible Markup Language (XML)	16
2.1.2	Simple Object Access Protocol (SOAP).....	17
2.1.3	Web Services Description Language (WSDL)	19
2.1.4	UDDI (Universal Description, Discovery and Integration).....	20
2.2	Mesh Object Service Protocol (MOSP).....	21
2.2.1	MOSP Service Architecture	21
2.2.2	MOSP Object Model.....	22
2.2.3	MOSP Interface Definition Language (MIDL)	23
2.2.4	MOSP Messages	24
2.3	Interoperation between CORBA and Web Service	27
2.3.1	Common Object Request Broker Architecture (CORBA).....	27
2.3.2	Gateway Systems between Web Service and CORBA	30
2.4	Brief Summary	36
Chapter 3	System Design	38
3.1	System Overview.....	38
3.1.1	Concept of MOSP Service	38
3.1.2	Design of Generic Gateway Service System.....	39
3.1.3	Design of Gateway Service	41
Chapter 4	System Implementation	44
4.1	MOSP Server, Service and Client.....	44
4.2	Gateway Factory Service	46
4.3	Gateway Service: onDesc().....	48
4.4	Gateway Service: onCall()	57
4.5	Performance Test	60
Chapter 5	Conclusion.....	63
5.1	Contribution.....	63
5.2	Future Work	65
5.2.1	Improve limitations	65
5.2.2	Interoperability between MOSP and Web Service	65

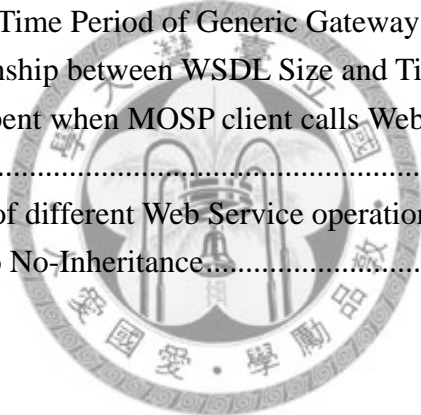
Bibliography68



List Of Figures

Figure 1-1: A Sample of MOSP Environment.....	4
Figure 1-2: MOSP Shopping Cart Service – Object Instance Creation	4
Figure 1-3: Web Service Shopping Cart Service.....	5
Figure 1-4: Web Service Shopping Cart Service – Instance Creation and Deletion ..	5
Figure 1-5: MOSP Shopping Cart Service – Inheritance	6
Figure 1-6: Web Service Shopping Cart Service – Inheritance does not work here ..	6
Figure 1-7: MOSP Shopping Cart Service – Polymorphism.....	7
Figure 1-8: Web Service Shopping Cart Service – Polymorphism does not work here	7
Figure 1-9: Web Service Shopping Cart Service – try another way to Override	8
Figure 1-10: MOSP Shopping Cart Service – Dynamic Binding.....	8
Figure 1-11: Web Service Shopping Cart Service – Dynamic Binding	9
Figure 1-12: XML Schema Data Types [30]	11
Figure 2-1: Web Service Framework [30].....	15
Figure 2-2: Web Service Processing Model	16
Figure 2-3: A sample of an XML document.....	16
Figure 2-4: SOAP Architecture	18
Figure 2-5: The sender transfers SOAP message	18
Figure 2-6: The SOAP Message which Client sends	19
Figure 2-7: The SOAP Message which Server responses	19
Figure 2-8: WSDL document Architecture	20
Figure 2-9: MOSP Service Architecture.....	21
Figure 2-10: The exchange of MOSP request/response messages of service descriptions or service calls between MOSP client and MOSP service.....	22
Figure 2-11: A sample MIDL	23
Figure 2-12: MOSP message framework	24
Figure 2-13: MOSP client request for MIDL	25
Figure 2-14: MOSP service response of the MIDL request	25
Figure 2-15: MOSP client request message for an operation call	26
Figure 2-16: MOSP service response message of the operation call	26
Figure 2-17: Components in the CORBA 2.x Reference Model [2]	28
Figure 2-18: Generic SOAP/HTTP to IIOP Bridge Diagram [20]	30
Figure 2-19: Static Dedicated SOAP/HTTP to SCOAP/ORB Bridge Diagram [20].	31
Figure 2-20: SOAP-CORBA Interoperability Interaction Model [20].....	31
Figure 2-21: XORBA Architecture [23].....	33
Figure 2-22: IDL Fragment of a XORBA Message Example [13]	33

Figure 2-23: A Sample SOAP Request [13]	34
Figure 2-24: CORBA/SOAP Integration Model [7]	35
Figure 3-1: How a MOSP Service works	38
Figure 3-2: Generic Gateway Service System	39
Figure 3-3: Generic Gateway Service System (when <code>createGateway()</code> is called) ..	39
Figure 3-4: Gateway Service Work Model.....	41
Figure 4-1: MOSP Server	44
Figure 4-2: MOSP Service	45
Figure 4-3: MOSP Client.....	45
Figure 4-4: MIDL document of Gateway Factory Service	46
Figure 4-5: A sample of WSDL document	53
Figure 4-6: The description part of a WSDL document.....	54
Figure 4-7: An MIDL document transformed from WSDL document.....	55
Figure 4-8: A sample SOAP request message	57
Figure 4-9: A sample SOAP response message.....	58
Figure 4-10: The sample Time Period of Generic Gateway Service System.....	60
Figure 4-11: The Relationship between WSDL Size and Time Spent in Each Step ..	61
Figure 4-12: The Time spent when MOSP client calls Web Service operation (Step 3)	61
Figure 4-13: Time spent of different Web Service operation call	63
Figure 5-1: Inheritance to No-Inheritance.....	66



List Of Tables

Table 1-1: Comparison between Web Service, JAVA and MOSP	10
Table 1-2: Java Types marshal to MOSP Data Types and unmarshal to Java Types....	12
Table 2-1: Comparison between Web Service and CORBA [10]	29
Table 2-2: Comparison between WSDL and IDL [13]	29
Table 2-3: Comparison between XML Schema and CORBA Object Model [13].....	29
Table 4-1: The algorithm of <code>createGateway()</code>	47
Table 4-2: The WSDL input/output message type to MOSP type	50
Table 4-3: The type transformation algorithm.....	51
Table 4-4: Mappings between XML and MOSP data types – 1	55
Table 4-5: Mappings between XML and MOSP data types – 2	56
Table 4-6: Mappings between XML and Java data type – 1.....	58
Table 4-7: Mappings between XML and Java data type – 2.....	59



Chapter 1 Introduction

1.1 Background

With the intensively progressing of technology, Internet has helped forward the development of all kinds of services. However, the traditional independent single service can no longer fulfill the increasing growth of user needs. Imagine that we have got a long vacation and plan to go traveling abroad. Before departing, we may need some services like traveling information query, flight tickets and hotel room booking, and weather forecast, etc. To get these services, we will usually surf on the websites of the airline companies and hotels. Thus we may need an integrated digital assistant to help us arrange the schedule, and all we have to do is just confirm and pay the bill. So, how does information technology solve this kind of problem?

At the time when networks are more and more universal, the above demands will obviously increase. However, the data are dispersed in different websites at everywhere in the world. To accommodate this tendency and to solve this kind of problems, Web Service [29] has been born at this era when Internet rose and developed and when XML [26] was mature.

Web Service is a software system, which provides a systemized and extensible framework with network communication protocol and an open standard of data format. As the component providing services, Web Service can be used to build distributed systems. Moreover, open standards make Web Service more interoperable, which also gives the systems, which are on different platforms and developed with different language, the ability to integrate and thus solves the problems distributed systems may face when integrating.

Furthermore, in many circumstances, we can easily observe the benefits Web Service brings. First of all, Web Service can communicate across the firewall. Now suppose that we need to provide an application service, whereas there are thousands of uses spread around the world. There are usually firewalls and proxy servers between clients and servers. Besides, application service providers generally are not willing to release the programs to each one of the great amount of users. Eventually, we can only choose to use browser and ASP to reveal the programs to client, which only result in a rise in difficulty of system development and maintenance. Under such condition, we can use Web Service to easily simplify this problem. We can build a

SOAP [27] client with Microsoft SOAP Toolkit [15] or .NET [14] and connect it with the application. In this way, we can not only shorten the development terms and reduce the complexity of codes but also increase the maintainability of the application.

Secondly, Web Service can be used to integrate different applications. One of the challenges which application developers often face is that they usually need to integrate all kinds of applications developed with different languages and implemented on different platforms, which generally takes a large amount of cost. With Web Service, we can use a standardized method to reveal the operations and data of applications for other application to use. Assume that there is an order program responsible for the registration of new orders from clients and another program used to manage the shipping of merchandise. Whenever a new order comes out, the order-registration program has to notify the order-administration program to deliver the cargoes. If these two programs come from different software manufacturers, traditional application merge way must take lots of cost. But now with Web Service, we can simply reveal some operations of order-administration program like “`addOrder ()`” for order-registration program to deliver cargoes.

In addition, Web Service also improves the reuse of software. In the past, the reuse of software was limited to program codes but not the data. It's because that we can readily publish the source code but hard to do the same thing on data, unless they are static and rarely change. In contrast, Web Service allows the reuse of codes and the data behind them. With Web Service, we no longer need to purchase, install and make use of software components in applications. All we have to do is merely invoke the remote Web Service.

For instance, if we want to confirm the address that user inputs in our own applications, we can directly pass this address to relative Web Service to confirm it by looking up the street, city, country and zip code, etc. Web Service provider can charge for this service according to the using time and frequency. It is not practical to achieve this with traditional way, since users consequently have to download and install the database, which do not provide real-time data, containing all of the above address information.

Another case of software reuse is just like the example we give at the beginning of this chapter. Nowadays, there are many application suppliers who provide services such as traveling information query and ticket reservation. Once they expose those functions through Web Service, programmers can easily integrate all of which in a traveling website to provide a united and friendly interface to customers.

Hence, by integrating each kind of Web Service that we need, we can quickly solve problems which originally seem to be difficult to handle. Moreover, Web Service has depicted a new blueprint to the future of the software world, which enables the application to share its API to other applications through the window of Internet.

Nevertheless, even if Web Service is so convenient, it still has some bottlenecks. First of all, Web Service is a “static” and “stateless” service. It means that Web Service is more likely to provide static service, such as the search service Amazon Web Service [1] provides, the key-word-search service Google provides, or the stock-price-enquiry service E*TRADE provides, which only enable users to extract the information but not to use dynamical service, such as editing or saving files, which interacts with users. This is because that Web Service does not have the concept of object instance. It is just like a Java class with many static methods. To provide dynamical service, Web Service has to record the state of each user, which can be achieved by adding a unique user id parameter on all methods. However, it requires the administration of user id controlled by users themselves, which is not suitable for public service. Therefore, if we want to provide a stateful service, Web Service is no longer capable. We need a new protocol to record user states more conveniently and to support dynamical service, which is so-called MOSP (Mesh Object Service Protocol.)

MOSP is a new network service protocol for distributed services. Its main character is that it is developed in object-oriented way with Java [25] language. Similar to Web Service, it allows different services to exchange messages through XML-based document in distributed environment and simple services to integrate into a complex value-added service. The difference between them is that MOSP has the object-oriented concept, and each object instance represents an encapsulation of a state, the object instance each user holds will directly record the user state. The garbage collection of Java will delete the object instance no longer referenced, so we do not have to administer objects. Moreover, MOSP also uses the concept of inheritance, which together with object-oriented increase the convenience of development.

While using a MOSP service, the MOSP user will login from an integrated port user interface. Because of the transparency MOSP Service provides, the user will not notice the differences between the local and the remote side. One single service that a user uses might be integrated from a variety of remote services. Figure 1-1 is an example of MOSP environment presenting a file-saving service, which in fact is integrated from file service and storage service.

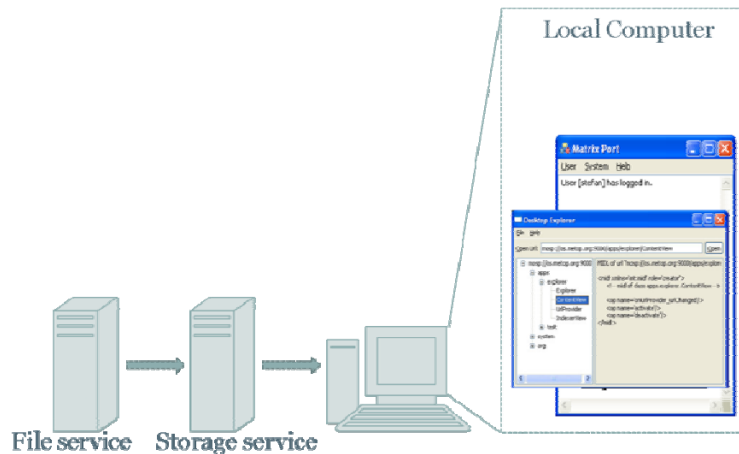


Figure 1-1: A Sample of MOSP Environment

We have mentioned the superiority of MOSP, and the contrast between MOSP and Web Service is listed below.

Take a book store and a shopping-cart service for example. Firstly, Figure 1-2 to Figure 1-4 show the differences between MOSP and Web Service in instance creation. We use Java pseudo code to present it. The left side of the figure is the shopping-cart service, which contains **Product** and **Cart** class. The Cart class provides lots of operations for the user, the Book Store Inc. in right side of the figure, to get information of Cart through operation calls. Figure 1-2 shows that in the MOSP environment, user can create a Cart object instance, add different products into Cart, and get information of Cart through calling different operations of Cart. Cart object can thus record the shopping state of each user.

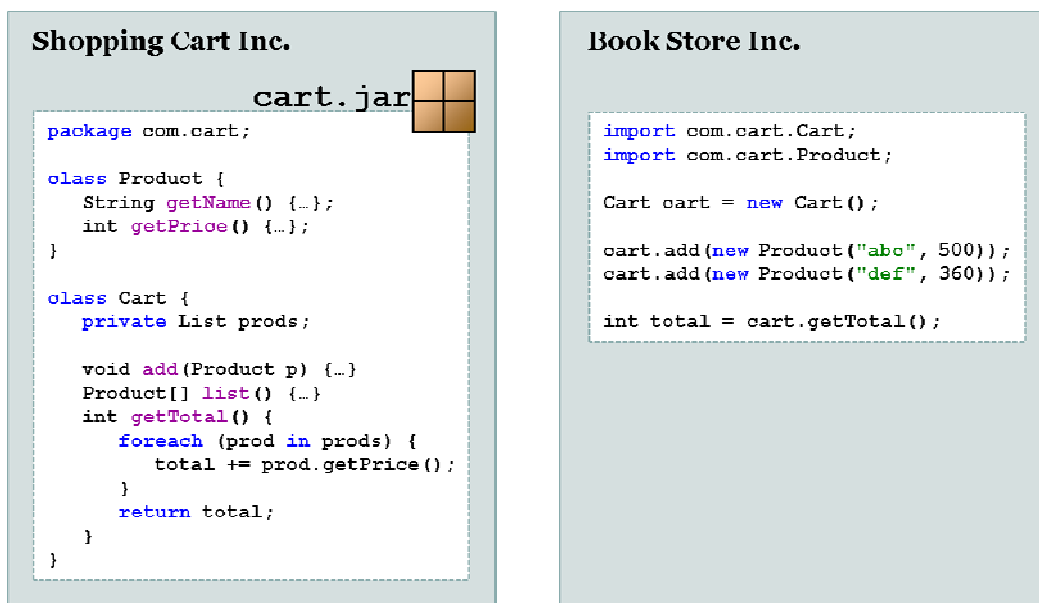


Figure 1-2: MOSP Shopping Cart Service - Object Instance Creation

However, Figure 1-3 shows that in the Web Service environment, we can not create an object instance. Consequently, different users will hold the same shopping cart while they use this service. In order to solve this and let different users manage their own shopping cart, we can use the way shown in Figure 1-4. The server provides a parameter `id` to present different instances and operations `newCart()` and `deleteCart()` to create and delete `id`. Users can get a unique `id` and return it after usage from the code like this: `int id = cart.newCart()` and `cart.deleteCart(id)`. This kind of method requires users to create and delete the instance themselves and thus does not apply to public service. Obviously, we can figure out that MOSP is more suitable for dealing with dynamical and stateful service, enables users not only to use the static search service but also to handle different objects in the service.

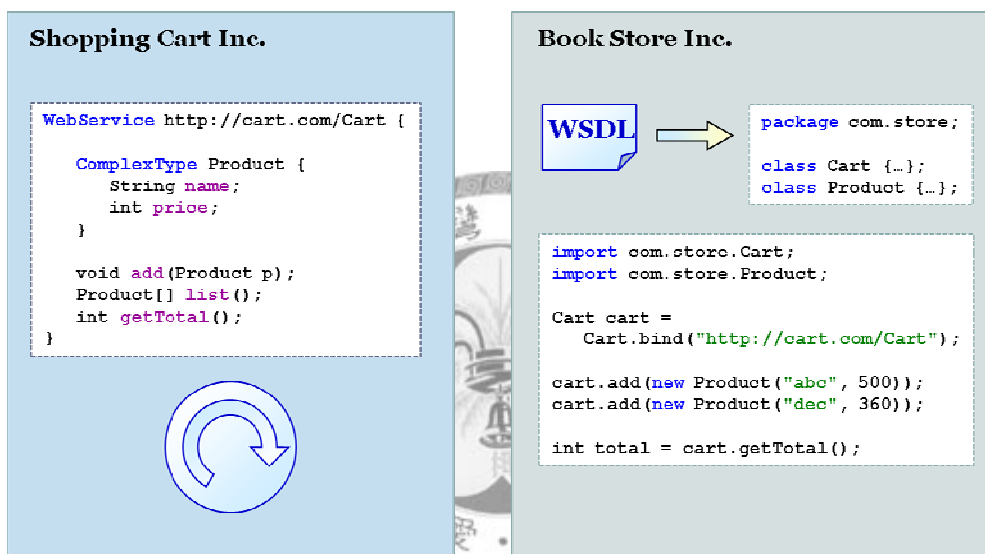


Figure 1-3: Web Service Shopping Cart Service

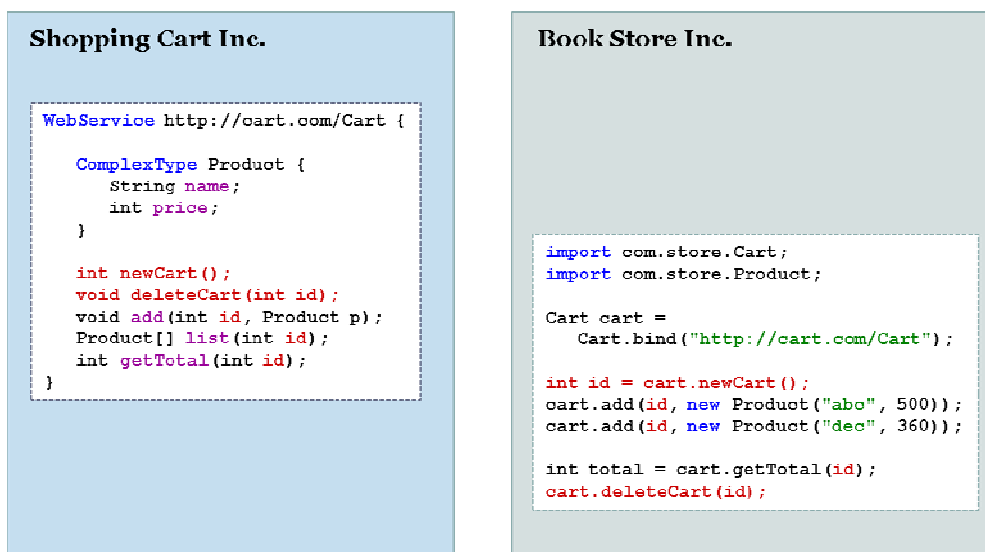


Figure 1-4: Web Service Shopping Cart Service - Instance Creation and Deletion

Secondly, Figure 1-5 and Figure 1-6 shows their difference in inheritance aspect. Figure 1-5 shows that in MOSP environment, we can create a class **Book** which inherits from **Product**, and adds an operation `getISBN()` let users look up the ISBN of the book. So users can directly cast **Product** to **Book** to get its ISBN.

