An Adaptive Communication Mechanism for Heterogeneous Distributed Environments Using XML and Servlets

Vincent Wing-hang CHEUNG

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Department of Computer Science & Engineering

Supervised by:
Prof. Michael R. LYU and Prof. Kam Wing NG

An Adaptive Communication Mechanism for Heterogeneous Distributed Environments Using XML and Servlets

submitted by

Vincent Wing-hang CHEUNG

for the degree of Master of Philosophy

at the Chinese University of Hong Kong

Abstract

Nowadays, distributed systems are becoming more and more popular in the provision of enriched information to the increasingly demanding users. Yet, many communication obstacles hinder the expansion of distributed systems. First, the use of firewalls has become the barricades for many different communication protocols, CORBA IIOP is one good example. Another problem is the lack of a simple and generic method to solve the problems that arise when integrating heterogeneous systems with different communication protocols, such as integrating CORBA systems with DCOM systems. In this thesis, we describe our mechanism of using XML and Java Servlet components to support various communication protocols in distributed systems and solve the two problems mentioned above.

Regarding firewall matters, we use CORBA systems to demonstrate our approach. People are trying to use XML to represent the communication protocols and to transmit the XML messages by HTTP, which is a common communication protocol recognized by most firewalls. SOAP, XML-RPC and XIOP are good examples of this approach. Yet, they have some deficiencies. SOAP and XML-RPC may not be compatible with some traditional systems, such as CORBA systems or DCOM systems; XIOP may require modification of existing components, and does not support complicated mechanisms, such as callbacks in CORBA.

We have developed a mechanism which supports CORBA general calls tunneling through firewalls with HTTP and XML, and does not require modification to the existing components. Then, we have extended our mechanism to support CORBA callbacks, which even XIOP and many firewalls dedicated for CORBA IIOP cannot handle. Moreover, we have developed a schema and implemented a translator for mapping CORBA IDL to XML format. These XML documents can help in creating add-on components in our mechanisms, and help in setting up a standard in the transmission of messages in communication.

In the later part, we further describe how we expand our mechanism to heterogeneous communication protocols. XML has flexible semistructure that can be the communication bridge between different protocols. We try to use XML as a common communication protocol among DCOM and Java RMI.

We demonstrate our mechanism by applying it to the integration of a practical system. We have implemented a scalable mediator-based query system with CORBA and we apply the proposed tunneling method to integrate different components across firewall and perform callbacks. We then demonstrate the expansion to other protocols by integrating our CORBA-based mediator query system with other DCOM and Java RMI objects. We also give overview of some related technologies (e.g. XML and Java Servlets), compare our approach to other similar approaches (e.g. XIOP and SOAP), and evaluate the performance of our mechanism in this thesis.

Acknowledgments

I would like to take this opportunity to express my gratitude to my supervisors, Prof. Michael R. Lyu and Prof. Kam Wing Ng, for their generous guidance and patience given to me in the past two years. Their numerous support and encouragement, as well as their inspiring advice are extremely essential and valuable in my research papers (published in CISST'2000, JCDL'01, IC'2001 and SCI'2001/ISAS'2001) and my thesis.

I am also grateful for the time and valuable suggestions that Prof. Kin Hong Lee and Dr Yiu Sang Moon have given in marking my term papers. Without their effort, I will not be able to strengthen and improve my research projects and papers.

I would also like to show my gratitude to the Department of Computer Science & Engineering, CUHK, for the provision of the best equipment and pleasant office environment required for high quality research.

Finally, special thanks to my fellow colleagues, who have helped me in solving programming and computer problems, enlightened me with new research ideas, and given me encouragement and supports. They have given me a joyful and unforgettable university life.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Nowadays, distributed systems are becoming more and more popular than centralized systems because of their global nature, scalability, openness, heterogeneity and fault-tolerance. [1] A distributed system will have components that are distributed over various computers. These components need to interact with each other for providing access to other's services or requesting services from others. By using distributed systems in the provision of different services in different hosts, we can enhance the system scalability and increase fault-tolerance.

To further enhance the system scalability and fault-tolerance, and also to provide better services to the demanding users, there is a trend of integrating several distributed information systems into a single one. In spite of many benefits of integrating multiple distributed systems, we first have to tackle many challenges in communication among different components and different environments, where the situation is far more complicated than building a distributed system in a single enclosed area.

In this thesis, we focus on two communication problems in system integration. They are:

- The common use of firewall which blocks the integration of information systems;
- The integration of several systems with different communication protocols.

Though currently, there are a number of solutions for these two problems, they have their deficiencies. Our research motivation is to use XML and Servlet technologies to provide better solutions to those problems. We explain these two problems with more details below.

1.1 Firewall Issue in Distributed Systems

With the rapid expansion of the Internet, the use of firewalls is also becoming more and more common nowadays. Firewalls are used in the gateways between the local networks and the public Internet, in order to protect the computers in the internal networks by enforcing some security policies [2]. Their role is to control external access to internal information and services. Using packet filtering by a router in the network layer to enforce certain rules is one of the most common mechanism used by the firewalls. But firewall systems can include elements that operate at layers above the network layer in the application level. Application level gateways for Telnet, File Transfer Protocol (FTP), and Hypertext Transfer Protocol (HTTP) are in common use.

Common firewalls block many less common applications, such as the communication protocols for agents, and also the Internet InterORB Protocol (IIOP) used in Common Object Request Broker Architecture (CORBA) [3]. This is because these common firewalls may not be able to decode the message bodies of those protocols. Using CORBA IIOP as an example: IIOP is the Object Management Group's (OMG) specified network protocol for communication between object request brokers, which employs TCP/IP and can be handled by common firewalls at the network and transport level with packet filters. But at the application level, the message body of IIOP is encoded in Common Data Representation (CDR) and firewalls are unable to decode it. Therefore, firewalls cannot base filtering decisions on IIOP messages [4].

With the blocking of some protocols by firewalls, the scalability of system development and system integration would be limited. There exist specific firewalls dedicated for certain protocols, but they are usually not generic and may have some limitations. Take CORBA IIOP as an example again: There are a number of firewalls for CORBA IIOP, such as IONA Orbix Wonderwall [5] and Visibroker Gatekeeper [6], but they cannot solve all firewall problems. As they are not commonly used, both server and client sides must be using them. They may also be vendor-dependent and proprietary. Finally, some CORBA features, such as callbacks, may not be handled.

Elenko and Reinertsen [7] have suggested a communication perspective for the cooperation between XML and CORBA by employing XML, Servlet and HTTP calls to substitute for CORBA IIOP communications. Applying HTTP calls to transport XML parameter contents can eliminate the complicated firewall issue of IIOP, as application level gateways for HTTP are in common use.

SOAP [8] and XML-RPC [9] are proposed specifications which use XML for distributed system protocol. But they are not compatible to other existing distributed systems, such as CORBA or DCOM. XIOP [10] is a proposed substitute of IIOP by XML data, which is dedicated for CORBA. But it does not propose a mechanism to avoid great modifications to the existing components. Also, it does not have a mechanism to perform callbacks.

Table 1.1 summarizes the pros and cons of the above methods. Our target

Table 1.1: Pros and cons of existing methods for remote method callings across firewalls

Solutions	**Strengths**	**Weaknesses**
CORBA dedicated firewalls	• Fast to handle IIOP • Capable to handle complicated CORBA services	• Not popular • Vendor-dependent • Not able to handle callbacks
SOAP and XML-RPC	• Flexible semistructured XML to represent data • Simple; Complicated services are not required	• standalone protocol; not designed for existing protocols
XIOP	• compatible with CORBA	• modification to existing objects is needed • no mechanism for callback is suggested

is to develop a solution which can cover their weaknesses, i.e., a generic mechanism that can bind to the existing systems, without any modifications to the existing components in the systems.

Consequently, we took CORBA IIOP as our target and developed a simple solution by using HTTP, XML and Java Servlets for tunneling through the firewalls to support the CORBA IIOP calls in a more generic way. We then further enhance our mechanism to allow CORBA callbacks which are not feasible behind many CORBA firewalls. We briefly describe how we can automatically generate the necessary components to support our mechanism, by referring to the design of Interface Definition Language (IDL) for a CORBA system. In general, our approach can be applied to other communication protocols as well.

1.2 Heterogeneous Communication Protocols

Currently, we have many different ways to build distributed systems, and the most prominent middlewares are CORBA [3], DCOM [11] and Java RMI [12]. CORBA is defined by the Object Management Group, which supports heterogeneous and distributed objects. The CORBA objects are using the IIOP communication protocol to interact with each other. The Distributed Component Object Model (DCOM) protocol is an application-level protocol for object-oriented remote procedure calls which is useful for distributed, component-based systems of all types. It is a Microsoft technology. Java RMI is developed by Sun Microsystem, which uses the Java Remote Method Protocol (JRMP) to communicate.

All of them have different architectures and different protocols for communication, hence it is very difficult to integrate systems with different middlewares directly. Integrating them requires some bridging tools. Though there are quite a number of bridging tools developed, they may not be able to solve the problems nicely. Some bridges can only map the CORBA objects to the COM/DCOM objects, or vice versa. They do not support interworking. Some bridges actually can only support interworking between the CORBA objects and COM objects; they don't support the interworking between the CORBA objects and DCOM objects.