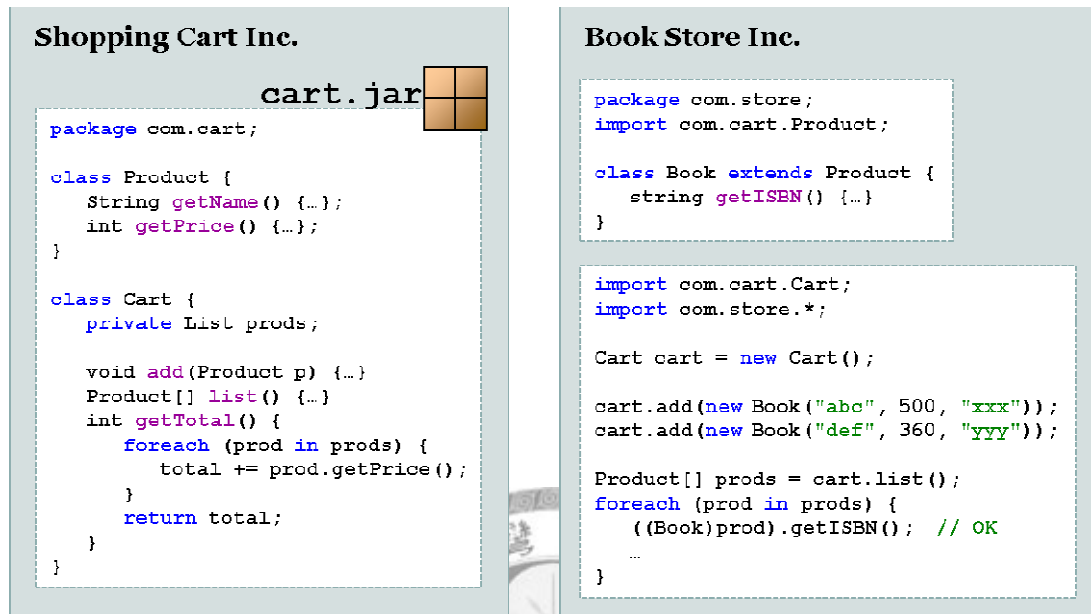


Figure 1-5: MOSP Shopping Cart Service – Inheritance

However, we can not use inheritance under Web Service environment. As Figure 1-6 shows, even if users copy the content of **Product** to **Book** and add a new parameter `ISBN` in **Book**, users can not cast **Product** to **Book** and get its ISBN.

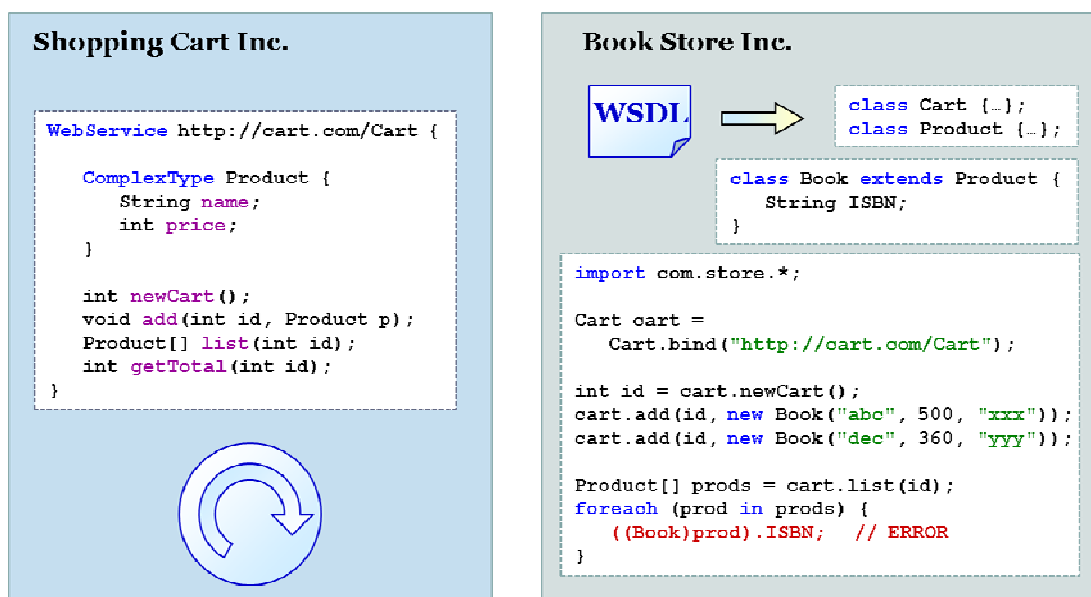


Figure 1-6: Web Service Shopping Cart Service – Inheritance does not work here

Thirdly, Figure 1-7 to Figure 1-9 shows their difference in polymorphism aspect. In MOSP environment, as Figure 1-7 shows, if users want to use dynamic price for each book: make a discount of 30% while the book is stocked for more than two years, they can use the polymorphism concept to override `getPrice()` to achieve this.

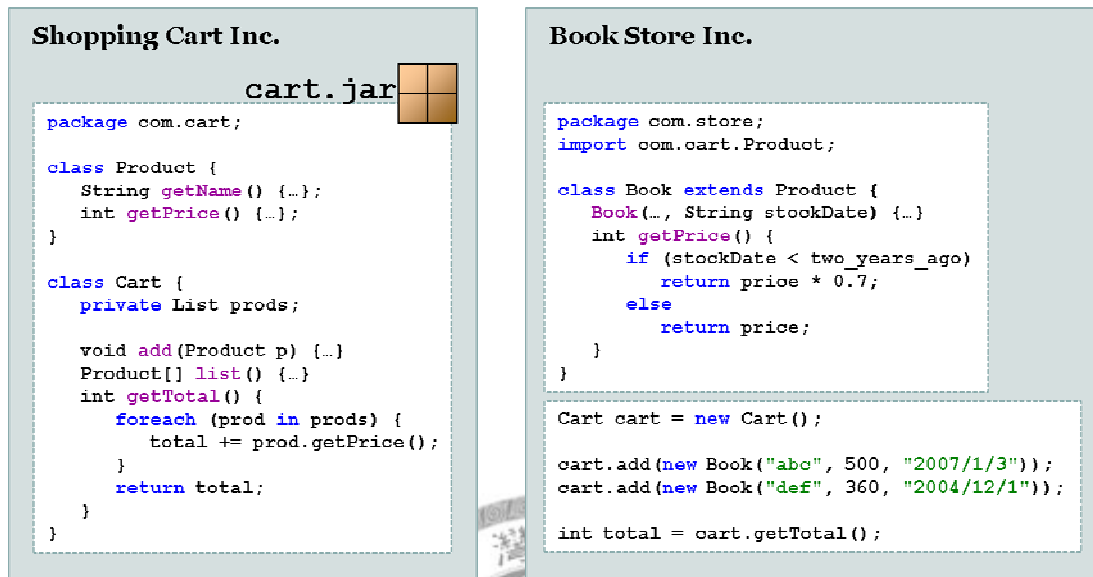


Figure 1-7: MOSP Shopping Cart Service – Polymorphism

However, in Web Service environment, showing in Figure 1-8, users can not override the price argument, since the lack of inheritance. Thus Web Service can not get such information of the new price as well.

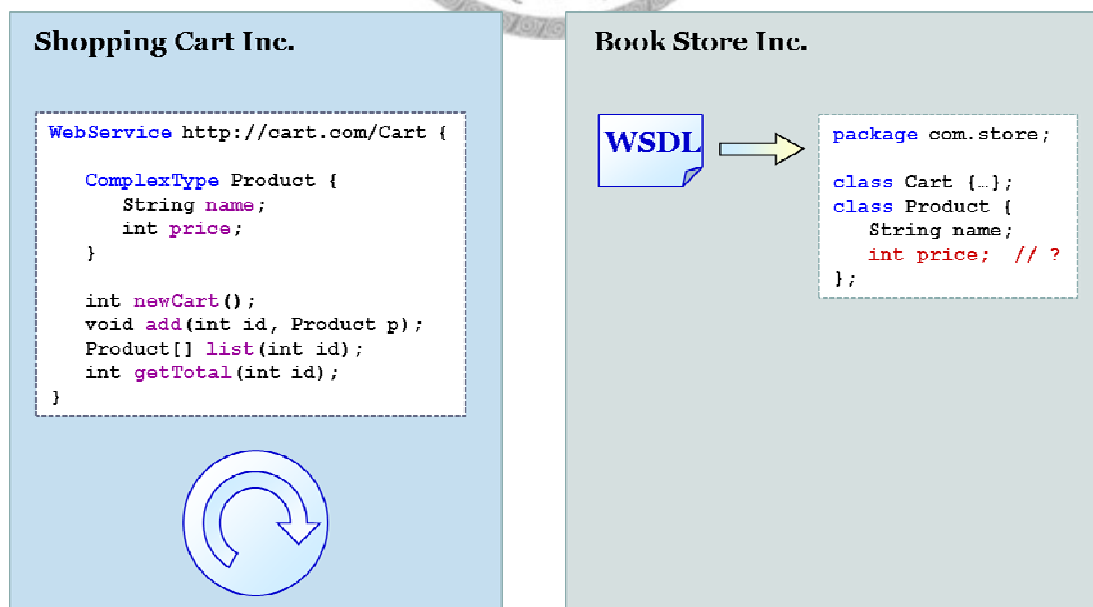


Figure 1-8: Web Service Shopping Cart Service – Polymorphism does not work here

Another possible way to solve this problem, shown in Figure 1-9, is turning **Book** class into a new Web Service using `getPrice()` to represent dynamic price. Unfortunately, Web Service can not be passed as an argument to another Web Service; otherwise we have to build specific Web Services for all users. Hence this solution is also not capable.

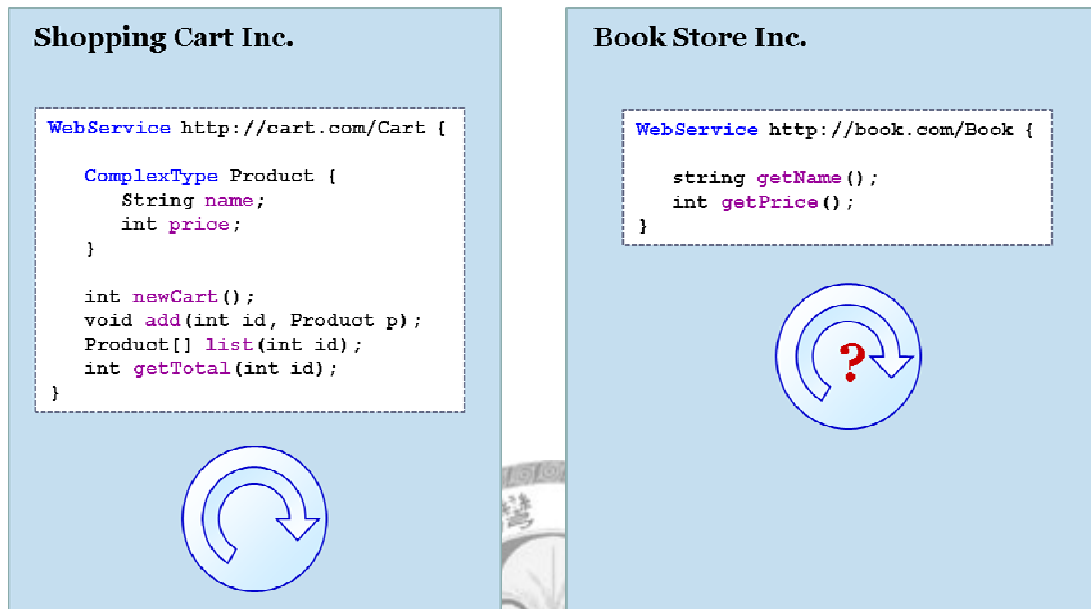


Figure 1-9: Web Service Shopping Cart Service – try another way to Override

Lastly, Figure 1-10 and Figure 1-11 shows the situation of dynamic binding in MOSP and Web Service environment.

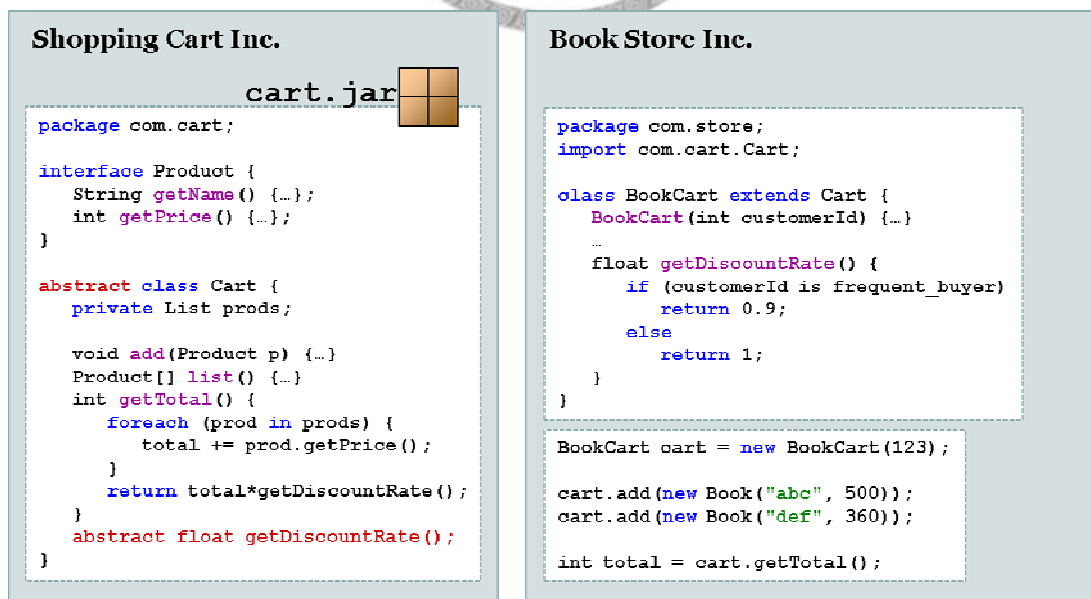


Figure 1-10: MOSP Shopping Cart Service – Dynamic Binding

Figure 1-10 shows how to enable users to make different discount for different clients. Because MOSP inheritance includes not only interface inheritance but also implementation inheritance, users can inherit the implementation details of parent class. To achieve this, all we have to do is just add an abstract operation `getDiscountRate()` for users to implement, and users can easily adjust discount for each client on their demand.

However, if we want to achieve such dynamic binding requirement in Web Service environment, shown in Figure 1-11, we have to build a new Web Service for every single user to maintain `getDiscountRate()`, which is obviously very unpractical.

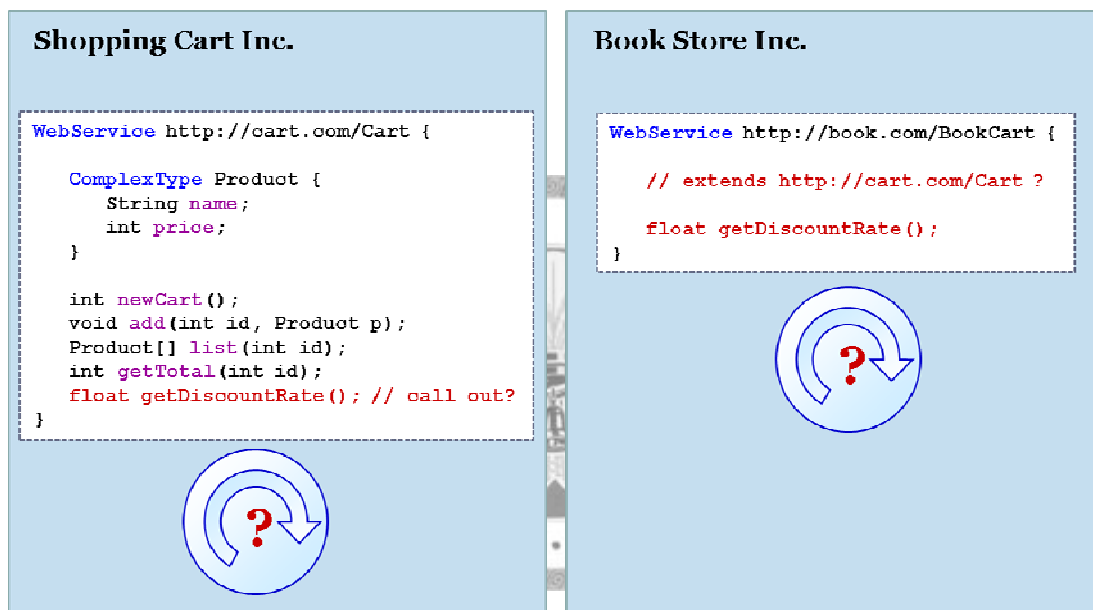


Figure 1-11: Web Service Shopping Cart Service – Dynamic Binding

In conclusion, we list the comparison between Web Service, Java and MOSP in Table 1-1. We can see that MOSP receives the advantages of Web Service and Java. Web Service is a more independent service than MOSP. Each Web Service is a single individual, which does not depend on other service. MOSP services are more dependant, they may inherit from and reference to each others.

	Web Services	Java	MOSP
Encapsulation	Y	Y	Y
Polymorphism		Y	Y
Interface Inheritance		Y	Y
Implementation Inheritance		Y	Y
Function Overloading	Y	Y	Y
Firewall Friendly	Y		Y
Instance Creation		Y	Y
Instance Lifetime		Garbage Collection	Y

Table 1-1: Comparison between Web Service, JAVA and MOSP

1.2 Motivation

From the above section, we can see that MOSP is much more powerful and with more capability than Web Service. For the environment that server and client side have more interaction, MOSP fits better than Web Service. Even though MOSP seems to be better than Web Service, it is not yet that popular as Web Service.

We have known that Web Service is the most general SOA technology at present, and MOSP is relatively fresher and less-known technology. To promote MOSP, attract new users and make the old Web Service still available in new MOSP environment, we need to provide a gateway system to **enable MOSP clients to call Web Services**.

On developing this gateway system, we may face some challenges shown as follows:

1.2.1 Challenges

1.2.1.1 Different Data Types

Since the documents Web Service delivers are XML based, the data types we use also need to be accepted by XML Schema. We list XML Schema data types in Figure 1-12.

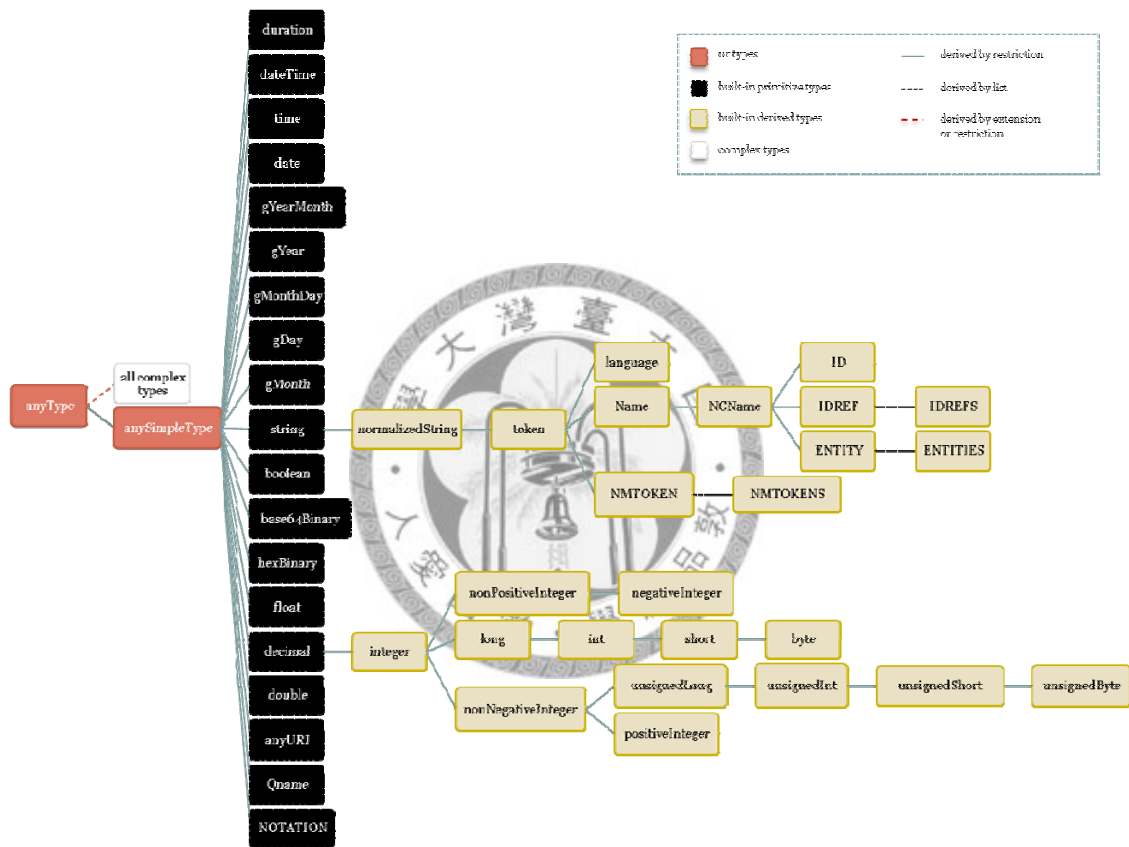


Figure 1-12: XML Schema Data Types [30]

XML Schema separates all data types into primitive types and complex types. Primitive types are the subtypes of **anySimpleType** listed in Figure 1-12, and complex types are composed from multiple primitive types and complex types.