With the release of new COM-CORBA interworking specification by OMG, many vendors have developed some better applications for bridging, there are still some area that can be improved. Let us discuss one of the very famous applications, OrbixCOMet 2000 [13], which is developed by IONA Technologies. OrbixCOMet 2000 implements the COM/CORBA Interworking specification by enabling transparent communication between COM/Automation clients and CORBA servers. There is a COMET component located between the CORBA

Table 1.2: Pros and cons of existing methods for communication in heterogeneous environments

Solutions	**Strengths**	**Weaknesses**
OrbixCOMet	• Fast	• Not generic to other protocols
RMI/IIOP	• Fast; no real-time protocol conversion overhead	• Not generic to other protocols • Not reversible to communicate with RMI objects

enclave and COM/DCOM enclave, and it acts as a bridge which provides the mappings and performs translation between CORBA and COM/Automation types.

Though OrbixCOMet is already a good implementation in bridging between CORBA and COM, it allows only a limited number of connections for DCOM, as DCOM is distributed while COM is not. Moreover, the weakest point of OrbixCOMet is that it only supports the communication between CORBA and COM/DCOM, and not other protocols. It is because the COMET component would only convert a binary communication protocol message to another binary communication protocol message. As they use binary messages, protocols other than CORBA and COM/DCOM are not able to read them.

Another example of is RMI/IIOP package [14] of Java RMI, which helps RMI objects to communicate with CORBA objects. With modifying the existing RMI objects with RMI/IIOP package, the communication protocol of those RMI objects would be substituted by IIOP, such that they are communicate with CORBA objects. But the drawback is that those modified RMI objects are no longer be able to invoke other RMI objects, as they have given up the original communication protocol.

Table 1.2 summarizes the pros and cons of the two methods mentioned. Again, our target is too cover the weaknesses of those methods, i.e. to give a generic bridging solution to heterogeneous distributed environments and communication protocols.

Here, we extend the mechanism for tunneling across firewalls, and use XML as the bridging messages between different distributed system environments. Based on the generic CORBA IDL, we design a mapping schema from CORBA IDL to XML. Also, we have developed some rules for mapping other Interface Definition Languages, such as MIDL of DCOM and Java Interface of JavaRMI, to the same XML schema.

By sharing the same schema, different distributed environments can communicate with that ”common language” for remote object method calling. The easily-manipulated and human-readable XML messages are not only limited to the usage of CORBA, DCOM or Java RMI, but can be also applied to other web-based applications, such as Active Server Pages (ASP), Java Server Pages (JSP), etc. It is because all of them can use the same XML method calling schema to invoke those CORBA or DCOM objects. Hence, our approach can provide a generic bridge for communication among different distributed system environments and web applications, without modifying the existing components.

1.3 Translator for Converting Interface Definition to Flexible XML

As we have mentioned, to tackle the firewall and heterogeneous distributed environments problems, we have to make use of passing XML messages with HTTP. With the flexible semi-structured XML, messages of remote object callings can be well-represented. However, we still need a standard for the transmission of messages, otherwise the objects in different enclaves are not able to communicate.

We have designed a schema for mapping CORBA IDL to XML format, and implemented a translator to convert the IDL files and generate the XML documents. By making use of the XML documents that follow an agreed Data Type Definition (DTD), we can have a standard for message transmission. Moreover, these XML documents can help to generate the add-on components automatically. The generation of these source codes can help to reduce the extra programming work for those add-on components as they usually contain many similarities, especially in the part of converting the internal data structures to XML formats.

CORBA has a very generic IDL as it supports a variety of programming languages, such as C++, Java, COBOL, etc. As CORBA IDL is so generic, we use the XML mapping scheme of CORBA IDL as the fundamental, and map other interface definition languages of other distributed environments to the same XML schema. By using the same schema, different distributed system environments can have a "common language" and hence be able to communicate with each other.

1.4 An Implementation of a Scalable Mediator Query System

Nowadays, there is a trend to integrate several information systems to offer richer information. As we have mentioned, the firewall problem, and the heterogeneous distributed environments, are often the obstacles in building or integrating a scalable large system.

We have proposed the solutions for those problems and we would like to demonstrate our work by a mediator-based query system and applying our mechanism onto it, such that it can be scalable across the firewalls and heterogeneous distributed system environments.

We use the mediator architecture to integrate multiple query systems via the Internet. Mediators forward the client queries to the appropriate digital libraries or mediators, and then integrate the returned answers and forward them back to the clients. We use the mediators to make queries across the firewalls by making use of XML and Java Servlets, and also querying across the heterogeneous systems with some components which are programmed in DCOM or Java RMI.

By building this query systems, we can demonstrate the advantages of our approach. Also, we will evaluate our approach by measuring the performance of this system.

1.5 Our Contributions

Briefly speaking, we have the following contributions in our research work:

- We have proposed a generic mechanism to enable distributed objects to communicate across firewalls by using XML and Java Servlets. We use CORBA as an example, but this mechanism is generic and can be applied to other distributed environments.
- We have extended the mechanism to support the callback feature in CORBA, which is not supported by other XML-based protocols nor many CORBA-dedicated firewalls.
- We have proposed a schema for mapping CORBA IDL to XML format.

With this schema, we can automatically generate some add-on components in our mechanism. Also the schema can provide a standard grammer for the transmission messages of method callings.

- We have extended the mechanism to support remote object calling in heterogeneous environment. By mapping different interface definition languages of different distributed environments to the schema we have designed, objects of different distributed environments can have communication.

- We have implemennted a mediator-based query system to demostrate our work. This mediator-based query system has applied our mechanism thus it can make queries to a remote object beyond the firewalls, and have callback feature support. Also, the system can make queries to objects from heterogeneous distributed environments, such as DCOM objects, JavaRMI objects, or even other web applications (JSP, ASP etc).

1.6 Outline of This Thesis

We would explain the contributions described above in details in the coming chapters. First, we have an overview of some related work and technologies in Chapter 2. We describe XML and Java Servlets technologies there, as they are closely related to our approach. Also, we look at SOAP, XML-RPC and XIOP, which are similar appoarches that use XML as a protocol. We will discuss their pros and cons there.

In Chapter 3, we introduce our tunneling mechanism, and how we support the callback feature there. Chapter 4 will cover our schema for mapping CORBA IDL to XML format, and outline how we generate the add-on components. For Chapter 5, we focus on the way we support communication among heterogeneous distributed envrionments.

We demonstrate our mechanism with a mediator-based query system in Chapter 6, and we describe in details the components of that system and how we apply our mechanism to enhance it to be a more scalable system.

In Chapter 7, we evalute the performance of our approach, and also the advantages and disadvantages. We will also address the enhancement on security issue and perfomance issue in our mechanism. We then conclude our work in Chapter 8.

Chapter 2

Related Work and Technologies

In this chapter, we will present an overview of some XML technologies and Java Servlets technologies as they are closely related to our research project. We have used XML and Java Servlets technologies heavily in our research. XML has a flexible structure and strong capability in representing data, hence it plays a very important role in our research project. Java Servlet technology is a popular choice for building interactive Web application, thus we use it to transmit XML messages in the Internet.

There are some protocols which are similar to our approach, such as SOAP, XML-RPC and XIOP. We will also give a brief overview of these technologies in this chapter, and discuss their strengths and weaknesses.

We hope this chapter can help you to understand our research work better.

2.1 Overview of XML Technology

In the age of worldwide information networks, documents must be easily accessible, portable and flexible. The information documents must also be system- and platform-independent. XML possesses these features and offers documents an advantage not found in other document description languages.

Extensible Markup Language (XML) [15, 16, 17] is a new standard adopted by the World Wide Web Consortium (W3C) in 1998, and it is a kind of generalized markup language. Some of the design goals of XML are [18, 19]:

- XML shall be straightforwardly usable over the Internet;
- XML shall be able to store complex data structures.
- XML shall support a wide variety of applications

These goals make XML to be a data exchange and representation standard. Also, XML can be widely used in various kinds of applications, and exchange information among different applications, and also heterogeneous platform.

Our research mainly focuses on using XML and Java Servlet to support various communication protocols. XML is used because it can provide flexible structure description to complex protocol structures and data structures. Also, XML is platform-independent and system-independent that would be very suitable to be used in distributed heterogeneous environment. Moreover, we can foresee that the Internet would be a platform in building large and scalable distributed in future, for which XML can work well. Hence, we use XML heavily in the system implementation of our research work.

In the following sections, we will address the basic syntax of XML, the use of DTD, and how XML represents complex data structures.

2.1.1 XML Basic Syntax

In this part, we overview the syntax of XML data, which is based on the specification of XML1.0 by W3C [15]. We will only cover those standards that will be used in our research project.

```
<news>
    <source>South China Morning Post</source>

    <date>
        <day>15</day>
        <month>4</month>
        <year>2000</year>
    </date>

    <title>Press warning appropriate, says Beijing</title>

    <reporter location="Hong Kong">
        <firstname>Greg</firstname>
        <lastname>Torode</lastname>
    </reporter>

    <content>Beijing yesterday defended remarks made by senior
        SAR-based official Wang Fengchao that local media should
        avoid reporting separatist views.
    </content>

</news>
```

Figure 2.1: An example of XML document

XML is a textual representation of data. The basic component in XML is the element, that is, a piece of text bounded by matching tags. Users can define new tags for their needs, which should appear in pairs with a start tag and an end tag. For example, to describe a piece of news article, we can define a pair of tags `<news>` and `</news>`, and then we can put all of the news contents inside this tag pair. Inside an element we may have text, other elements, or even a mixture of both. Figure 2.1 shows a typical XML document. You can see we have defined new tags like `<date>, <source>`, etc.