MOSP separates all data into three main data types: `mt:/ref`, `mt:/val` and `mt:/void`. The characters behind slash represent subtype of the characters before it. If receivers can recognize the subtype, they can absolutely further understand the data type. For example, `mt:/val/num` represents numeric data, and `mt:/val/num/int` shows that `int` is one of the numeric data types. When receiver can not recognize `mt:/val/num/int`, they can take it as `mt:/val/num` or `mt:/val` data type.

Java	→ mosp	→ Java
// Special type		
Array	#[] (urt of base type)	Array (use primitive base type when possible)
org.mosp.OutArg	OutArg.getType()	(to corresponding java type)
(null value)	mt:/void	(null value)
// Reference type		
org.mosp.Instance	mt:/ref/mosp/interface	org.mosp.Instance
org.mosp.Creator	mt:/ref/mosp/creator	org.mosp.Creator
org.mosp.Typedef	mt:/ref/mosp/typedef	org.mosp.Typedef
org.mosp.MeshObject	mt:/ref/mosp	org.mosp.MeshObject
java.net.URL	mt:/ref/url	java.net.URL
java.net.URI	mt:/ref	java.net.URI
// Value type		
(string literal) java.lang.String	mt:/val/str	java.lang.String
boolean java.lang.Boolean	mt:/val/bool	Boolean
int java.lang.Integer	mt:/val/num/int	int
double java.lang.Double	mt:/val/num/double	double
float java.lang.Float	mt:/val/num/float	float
long java.lang.Long	mt:/val/num/long	long
short java.lang.Short	mt:/val/num/short	short
byte java.lang.Byte	mt:/val/num/byte	byte
java.lang.Number	mt:/val/num	(subclass of number based on data)
char java.lang.Character	mt:/val/char	char
org.mosp.Struct	mt:/val/struct	org.mosp.InStruct
org.mosp.MidlDoc	mt:/val/xml/midl	org.mosp.MidlDoc
org.w3c.dom.Document	mt:/val/xml	org.w3c.dom.Document
byte []	mt:/val	byte []

Table 1-2: Java Types marshal to MOSP Data Types and unmarshal to Java Types

MOSP is developed from Java language, thus its data types are marshaled from Java data types. Table 1-2 lists the Java data types marshaling to MOSP data types and unmarshaling back to Java data types. The left columns lists Java types, the middle columns show the MOSP data types marshaled respectively from the Java data types in left columns, and the right columns represent the Java data types unmarshaled from MOSP data types. We can see that some Java types like `java.lang.Integer` will be lost during the marshaling and unmarshaling process. The detailed MOSP data types can be referred to in Section 2.2.2.

Compare MOSP data types with XML data types, we will discover that some XML data types, such as `dateTime`, `time`, `date` and so on do not exist in MOSP. Therefore we need to define a transformation rule. Besides, the `mt:/val/struct` and array types (`#[]`) in MOSP represent the structure (composition of multiple structures and primitive types) and array (composition of one single type such as structure or primitive type), which can be transformed into XML complex types and array type respectively since they have similar ideas.

1.2.1.2 Parameter Passing (Marshalling)

While implementing the gateway which enables MOSP clients to call Web Service, we need to marshal one data type to another in order to describe the idea of data types in different environment or to transfer data through networks. This may cause some challenges, like how to transform one data type into another to deliver, and then transform it back into the original data type or near-original data type (the data type which performs just as the original one does).

1.2.1.3 Protocol Binding

The protocol bindings that Web Service uses are open. The binding protocol in most common use is SOAP binding. While facing different protocol binding of Web Service, not only the action it performs but also the user requirements may change from it. For example, if Web Service is transferred via e-mail (using MIME binding), the delay time may be longer, thus we have to adjust the timeout of the MOSP service. How the MOSP service should perform each different action of Web Service while Web Service uses different bindings is also one of the challenges.

1.3 Research Goal

➤ **Solve Problems**

Because of the difference between MOSP and Web Service, some problems and difficulties or even limitations may be brought about. We want to solve these kinds of problems.

➤ **Transparency**

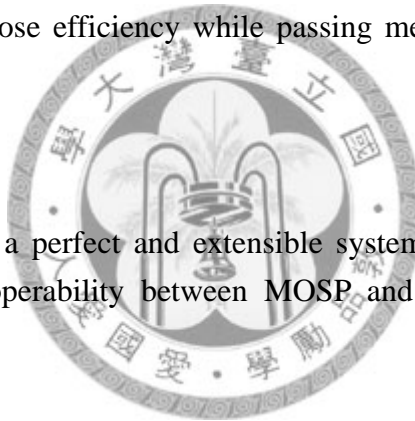
We want to reach transparency between MOSP and Web Service message passing, which means that users will not be aware of the transformation.

➤ **Efficiency**

We do not want to lose efficiency while passing message between MOSP and Web Service.

➤ **Extensibility**

We hope to provide a perfect and extensible system design to ease the future extension, such as interoperability between MOSP and RMI [22] or MOSP and CORBA [4].



Chapter 2 Related Work

2.1 Web Service

Web Service is composed of a series of standards and developing standards, which is established and assigned to advance inter-platform language to language communication by World Wide Web Consortium (W3C [30].) Its definition to Web Service is as follows, “A Web service is a software system identified by a URI, whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML based messages conveyed by internet protocols.” More specifically, W3C has already formulated a model (Web Service Description Language, also known as WSDL [28]) and a procedure calling protocol (SOAP [27] API) as formal standards of Web Service.

Web Service Framework is shown in Figure 2-1, including XML [26], SOAP, WSDL and UDDI [17], and each will be introduced later. As the figure shows, SOAP, WSDL and UDDI are all described through XML. The processing model of Web Service is as follows: Firstly, we need to transform data into XML-based type. Then we use WSDL to describe the contents of service and make the receiver side understand the information of this service from which. Lastly, we use SOAP to transfer operation requests and responses. Also, we can use UDDI to search for or register a service.

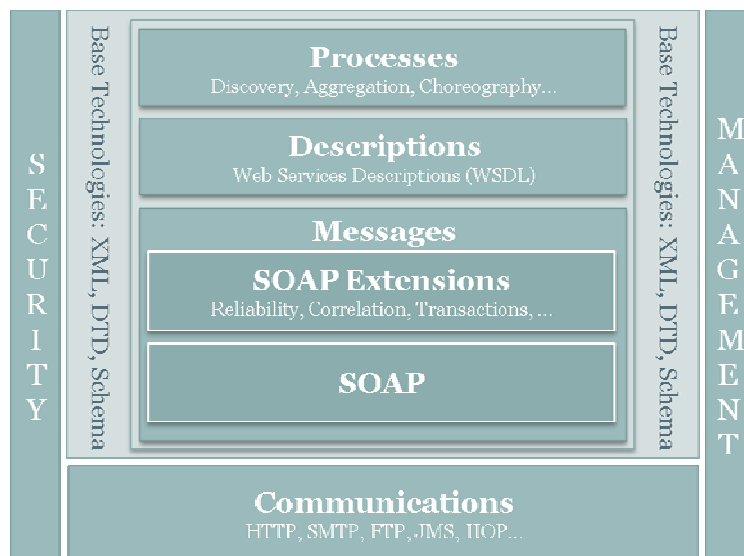


Figure 2-1: Web Service Framework [30]

We can describe the Web Service Processing Model more specifically. As Figure 2-2 shows, Web Service Provider will describe its service with WSDL, and register to UDDI of Service Broker. UDDI is just like an index catalog, which Service Requester can inquire about the service it needs, and get a WSDL file to know the information of the Web Service which describes. When the Service Requester find the service it needs, it can directly communicate with Service Provider and use the services via SOAP messages passing.

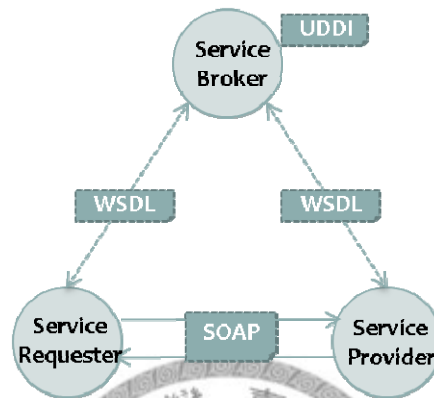


Figure 2-2: Web Service Processing Model

Web Service is based on Web open standard, and the essentials of which are HTTP and XML. However, more standards are needed to build a complete Web Service. We introduce XML and those important standards based on XML as follows.

2.1.1 Extensible Markup Language (XML)

XML is a series of principles which allows its users to define tags and simplifies data access, processing, exchange and transforming. In a word, XML is a kind of language whose document format can be defined freely. XML itself is so-called hyper-language. Figure 2-3 is a simple example of an XML-format document representing a quiz.

```

<?xml version="1.0" encoding="UTF-8" ?>
<quiz>
  <question>
    Who was the first president of the U.S.A.?
  </question>
  <answer>
    George Washington
  </answer>
  <!-- Note: We need to add more questions
  later. -->
</quiz>
  
```

Figure 2-3: A sample of an XML document

XML can cross different platforms, networks and languages. Before its showing up, every remote procedure call of message passing have to be accomplished through the communication protocol and API which are supported by both sides. XML provides better elasticity, whose open standards allow different systems to exchange data and communicate with each others. The applications build in different platforms and languages can interoperate by using XML. Web Service can combine services from XML interoperability and extensibility to provide more complex value-added services.

2.1.2 Simple Object Access Protocol (SOAP)

SOAP, as implied by its name, is a simple communication protocol which allows users to access objects in networks. SOAP use XML format together with other Internet protocols, like HTTP, SMTP and TCP, to transfer messages. SOAP messages of Web Service are usually transferred by HTTP so that SOAP messages can cross firewalls and support SSL.

SOAP is a lightweight data transfer protocol. Its way of passing data which can cross different platforms greatly simplifies information exchange in distributed environment. As long as both sides support SOAP, they can talk to each others. Thus SOAP becomes a great tool of Web Service to cross the platforms and languages.

SOAP message is transferred in Request/Response way which we familiar to. It also defines an XML framework to call an operation and pass the arguments of the operation. In the meantime, SOAP defines an XML framework to respond the return value or exceptions. However, SOAP does not define how do the sender and receiver send and receive messages, but let the developers to decide how to deal with it. SOAP does not define how the operation be implemented. In a word, SOAP leaves the implementation details to the developers.

Figure 2-4 is the architecture of SOAP Message. SOAP Envelope is a standard XML document, dividing into Header and Body. Header is used to define the SOAP contents, data types and codes; Body is used to transfer the contents of client side request or server side response, and SOAP Fault in Body is used to transfer the error messages. SOAP Message in fact is the document which packs the requests of senders and the responses of the receivers in XML format, and enables both sides to communicate with each others.

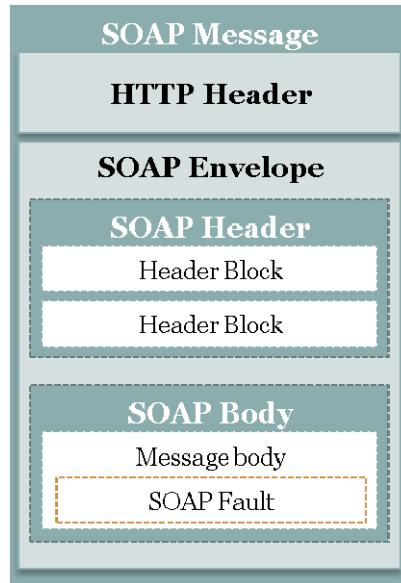


Figure 2-4: SOAP Architecture

Take an example in Figure 2-5 to describe how SOAP message works. Client wants to call `addTwoNums()` in Server through Internet. Client firstly composes a SOAP Message and sends it with HTTP as Figure 2-5 shows.

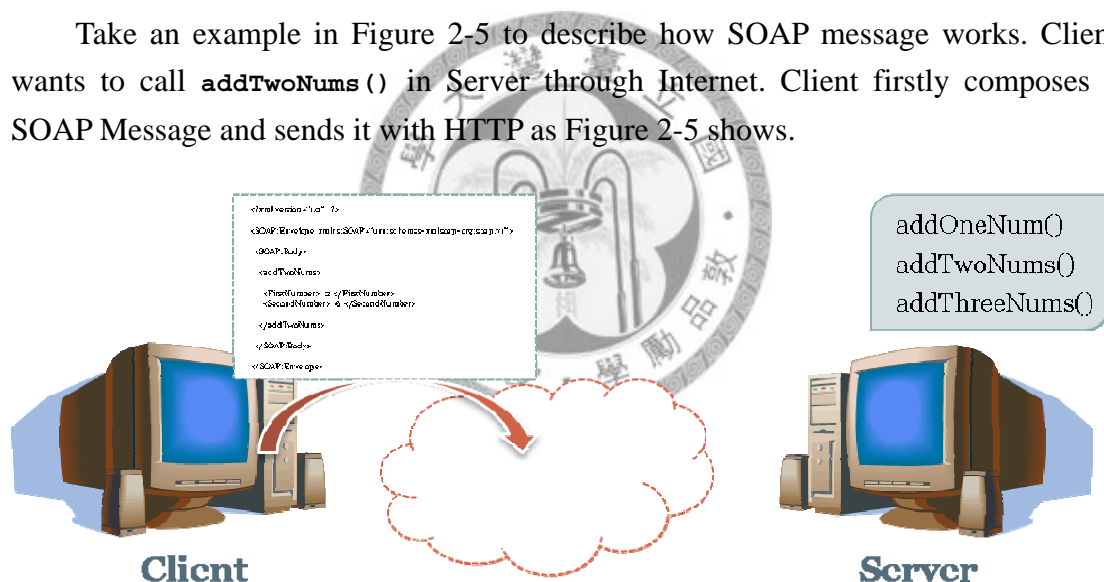


Figure 2-5: The sender transfers SOAP message

The message content which Client side sends is shown in Figure 2-6. The first line of it shows that the version of this XML document. The second line is the SOAP Envelope tag. The content “`xmlns:SOAP=...`” in this tag defines the prefix and namespace URL of SOAP. The third line is the tag of SOAP Body.

Between SOAP Envelope tag, there are tags with operation names (`<addTwoNums>`) and arguments (`<FirstNumber>` and `<SecondNumber>`.) The contents between argument tags (the 2 between `<FirstNumber>` and `</FirstNumber>`) in the argument values transferred to the operation. When Server receives this SOAP

message, it will call the operation with the arguments shown in this message (call operation `addTwoNums(2, 6)`.) Then Server will produce a SOAP message to return the execution result to Client, as Figure 2-7 shows. The message contains the name of response and the return values.

```
1 <?xml version="1.0" ?>
2 <SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">
3   <SOAP:Body>
4     <addTwoNums>
5       <FirstNumber> 2 </FirstNumber>
6       <SecondNumber> 6 </SecondNumber>
7     </addTwoNums>
8   </SOAP:Body>
9 </SOAP:Envelope>
```

Figure 2-6: The SOAP Message which Client sends

```
<?xml version="1.0" ?>
<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-org:soap.v1">
  <SOAP:Body>
    <addTwoNumsResponse>
      <Value> 8 </Value>
    </addTwoNumsResponse>
  </SOAP:Body>
</SOAP:Envelope>
```

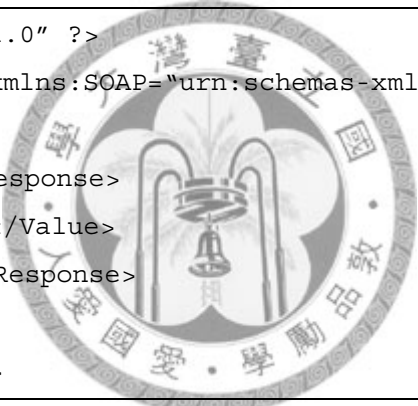


Figure 2-7: The SOAP Message which Server responses

One of the advantages of SOAP is its simplicity. Users can easily enjoy the convenience it brings even if he/she does not have professional knowledge. SOAP also helps accomplishment of distributed system, which enables developers use software services others provide more easily.

2.1.3 Web Services Description Language (WSDL)

WSDL is also an XML format document, which is mainly used to describe the details of Web Service. It enables Web Service program a standard way to describe what abilities it has and how the clients use Web Service. The content it describes includes the Web Services and the operations the service provider provides, and how the service requests communicate with these Web Service operations, including transmission protocol, data types and arguments.

Figure 2-8 shows the standard architecture of WSDL document, which is developed by IBM and Microsoft.

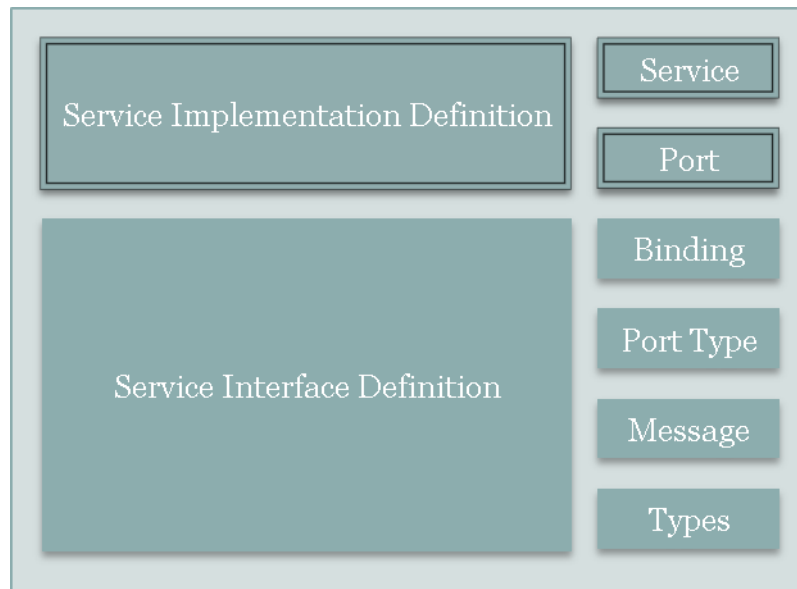


Figure 2-8: WSDL document Architecture

In this figure, we can see that WSDL document includes two parts: **Service Interface Definition** and **Service Implementation Definition**. Service Interface Definition describes the interface of Web Service. **Types** is used to describe definitions of data types, which may come from all kinds of type systems. **Message** defines the arguments and types of input and output messages. **PortType** defines the operations of endpoints of this service. **Binding** defines the communication protocol and data format of each PortType. Service Implementation Definition describes the name of the service, the company which provides the service and the location of the service. **Port** represents an endpoint for users to communicate with the service, which will assign its binding and location. **Service** is the collection of all endpoints, which is the collection of all Web Services provided as well.

2.1.4 UDDI (Universal Description, Discovery and Integration)

UDDI [16] is a developing registration center and catalog standard. It is proposed as one of the core standards of Web Service at first. It is designed to provide SOAP message inquiry service and the access of WSDL, to let users get the information like service binding and data formats to interact with services in the catalog.

UDDI is also based on XML. It can not only let service providers register to it and announce the Web Services it provides, but also let service requesters get the Web Service they need with the search service it provides.

UDDI registration includes three components. White page is the basic information of the enterprises, such as addresses, contact information; yellow page is the standard classification of enterprises; green page is the technical information about the services enterprises provide.