XML also allows us to associate attributes with elements. Attributes in

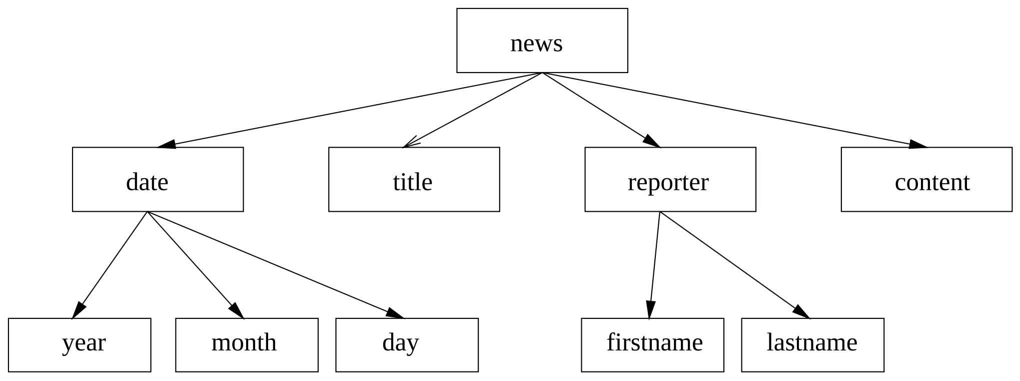


Figure 2.2: The tree hierarchy of the XML document in Figure 2.1

XML are like properties in data models. In XML, attributes are defined as (*name, value*) pairs. With tags, users may define arbitrary attributes, which can enrich the meaning of an element. In the example of Figure 2.1, the tag `<reporter>` has an attribute `location` which indicates the location of that reporter.

There are some differences between tags and attributes. A given attribute may occur only once within a tag, while sub-elements with the same tag may be repeated. Also the value associated with an attribute is a string, while that associated with an element can contain sub-elements.

XML data can always be viewed as a tree structure. For example, in Figure 2.2, the tree hierarchy is the representation of the XML document in Figure 2.1.

2.1.2 DTD: The Grammar Book

We have given an overview of some simple syntax in the previous section. But for most of the time, just following the syntax would not be enough for real-life applications. We usually have to give rules to the XML documents in order to

```
<!DOCTYPE database [
    <!ELEMENT database (news*)>
    <!ELEMENT news (date,title,reporter*,content)>
    <!ELEMENT date year CDATA #REQUIRED
                  month CDATA #REQUIRED
                  day CDATA #REQUIRED>
    <!ELEMENT title (#CDATA)>
    <!ELEMENT reporter (firstname, lastname)>
        <!ATTLIST reporter location (#CDATA)>
    <!ELEMENT firstname (#CDATA)>
    <!ELEMENT lastname (#CDATA)>
    <!ELEMENT content (#PCDATA)>
]>
```

Figure 2.3: The DTD of the XML document in Figure 2.1

regulate them to have specified numbers of specific tags or attributes, and also to have specific structures. To do this, we can use Document Type Definition (DTD) [15]. A DTD serves as a grammar for the underlying XML document, and it is part of the XML language. To some extent, a DTD can also serve as a schema for the data represented by the XML document; hence we are interested in DTD also.

Consider the example in Figure 2.1, it may follow the DTD in Figure 2.3. The meanings of some regular expressions in DTD are shown in Table 2.1. Based on the DTD, we can hence define more documents of a similar schema. Also, different sources can be compromised to use a common schema for their standard.

Table 2.1: Meanings of some regular expressions in DTD

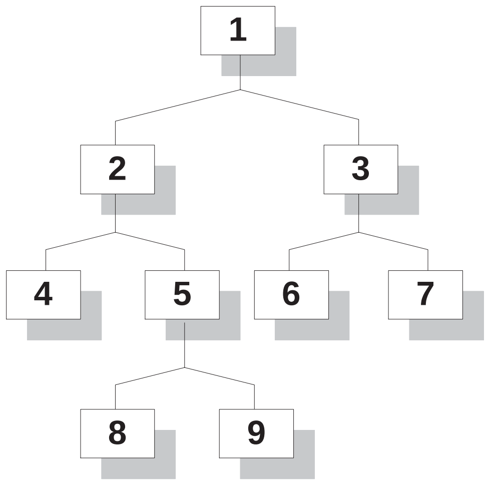
Regular Expressions	Meanings
`test*`	any number of `test` element
`test+`	one or more occurrence
`test?`	zero or one
`test | test'`	alternation
`test , test'`	concatenation

2.1.3 Representing Complex Data Structures

XML plays a very important role in the transmission of HTTP messages. XML has the flexibility in defining new tags on top of its semi-structured feature, so that it can well represent most of the complicated data structures [19]. Even in the case of unlimited-multilevel recursive data structures, such as tree structures, XML can still handle them nicely. Figure 2.4 shows a tree structure and its corresponding XML representation. We can see that the XML data can represent data with complex structures with great flexibility. Hence, we use HTTP to send streams of XML data between the client and server sides to represent the parameters in the remote procedural calls.

By using the DTD of XML data, we can further provide a grammar for the XML data transmission format. Hence we can make a compromise on the interpretation of data transmission of complicated data structure formats for both client and server sides.

Besides the flexibility of data representation, the readability and the ease of manipulation of XML information also provide great flexibility for server as well as client implementation. As long as we follow the DTD of the data transmission format, programmers can have a high degree of freedom to choose different implementation methods.



```
<node> 1
    <node> 2
        <node> 4 </node>
            <node> 5
                <node> 8 </node>
                <node> 9 </node>
            </node>
        </node>
    <node> 3
        <node> 6 </node>
        <node> 7 </node>
    </node>
</node>
```

Figure 2.4: A tree structure and its corresponding XML data describing its structure

2.2 Overview of Java Servlet Technology

Currently, Java Servlet Technology [20, 21, 22] has become a popular choice for building interactive Web applications. In our research project, we also use Java Servlets to support different communication protocols and build the distributed systems upon an Internet-based environment. As Java Servlet plays an important role in our research, we would like to present a brief overview of it before the following chapters.

According to the Java Servlet Specification [20], a Servlet is a web component, managed by a container, that generates dynamic content. Servlets are small, platform-independent and are able to cooperate with web servers. They interact with web clients via a request-response paradigm implemented by the Servlet container. This request-response model is based on the behavior of the Hypertext Transfer Protocol (HTTP). It provides a simple, consistent mechanism to Web developers for extending the functionality of a Web application and for accessing existing business systems.

Servlets provide a component-based, platform-independent method for building Web-based applications, without the performance limitations of Common Gateway Interface (CGI) programs. And unlike proprietary server extension mechanisms (such as the Netscape Server API or Apache modules), Servlets are server-independent and platform-independent. This leaves the programmers free to select a "best of breed" strategy for the servers, platforms, and tools.

When compared to other traditional server extension mechanisms, Servlets have the following advantages:

- They are generally much faster than CGI scripts because a different process model is used.
- They use a standard API that is supported by many web servers.
- They have all the advantages of the Java programming language, including ease of development, portability, performance, reusability, and crash protection
- They can access the large set of APIs available for the Java platform, such as JDBC.
- A Servlet module would be loaded once the first time it is invoked and then it stays loaded until the HTTP server task is shut down or restarted. But a CGI script is loaded every time it is invoked and unloaded when it has finished, hence the performance is worse.

For the mechanism in our research, the Java Servlets modules can actually be substituted by other server extension mechanisms. But as Java Servlets have more advantages when compared to CGI, especially in terms of system- and platform-independence, and memory management, we chose to use Java Servlets in our implementation.

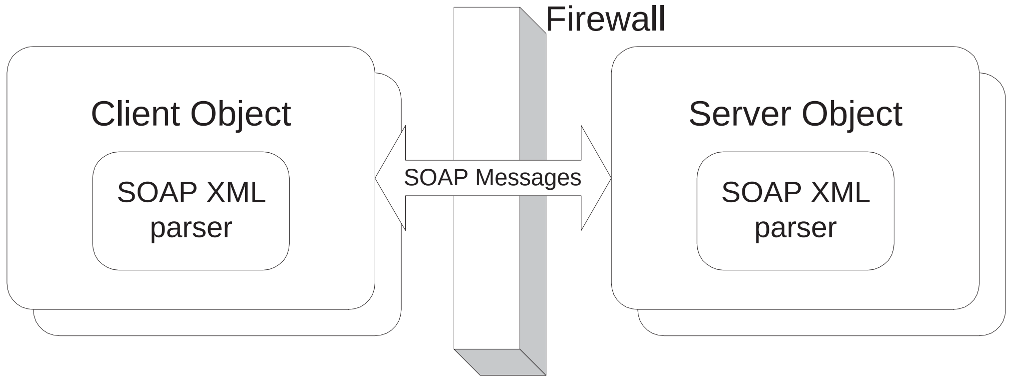


Figure 2.5: Diagram showing the mechanism of SOAP

2.3 Overview of Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) [8, 23] is a lightweight protocol for the exchange of information in a decentralized, distributed environment, which has been accepted by World Wide Web Consortium as a standard. People started discussing XML-based protocol in early 1998, and SOAP specification finally shipped at the end of 1999 by W3C.

It is an XML based protocol that consists of three parts: an envelope that defines a framework for describing what is inside a message and how to process it, a set of encoding rules for expressing instances of application-defined datatypes, and a convention for representing remote procedure calls and responses. Objects need to integrate with some XML-parsers to create messages for making requests or responses. Figure 2.5 shows the mechanism of using SOAP in a distributed system.

Due to XML features, building distributed systems with SOAP can provide many advantages:

- Adaptable to widely distributed networks, as XML is platform- independent and system-independent;
- Able to work through firewalls with use of HTTP in transmission of XML;
- Flexible in implementation of different components. Components can be developed in Perl, Java, PHP, ASP etc.

But SOAP still has some deficiencies when compared to our approach or working with traditional distributed systems:

- SOAP is not designed to give support to those popular platform types, such as CORBA, DCOM, etc. Hence, it cannot be combined to existing CORBA systems, DCOM systems, or Java RMI systems naturally.
- SOAP is a definition of the communication protocol contents. The calling mechanism has to be defined by users. It would be hard to support some complicated calling methods, such as callbacks.
- Programmers need to deal with XML details while developing distributed systems, such as dealing with XML parsers, etc.

For our approach described in this thesis, we would try to maintain the advantages of using XML in communication messages protocol, and to avoid the deficiencies of SOAP.

2.4 Overview of XML-RPC

XML-RPC is another XML-based protocol for communication in distributed systems across Internet firewalls. It is developed by UserLand Software, Inc at 1998. Though it is not a standard of W3C, it has a little-bit longer history than SOAP.

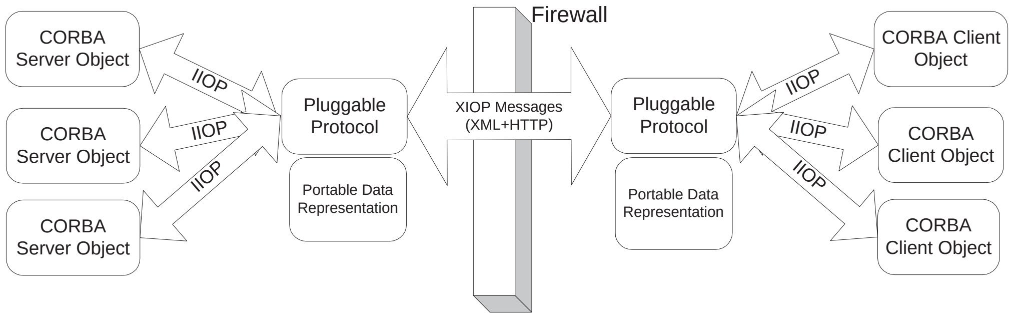


Figure 2.6: Diagram showing the mechanism of XIOP

It provides very similar functionalities as SOAP (refer to section 2.3). XML-RPC also works by marshaling procedure calls over HTTP as XML documents. It is even more lightweighted than SOAP as SOAP supports XML Schemas, enumerations, strange hybrids of structs and arrays, and custom types which XML-RPC does not support. At the same time, several aspects of SOAP are implementation defined. So, XML-RPC has less features than SOAP, but the advantage of it is having more compact XML message structures.

As XML-RPC is very similar to SOAP, they both have similar advantages and deficiencies.

2.5 Overview of XIOP

Different from SOAP and XML-RPC, XIOP is designed as a substitute of CORBA IIOP in XML format. It is a pretty new protocol developed by Financial Toolsmiths AB, which was introduced in April 2000. Besides working well across the Internet firewalls, XIOP is compatible with existing CORBA systems. XIOP requires a pluggable protocol framework to make conversion between IIOP and XIOP. Figure 2.6 shows the mechanism of the use of XIOP.

The advantages of XIOP are:

- Integrates HTTP and XML into a distributed object framework.
- Fits into an existing, open and well established distributed object framework: OMG CORBA.
- Uses the OMG IDL type system and therefore is more suitable for mappings to programming languages "natural datatypes"
- Leverages existing mappings to programming languages such as C, C++, Java, ADA, (Python, Perl etc).
- Leverages existing object serialization standard.

In spite of the many advantages of XIOP, it still has many rooms for improvement. XIOP development is mainly focusing on the conversion mapping of traditional IIOP and XML-based XIOP, but for the mechanism of conversion and callings, there are still some deficiencies that can be improved.

- The pluggable protocol framework increases the complexity of the CORBA environment.
- The pluggable protocol framework centralizes all protocol conversion jobs which may be the bottle-neck in message transmission.
- We need to modify the original CORBA components in order to use the message-conversion framework.

Our approach described in this thesis is trying to maintain the advantages of XIOP messages, but using a simple architecture, avoiding modification to the original components, and providing methods for workload distribution.

Chapter 3

Using XML and Servlets to Support CORBA Calls

3.1 Objective

In Chapter 1, we have described the need of integrating different distributed systems and how this would induce some communication problems. One problem we have mentioned is that the common use of Internet firewalls would block the communication with many traditional distributed system platforms, such as IIOP in CORBA. Using IIOP as an instance, it is the Object Management Group's (OMG) specified network protocol for communication between object request brokers. It employs TCP/IP and can be handled by common firewalls at network and transport level with packet filters. But at the application level, the message body of IIOP is encoded in Common Data Representation (CDR), which is different from the packet formats with other common protocols, such as FTP or HTTP. Firewalls are unable to decode it because they cannot base filtering decisions on IIOP messages.

There are some firewalls which are dedicated for some special protocols, for example, Orbix Wonderwall [5] and Visibroker Gatekeeper [6] are dedicated for

CORBA IIOP. But there are some deficiencies for these two firewalls, as they are generally vendor-dependent and may not support certain CORBA features like callbacks.

There are proposals to use SOAP, XML-RPC or XIOP to tackle this firewall problem. But SOAP and XML-RPC are not specified to work with other protocols and programmers may have to deal with XML parsers or other XML tools in order to use it. And for XIOP, programmers also need to deal with the pluggable protocol framework, which is not transparent to them.

Here, we try to use XML, Java Servlets and HTTP to simulate IIOP calls. Modifications to the existing components are avoided in order to give great transparency to users about our newly added implementation.

In this chapter, we will introduce our mechanism which can achieve this target. First, the general concept of our approach will be introduced, and we will explain what the server and the client sides will do in details. Then, we will describe how callbacks can be done. We also describe what would be the contents in the XML messages and how we automatically generate some source codes of our newly added components, and make those components transparent to programmers.

3.2 General Concept of Our Mechanism

Here, we use CORBA as an example to demonstrate our tunneling mechanism. Actually, the same mechanism can be applied to other distributed system environments, such as DCOM or other agent environments.

Let us assume that we are having two CORBA enclaves, each of them is located in a Local Area Network (LAN). For each LAN, it has a firewall that separates them from the outside Internet. Now, we want to let the objects

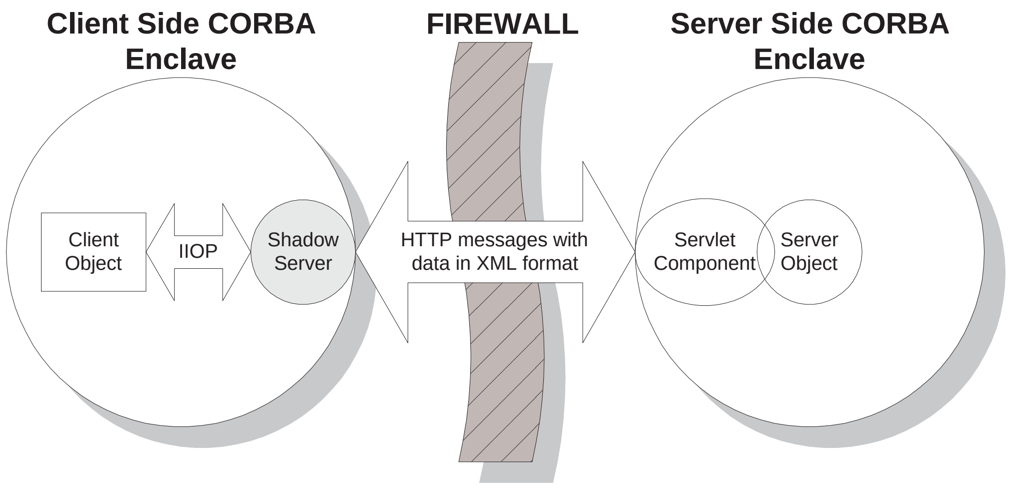

Client Side CORBA Enclave
FIREWALL
Server Side CORBA Enclave
Client Object
IIOP
Shadow Server
HTTP messages with data in XML format
Servlet Component
Server Object

Figure 3.1: Our mechanism to support general CORBA IIOP across the firewalls

in these two enclaves to be able to communicate with each other. Unfortunately, CORBA IIOP cannot pass through those common firewalls. In order to support IIOP calls between two CORBA enclaves separated by firewalls, the main approach we use is to convert the contents of IIOP calls into HTTP calls, as HTTP calls can go through the firewall blocking. Figure 3.1 shows the mechanism of our tunneling solution. The object that issues a request is named as client object, while the object that gives a response is named as the server object. The enclaves they are located are named client side and server side respectively.

In this case, we need two components to do the conversions from IIOP to XML-based data automatically:

- one is at the client side to convert the request messages from IIOP messages to HTTP messages (i.e., the one named as *Shadow Server* in Figure 3.1),
- another one is at the server side to convert the HTTP request messages

back to normal IIOP messages (i.e., the one named as *Servlet Component* in Figure 3.1).

Their duties will be inter-changed when the server returns the computation results back to the client side. We now explain the details of these two components.

3.2.1 At Client Side

At the client side, we add a new CORBA object which is used to convert IIOP messages into XML-based messages and vice versa. We call this client-side conversion component as *Shadow Server*, as it will perform exactly the same functions as the actual target server object. This conversion component allows client objects to make requests to it, with request methods which are exactly the same as in the actual server object. And this component will immediately return the results to the client objects, with the returned data in the same type and format as the actual server would return. So, in the viewpoint of the client objects, this conversion component performs exactly the same as the original target server object, and we can just regard this conversion component object as a proxy. That is why we call it a *Shadow Server*.

Figure 3.2 shows the details of what is happening at the client side. Client object first sends a request to the *Shadow Server*. The *Shadow Server* object provides the same interface as the real target server object. They are sharing the same interface definition of the target server IDL file. By using the same interface, the client objects will not notice the differences between these two objects while making requests.