2.2 Mesh Object Service Protocol (MOSP)

2.2.1 MOSP Service Architecture

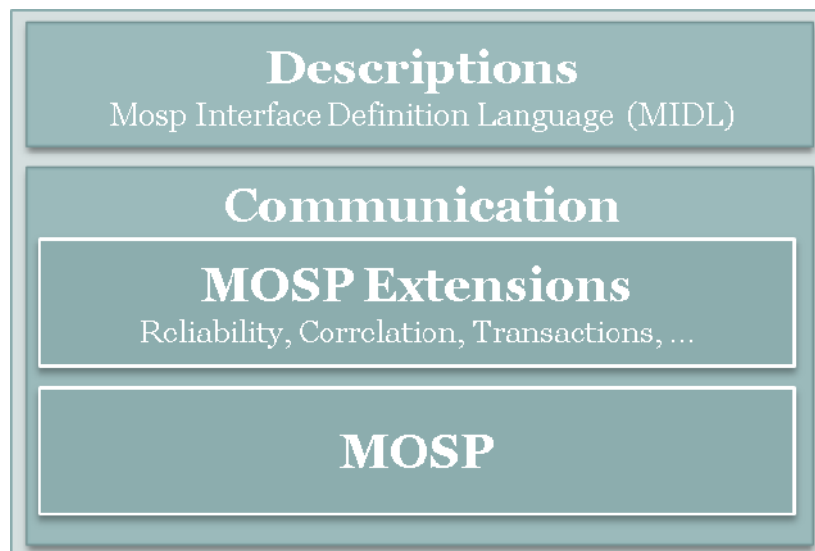


Figure 2-9: MOSP Service Architecture

MOSP, like HTTP, is the network application layer protocol responsible for transferring data on the networks. Figure 2-9 describes MOSP Service architecture. We can clearly observe the difference and similarity between MOSP and Web Service from this figure. First of all, MOSP service does not contain component like UDDI. In MOSP service, with the URL of a service, users can directly access the service and request for its MOSP Interface Definition Language (MIDL). MIDL, a little bit like but simpler than WSDL, is used to describe MOSP service. MIDL contains the operations and arguments which the MOSP service provides. Without SOAP Messages and HTTP Messages, MOSP service uses MOSP Messages to transfer data, such as the request/response messages of requesting for MIDL documents or operation calls, through networks.

Figure 2-10 shows the scenario that MOSP client and MOSP service exchange MOSP messages. Note that MOSP (DESC) represents the request for MIDL, and the response of it is a MOSP message containing MIDL; MOSP (CALL) represents the request for an operation call, and the response is a MOSP message containing the execution result of the operation call.

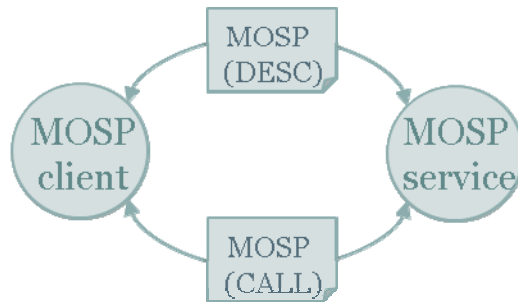


Figure 2-10: The exchange of MOSP request/response messages of service descriptions or service calls between MOSP client and MOSP service

2.2.2 MOSP Object Model

We list the main data types of MOSP in middle columns of Table 1-2. MOSP separates its data types to value type (`mt:/val`) and reference type (`mt:/ref`). The slash sign in each data types represents the inheritance relationship. That is to say, the string behind the slash sign represents the derived-type of the string before it. If the receivers can understand the meaning of the derived-type, they can thus get what this data type represents more clearly; otherwise, they can just deal with it as the base type. For example, `mt:/val/num/int/temp` represents an integer temperature data type. If the receivers can not recognize this data type, they can just see it as an integer type (`mt:/val/num/int`) or even a numeric type (`mt:/val/num`).

In MOSP environment, we name the nodes “Peer.” Every peer will be referred to by its location, noted as a MOSP URL, such as `mosp://foo.com/`. The MOSP service object of a peer, named “Mesh-Object,” is referred to by the MOSP URL of the peer and the path to it, such as `mosp://foo.com/bar`. Since MOSP is object-oriented, every mesh-object in MOSP environment is seen as a specific data type, represented by its MOSP URL. That is to say, `mosp://foo.com/bar` is a data type which can be accessed by any peer in MOSP. We class such data type as reference type. For example, if the argument data type is assigned as `mt:/ref`, we can put any data as an argument whose type belongs to reference type.

Moreover, MOSP contains the idea of inheritance. MOSP mesh-objects can not only access but also inherit from each others. This means that the mesh-object <mosp://foo.com/bar> may inherit from <mosp://zoo.com/abc>, which inherit from <mosp://koo.com/xyz>, and so on.

Table 1-2 also shows the data type marshaling and unmarshaling process between MOSP data types and Java data types. We can discover that there exists some information loss situation. So in order to enable MOSP client to call Web Service, we need to define a marshaller to deal with the marshaling and unmarshaling among MOSP, Java and Web Service data types. Note that Web Service data types do not contain the concept of scoping like MOSP. Besides, there are some other differences among these three types, which may cause some limitations.

2.2.3 MOSP Interface Definition Language (MIDL)

Figure 1-1 shows a simple example of MIDL document. We can easily discover that MIDL architecture is much simpler and easier to read than WSDL, since it only contains operation and argument information, and combines operation and arguments together. The first line is the `midl` tag, which represents that it is an MIDL file. The `op` tag in the second line represents the operations provided by this MOSP service; `name` represents the operation name, and `type` represents the return type. The third line represents the input argument of this operation. Note that `#[]` represents MOSP array type, and `st` represents MOSP struct type, which can be referred to by a `#` sign and its name, such as `#power` shown in the fourth line. While implementing a MOSP service in Java, we only have to add a `@midl` label before a class to announce it as a MOSP service and a `@op` label before each operation open to MOSP clients.

```
1 <midl role="instance" xmlns="mt:/val/xml/midl">
2   <op name="calculate" type="mt:/val/int">
3     <arg type="mt:/val/int#[]" name="intArray">
4     <arg type="#power" name="powerOfNumbers">
5   </op>
6   <st name="power">
7     <arg type="mt:/val/int#[]" name="powerArray">
8     <arg type="mt:/val/bool" name="usePower">
9   </st>
10 </midl>
```

Figure 2-11: A sample MIDL

2.2.4 MOSP Messages

We have described the exchange of MOSP messages between MOSP client and service in Figure 2-10. Web service uses HTTP message to pack SOAP message and transfer on networks, while MOSP considers that way too complicated and bothering, which makes messages become heavy and cause time-waste while unpacking them. Hence MOSP merges HTTP message and SOAP message into a single message, which is so-called MOSP message. MOSP message passes data in plaintext form.

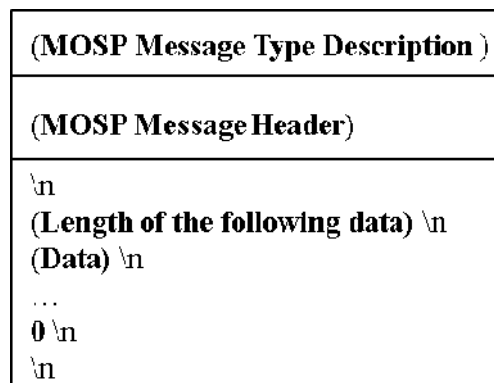


Figure 2-12: MOSP message framework

Figure 2-12 shows the MOSP message framework. The first line in MOSP message is the description of this message, such as operation call or response message. Later part of MOSP message is the message header, which is optional, used to reveal some information such as operation name. The rest of the message is the data to transfer. The data content is described by a combination of a number representing the length of the data and then the data itself. Such combination may show up over and over in this part of MOSP message. 0 represents the end of this message, since 0 means that the length of the following data is 0. Information in this message is separated with a line feed.

Figure 2-13 and Figure 2-14 shows the request and response MOSP message while MOSP client requests for MIDL respectively. The first line in Figure 2-13 shows the MOSP version this message uses is **MOSP 0.8**, and **DESC** represents the request for the description of the service, MIDL. This message does not contain other information, thus 0 is used to represent the end of this message.

MOSP service uses an operation **onDesc()** with arguments **InMsg** and **OutMsg** to receive the MOSP description request message. Once the service receives MOSP description request message, it will be received as an **InMsg** object. Then service can pack the response message containing MIDL file as an **OutMsg** object to transfer.

Figure 2-14 is the response of the above request message from MOSP service. 200 is the standard response code (a similar idea with HTTP), and each number represents different meaning. OK is the meaning of the response code 200, which means that the receiver successfully resolves the receipt message and returns the correct result, which is the content of MIDL (in plaintext form) here. Other response codes such as 403 Forbidden, 404 Not Found, and so on, can be referred to in the **RespCode** class in MOSP source code.

```
MOSP/0.8 DESC /
0
```

Figure 2-13: MOSP client request for MIDL

```
MOSP/0.8 200 OK
102
<midl>
.....
</midl>
0
```

Figure 2-14: MOSP service response of the MIDL request

Figure 2-15 and Figure 2-16 shows the request and response MOSP messages while MOSP client requests for an operation call respectively. In Figure 2-15, **CALL** represents the request for an operation call to MOSP service, and **/path/service** represents the location of the service in server peer. The message header **Op:fun1** represents the operation which is called, and the input arguments are shown in the later message content. **3** represents the length of the value of the argument, **type=mt:/val/num/int** represents the data type of the argument, and **123** represents the value of the argument.

MOSP service uses an operation **onCall()** with arguments **InMsg** and **OutMsg** to receive the MOSP operation call message. Once the service receives MOSP call message, it will be received as an **InMsg** object from which receiver can get **Arg** objects, which represent arguments, to get the data content. After the execution of the operation, service can pack the response message as an **OutMsg** object to transfer.

Figure 2-16 is the response of the above request message from MOSP service. `200 OK` means that the receiver successfully resolves the receipt message, calls the operation and returns the correct result. `18` represents the length of return value, and `mosp://foo.com/xyz` represents the data type of return value: `mosp://bar.com/abc`. From this message, we can also discover that object `mosp://bar.com/abc` inherits from `mosp://foo.com/xyz`. It is the concept of the inheritance on networks in MOSP.

```
MOSP/0.8 CALL /path/service
Op:fun1

3; type=mt:/val/num/int
123
0
```

Figure 2-15: MOSP client request message for an operation call

```
MOSP/0.8 200 OK

18; type=mosp://foo.com/xyz
mosp://bar.com/abc
0
```

Figure 2-16: MOSP service response message of the operation call

2.3 Interoperation between CORBA and Web Service

The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG [21]) that enables software components written in multiple computer languages and running on multiple computers to work together. CORBA, a little bit similar to MOSP, also has the concept of object-oriented and inheritance. We can use the integration experience of CORBA and Web Service as consultation.

2.3.1 Common Object Request Broker Architecture (CORBA)

CORBA was built by OMG in 1992, which is an open standard used on distributed objects. CORBA enables clients to call operations of remote objects, regardless of the language binding and locations of the objects.

Interaction between CORBA client and server is regulated by Object Request Brokers (ORBs) on both sides. CORBA client and server communicate through Internet Inter-ORB Protocol (IIOP) or General Inter-ORB Protocol (GIOP). CORBA objects can be on client side or server side without affecting the execution and use of it. The operations provided by CORBA objects are defined by Interface Definition Language (IDL). The operations defined on the interface will accept input arguments and get return values or exceptions.

The language CORBA supports includes C, C++, Java, Ada95 and COBOL, and some scripting languages like Perl, Python and JavaScript. Besides, CORBA is independent of operating systems, which can work on many platforms, such as Win32, UNIX and real-time embedded system. The communication protocols CORBA uses in ORB communication includes TCP/IP, IPX/SPX, ATM and so on.

Figure 2-17 shows the main components of CORBA reference model. These components provide portability, interoperability and transparency for CORBA.

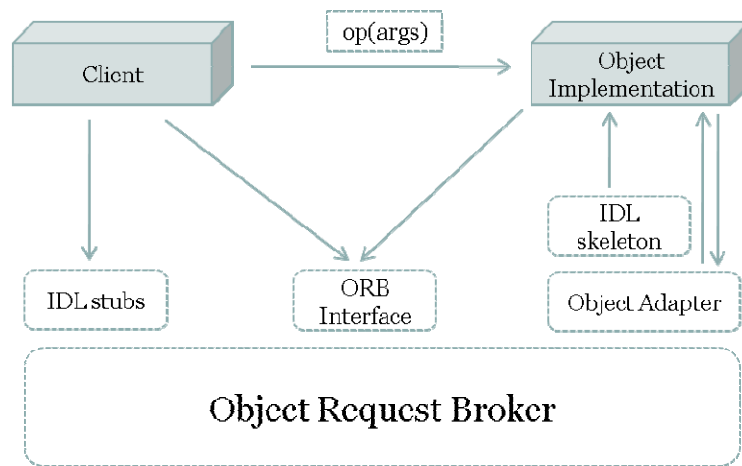


Figure 2-17: Components in the CORBA 2.x Reference Model [2]

CORBA application life cycle is as follows:

1. Define the provided service interface in IDL.
2. Compile the IDL and create client stub and server skeletons.
3. Execute the service and connect it to skeletons with Object Adapter in Figure 2-17.
4. Publish this service with Naming Service or Trading Service.

The execution process of CORBA client is as follows:

1. Connect to the service with Naming Service and get object reference.
2. Create client stubs with IDL-compiler to call the operations of the object reference. Besides, client can also look up the operations the service provides in Interface Repository (IR), and dynamically generate request with Dynamic Invocation Interface (DII).
3. Deal with the responses or exceptions the server transferred.

Table 2-1 lists a comparison between Web Service and CORBA, from which we can see that even though these two systems use different technology, the idea and the architecture are similar. Table 2-2 lists the comparison of WSDL and IDL, and the transformation between them can be designed based on this table. Table 2-3 lists the comparison between XML schema and CORBA object model, we can refer to it as the comparison between XML schema and MOSP object model.

Item	Web Services	CORBA
Protocol	SOAP, HTTP, XML, Schema	IIOP, GIOP
Location Identifiers	URLs	IORs, URLs
Interface Spec.	WSDL	IDL
Naming, Directory	UDDI	Naming Service, Interface Repository, Trader Service

Table 2-1: Comparison between Web Service and CORBA [10]

WSDL	IDL
No mappings to programming languages	Mappings to many programming languages
Describes mapping to transport layer	Mapping to transport layer defined by CORBA specification
XML Schema “object model” for XML documents	Object model for programming languages defined by CORBA specification
Draft	OMG standard, many implementations

Table 2-2: Comparison between WSDL and IDL [13]

XML Schema	CORBA Object Model
Deriving by extension, equivalence classes	Multiple interface inheritance
Defines elements and attributes of instances	Defines objects, operations and operation parameters of instances
No stable validators yet	Many IDL compiler implementations

Table 2-3: Comparison between XML Schema and CORBA Object Model [13]

2.3.2 Gateway Systems between Web Service and CORBA

Since Web Service and CORBA have similar architecture, directly use Web Service to replace CORBA is not practical. We can just build the interoperability between Web Service and CORBA service by implementing the gateway service, which can transform SOAP and CORBA IIOP messages automatically. Present SOAP-CORBA gateway includes SCOAP [20], XORBA [23] and soap2corba bridge [24]. The actions gateway performs here are accepting SOAP requests, transferring it to CORBA server and transferring the return result to SOAP response.

SCOAP combines SOAP with CORBA by mapping CORBA IDL with Web Service SOAP, and its architectures are shown in Figure 2-18 and Figure 2-19, Generic SOAP/HTTP to IIOP Bridge and Static Dedicated SOAP/HTTP to IIOP Bridge respectively. The former translates SCOAP message into IIOP and transfer it to IIOP domain, as Figure 2-18 shows, and this kind of translation needs to use IR or IDL to generate the mapping, and SCOAP types mapping to IDL types can be referred to in [20], which can be returned only when IIOP message body is accessible in SCOAP body. The later regulates the access to arbitrary CORBA/SOAP servers, which can only bridge operations of specific interfaces.

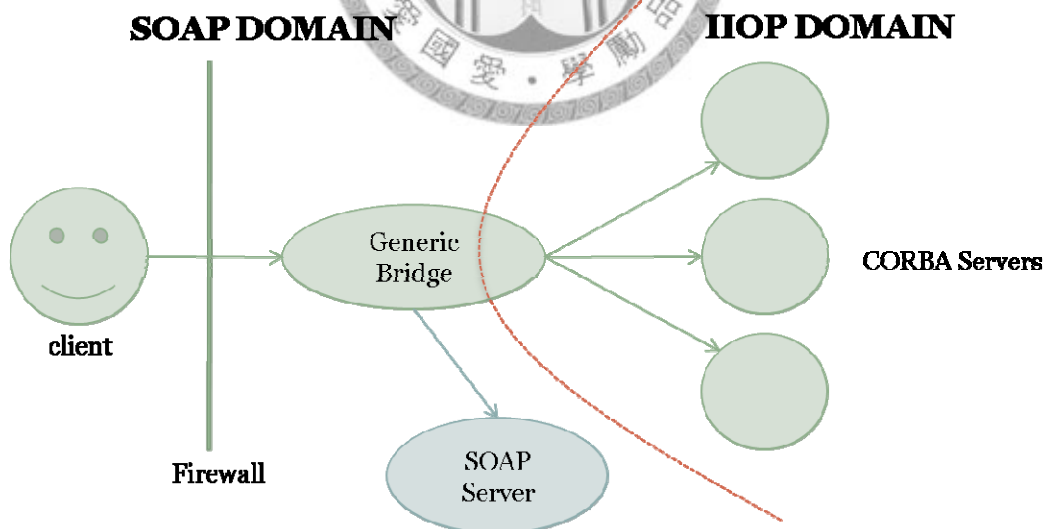


Figure 2-18: Generic SOAP/HTTP to IIOP Bridge Diagram [20]

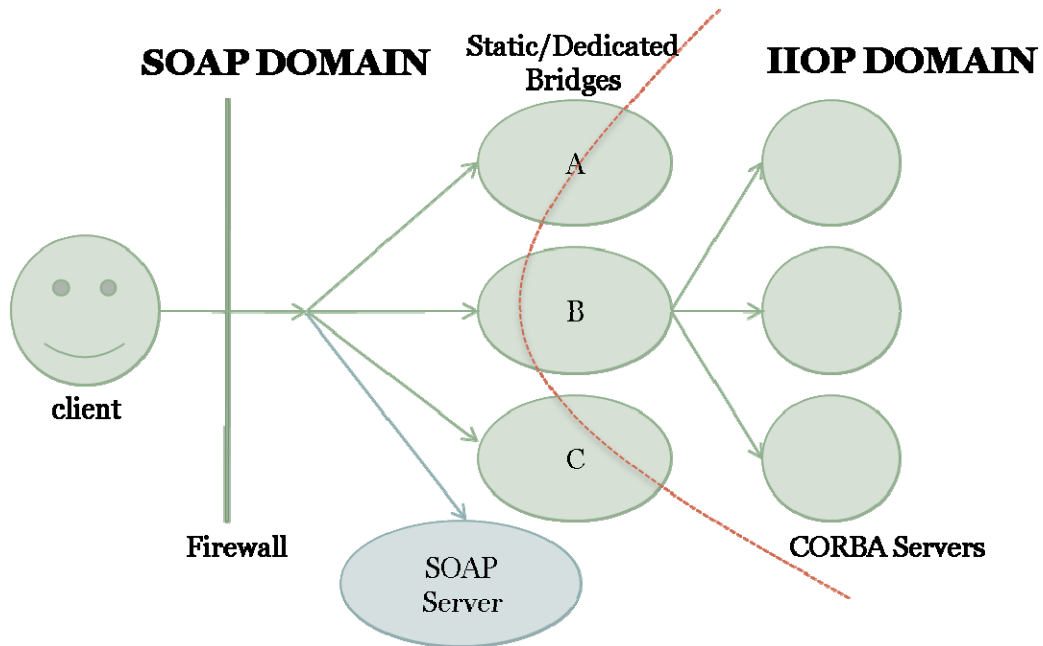


Figure 2-19: Static Dedicated SOAP/HTTP to SCOAP/ORB Bridge Diagram [20]

The above scenario shows how SOAP client interact with CORBA server through bridge. However, this interaction works only when SOAP and CORBA types are compatible. (For instance, there will be some problems when object reference is concerned.)