Other than the common interface, all the internal implementation of the methods would be different. The Shadow Server will not do any real com-

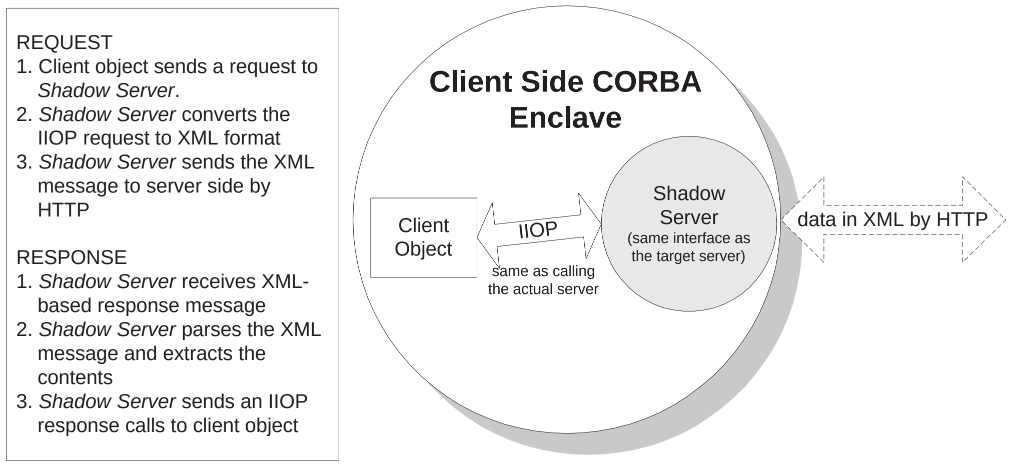

REQUEST
1. Client object sends a request to *Shadow Server*.
2. *Shadow Server* converts the IIOP request to XML format
3. *Shadow Server* sends the XML message to server side by HTTP
RESPONSE
1. *Shadow Server* receives XML-based response message
2. *Shadow Server* parses the XML message and extracts the contents
3. *Shadow Server* sends an IIOP response calls to client object
Client Side CORBA Enclave
Client Object
IIOP
same as calling the actual server
Shadow Server (same interface as the target server)
data in XML by HTTP

Figure 3.2: The details of our tunneling mechanism at client side

putation or manipulation to the data passed by the clients, instead it will convert the parameters and other related information to XML-based messages and send them to the real server object via HTTP. The details of the XML message contents will be described in section 3.3.

When the server side returns a response message to the client side, no matter it is a normal response, or an exception, it will also be a XML-based message via HTTP. The response message will be returned to the *Shadow Server* and then the *Shadow Server* converts all received HTTP messages into ordinary IIOP messages and returns them to the client objects.

3.2.2 At Server Side

At the server side, we add a new Java Servlet component which is used to convert IIOP messages into XML-based messages and vice versa. This Java Servlet component on the server side communicates with the *Shadow Server* on the client side. Servlets interact with web clients via a request-response paradigm implemented by the Servlet container. This request-response model

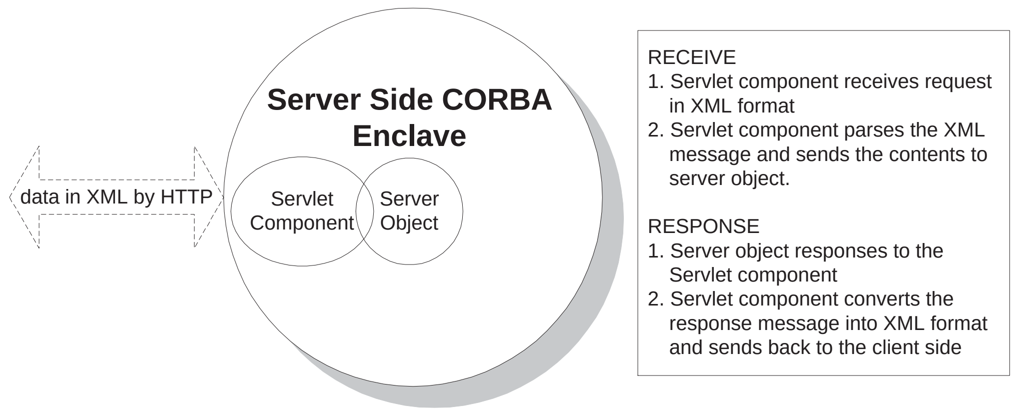

Server Side CORBA Enclave
data in XML by HTTP
Servlet Component
Server Object
RECEIVE
1. Servlet component receives request in XML format
2. Servlet component parses the XML message and sends the contents to server object.
RESPONSE
1. Server object responses to the Servlet component
2. Servlet component converts the response message into XML format and sends back to the client side

Figure 3.3: The details of our tunneling mechanism at server side

is based on the behavior of HTTP.

Each server object, which is ready for outside calls, will have a corresponding Servlet component associated with it. Figure 3.3 shows the detailed situation on the server side. When the client side sends a message, it will directly send to the Servlet component, which is already associated with the target server object. This Servlet component will parse the XML-based request message, extract the necessary parameters and the related information from it, and then convert it to an ordinary IIOP call and invoke the target server object.

When the server object has finished the computation, it will send the response to the Servlet component. The Servlet component will convert the response to XML format and return it back to the client side via HTTP. It is expected that the *Shadow Server* at the client side will receive that response message.

It would be very similar for the server to return *exception* messages. The *exception* messages will also be converted into XML format by the Servlet component and then return to the client side via HTTP. Also, the *Shadow Server* at the client side will receive those *exception* messages. In section 3.3, we will describe the details of the data contents in XML messages.

3.3 Data in Transmission

3.3.1 Using XML

Extensible Markup Language (XML) plays a very important role in the transmission of HTTP messages. XML is semi-structured and hence has the flexibility to well represent most of the complicated data structures. Hence, we use HTTP to send streams of XML data between the client and server sides.

Further to the flexibility of data representation, the readability and the ease of manipulation of XML information provide great flexibility for server as well as client implementations. That is the reason why we convert the binary stream of IIOP messages into XML.

By using the Data Type Definition (DTD) of XML data, we can provide a grammar for the XML data transmission format. Hence we can make a compromise on the interpretation data transmission formats for both client and server sides. As we have DTD to provide rules and guidelines of transmission message format for decoding and encoding, there is no limitation for the client side or the server side to be implemented by CORBA objects. Hence, programmers can have great freedom to choose different implementation methods. We will give more details of this in the next chapter.

3.3.2 Format of Messages in Transmission

If a client object needs to make a request to a server object, it has to first send the request message to *Shadow Server*. An ordinary request message is sent to

Shadow Server by the client object, and then *Shadow Server* will get the values of the parameters.

Based on the corresponding DTD of the target object, the *Shadow Server* constructs an XML document which describes the parameter types and values, and the object method being requested. There is a generic component in the *Shadow Server* that can construct the XML message based on the regulations stated in the DTD. For the details about the DTD format and the XML data format, please refer to Chapter 4.

After the XML message is constructed, it will be sent to the Servlet component on the server side by the POST method calls of HTTP. Then the whole XML message will be sent to the server side immediately. Each message in the POST method calls contains the following information:

- the IP address or domain name that the Servlet component is located;
- the port number to access that Servlet component;
- the path name and the name of the Servlet component; and
- the encoded XML message.

The first three items should be known by the *Shadow Server* during its initialization. The last item can only be determined at run time. We will give more details about the formation of the XML messages in the next chapter. For example, if we want to send a piece of XML request message form the client side to the server side, which calls the `deposit` method of object `Account`, we would have the following XML message:

```
<request>
   <Account type="interface">
      <deposit type="operation">
         <parameter ref="in" order="1">
            <float name="amount">23000.45</float>
            </parameter>
         </deposit>
      </Account>
   </request>
```

Assume the domain name of the server host is `pc90003.cse.cuhk.edu.hk`, path name is `research/`, port number is 8000 and the name of the Servlet component is `testing`. After encoding the XML message, the *Shadow Server* on the client side would use the HTTP POST method to send the encoded XML message to the Servlet component:

```
http://pc90003.cse.cuhk.edu.hk:8000/research/testing?%3C
request+type%3D%22interface%22%3E+%3Caccount+type%3D%22i
nterface%22%3E+%3Cdeposit+type%3D%22operation%22%3E+%3Cp
arameter+ref%3D%22in%22+order%3D%221%22%3E+%3Cfloat+name
%3D%22amount%22%3E23000.45%3C%2Ffloat%3E+%3C%2Fparameter
%3E+%3C%2Fdeposit%3E+%3C%2Faccount%3E+%3C%2Frequest%3E
```

At the server side, when the Servlet component gets the message by the HTTP POST method, it will extract and parse the XML message, and then invoke the corresponding method of the server object, by passing the extracted parameters to it. The server object will perform the computation immediately and pass the results or any exception message back to the Servlet component.

The Servlet component will then convert the results, or the exception signal into XML message again, based on the DTD of the server object. The returning stream would be the response part of the HTTP POST method that the *Shadow Server* issued. When *Shadow Server* gets the returned XML message, it will parse it, and then return the results or raise an exception to the caller client object.

3.4 Supporting Callbacks in CORBA Systems

The mechanism introduced in section 3.2 can handle all of the basic types of method calls. But only applying this mechanism may not be able to handle other more complicated calling features. For example, CORBA provides an interesting and useful feature, named Callbacks, which needs an enhancement of our mechanism in order to handle it.