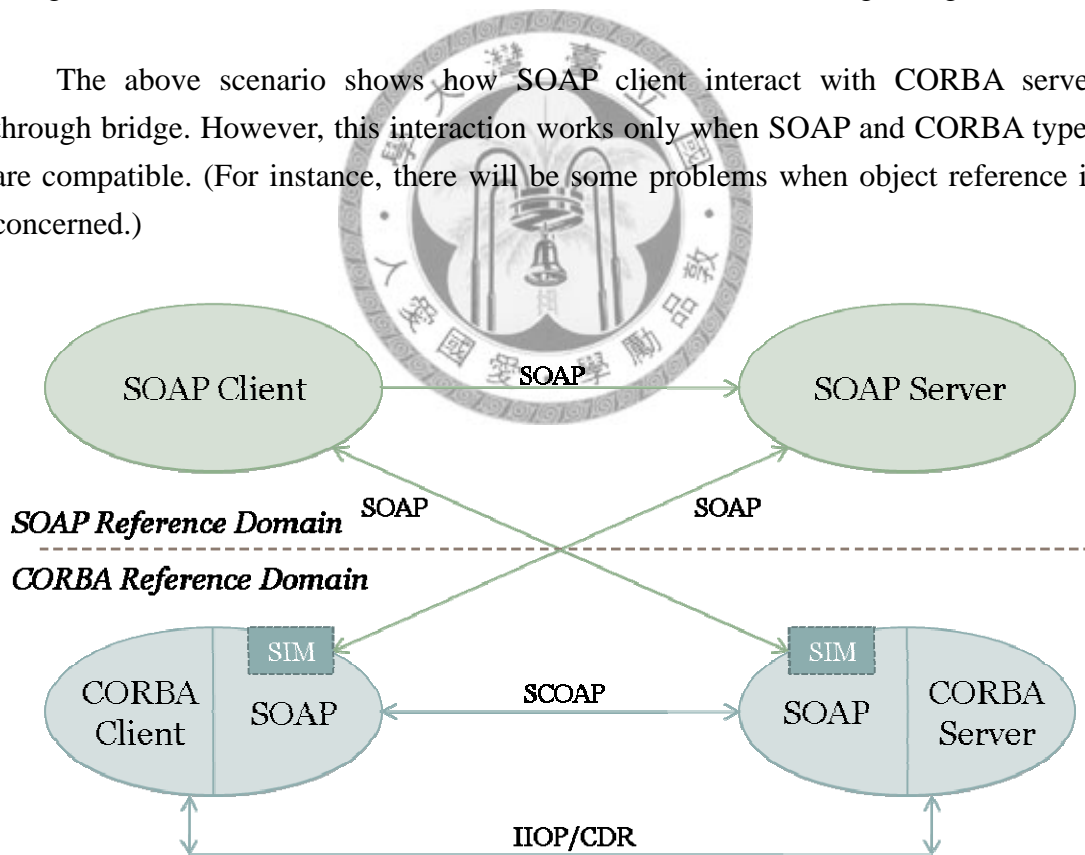
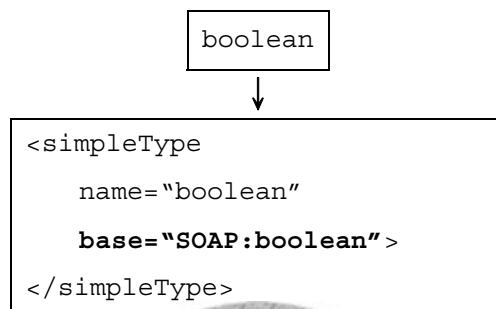


Figure 2-20: SOAP-CORBA Interoperability Interaction Model [20]

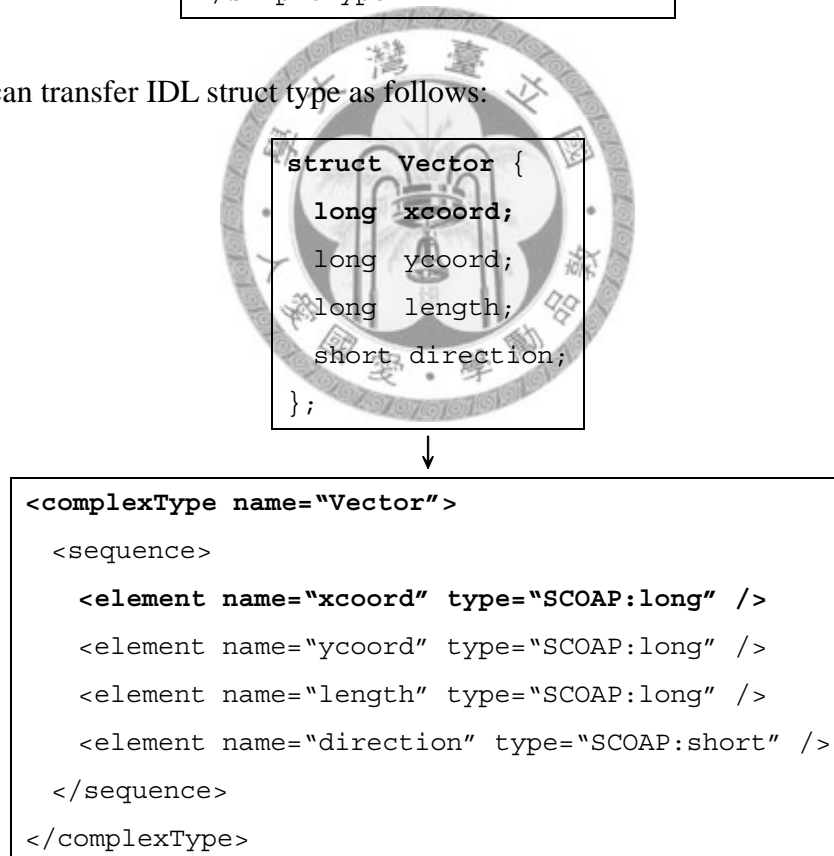
Figure 2-20 shows the interaction model between SOAP, SCOAP, CORBA clients and servers. SIM represents SOAP-IDL mapping, which plays the role to translate SOAP encoded arguments and messages to the format which system can

interpret. Arrows represents the interaction between entities, and the annotations on arrows shows the encoding technology used during interaction. We can see from this model that we can package CORBA with SOAP, which means Web Service. Thus CORBA can be transferred in Internet by tunneling with SOAP messages. Besides, SOAP and CORBA and communicate through IDL-SOAP mapping. We can design the mapping between MOSP and Web Service by referring to this model.

We show a simple IDL-SOAP mapping below. CORBA IDL type “`boolean`” is similar to SOAP type “`SOAP:boolean`”, so we can transfer it as follows:



We can transfer IDL struct type as follows:



We can see that data types of CORBA and MOSP are similar. The above mapping way may be feasible in MOSP-Web Service gateway.

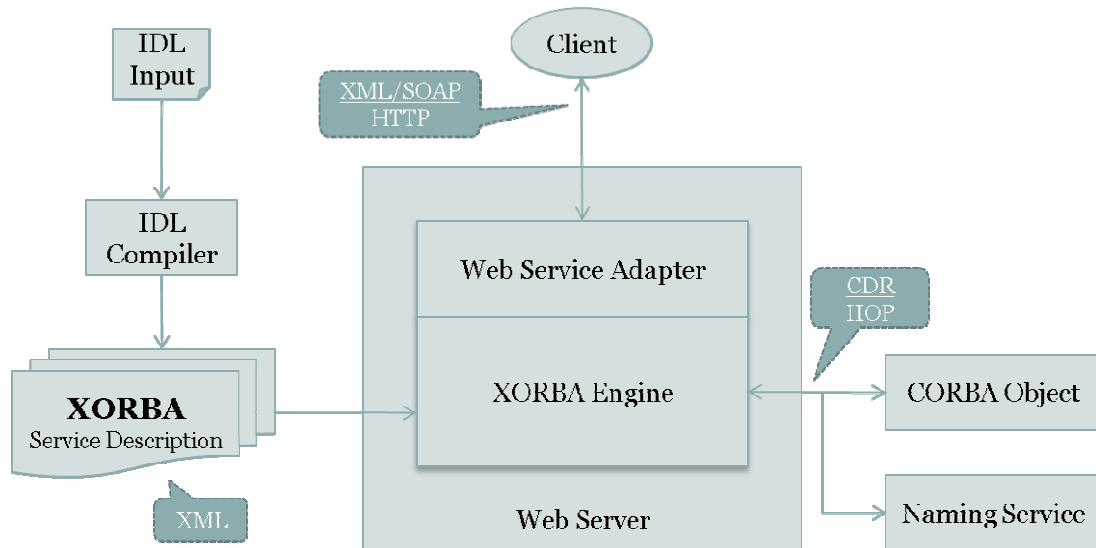


Figure 2-21: XORBA Architecture [23]

XORBA, also known as XML-CORBA Link, is developed by Rogue Wave Software, shown in Figure 2-21. XORBA is a Generic SOAP to CORBA Bridge, using SOAP Envelope to transfer in network, which can translate SOAP request to IIOP request and translate IIOP response to SOAP response. CORBA interface will be translated to the sub-elements **Interface** in SOAP message header, as the interface shown in Figure 2-22 and the SOAP message header in Figure 2-23.

In XORBA, CORBA object reference can be represented as string format or URI in Naming Service, which solves the predicament that SCOAP can not use object reference. In MOSP, object reference is represented by its MOSP URL. However, MOSP contains the concept of implementation inheritance, while CORBA only supports interface inheritance. The situation in MOSP is not as simple as in CORBA.

```

module PatientCare {
    enum ReturnValue { OK, Failed };

    interface PatientRecordManager {
        ReturnValue newPatient(in string fname, in string lname);
    }
}

```

Figure 2-22: IDL Fragment of a XORBA Message Example [13]


```

<SOAP:Envelope xmlns:SOAP="urn:schemas-xmlsoap-org:scap.v1">
  <SOAP:Header>
    <Interface SOAP:mustUnderstand="1">
      PatientCare.PatientRecordManager
    </Interface>
  </SOAP:Header>
  <SOAP:Body>
    <newPatient>
      <fname> Thomas </fname>
      <lname> Jefferson </lname>
    </newPatient>
  </SOAP:Body>
</SOAP:Envelope>

```

Figure 2-23: A Sample SOAP Request [13]

However, [7] proposed a more complete integration model referring to OMG standards. It adds a SOAP-CORBA gateway based on CORBA servant, completes the OMG standards and solves the performance bottleneck and single point of failure problems other gateways can not solve.

Firstly, two specifications [19] [18] of OMG about the interoperation between CORBA and WSDL/SOAP defines how IDL and WSDL/SOAP maps to each other. However, these standards omitted the support of CORBA client to SOAP service, which can not satisfy the requirement overall. [6] considered the single way situation from CORBA client to Web Service, and accomplish the static transformation between WSDL and IDL according to OMG specifications. However, the gateway it proposed can only be arranged at relative ORB server side, which will cause performance bottleneck and single point of failure problem at ORB server side.

The Proxy service proposed in [9] supports CORBA client to call CORBA service through Internet. However, there is only one proxy service responsible for the translation between SOAP and CORBA, which thus becomes the performance bottleneck of the entire system, and causes the single point of failure problem.

[7] proposed an overall solution, an Integration Model, which can solve the above problems, shown in Figure 2-24. This model provides two types of gateways: the CORBA-SOAP gateway is responsible for packing CORBA service into standard Web Service form; the SOAP-CORBA gateway is the part not concerned in OMG specification, which enables pure CORBA client access Web Service without any modifications. The mapping in this figure is responsible for translating IDL and WSDL documents, assigning the result to the relative repository in each domain, and saving a copy with relationships and attachments, such as the relationship between CORBA Interoperable Object Reference and Web Service Endpoint, in the original message repository, since IDL and WSDL focuses on different points.

CORBA-SOAP gateway is like a standard CORBA service object servant in CORBA domain, every translation and method of calling service is packaged in implementation. CORBA clients and servers do not have to change. This kind of design not only solves the problems of performance bottleneck and single point of failure but also balances the loading of the entire system.

The process of service call in SOAP-CORBA gateway can be described by a simple example. Firstly maps WSDL to IDL. Then saves the IDL document and the attachments to CORBA Interface Pool, and saves information related to implementation, such as Endpoint, to CORBA Implementation Pool. CORBA developer can implement the service depends on this IDL document, and register service object implementation information in CORBA Naming/Trading Service. CORBA clients can statically request with DII through the client stub compiled from IDL document. For CORBA client, this process is not much different from calling other CORBA service objects.

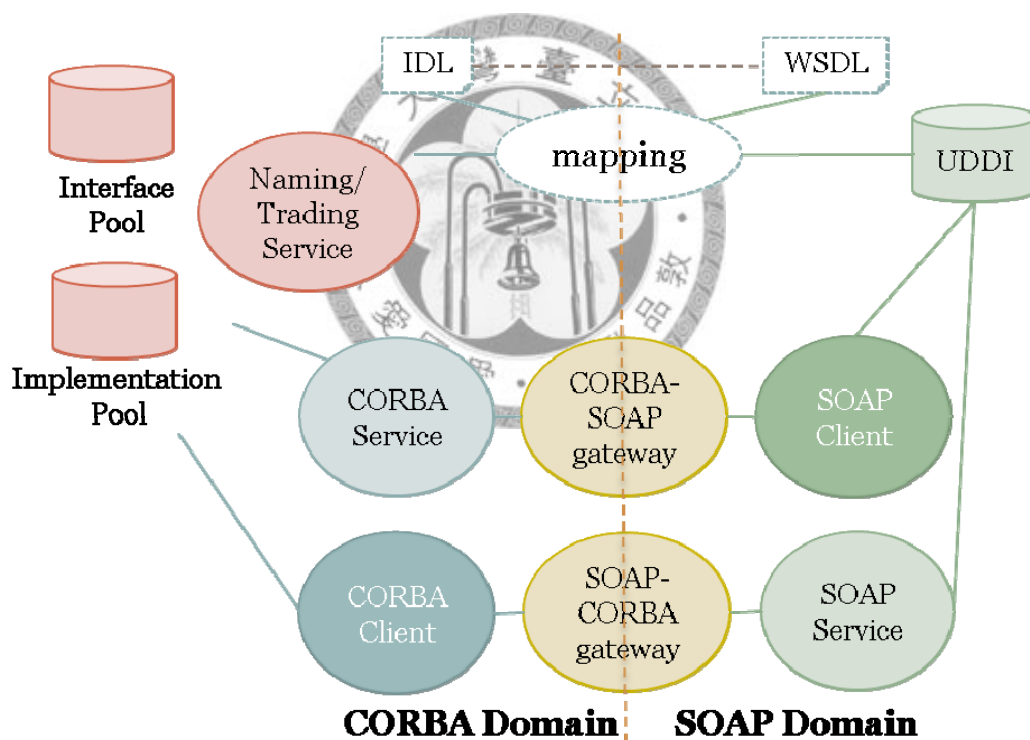


Figure 2-24: CORBA/SOAP Integration Model [7]

2.4 Brief Summary

Based on the earlier studies, we can make a brief summary. Firstly, Web Service provides a systemized and extensible architecture through the open standards of network communication protocols and data formats. Its main purpose is to simplify the integration of applications from different platforms in distributed systems. Web service is composed of three main components: SOAP, WSDL and UDDI, which all use XML format based on its open standard.

SOAP plays a role as the request message that Web Service client used to call Web Service and the response message the server returns to client. That is to say, SOAP is the message about service operation call and response. SOAP can easily fit the user needs for its convenient and simple architecture. WSDL is the document enabling Web Service applications describe itself with a standardize way, including the operations, protocol bindings, arguments and data types. UDDI is like a catalog of Web Services to which Web Service providers can register and publish its Web Services, and on which service requesters can search for services, get WSDL documents they need, and invoke the services as well.

A brand new protocol, MOSP, was born for the needs which Web Service can not satisfy. A great difference of MOSP from Web Service is the concept of object-oriented, inheritance and the dependence relationship between MOSP services. MOSP is stateful, it can record user states with object instances, thus can provide more complex and interactive services to users. It is hardly possible to transfer Web Service, which is independent, to MOSP service, which is interdependent.

CORBA is similar to MOSP since they both have the concept of object-oriented and inheritance. Note that CORBA inheritance is limited to interface inheritance, while MOSP enables implementation inheritance. We discussed some studies of the gateway between Web Service and CORBA.

SCOAP contains two models of publishing CORBA/SOAP service at server side: Generic SOAP/HTTP to IIOP Bridge and Static Dedicated SOAP/HTTP to IIOP Bridge. The former translates SCOAP message to IIOP and transfers it to IIOP domain, while the later regulates the access to arbitrary CORBA/SOAP server focusing on operations of specific interfaces. We can use the IDL-SOAP mapping in SCOAP as a reference for the data type mapping between MOSP and Web Service.

XORBA, XML-CORBA Link, is a Generic SOAP to CORBA Bridge, which uses SOAP Envelop to transfer in networks, translates SOAP request to IIOP request, and vice versa. The information of CORBA interface will be attached in SOAP message header, and CORBA object reference can be represented as string format or URI in the Naming Service, which solves the predicament that SCOAP can not use object reference.

Lastly, we mentioned a CORBA/SOAP Integration Model which combines OMG official specification and old gateway systems. It completes the OMG standards, which supports CORBA client calling SOAP service, and solves the performance bottleneck and single point of failure problems of old gateway systems. This system provides bidirectional gateway system, one is the CORBA-SOAP gateway which fits OMG specification, and the other is the SOAP-CORBA gateway. The concept of this system model and the benefit of load balancing can be a great prototype to build the gateway system between MOSP and Web Service.



Chapter 3 System Design

3.1 System Overview

3.1.1 Concept of MOSP Service

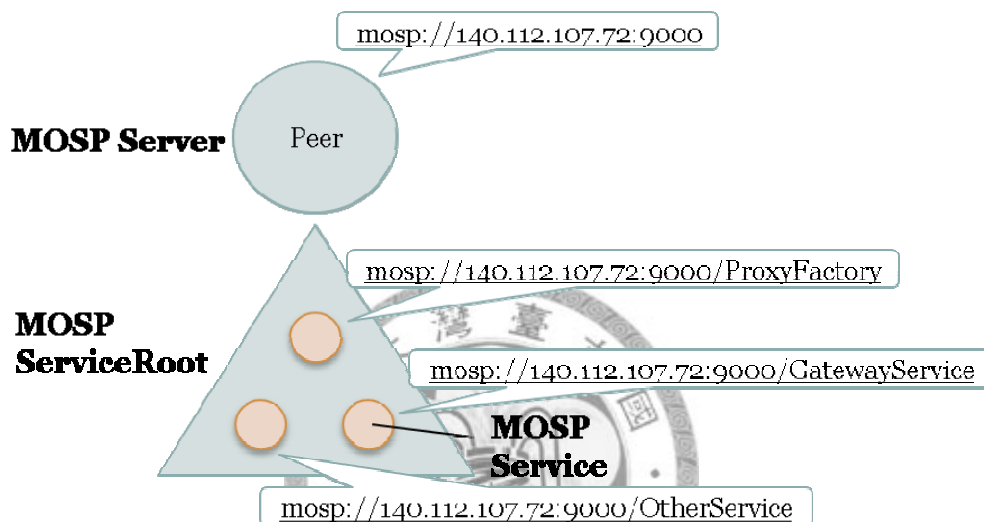


Figure 3-1: How a MOSP Service works

To introduce the architecture of our gateway system which enables MOSP clients to call Web Service, we need to start with the architecture of MOSP service. In MOSP environment, every node is called a **Peer**. Each peer may be a MOSP client or server.

As Figure 3-1 shows, the peer in this figure plays the role as a MOSP server, and each MOSP server contains a **ServiceRoot**, which contains many **MOSP Service** objects. Each MOSP Service object refers to one specific path to the MOSP server. ServiceRoot represents the entrance point of each server, like the root of a service tree. When the clients request for services, they will enter the tree from ServiceRoot and search for the services they need.

When a MOSP server peer starts, and opens a port (say, port:9000), the peer will be located to its specific MOSP URL (a link starts with `mosp://`, just as HTTP URL). MOSP clients can get a **MeshObject** representing a MOSP server by binding to the peer through its MOSP URL, and use the provided services through this MeshObject. If MOSP server peers want to provide some services, they can just add the service objects and the paths of the service objects to their ServiceRoot.

When MOSP clients want to use the services that MOSP servers provide, they can get a MeshObject object instance of each service from its MOSP URL (the MOSP URL of its server plus its path), and perform the actions directly to the object instance. For example, get the MIDL document of a MOSP service by `getMidl()` operation of the object instance, or call the MOSP service by `call()` operation.

3.1.2 Design of Generic Gateway Service System

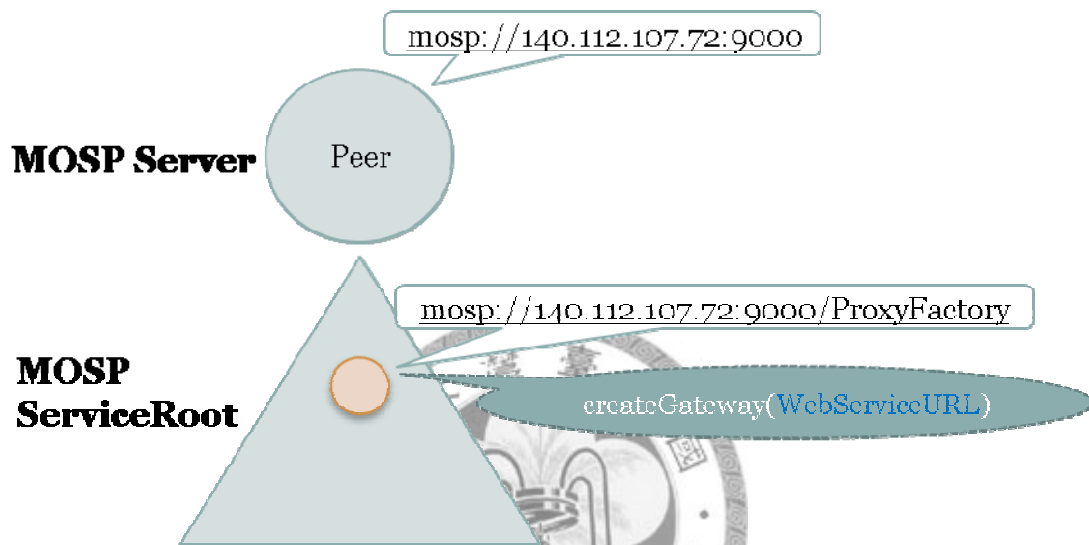


Figure 3-2: Generic Gateway Service System

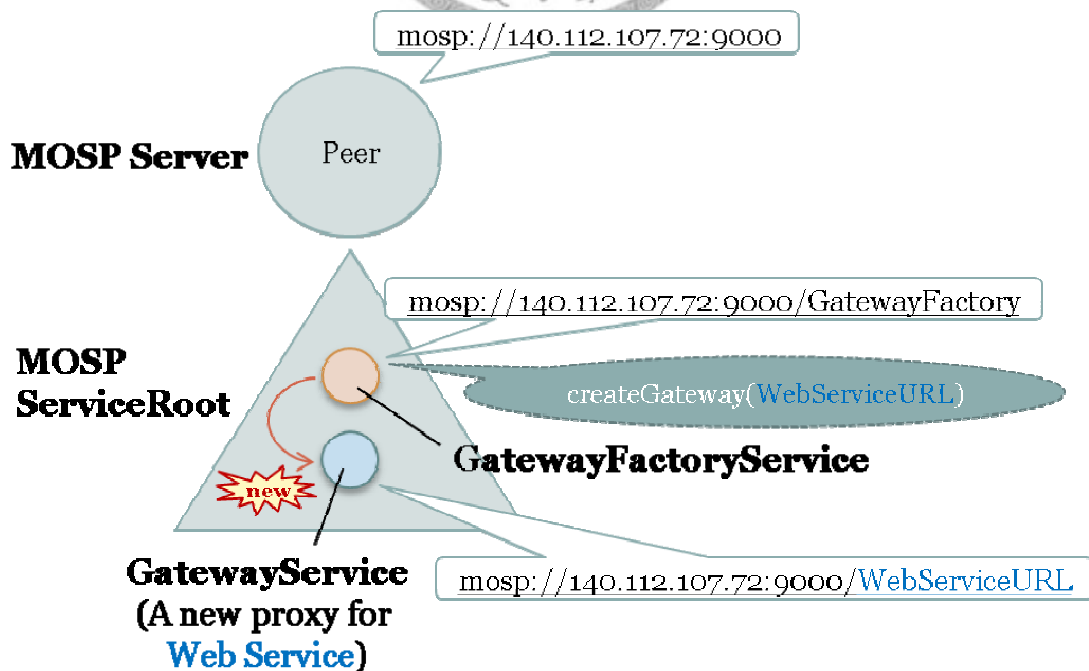


Figure 3-3: Generic Gateway Service System (when `createGateway()` is called)

The design goal of our gateway system is to enable MOSP clients to call Web Service, and make old Web Service still available in new MOSP environment without re-developing it. Based on this requirement, the gateway system needs to achieve the ability to dynamically transform WSDL documents of different Web Services into MIDL documents describing MOSP service; on the other hand, the system needs to achieve the ability to compose SOAP messages and transform which into and from MOSP messages as well. According to these requirements, we design this **Generic Gateway Service System** as Figure 3-2 and Figure 3-3 show.

The capability of the Generic Gateway Service System is to dynamically create a relative MOSP Service for MOSP client to call according to the Web Service WSDL URL. The relative MOSP Service provides the same operations as the Web Service. That is to say, the way of doing this is just like decorating a Web Service to a MOSP Service.

The MOSP peer in Figure 3-2 is the Generic Gateway Service System. The service root of the peer initially contains a MOSP service, which is named **Gateway Factory Service**. It provides an operation `createGateway()`, which firstly generates a specific **Gateway Service** object according to the input argument (a WSDL URL in string format), and then settles a unique path to this object and add it to the service root. After the execution, there is another MOSP Service, which is the dedicated Gateway Service for the specific Web Service, available for clients.

To create the unique path of each Gateway Service generated from different Web Service, the system hashes the WSDL URL to a string and uses it as the path. However, the hash values still may be the same, so the system saves those values for later comparison. Once a new path is generated, the system compares it to the old paths. If the path name already exists, the system repeatedly appends some string to it until the name becomes unique.

In fact, WSDL documents in different locations may refer to the same Web Service, because Web Service are not differentiated from its WSDL URL but from its **{target namespace, portType name}** combination in WSDL document. Therefore, whenever the operation `createGateway()` is called, firstly the system needs to check if the relative Web Service which the input WSDL describes already exists in the system. If so, the system has to update the Gateway of this Web Service. To achieve this, Gateway Factory searches for the information of target namespace and portType name in the WSDL, and saves it in a table. If the target namespace and portType name combination already appears in the table, it will update the Gateway by this new WSDL document, and returns the Gateway object, with the original path, to the client.

Hence, MOSP client can operate this Gateway object as an ordinary MOSP service object: to get the MIDL document of this service, or to call the operations it provides. Here, the MIDL and operations Gateway provides are respectively transferred from WSDL document and operations of the relative Web Service.

3.1.3 Design of Gateway Service

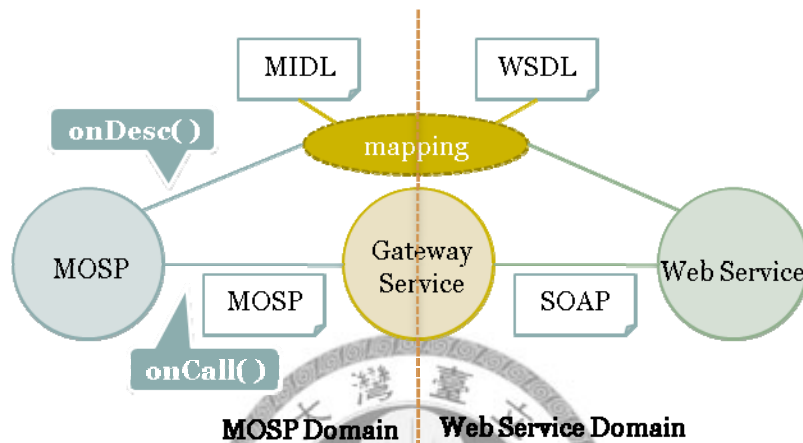


Figure 3-4: Gateway Service Work Model

Figure 3-4 is the work model of Gateway Service. Each MOSP service contains two parts: the description part, represented by the operation `onDesc()`, and the implementation part, represented by the operation `onCall()`. MOSP service uses `onDesc()` and `onCall()` to process the receipt and response of MOSP **DESC** message and MOSP **CALL** message respectively. By setting these two operations, we can set the MIDL document defining the operations provided and set the actions to perform and the return messages on each client operation call of the service.

In `onDesc()`, Gateway Service reads in a WSDL document and transfers it to MIDL file. We have described the architecture of WSDL and MIDL in Figure 2-8 and Figure 2-11 respectively, from which we can know that MIDL only describes the operation names, input arguments (including argument names and types) and return types. We can get this information from Service Interface Definition part of a WSDL document, which are **Types**, **Message** and **PortType**, with a WSDL parser.

Firstly, we can get operation names and input/output messages from PortType. Then we can get message types from Message. Lastly, we can get the definition of all types in the **schema** section of Types.

However, most types used in WSDL are the data types defined in XML schema. Therefore, we define some transformation rule to build mappings between WSDL data types and MOSP data types. Thus, we can transform a WSDL document into an MIDL document.

In `onCall()`, Gateway service firstly gets the operation name the client calls from the received MOSP CALL message header, and determines whether if this operation is provided by the Web Service. We can get the binding protocol Web Service uses from **Service**, **Port** and **Binding** of the WSDL. As we have mentioned, Web Service standards are open, and the binding in most common use is SOAP binding. Gateway service supports SOAP binding. While facing Web Service using other bindings, such as MIME binding, the Gateway service may not be available to it, since Web Service using MIME binding is transferred via e-mail and is asynchronous. Asynchronous services require the timeout mechanism of each service object access, which we do not define in our system.

We can get the binding protocol used by Web Service from the Binding part of WSDL. If SOAP binding is used, Gateway service will get the input arguments from the received MOSP message. The users will inputs the arguments according to the MOSP type defined in MIDL in Java data types, and those types will be marshaled to MOSP message to transfer. MOSP message will represent the argument type and value in plaintext, such as `mt:/val/num` and `123`.

Later, after receiving MOSP input message, Gateway service will decompose those arguments according to the argument structure in MIDL, and package it in the argument structure in WSDL in a SOAP request message to transfer to Web Service. Since SOAP message does not include the information of argument data types, marshaling is not used in this step. Besides, SOAP message also transfer argument value in plaintext, thus marshaling is not used here.

When Web Service gets the SOAP request message, it will return a SOAP response message, which may contain SOAP Fault while there are exceptions. Gateway will transfer SOAP fault to sting and return it to MOSP clients. In the common situation when the SOAP message is accepted successfully and the execution works correctly, Web Service will return the execution result packaging in SOAP response message, which will have the same structure as the return argument in WSDL. Therefore, Gateway will decompose the SOAP response message according to the structure shown in WSDL, marshal the argument values to Java type, and then marshal it to MOSP type to package it as MOSP return message for MOSP clients.

When MOSP message is returned to the client, it will be marshaled to MOSP data type object represented in Java binding format. Thus, clients can decompose the return object according to the argument structure in MIDL.

We can see that marshaling is used many times during transformation process. We can define a **Marshaller** responsible for doing this. Firstly, as Table 1-2 shows, Marshaller is responsible for mapping MOSP types and Java types. Then, we need the mapping among XML types, Java types and MOSP types. With those mappings, we can transform the Web Service data types into MOSP types represented as Java types by the Marshaller. Moreover, since MOSP messages and SOAP messages both represent argument values in plaintext, the Marshaller needs to have the ability to transform Java objects into string, and vice versa. Since Marshaller has defined the mapping rule between Java types and MOSP types, once getting the Java objects, we can directly transform them to MOSP data types with the Marshaller.



Chapter 4 System Implementation

The proposed system is developed with Java JDK 1.6, Java EE SDK [11], WSDL4J [31], Castor [3] and JDOM [12]. The Web Service server is build with Sun official application server GlassFish V2 [8].

4.1 MOSP Server, Service and Client

In MOSP environment, we named the remote objects accessed through MOSP URL **MeshObject**. MeshObject is separated into three kinds, which are Creator (similar to Java class), Instance (similar to Java object instance) and Typedef (similar to Java interface) and represented by three subclasses of MeshObject: **Creator**, **Instance** and **Typedef**. We can use MeshObject.bind() with MOSP URL as the input argument to bind to a MOSP object and get the object instance, the object of one of these subclasses, as Figure 4-3 shows.

We have introduced MOSP server and client work model in Section 3.1.1. Later, we describe it by the sketch codes of MOSP server, MOSP service and MOSP client in Figure 4-1, Figure 4-2 and Figure 4-3 respectively.

In Figure 4-1, we firstly create a service root object and a service object, and then set the service to the service root by assigning a path to it. Therefore, we can generate a MOSP peer which contains the service root object we created. After generating the peer, we can use `start()` and `openListener(":9000")` to enable other peers connect to it through its MOSP URL and port (the port is 9000 here). Also, MOSP clients can bind to the remote object instance of the service this server provides by the MOSP URL of the server plus the path to the service, as Figure 4-3 shows. Finally, we can use `shutdown()` to close the MOSP server peer.

```
FooServiceRoot root = new FooServiceRoot();
FooService service = new FooService();
root.setService("/Service", service);
Peer server = new Peer(root);
try {
    server.start();
    server.openListener(":9000");
} finally {
    peer.shutdown();
}
```

Figure 4-1: MOSP Server

```

FooService extends MetaService {

    @Override
    public void onDesc(InMsg req, OutMsg resp) {
        String midl = "...";
        OutArg arg = new OutArg();
        arg.setType(MOSP.HEADER_NAME);
        arg.setData(midl.getBytes());
        resp.appendArg(arg);
        resp.setCode(RespCode.OK);
        resp.setDescription(RespCode.OK_INFO);
    }

    @Override
    public void onCall(InMsg req, OutMsg resp) {
        String operationName = req.getHeader(MOSP.HEADER_NAME);
        /* do something ... */
        Object result = ...; // execution result
        OutArg arg = new OutArg();
        arg.setType(/* set arg type from result */);
        arg.setData(/* set arg type from result */);
        resp.appendArg(arg);
        resp.setCode(RespCode.OK);
        resp.setDescription(RespCode.OK_INFO);
    }
}

```

Figure 4-2: MOSP Service

```

Peer client = new Peer();
try {
    client.start();
    Creator creator = Creator.bind("mosp://140.112.107.72:9000/Service");
    MidlDoc midl = creator.getMidl();
    Object returnResult = creator.call("operation1", "arg1", "arg2");
} finally {
    peer.shutdown();
}

```

Figure 4-3: MOSP Client

Figure 4-2 shows how `onDesc()` and `onCall()`, which we have mentioned in Section 2.2.4 and 3.1.3, work in Java code. These two methods both have the arguments **InMsg** and **OutMsg**. **InMsg** represents MOSP messages received from MOSP clients, and **OutMsg** represents MOSP messages response to MOSP clients.

`onDesc()` is triggered by MOSP clients calling `getMidl()` method, which we mentioned in Section 3.1.1, as Figure 4-3 shows. The actions it performs is marshaling the MIDL content in string format to **OutArg** (the arguments in MOSP message: **OutArg** for **OutMsg**, and **InArg** for **InMsg**), setting the standard response code we mentioned in Section 2.2.4, and appending them to the return MOSP message **OutMsg**.

`onCall()` is triggered by MOSP clients calling `call()` method, which we also mentioned in Section 3.1.1, with operation name and input arguments as input parameters as Figure 4-3 shows. The actions it performs is decomposing the request MOSP message `InMsg` to get the operation name and input arguments which MOSP client requests, executing the requested operation with the input arguments, getting the execution result and marshaling it to `OutArg`, setting the standard response code and appending `OutArg` and the response code to `OutMsg`.

Now, we can see how the Generic Gateway Service System was built.

4.2 Gateway Factory Service

We first introduce the Gateway Factory service, which we mentioned in Figure 3-3. Gateway Factory service provides an operation `createGateway()`, which is responsible to create a Gateway Service and assign it to the service root of Gateway Factory server. Thus there is no need to enable each client to get an object instance of Gateway Factory service.

Figure 4-4 shows the MIDL document of Gateway Factory Service. `midl role="creator"` represents that this service is a Creator object (similar to Java Class). Then we use `static="1"` to set the operation to static, thus clients can directly call the static operations without creating an object instance first.

From this MIDL document, we can see the this service provide an operation `createGateway()` which gets a string of WSDL URL as the input and returns a MOSP Instance (similar to Java object instance). Each MOSP Instance represents a Gateway service instance transformed from Web Service. Since Web Service does not record user states, we directly transform it into MOSP Instance but not Creator, thus users can not create different object instances from the Gateway service they get, which achieves the stateless characteristic of Web Service.

```
<midl role="creator" xmlns="mt:/val/xml/midl" >
  <op name="createGateway" type="mt:/ref/mosp/instance" static="1" >
    <arg type="mt:/val/str" name="wsdlURL" />
  </op>
</midl>
```

Figure 4-4: MIDL document of Gateway Factory Service

Now we introduce `createGateway()` more specifically. In Section 3.1.2, we have introduced that it reads a WSDL document from the input WSDL URL, creates a Gateway service relative to the Web Service referred to by the WSDL, adds it to the service root of the Gateway Factory server, and returns it to clients.

As Figure 3-3 shows, we can set the Gateway service object to the Gateway Factory service root with a path to it. The Gateway service is created with the WSDL URL as input parameter. The path to each service needs to be unique, so we hash the WSDL URL into a string to achieve this. Since hash value may overlap, we save the paths into a path list, and check if the path already exists in the list whenever generating it. If so, we simply append a character to it until it becomes unique. However, Web Services do not differentiate from its WSDL URL but from the target namespace and portType name attributes in the WSDL document, since different WSDL URLs may reference to the same Web Service. Thus we will put the mapping of Web Service name (the target namespace plus portType name) and the path to its Gateway into a Web Service list. Whenever get a new WSDL URL, we check if the Web Service referred to by this WSDL already exists in the Web Service list. If so, we update the Gateway service in service root with the new WSDL URL. At last, we return the Gateway object. We list the algorithm of `createGateway()` in Table 4-1.

```

Get input MOSP message InMsg req;
String WSDL_URL = req.getArg(0);
GatewayService gateway = new GatewayService(WSDL_URL);
Get target_namespace and portType_name from the WSDL in WSDL_URL;

Object key = to_key(target_namespace + portType_name);
if ( WebService_Map.contains(key) ) {
    String path_to_service = WebService_Map.get(key)
    serviceRoot.setService(path_to_service, gateway);
}
else {
    String path_to_service = WSDL_URL.hash();
    while ( path_List.contains(path_to_service) )
        path_to_service += "1";
    path_List.add(path_to_service);
    WebService_Map.put(key, path_to_service);
    serviceRoot.setService(path_to_service, gateway);
}
return path_to_service; // use path to represent MOSP Instance object

```

Table 4-1: The algorithm of `createGateway()`

4.3 Gateway Service: onDesc()

The Gateway Service `onDesc()` is responsible for transforming the MIDL document from WSDL document, and returning it to MOSP clients. We choose to create the MIDL document in Gateway Service constructor, with a WSDL URL string as the input argument. It is because the MIDL document is generated from WSDL URL. If the constructor contains the WSDL URL string as input argument, we can set up the MIDL document for later use once creating the Gateway Service.

To parse a WSDL document, we build a Java class which imports **WSDL4J**, **JDOM** and **Castor** project. First of all, with WSDL4J, we can use the method `readWSDL(String WSDL_URL)` from class `javax.wsdl.xml.WSDLReader` to read a WSDL document and parse it to a `javax.wsdl.Definition` object. It is named “Definition” because a WSDL document is XML based, which ordinarily with a root element named “**definition**”, as the sample WSDL document shown in Figure 4-5. The definition element has child elements, with the same architecture as shown in Figure 2-8, which are **types**, **message**, **portType**, **binding** and **service** as Figure 4-5 shows. With the Definition object, we can easily get those child elements as Java objects by method call.