3.4.1 What is callback?

Just imagine an example of a stock-prices reporting system: You are using the client application to lookup the changes of the prices of your stocks. There are thousands of users like you, and hence there may be thousands of client programs that need to connect to the server for looking up the prices every minute in order to know the latest stock prices. Though the prices may not be changing all the time, there will still thousands of connection and lookups every minute. This would lead to a nightmare in network traffic.

When client objects need to react to changes or updates that occur on the server side, it would be rather inefficient for the client objects to lookup the server periodically. Instead, it would be more efficient if the server can notify the clients whenever there is an update on the server side, hence the client programs can react to changes with a faster response, and also can minimize the number of connections. That is, once the client programs have subscribed to certain stocks, whenever there are updates in stock prices, the server will inform the client programs. What those client programs need to do is just waiting for the server to call. This approach is called the callback feature.

The callback feature allows a client object to pass the reference of itself as one of the parameters when invoking the server object's methods. And then,

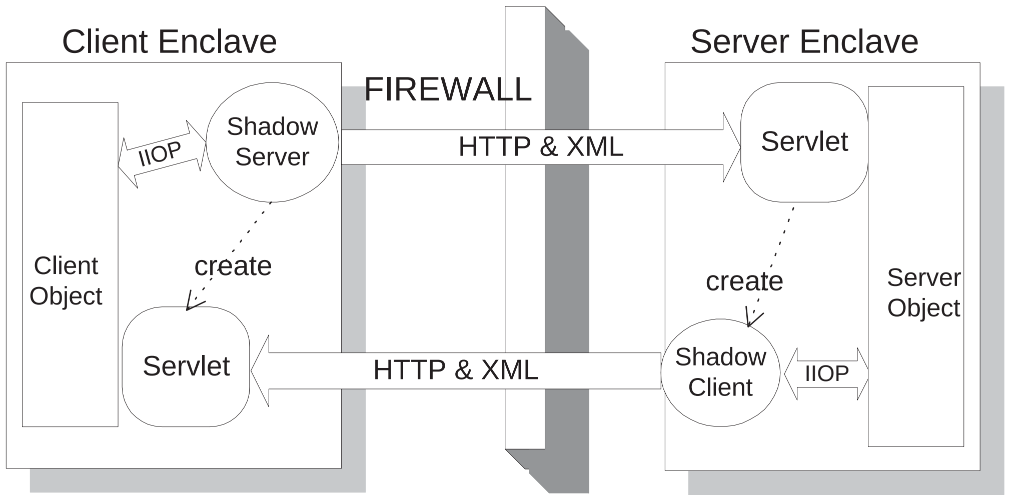


Figure 3.4: Mechanism that supports CORBA callbacks

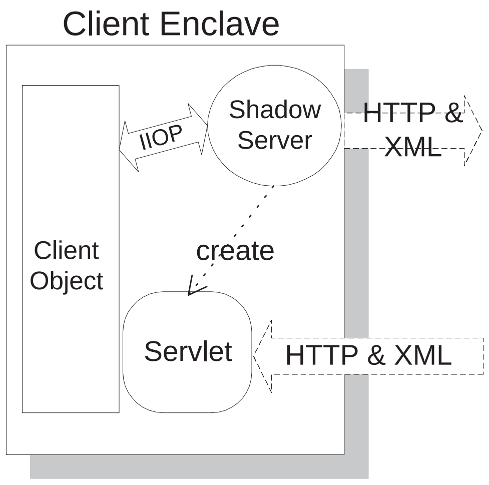
the server object can call the client object's methods by the reference. This requires both sides to be capable of starting a communication. Because of this, many CORBA-dedicated firewalls are not capable to do so. Here, we try to enhance the mechanism we described before to enable the callback features.

3.4.2 Enhancement to Allow Callbacks

As the callback feature needs both client and server objects to be capable of initializing a new communication, we implement both sides to have the shadow objects and Servlet components. We describe our mechanism for CORBA callbacks in Figure 3.4.

Enhancement on Client Side

On the client side, if the client object is expected to use the callback feature, it should have a Servlet component associated with it at the very beginning, which can be known from the system IDL design.



Procedure

1. Client object first sends a method call to the *Shadow Server*. For callbacks, the method call must contain the client object itself as one of the parameters.

2. Once the *Shadow Server* checks out there is a possibility of callbacks, it will create a new Servlet component assoicated with the client object for the server object to callback later.

3. *Shadow Server* sends the method request as normal to the server side, with some additional information of the new Servlet component, such as its location, port number, calling method list, etc.

4. The new Servlet component waits for callback from server side.

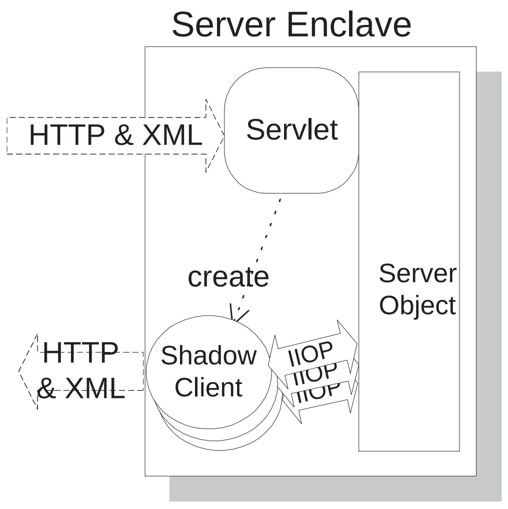
Figure 3.5: CORBA callback mechanism on client side

The client object will first get a reference to the *Shadow Server* on the client-side CORBA enclave. When the *Shadow Server* receives a method call from the client object that may request a callback (that is putting itself as one of the parameters), and if it is the first time, it will create a Servlet component to be associated with that client object, and will store the information such as IP address, port number, host name, calling methods, etc, in itself.

These information (IP address, port number, call method and calling methods, etc) of the Servlet component associated with the client object, will also be sent to the server side when invoking the server method. Figure 3.5 shows the details of what happens on the client side.

Enhancement in Server Side

On the server side, once the Servlet Component has received a message that includes the information of the location of the Servlet component of the calling



Procedure

1. Servlet Component in the Server side receives call from outside. If there is callback, this Servlet Component will create a *Shadow Client* immediately, which will be initialized by the info of location provided by client side.

2. Servlet Component will inform the server object the location of the *Shadow Client* that has required callback.

3. When there is a need to callback, server object will call the *Shadow Client(s)*.

4. The *Shadow Client(s)* will invoke the Servlet Component(s) on the client side.

Figure 3.6: CORBA callback mechanism in server side

client object, it will automatically generate a new *Shadow Client* object, which has the same interface as the calling client object. This *Shadow Client* object will be initialized by the information of the real client and its Servlet component, so that it will know how to set up the connection with the real client later.

The real server object then gets the reference of the *Shadow Clients* (the newly created ones on the server side) that requires callbacks. Whenever the server is updated, it can call the *Shadow Clients* to invoke and notify the client object. During the data transmission, we still employ similar XML data format as described in the previous section. By this mechanism, we can support IIOP calls for CORBA callbacks by integrating XML, Servlet and HTTP calls. Figure 3.6 shows the details of what is happening on the client side.

3.5 Achieving Transparency with Add-on Components

One of the advantages of our add-on components are their transparency to the whole system. They also help us to avoid any modifications to the existing components in the system. We have shadow objects that have exactly the same interface as the objects being called in another enclave. Shadow objects are located at the same enclave as the callers, and they perform exactly the same functions as the target objects that the callers want to call. Servlets components, which are located at the same enclave as the objects being called, should be able to convert the XML messages to appropriate calling methods that the objects being called can understand. Interfaces of the objects are very important for the function and the creation of these add-on components.

When building a CORBA system, programmers are first needed to design the interfaces of all CORBA objects and provide an IDL file to generate the necessary source codes for server skeletons, client stubs and other system architectures. The IDL design of a CORBA system provides the interface definitions of all the objects in the system. The IDL files can provide the following interface information:

- Interface names;
- Object method names provided by each interface;
- The return type of each method;
- All parameters types and their orders in prototypes of each method;
- All passing types of the parameters (i.e. if they are "passing by reference or passing by value);

- All exceptions in each method;
- All newly defined structures; and
- The possibility of having callbacks features (i.e., when a CORBA object interface has another CORBA object interface as one of its parameters).

With IDL providing adequate information about the interface, we can use these interface information to generate the XML message schema, and also the source code for the add-on components. As both the add-on Servlet components and the shadow objects have many common parts of source codes and they are both concerned only with the interfaces of the server and client objects, we can use the IDL files to generate these add-on components automatically.

We have developed a compiling tool which can compile the IDL files, analyze the interface design and then generate the following artifacts:

- Source code for `Shadow Server/Client` objects;
- Source code for Servlet components; and
- DTD of the transmitted messages.

The generation of these source codes can help to reduce the extra programming work for those add-on components as they usually contain many similarities, especially in the part of converting the internal data structures to XML formats. The generation of DTD files, on the other hand, provides a standard for information exchange in XML formats. Based on the DTD, system developers can have implementations other than using CORBA for their clients or servers components. These tools will be introduced in the next chapter.

Chapter 4

A Translator to Convert CORBA IDL to XML

4.1 Introduction to CORBA IDL

In order to generate the XML documents for data transmission in remote procedure calls and source code generation for the add-on components, we make use of the Interface Definition Language (IDL) files for CORBA object design. We try to use these IDL files to extract all the necessary information of the interfaces from IDL files for the auto-creation of our *Shadow Server* or *Shadow Client*, and the DTD standards for message passing. We are able to do this because the IDL files have already contained all the information about all the interfaces of all CORBA objects that would be involved in remote calling.