As we mentioned in Section 3.1.3, to transform WSDL into MIDL, we need to focus on the Service Interface Definition part, the **types** and **message** and **portType** elements shown in Figure 4-5. Figure 4-6 shows these three parts we abstracted from the WSDL document in Figure 4-5. First we can get the operations provided from portType, and each operation contains an input and an output message, each of which relates to one of the message elements by its name. Each message contains a message **part**, which relates to an “**element**” element in the child element **schema** of **types** by **element** name. Each element relates its type to a **complexType** by the complexType name. Furthermore, each complexType contains one or more **elements**, and each of which contains a type relating to another complexType or a defined type, such as XML string type shown in the first complexType in Figure 4-6 (`xs:string` represents the string type in namespace `xs`, which is the XML schema defined in the definition element of this WSDL document). Note that the name of each xml element differentiates from its local name and namespace, for instance, `tns` represents the namespace of the WSDL document itself, such that `tns:getCountry` refers to the complexType `getCountry` defined by this WSDL document. All these elements will be parsed to Java objects, which can be abstracted from the **Definition** object.

With the above steps, we can read the WSDL document in Figure 4-6 as follows. The portType element says that there is one service named `getCountryService`, which provides an operation named `getCountry`, with input message `getCountry` and output message `getCountryResponse`. The input message `getCountry` relates to the complexType `getCountry`, which contains an XML string array named `city` (`maxOccurs` represents the max occurrence times of this element, `unbounded` means the occurrence time is unbounded. We can see the elements whose `maxOccurs` is more than 1 or unbounded as an array); the output message `getCountryResponse` relates to complexType `getCountryResponse`, which contains the complexType `country`. `country` contains an XML string named `name` and another XML string named `president`. It means that the operation `getCountry` input parameter is the string `city`, and the return type is a complexType with two string parameters. We can transform WSDL into MIDL as shown in Figure 4-7.

As we can see in this MIDL document, it provides a more intuitive way of describing a service. To transform WSDL into MIDL document, we first transform each portType into a MIDL document, and then transform each operation in portType into the operation in the MIDL document. Then we set the return type and input arguments of each operation from the WSDL input/output message and types.

In WSDL Types element, there are some redundant complexType in a WSDL document, and we can trim it with the algorithm shown in Table 4-2 when transforming WSDL Types into MOSP types.


```

// On WSDL input message:
Get input message element in_msg_elem;
MOSPType[] in_mosp_types;
if ( in_msg_elem.getType().isComplexType() ) {
    element[] elems = in_msg_elem.getType().getSubElements();
    if ( no elem in elems is ComplexType ) {
        int i = 0;
        for each ( elem in elems ) {           // expends in_msg_elem
            in_mosp_types[i] = to_mosp_type(elem);
            ++i;
        }
    }
}
else {
    in_mosp_types[0] = to_mosp_type(in_msg_elem);
}

// On WSDL output message:
Get output message element type out_msg_elem;
MOSPType out_mosp_type;
if ( out_msg_elem.getType().isComplexType() ) {
    element[] elems = out_msg_elem.getType().getSubElements();
    if ( elems.size()==1 AND !elems[0].isComplexType() ) {
        out_mosp_type = to_mosp_type(elem); // expends out_msg_elem
    }
}
else {
    out_mosp_type = to_mosp_type(out_msg_elem);
}

```

Table 4-2: The WSDL input/output message type to MOSP type

```

MOSPType to_mosp_type( element element ) {
    MOSPType mospType;
    mospType.name = element.name;
    // XML primitive data types to MOSP primitive data types
    if ( element.getType().isPrimitiveType() ) {
        mospType = to_mosp_prim_type(element.getType());
        // The transformation rule is listed in Table 4-4 and Table 4-5
    }
    // XML complexTypes to MOSP struct types
    else if ( element.getType().isComplexType() ) {
        mospType = createMOSPStruct(element.getType());
    }
    // XML array types to MOSP array types
    if ( element.maxOccurs > 1 OR element.maxOccurs.isUnbounded() ) {
        mospType.toArrayType(); // append #[] sign to the end
    }
    return mospType;
}

MOSPStruct createMOSPStruct( complexType complexType ) {
    MOSPStruct MOSP_st;
    // MOSPStruct extends MOSPType and contains primitive and struct type
    // arguments in its argument List
    MOSP_st.type = complexType.name;
    element[] elems = complexType.getSubElements();
    for each ( elem in elems ) {
        if ( elem.getType().isPrimitiveType() ) {
            MOSPPrim arg;
            arg.name = elem.name;
            arg.type = to_mosp_prim_type(elem.getType());
            MOSP_st.argList.add(arg);
        }
        else if ( elem.isComplexType() )
            MOSP_st.name = elem.name;
            MOSP_st.argList.add(createMOSPStruct(elem.getType()));
    }
    return MOSP_st;
}

```

Table 4-3: The type transformation algorithm

We have described the meaning of each element in MIDL in Section 2.2.3. Some data type transformation rules need to be made during the transformation. First is the transformation between XML primitive data types and MOSP primitive data types. Then is the transformation between XML array types and MOSP array types. Last is the transformation between XML complexTypes and MOSP struct types. We list the type transformation algorithm in Table 4-3.

We define an **OperationInfo** class to record each operation name, and its return type and input arguments while parsing the Definition object of WSDL. The input arguments and return type is a list with **MOSPArg** object or **MOSPStruct** object. We define **MOSPArg** and **MOSPStruct** classes to record the argument types, which are primitive type and struct type respectively. **MOSPStruct** may contain one or more **MOSPArg** or **MOSPStruct** objects, just as complexType which may contain one or more primitive type arguments or complexType arguments. This transformation is listed in the second part of Table 4-3.

To get the operation information we mentioned above, as we know, we need to get the portType information first. With **Definition.getPortTypes()**, we can get the **PortType** objects; with **PortType.getOperations()**, we can get **Operation** objects. With **Operation** object, we can finally get the information such as operation name and input/output message.

To transform input/output message into arguments, we need to parse **Message** and **Types** object, from **Definition.getMessages()** and **Definition.getTypes()** respectively. **Types** object only contains a **DOM** object of the **schema** element, thus we use **JDOM** to transform this **DOM** object into **JDOM** object, and then use **Castor** project to transform **JDOM** object into **Schema** object, which contains objects such as **Element** and **ComplexType**. With **Schema** object, we can thus get argument information from method calls. With the information we need, we can transform WSDL data types into MIDL data types with the algorithm shown in Table 4-3.

Now, we can finally transform WSDL document into MIDL document. However, there exist some limitations. For example, **element** may contain some attributes such as **minOccurs** (**minOccurs** represents the minimum occurrence time of this element) or **nillable** (**nillable** is a boolean value, representing if this element can be set to null), which we can not transform into MIDL since MIDL does not contain these properties.

After generating the MIDL document, Gateway service will directly marshal it to MOSP response message transferred to MOSP clients.

```

<definitions xmlns= "http://schemas.xmlsoap.org/wsdl/"
  xmlns:xs= "http://www.w3.org/2001/XMLSchema" ...
  xmlns:tns= "http://service/" targetNamespace="http://service/">
  {
  <types>
    {
    <schema>
      ...
    </schema>
    }
  </types>
  {
  <message name="...">
    <part name="parameters" element="..." />
  </message>
  {
  <portType name="getCountry">
    <operation name="...">
      <input message="..." />
      <output message="..." />
    </operation>
  </portType>
  {
  <binding name="..." type="...">
    <soap:binding transport="..." style="document" />
    <operation name="...">
      <soap:operation soapAction="..." />
      <input>
        <soap:body use="..." />
      </input>
      <output>
        <soap:body use="..." />
      </output>
    </operation>
  </binding>
  {
  <service name="getCountryService">
    <port name="..." binding="...">
      <soap:address location="..." />
    </port>
  </service>
  }
  </definitions>

```



Figure 4-5: A sample of WSDL document

```

<types>
  <schema>
    <element name="getCountry" type="tns:getCountry" />
    <element name="getCountryResponse" type="tns:getCountryResponse" />
    <complexType name="getCountry">
      <sequence>
        <element name="city" type="xs:string" minOccurs="0"
          maxOccurs="unbounded" />
      </sequence>
    </complexType>
    <complexType name="getCountryResponse">
      <sequence>
        <element name="return" type="tns:country" minOccurs="0" />
      </sequence>
    </complexType>
    <complexType name="country">
      <sequence>
        <element name="name" type="xs:string" minOccurs="0" />
        <element name="president" type="xs:string" minOccurs="0" />
      </sequence>
    </complexType>
  </schema>
</types>

<message name="getCountry">
  <part name="parameters" element="tns:getCountry" />
</message>

<message name="getCountryResponse">
  <part name="parameters" element="tns:getCountryResponse" />
</message>

<portType name="getCountryService">
  <operation name="getCountry">
    <input message="tns:getCountry" />
    <output message="tns:getCountryResponse" />
  </operation>
</portType>

```

Figure 4-6: The description part of a WSDL document

```

<midl role="instance" xmlns="mt:/val/xml/midl">
  <op name="getCountry" type="#country">
    <arg type="mt:/val/str#[]" name="city"/>
  </op>
  <st name="country">
    <arg type="mt:/val/str" name="name"/>
    <arg type="mt:/val/str" name="president"/>
  </st>
</midl>

```

Figure 4-7: An MIDL document transformed from WSDL document

XML Data Type	MOSP Data Type
anySimpleType	mt:/val
duration	mt:/val/str/time/duration
dateTime	mt:/val/str/time/dateTime
time	mt:/val/str/time/hours
date	mt:/val/str/time/date
gYearMonth	mt:/val/str/time/gYearMonth
gYear	mt:/val/str/time/gYear
gMonthDay	mt:/val/str/time/gMonthDay
gDay	mt:/val/str/time/gDay
gMonth	mt:/val/str/time/gMonth
String	mt:/val/str
normalizedString	mt:/val/str/normalizedString
token	mt:/val/str/normalizedString/token
language	mt:/val/str/normalizedString/token/language
Name	mt:/val/str/normalizedString/token/Name
NCName	mt:/val/str/normalizedString/token/Name/NCName
ID	mt:/val/str/normalizedString/token/Name/NCName/ID
IDREF	mt:/val/str/normalizedString/token/Name/NCName/IDREF
IDREFS	mt:/val/str/normalizedString/token/Name/NCName/IDREF/IDREFS
ENTITY	mt:/val/str/normalizedString/token/Name/NCName/ENTITY
ENTITIES	mt:/val/str/normalizedString/token/Name/NCName/ENTITY/ENTITIES
boolean	mt:/val/bool
base64Binary	mt:/val
baseBinary	mt:/val

Table 4-4: Mappings between XML and MOSP data types – 1

XML	MOSP
float	mt:/val/num/float
decimal	mt:/val/num
integer	mt:/val/num/int
nonPositiveInteger	mt:/val/num/int/nonPositiveInteger
negativeInteger	mt:/val/num/int/nonPositiveInteger/negativeInteger
long	mt:/val/num/long
int	mt:/val/num/int
short	mt:/val/num/short
byte	mt:/val/num/byte
nonNegativeInteger	mt:/val/num/int/nonNegativeInteger
unsignedLong	mt:/val/num/long/unsignedLong
unsignedInt	mt:/val/num/int/nonNegativeInteger/unsignedInt
unsignedShort	mt:/val/num/short/unsignedShort
unsignedByte	mt:/val/num/byte/unsignedByte
positiveInteger	mt:/val/num/int/nonNegativeInteger/positiveInteger
double	mt:/val/num/double
anyURI	mt:/ref
Qname	mt:/val/str/Qname
NOTATION	mt:/val/str/NOTATION

Table 4-5: Mappings between XML and MOSP data types – 2

When deciding the mappings between XML and MOSP data types, we face a dilemma of getting more precise types or clarifying the inheritance relationship between types. For instance, xml **unsignedLong** type is actually derived from **unsignedInteger** and contains value space from 0 to 18,446,744,073,709,551,615, while **long** value space is - 9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. If we let **unsignedLong** inherits from **long** in MOSP data type hierarchy, its value space will limit from 0 to 9223372036854775807, which is half of the original value space. After consideration, we think the value space is already sufficient for users. Clearer inheritance relationship is more important for MOSP users.

4.4 Gateway Service: onCall()

The Gateway Service `onCall()` is responsible for creating the SOAP request message from MOSP CALL message, passing it to Web Service, getting the SOAP response message from Web Service and then transforming it into MOSP CALL response message for clients.

From MOSP CALL message, we can get the name and the input argument value of the Web Service operation which is called. With the same structure as the structure of arguments in WSDL and the input argument value, we can generate a SOAP request message. Figure 4-8 shows a sample SOAP request message of the Web Service which the WSDL document in Figure 4-6 describes. The SOAP Envelope wrapped up SOAP Header and SOAP Body. SOAP Body will contain a child element whose name is the same as the operation name. Then this element will contain child elements, which are also the input arguments of the operation. Each of those elements contains text content (such as the text **Taipei** in SOAP message), which represents the argument value of each argument. The element name is the same as the element name of each primitive or complexType type. Figure 4-9 shows a sample SOAP response message of the same Web Service. It is similar to SOAP request message, while name of the child element of its SOAP Body is the operation name plus the string "Response". Note that we only have to repeat the element in SOAP message to represent an array type. Since SOAP message use text to represent the argument value, the original argument types will be transformed into String type.

```
<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:getCountry xmlns:ns2="http://service/">
      <city> Taipei </city>
      <city> New York </city>
      <city> Tokyo </city>
    </ns2:getCountry>
  </S:Body>
</S:Envelope>
```

Figure 4-8: A sample SOAP request message


```

<?xml version="1.0" encoding="UTF-8"?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
  <S:Header/>
  <S:Body>
    <ns2:getCountryResponse xmlns:ns2="http://service/">
      <ns2:return>
        <name> USA </name>
        <president> George W. Bush </president>
      </ns2:return>
    </ns2:getCountryResponse>
  </S:Body>
</S:Envelope>

```

Figure 4-9: A sample SOAP response message

The MOSP CALL request message is marshaled to **InMsg** class in Java. We can use **InMsg.getArgs()** to get the arguments **InArg** in MOSP message. InArg may be Struct (**InStruct** object), Array (**InArg[]** object) or String type, depending on the original role each argument plays in MIDL. SOAP request messages are composed from the content parsed from **InArg**.

After that, we use the WSDL URL and the SOAP request message as the input parameter to call a Web Service using SAAJ (SOAP with Attachments API for Java), and then get a SOAP response message. From the SOAP response message, we can use the same principle, which we used to deal with InArg and SOAP request message, to decompose the SOAP response message and marshal the argument values to **OutArg**. Then set **OutArg** to **OutMsg** to return to clients.

However, the argument data types are no longer String types when we marshal it to OutArg. We have to transform those arguments into the original data types defined in WSDL document from String. This transformation rule is listed in Table 4-6 and Table 4-7.

XML Data Type	Java Data Type
anySimpleType	java.lang.Object.
duration	javax.xml.datatype.Duration
dateTime	java.util.Calendar
time	java.util.Calendar
date	java.util.Calendar
gYearMonth	javax.xml.datatype.XMLGregorianCalendar
gYear	javax.xml.datatype.XMLGregorianCalendar
gMonthDay	javax.xml.datatype.XMLGregorianCalendar

Table 4-6: Mappings between XML and Java data type – 1

XML Data Type	Java Data Type
gDay	javax.xml.datatype.XMLGregorianCalendar
gMonth	javax.xml.datatype.XMLGregorianCalendar
String	java.lang.String
normalizedString	java.lang.String
token	java.lang.String
language	java.lang.String
Name	java.lang.String
NCName	java.lang.String
ID	java.lang.String
IDREF	java.lang.String
IDREFS	java.util.List
ENTITY	java.lang.String
ENTITIES	java.util.List
boolean	boolean
base64Binary	byte[]
baseBinary	byte[]
float	float
decimal	java.lang.Number
integer	java.lang.Integer
nonPositiveInteger	java.lang.Integer
negativeInteger	java.lang.Integer
long	long
int	int
short	short
byte	byte
nonNegativeInteger	java.lang.Integer
unsignedLong	long
unsignedInt	java.lang.Integer
unsignedShort	short
unsignedByte	byte
positiveInteger	java.lang.Integer
double	double
anyURI	java.net.URI
Qname	javax.xml.namespace.QName
NOTATION	java.lang.String

Table 4-7: Mappings between XML and Java data type – 2

4.5 Performance Test

The operation steps of Generic Gateway Service System are as follows:

- Step 1. Gateway Factory:** MOSP client binds to the MOSP URL of Gateway Factory, and gets a MeshObject instance, through which the client can invoke the operations provided by Gateway Factory Service.
- Step 2. Gateway:** MOSP client invokes `createGateway()`, the operation provided by Gateway Factory Service, which reads in a WSDL URL text-string as the input argument and receives a Gateway Service, the object instance of the MOSP service transformed from Web Service.
- Step 3. Gateway calls Web Service:** MOSP client invokes the operations provided by Gateway service (the same operations as the ones provided by relative Web Service). Gateway translates the requests to Web Service and translates the response from Web Service to MOSP client.

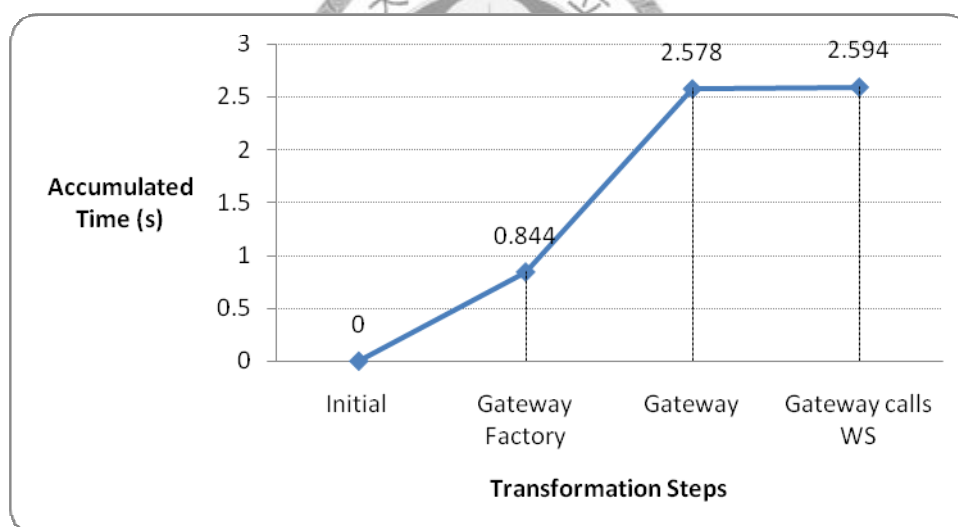


Figure 4-10: The sample Time Period of Generic Gateway Service System

Figure 4-10 shows the time period of each steps we mentioned above. However, when a MOSP client uses the system, the generation of Gateway Factory (Step 1) will only be executed once, and the execution time of Step 1 for different MOSP clients is almost the same; the generation of Gateway (Step 2) will be executed once for a specific Web Service. The time spent for generating Gateway is increased by complexity of the relative Web Service, which we represent with whose WSDL document size. The relationship between the generation time of Gateway and WSDL document size is shown in Figure 4-11 in which we also show the Gateway Factory generation time and the WSDL document download time.

However, the time spent when the Gateway invokes Web Service (Step 3) is much less than which in Step 1 and Step 2. Therefore, the time spent when a MOSP client calls the operations of the same Web Service for several times will be similar to the time spent when the MOSP client only calls the operation for one time. Figure 4-12 shows the time spent when a Gateway calls the same operation of its relative Web Service for 100 times. Obviously, we can observe that the time spent of each operation call is almost the same from this figure. Also, we can see that the total time spent for calling an operation once is similar to which for calling an operation for many times. Thus the performance of the system will be relatively better when clients use it for more times.