The CORBA IDL is used to define interfaces to objects in a distributed environment [3, 24, 25]. The first step in developing a CORBA application is to define the interfaces to the objects required in the distributed system. IDL allows programmers to define interfaces to CORBA objects without specifying the implementation of those interfaces. In fact, programmers can implement IDL interfaces using any programming language for which an IDL mapping is available. CORBA applications written in different programming languages are fully interoperable. CORBA defines standard mappings from IDL to several programming languages, including C++, Java, and Smalltalk, hence, we can say that CORBA IDL is very generic.

A translator, which is written in *Perl*, is implemented in our research project for XML documents generation. Here, we will describe how this translator works to produce XML documents and the DTD files of those XML documents.

In the following sections, we introduce different elements of CORBA IDL and describe one by one in details about how we convert different items into XML format. We will also discuss how the add-on components work, and how they can be generated by the XML files.

4.2 Mapping from IDL to XML

Here, we will describe the schema of mapping CORBA IDL to XML in details. First, we explain how we represent some data types in XML formats. Then, we describe the schema for mapping all information in `interface`. Basically, `interface` in CORBA IDL is representing an object in CORBA. We will also address the inheritance issue in our mapping.

We will explain two uses of the XML documents. One is for representing the IDL, which mainly contains only the structural information of different object interfaces. With this XML file, we can generate the add-on components, such as shadow objects and Servlet components, automatically.

Another one is for transmitting request or response messages in method calling. For these XML messages, they are not just representing the structure, but also data values. DTD is provided to give the grammar for these XML messages.

Table 4.1: Basic types in IDL and their corresponding XML tags

IDL Type	Representation Size	Corresponding XML
short	16-bit	`<short> </short>`
unsigned short	16-bit	`<ushort> </ushort>`
long	32-bit	`<long> </long>`
unsigned long	32-bit	`<ulong> </ulong>`
long long	64-bit	`<longlong> </longlong>`
unsigned long long	64-bit	`<ulonglong> </ulonglong>`
float	IEEE single-precision floating point numbers	`<float> </float>`
double	IEEE double-precision floating point numbers	`<double> </double>`
char	An 8-bit value	`<char> </char>`
boolean	TRUE or FALSE.	`<boolean> </boolean>`
octet	An 8-bit value that is guaranteed not to undergo any conversion during transmission	For simplicity, we do not support it right now. In fact, `uuencode` can convert binary messages to character strings
any	The any type allows the specification of values that can express an arbitrary IDL type	For simplicity, we do not support it right now.

4.2.1 IDL Basic Data Types

Same as many programming languages, IDL also provides a number of basic types for defining interfaces, such as integers, floating point numbers, etc. Table 4.1 lists the basic types supported in IDL.

To convert these values into XML data, we simply use the variable type names as the tag names. For example, short becomes `<short>, </short>`, unsigned short becomes `<ushort>, </ushort>` etc.

In transmission messages, we will put the value of a variable inside the tags, and state the variable name in an attribute of the tags as `name`. For example, a short integer named `abc` is carrying 14, then the representation is:

```
<short name="abc">14</short>
```

In the XML representation of attribute definition in IDL files, no data is in `short` variable, hence we use a short form of tags: `<short name="abc" />`.

4.2.2 IDL Complex Data Types

This section describes the IDL complex data types including: `enum`, `struct`, `string`, `sequence`, and `array`. They may consist of a number of basic type elements. In IDL, for the definition of these complex types, a new type name may be assigned. We use these new type names as the new tag names in XML documents. And all new tags would bound all their detailed information and have the attribute `complex` to indicate what category of complex data type it is. Here, we will describe different complex data types in details.

Enum Type

An enumerated type allows you to assign identifiers to the members of a set of values. For example, a variable of `Color` type below may have `red`, `blue` or `green` as its value:

```
enum Color {red, blue, green};
```

To use XML in representing the IDL design, we will use tags `<element>`, `</element>` to present the possible values as follow.

```
<Color complex="enum">
   <element>red</element>
   <element>blue</element>
   <element>green</element>
</Color>
```

When we use XML in message transmission, we will use only one `<element>`

`</element>` pair to represent the current value of the variable. For instance, a variable `pixel` of `Color` type in the previous example containing a value as `blue`, would be represented in the XML message in transmission like this:

```
<Color complex="enum" name="pixel">
   <element>blue</element>
</Color>
```

Struct Type

A struct data type allows programmers to package a set of named members of various types, for example:

```
struct Customer{
   long id;
   short age;
   boolean ismale;
};
```

The `Customer` type contains three members, including a long integer named as `id`, a short integer named as `age`, and a boolean named as `ismale`.

Shown here is the way we represent the definition above in XML format:

```
<Customer complex="struct">
   <long name="id"/>
   <short name="age"/>
   <boolean name="ismale>"/>
<Customer>
```

If we are using the XML to represent a struct variable, say with variable name as `peter` of type `Customer` in the previous example, and `id` is 123, `age` is 28 and `ismale` is `TRUE`, the information will be presented as follow.

```
<Customer complex="struct" name="peter"/>
   <long name="id">123</long>
   <short name="age">28</short>
   <boolean name="ismale">TRUE</boolean
</Customer>
```

String

An IDL string represents a character string, where each character can take any value of the `char` basic type. If the maximum length of an IDL string is specified in the string declaration, then the string is bounded, otherwise the string is unbounded.

```
string<10> place;       //Bounded String
string name;            //Unbounded String
```

The usage of `<string>` tags is very similar to those basic data types. To represent a string with maximum length defined, we use an attribute `size` in the `<string>` tags. (Otherwise, this attribute will be omitted.)

```
<string size="10" name="name"/>
```

Sequence and Arrays

In IDL, you can declare a sequence and array of any IDL data types. An IDL sequence is similar to a one-dimensional array of elements, but it does not have a fixed length. If the sequence has a fixed maximum length, then the sequence is bounded. Otherwise, the sequence is unbounded.

```
sequence<short, 6> marksix;     //Bounded Sequence of Short Integers
sequence<string> name;          //Unbounded Sequence of String
```

Shown here is how we represent a sequence in XML for IDL definition.

```
<sequence bounded="6">
   <short/>
</sequence>
```

If we represent data, the elements bounded by tag `sequence` should be repeated with corresponding values. Say, a sequence of `short` with variable name as `number`, with values in the sequence (1,2 and 3) will be represented as follow:

```
<sequence bounded="3" name="number">
   <short index="1"> 1 </short>
   <short index="2"> 2 </short>
   <short index="3"> 3 </short>
</sequence>
```

XML is so flexible that it can well represent structures even as complex as a sequence of struct type data. Using `customer` in the struct type section as an example:

```
<sequence bounded="6" name="number">
   <Customer complex="struct" index="1">
      <long name="id">111</long>
      <short name="age">24</short>
      <boolean name="ismale">TRUE</boolean>
   </Customer>
   <Customer complex="struct" index="2">
      <long name="id">112</long>
      <short name="age">39</short>
      <boolean name="ismale">FALSE</boolean>
   </Customer>
   <Customer complex="struct" index="3">
      <long name="id">113</long>
      <short name="age">23</short>
      <boolean name="ismale">TRUE</boolean>
   </Customer>
</sequence>
```

For arrays, they are very similar to sequence, but they are multi-dimensional and always have a fixed size. Using an example of a 3×2 array:

```
short test[3,2]
```

We use attributes `size1`, `size2`, `size3`,..., etc. to represent the size in different dimensions, in which `size1` means the first dimension, `size2` means the second dimension, etc. To represent this statement of IDL in XML format:

```
<array size1="3" size2="2" name="test"/>
   <short/>
</array>
```

To represent the data for transmission in XML format, we use attributes `index1`, `index2`, `index3`,..., etc. to represent the indexes of the elements with different dimensions. We can have the following structure:

```
<array size1="3" size2="2" name="test">
   <short index1="0" index2="1"> 123 </short>
   <short index1="0" index2="2"> 124 </short>
   <short index1="1" index2="1"> 125 </short>
   <short index1="1" index2="2"> 126 </short>
   <short index1="2" index2="1"> 127 </short>
   <short index1="3" index2="2"> 128 </short>
</array>
```

TypeDef

The `typedef` keyword allows programmers to define a meaningful or more simple name for an IDL type. The following IDL provides a simple example of using this keyword:

```
typedef float Money
typedef MarkSix short[6];
```

We would just use the new type name to create a pair of tag to bound the data type it representing.

```
<MarkSix complex="typedef">
   <array size1="6" name="test"/>
      <short/>
   </array>
</MarkSix>
```

To represent the `typedef` data in transmission, tags of the `typedef` name would bound the original message.

```
<MarkSix complex="typedef">
   <array size1="6" name="test"/>
      <short index1="1"> 1 </> <short index1="2"> 13 </>
      <short index1="3"> 17 </> <short index1="4"> 24 </>
      <short index1="5"> 29 </> <short index1="6"> 38 </>
   </array>
</MarkSix>
```

4.2.3 IDL Interface

An IDL interface describes the functions that an object supports in a distributed application. Interface definitions provide all of the information that clients need in order to access the object across a network.

```
interface Account {
   // The account owner and balance.
   readonly attribute string name;
   attribute float balance;

   // Operations available on the account.
   void deposit (in float amount);
   boolean withdraw (in float amount);
};
```

The example above shows the interface `Account`. The objects which implement this interface will have two attributes, and two operations. To represent by XML, we just use the interface name as the new tag name and bound everything inside.

```
<Account complex="interface">
   ...
   Details of Attributes and Operations
   ...
</Account>
```

We will talk about the detailed XML representation of attributes and operations in the following parts.

4.2.4 Attributes

Attributes correspond to variables that an object implements. They indicate that these variables are available in an object and that clients can read or write their values.