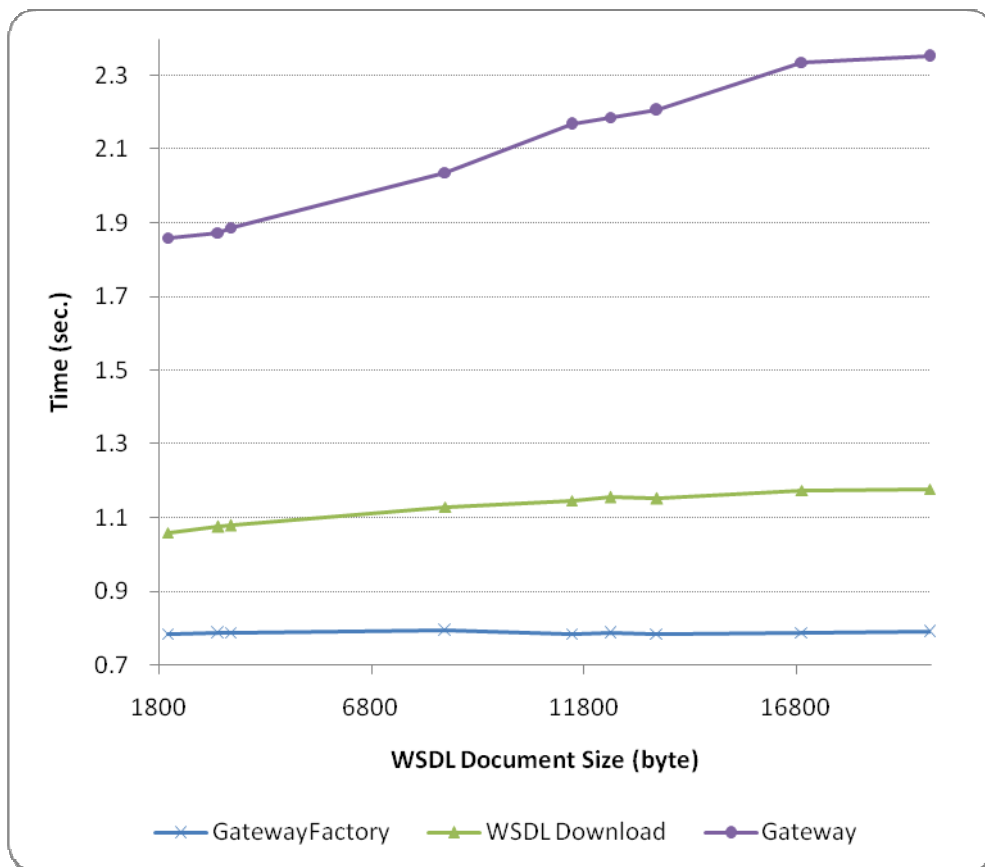


Figure 4-11: The Relationship between WSDL Size and Time Spent in Each Step

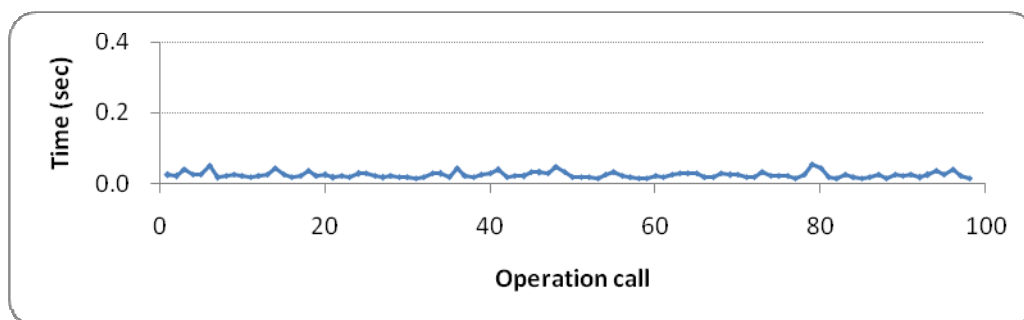
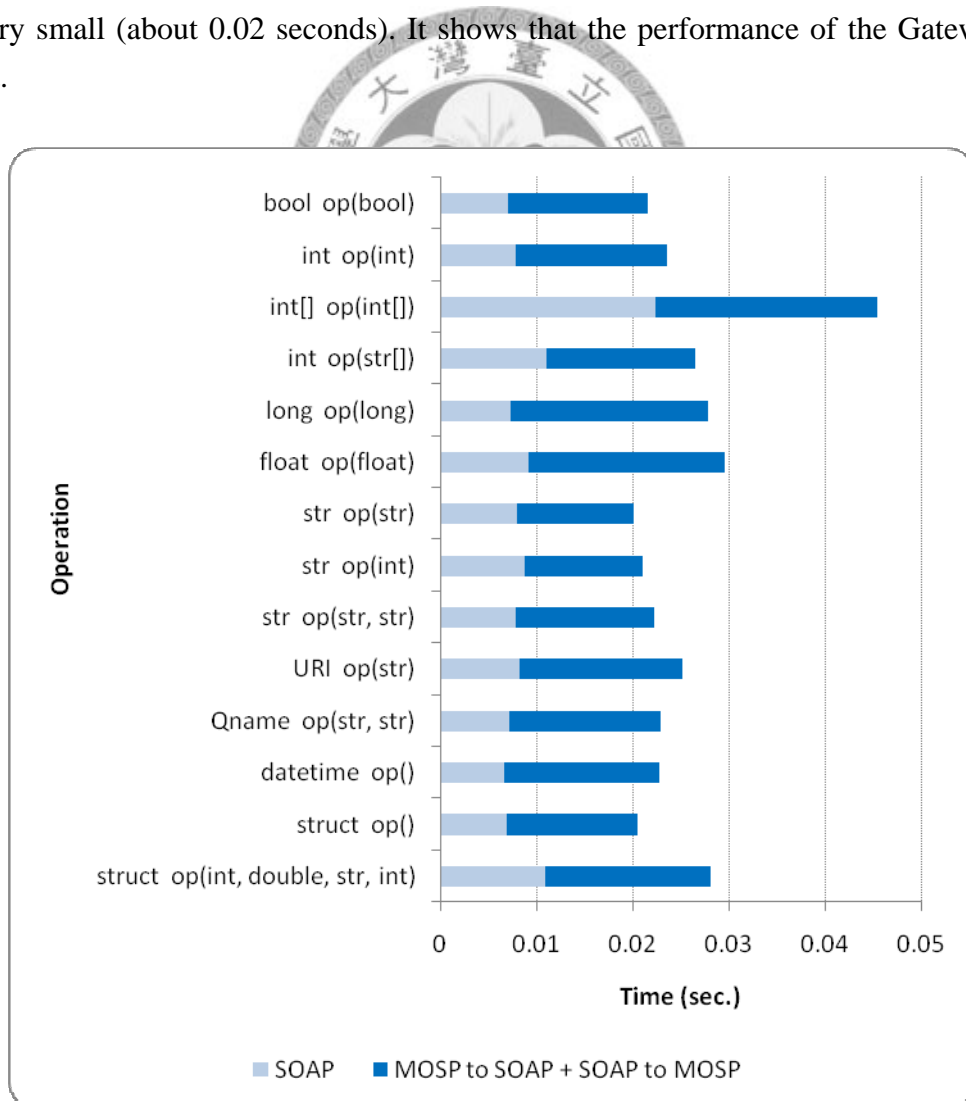


Figure 4-12: The Time spent when MOSP client calls Web Service operation (Step 3)

Figure 4-13 shows the difference between the time spent when a Gateway calls different operations (with different input argument types or return types) of the same Web Service. We separate the time into two parts: one is the time for the Gateway to call the Web Service with a generated SOAP request message and get a SOAP response message; the other is the time for the Gateway to transform the SOAP request message from MOSP input message and transform the SOAP response message into MOSP output message. We can observe that the transformation time is very close to the SOAP message passing time. Also, the transformations between the operations whose have input argument types and return types are primitive. Moreover, the SOAP message passing time and transformation time for array type is more than primitive type. The operation with more input arguments also needs more transformation time. Some types which are not MOSP original types, such as datetime, may need more transformation time. The transformation time of MOSP struct type (transformed from XML complexType) is similar to the transformation time of primitive data type. Clearly, we can also see in this figure that the transformation time is very small (about 0.02 seconds). It shows that the performance of the Gateway is great.



Chapter 5 Conclusion

5.1 Contribution

The growth of Internet technologies has unleashed a way of innovations that change the way people communicate and collaborate. Most people can barely imagine life without networks. The rapid rise of Internet has ushered in a new era. Nowadays, companies are moving their main operations to web for better automation, efficient business processes and global visibility. We need an integrated, robust solution for leveraging the existing applications, rapidly adapt to the unique needs and continually evolve as requirements change over time.

The current trend of such solution is moving away from tightly coupled systems towards systems with loosely coupled, dynamically bound components. **Web Service** is the present evolution of this new category of services. It is an interface describing a collection of operations which are network-accessible through standardized XML messaging. Web Service technology provides a language-neutral, platform-neutral programming model accelerating application integration above the networks.

Although Web Service is very popular and in general use, which solves many problems, there are still some insufficiency. For example, it is a stateless service system, which does not record the state of each client using it and can only provide services with simpler interaction with clients, such as key word search.

A brand-new solution, **MeshObject Service Protocol (MOSP)** provides another choice now. First of all, in MOSP world, each node (also known as **peer**) and each service it provides in the networks can be identified by a unique MOSP URL, which means we replace the original Hypertext Transfer Protocol (HTTP) with MOSP. Moreover, MOSP uses the concept of object-oriented, which enables users to obtain an object instance of the service provided by a peer by binding to its MOSP URL. MOSP can provide stateful services with such way. Besides, MOSP contains the concept of inheritance as well, which enables MOSP services to be reused more freely and easily, and therefore reduces the cost and time to develop applications.

Although MOSP brings so many benefits, Web Service is still the most popular

service system in general use. To promote MOSP service, we need to make the old services, which Web Service provides, still available in MOSP environment. This way can lower the entrance barrier to newcomer. Obviously, it is impractical to rebuild and develop new MOSP services which provide the same service as Web Service does. It is much better to enable MOSP clients to call Web Services. For this purpose, we proposed a **Generic Gateway Service System** which directly transforms Web Services into MOSP services. With the gateway system we proposed, entrance barrier to MOSP can be reduced, and new users are more willing to join the MOSP environment. Furthermore, they can enjoy the profits and convenience MOSP brings and also utilize the old Web Services they need.

The Gateway system we proposed fulfills our requirements by providing a MOSP server (the gateway), which is responsible for transforming a Web Service into a relative MOSP service on reading in a WSDL URL text-string which MOSP clients input, and then managing the interactions between MOSP clients and that relative MOSP service.

This gateway system provides two main functions: one is the **function for description**, to transform Web Service Description Language (WSDL) documents into MOSP Interface Definition Language (MIDL); the other is the **function for call**, to transform Simple Object Access Protocol (SOAP) messages into and from MOSP messages. WSDL and MIDL are used to describe Web Services and MOSP services, respectively. SOAP messages and MOSP messages (MOSP CALL message) are used to transfer the request and response between services and clients. We can clearly see that the main procedure in each function is the transformation, and since Web Service and MOSP use different data type scheme and object model, the transformation between data types and the marshaling of arguments are needed.

We have made some translation rules, which define the mappings between XML and MOSP data types in gateway function for description, since most Web Services use XML scheme data types. Also, the translation rules define the mappings among Java data types, XML data types and MOSP data types, in order to handle the marshaling between arguments in gateway function for call.

Also, we provide a way to abstract WSDL information and construct SOAP messages, which helps other service systems such as RMI or CORBA implement their gateway for calling Web Service.

However, there exist some limitations in the gateway system we proposed. First of all, the Generic Gateway Service System provides only one MOSP server responsible for handling the request for Web Services, which may cause performance

bottleneck and single point of failure problems when MOSP users and the demands for invoking Web Service increase.

Secondly, the gateway system we proposed only works when the Web Service uses SOAP binding. However, since almost all Web Services support SOAP binding, the gateway system is available in most cases. Thirdly, the XML data type information, such as the value spaces or the original relationship between data types, may be lost during transformations. Also, this gateway system does not support Web Service using data type schemes other than XML scheme, unless the formers are based on the later. Last is the loss of WSDL information, such as the `minOccurs` or `nullable` attributes of argument elements, which contain ideas that MOSP does not have.

5.2 Future Work

In the following, we highlight several issues and concepts that could be studied further.

5.2.1 Improve limitations

In this research, we proposed a gateway system which enables MOSP clients to call Web Service. However, there exist some limitations which we have mentioned above. We hope to solve those by providing a more complete and full-scale gateway system which support other protocol bindings such as MIME binding and other data types. Also, we hope this gateway system can provide a better transformation rule which can reduce the information loss during transformations or marshaling procedures. Furthermore, we need a better design of gateway system which can solve the possible performance bottleneck and single point of failure problem of the gateway system we proposed.

5.2.2 Interoperability between MOSP and Web Service

To provide a more comprehensive gateway system and to achieve the interoperability between Web Service and MOSP service, we need to enable Web Service clients to call MOSP services. However, it needs lot of works since MOSP services is interdependent while Web Service is distributed and independent.

We discuss the challenges we may face when implementing the complete

interoperability between MOSP and Web Service as follows.

5.2.2.1 OO (stateful) to Non-OO (stateless)

The most different part between MOSP and Web Service is that the former contains the concept of object-oriented and is stateful while the later does not. MOSP use different object instances held by different users to record their state. For example, MOSP server can easily differentiate the states of its client A and client B, such as the name of client A, whereas Web Service can not do this since it is stateless.

Transforming a Web Service into a MOSP Service is more feasible. However, the transformation from a MOSP Service to a Web Service is much more complex and more likely to be impossible to accomplish, since MOSP Services are interdependent while Web Services are distributed and independent. One simple feasible solution is to create a unique parameter for each user to distinguish them, and add it as an argument into all operations to record the states of each user in Web Service. Another more intuitive way is to directly build each MOSP object instance into an independent Web Service. However, all these solutions do not solve the problem caused from trying to transform MOSP Service with the inheritance and dependency characteristics into Web Service which do not contain them.

5.2.2.2 Inheritance to Non-Inheritance

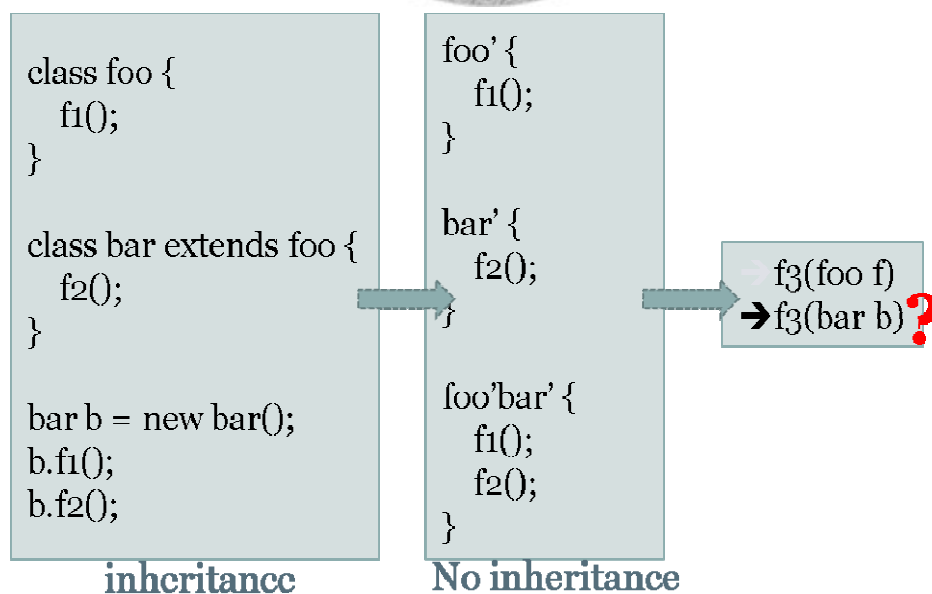


Figure 5-1: Inheritance to No-Inheritance

One of the differences between MOSP and Web Service is the concept of inheritance. To transform the system with inheritance to a system with no inheritance is also a difficult problem. Take Figure 5-1 for example, there is a class **foo** and a class **bar** inheriting from **foo**. Then we can get that **bar** will include all operations in **foo**, and **bar** is seen as **foo** class from the concept of polymorphism. To convey such idea with no inheritance way, we may need to create **foo'** and **bar'** which represent **foo** and **bar** (not include the operations inherited from **foo**) respectively, and **foo'bar'** which contains all operations in **foo'** and **bar'** classes. Thus **foo'bar'** may be able to express **bar** inheriting from **foo** which contains both operations of **foo** and **bar**. However, things are not so easy. Since we can not see **foo'bar'** as a **foo** class, which also means that we lose the characteristic of polymorphism, we can not use **foo'bar'** as a **foo** input parameter while doing operation calls. Consequently, the transformation of the concept of inheritance is also a huge challenge.



Bibliography

- [1] Amazon.com. Amazon Web Services
<http://www.amazon.com/gp/browse.html?node=3435361>.
- [2] Aniruddha Gokhale, Bharat Kumar and Arnaud Sahuguet. *Reinventing the Wheel? CORBA vs. Web Services*.
<http://www.2002.org/CDROM/alternate/395>.
- [3] Castor. Open Source Data Binding Framework for Java.
<http://www.castor.org/>.
- [4] Common Object Request Broker Architecture (CORBA)
<http://www.corba.org/>.
- [5] Dynamic Invocation Interface (DII)
<http://users.skynet.be/pascalbotte/rcx-ws-doc/dii.htm>.
- [6] Fan Yu, Yang Fang, Liu Bixin, Zhou Bin. Research and implementation of a SOAP-CORBA gateway system [J]. *Computer Engineering and Applications*, 2004, 40 (29): 97-100.
- [7] Feng Mingzheng. Integration of Web Services and CORBA. *Journal of Southeast University (National Science Edition)*. 2005-04-04.
- [8] GlassFish. Open Source Application Server.
<https://glassfish.dev.java.net/>.
- [9] Gu Haiqun, Gu Qingfan, Wu Jieyi, Li Yu. Design and realization of CORBA calls encapsulated by Web Services [J]. *Computer Engineering*, 2005, 31 (1): 114-116.
- [10] Irmen de Jong. Web Services/SOAP and CORBA. April 27, 2002
http://www.xs4all.nl/~irmen/comp/CORBA_vs_SOAP.html.
- [11] Java Platform, Enterprise Edition (Java EE)
<http://java.sun.com/javae/>.
- [12] JDOM. Provides a complete, Java-based solution for accessing, manipulating, and outputting XML data from Java code.
<http://www.jdom.org/>.

- [13] Mapping CORBA and SOAP.
<http://dsrg.mff.cuni.cz/seminars/2000-10-31-kalibera-corba-soap/referat.html>.
- [14] Microsoft .NET <http://www.microsoft.com/taiwan/net>.
- [15] Microsoft SOAP Toolkit 3.0
<http://www.microsoft.com/downloads/details.aspx?FamilyID=c943c0dd-ceec-4088-9753-86f052ec8450&DisplayLang=en>.
- [16] OASIS. Organization for the Advancement of Structured Information Standards. <http://www.oasis-open.org/home/index.php>.
- [17] OASIS. UDDI Version 3.0.2 http://www.uddi.org/pubs/uddi_v3.htm.
- [18] Object Management Group, Inc. Formal/04-04-01 WSDL-SOAP to CORBA interworking, Version 1, 0 [S]. U.S.A., 2004.
- [19] Object Management Group, Inc. Formal/05-02-01 CORBA to WSDL/SOAP interworking specification [S]. U.S.A., 2005.
- [20] Object Management Group, Inc. Simple CORBA Object Access Protocol (SCOAP). orbos/00-09-03, orbos/00-09-04.
- [21] OMG. The Object Management Group. <http://www.omg.org/>.
- [22] Remote Method Invocation (RMI)
<http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp>.
- [23] Rogue Wave Software Inc. XML-CORBA link (XORBA) [EB/OL].
<http://www.roguewave.com/support/legacy/>. 2005-09-23.
- [24] SourceForge. SOAP to CORBA bridge. <http://soap2corba.sourceforge.net/>.
- [25] Sun Microsystems. JAVA.com <http://www.java.com/en/>.
- [26] W3C. Extensible Markup Language (XML) 1.0 (Fourth Edition)
<http://www.w3.org/TR/xml/>.
- [27] W3C. Simple Object Access Protocol (SOAP) 1.2
<http://www.w3.org/TR/soap/>.
- [28] W3C. Web Service Description Language (WSDL) 1.1
<http://www.w3.org/TR/wsdl>.
- [29] W3C. Web Services. <http://www.w3.org/2002/ws/>.

[30] W3C. World Wide Web Consortium. <http://www.w3.org/>.

[31] Web Services Description Language for Ja (WSDL4J)
<http://sourceforge.net/projects/wSDL4j>.