In general, attributes map to a pair of functions in the programming language used to implement the object. These functions allow client applications to read or write the attribute values. However, if an attribute is preceded by the keyword readonly, then clients can only read the attribute value. These read and write functions will also be prepared in our `Shadow Client` and `Server` objects, and Servlet components.

With reference to the interface `Account` shown in the previous section, it contains two attributes:

```
readonly attribute string name;
attribute float balance;
```

In the example, the string attribute is read-only while another one is read-write. To represent these, we use a tag attribute `readonly`.

```
<string type="attribute" readonly="true" name="name"/>
<float type="attribute" name="balance"/>
```

4.2.5 Operations (Methods)

IDL operations define the format of functions, methods, or operations that clients use to access the functionality of an object. An IDL operation can take parameters and return a value, using any of the available IDL data types.

In our representation of IDL files using XML, we use the operation name as the tag name, and have an attribute `type` to indicate that it is an object method. Within the pair of tags, we have two more kinds of element tags, they are `<parameter>` and `<return>`. `<parameter>` tags describe one of the parameters of the operation. It has attribute `ref` to state if it is passed by reference or values, and attribute `order` to describe its listing order with other parameters in the method call.

Here is an example with the two operations shown in the example in the previous section.

```
<deposit type="operation">
   <parameter ref="in" order="1">
      <float name="amount"/>
   </parameter>
</deposit>
<withdraw type="operation">
   <return>
      <boolean/>
   </return>
   <parameter ref="in" order="1'>
      <float name="amount"/>
   </parameter>
</withdraw>
```

If the operation takes an object as one of its parameters, we pass the object's interface in the method call, which is also bounded by `parameter` tags. The XML-based interface definition passed should contain all the information of its attributes, operations and exceptions. This would be useful in callback features.

4.2.6 Exceptions

IDL operations can raise exceptions to indicate the occurrence of an error. CORBA defines two types of exceptions: System exceptions are a set of standard exceptions defined by CORBA. User-defined exceptions are exceptions that you define in your IDL specification.

Implicitly, all IDL operations can raise any of the CORBA system exceptions. No reference to system exceptions appears in an IDL specification. Also, as all objects may throw system exceptions, it is not necessary to put the definition of system exceptions in the XML documents.

An IDL exception is a data structure that contains member fields. In the following example, the exception `notEnoughMoney` includes a single member of type string.

```
interface Account {
    exception notEnoughMoney {
        string reason;
    };
    void withdraw(in CashAmount amount)
        raises(notEnoughMoney);
    ...
};
```

Inside the `<Account>` tags, we will have the definition of the exception notEnoughMoney as follow:

```
<notEnoughMoney type="exception">
    <string name="reason"/>
</notEnoughMoney>
```

And inside the method `<withdraw>` tags, we will add a new tag for the exception, `<raises>`, to bound the exceptions (similar to `<parameter>` and `<return>` tags).

4.2.7 Inheritance

IDL supports inheritance of interfaces. An IDL interface can inherit all the elements of one or more other interfaces. In our mapping scheme, we will simply put all information of the parent interface into the child interface. For example, we have two interfaces in IDL, where `child` is inherited from `parent`.

```
interface parent {
    short op(in a);
};
interface child : parent {
    readonly attribute short cat;
};
```

In our XML file, the interface `child` would be:

```
<child type="interface" parent1="parent">
    <short type="attribute" name="cat">
    <op type="operation">
        <parameter ref="in" order="1">
            <short name="a" />
        </parameter>
    </op>
</child>
```

4.2.8 IDL Modules

An IDL module defines a naming scope for a set of IDL definitions. Modules allow you to group interfaces and other IDL type definitions in logical name spaces, and prevent name clashes with other modules.

As IDL module is the outermost bounding of the whole IDL file in the XML document, we also create a pair of tags with the name of the module to be the root tags. The tag would contain an attribute `type` with value as `module`.

```
<bank type="module">

    <!-- ... definition interfaces -->

</bank>
```

4.2.9 A Sample Conversion

We have described the schema of the conversion from CORBA IDL to XML. Here, we give a sample IDL file, and then demostrate how we convert it into XML format. Figure 4.1 shows a sample IDL of the information system of a small shop. Figure 4.2 is the XML format of the IDL in Figure 4.1.

```
module OrderProcessing {

   typedef string ProductCode;
   enum PriceType { retail, contract, promotion };

   struct Customer {
      string customerCode;
      string customerName;
   };

   interface Price {
      attribute float price;
      attribute PriceType priceType;
      attribute string expiryDate;
   };

   typedef sequence < Price > PriceSequence;

   interface PriceCalc {
      attribute Customer customer;
      attribute ProductCode product;
      void getPrices(out PriceSequence prices);
      void recordUseOfPrice(in Price priceUsed);
   };
};
```

Figure 4.1: A sample IDL file for an information system

4.3 Making a Request or Response

If a client object wants to make a request to the server object, it will make a request call to the `Shadow Server` instead. The `Shadow Server` will generate a XML request message to the server side, which uses a pair of `<request>` tags as the root tags to bound the method calling information.

After the server object has performed the request, it will return a returned value or throw an exception to the client side. The returned messages would be bounded by a pair of `<response>` tags as root tags.

Both XML-based request and response messages will follow the generated

DTD. It is shown in Figure 4.3.

4.4 Code Generation for Add-on Components

In the previous sections, we have shown the complete schema for the mapping of CORBA IDL to XML messages. With that XML data, we can generate the add-on components automatically, as they already have all the information about the interfaces.

For instance, in CORBA callback calls, a client would put itself as one of the parameters in the server's method call. Here, the client putting itself in the parameters means that on the client side, the `Shadow Server` would send the interface definition of that client to the server side. The Servlet component on the server side would immediately use the interface definition of the client, which is in XML format, to create a `Shadow Client` immediately. While on the client side, the `Shadow Server` will also generate a Server component. Here, we will discuss the generation of these two components.

4.4.1 Generation of Shadow Objects

A Shadow Object has the same interface as the actual target object. To generate it, we have to analyze the XML file that represents the object interface. For each operation in the interface, we create the same operation which takes the same parameters and returns a value of the same data type.

Inside each operation, we would have three parts of source code:

1. To convert the parameters into text strings, and form a XML message which describes a request call.

2. To start a HTTP connection with and send the XML message to the server side, and wait for the response.

3. When the XML-based response message is returned, either a returned value form or an exception, it is parsed and returned to the caller object by returning it or throwing an exception.

4.4.2 Generation of Servlet Components

The mechanism in Servlet components is much simpler than that in `Shadow Objects`. A Servlet component is to take the XML-based request messages from the client side, and then make a corresponding call to the server object. Hence, it has to know how to call all the operations of the server object.

Inside a Servlet component, we would have three parts of source code.

1. To wait for the client requests by HTTP connection.

2. When the XML-based request message is received, which contains information about the calling method and its parameters, the Servlet parses the message and makes a method call with corresponding parameters to the server object.

3. To convert the returned values or any exceptions into XML format, and then return it back to the client side.

```
<OrderProcessing type="module">

    <ProductCode complex="typedef"> <string/> </ProductCode>

    <PriceType complex="enum">
        <element>retail</element>
        <element>contract</element>
        <element>promotion</element>
    </PriceType>

    <Customer complex="sequence">
        <string name="customerCode">
        <string name="customerName">
    </Customer>

    <Price complex="interface">
        <float type="attribute" name="price"/>
        <PriceType type="attribute" name="priceType"/>
        <string type="attribute" name="expiryDate"/>
    </Price>

    <PriceSequence complex="typedef">
        <sequence> <Price/> </sequence>
    </PriceSequence>

    <PriceCalc type="interface">
        <Customer type="attribute" name="customer"/>
        <ProductCode type="attribute" name="product"/>
        <getPrices type="operation">
            <parameter ref="out" order="1">
                <PriceSequence name="price"/>
            </parameter>
        </getPrices>
        <recordUseOfPrice type="operation">
            <parameter ref="in" order="1">
                <Price name="priceUsed"/>
            </parameter>
        </recordUseOfPrice>
    </PriceCalc>
</OrderProcessing>
```

Figure 4.2: XML format of the IDL in Figure 4.1

```
<!DOCTYPE OrderProcessing [
    <!ELEMENT OrderProcessing (ProductCode, PriceType,
              Customer, Price, PriceSequence, PriceCalc)>
        <!ATTLIST OrderProcessing type (#CDATA)>
    <!ELEMENT ProductCode (string)>
        <!ATTLIST ProductCode type (#CDATA)>
        <!ATTLIST ProductCode complex (#CDATA)>
        <!ATTLIST ProductCode name (#CDATA)>
    <!ELEMENT PriceType (element*)>
        <!ATTLIST PriceType complex (#CDATA)>
    <!ELEMENT Customer (string*)>
        <!ATTLIST Customer complex (#CDATA)>
        <!ATTLIST Customer type (#CDATA)>
        <!ATTLIST Customer name (#CDATA)>
    <!ELEMENT Price (float, PriceType, string)>
        <!ATTLIST Price type (#CDATA)>
    <!ELEMENT PriceSequence (sequence)>
        <!ATTLIST PriceSequence type (#CDATA)>
    <!ELEMENT sequence (Price*)>
    <!ELEMENT PriceCalc (Customer, ProductCode,
                         getPrices, recordUseOfPrice)>
        <!ATTLIST PriceCalc complex (#CDATA)>
    <!ELEMENT getPrices (parameter)>
        <!ATTLIST getPrices type (#CDATA)>
    <!ELEMENT recordUseOfPrice (parameter)>
        <!ATTLIST recordUseOfPrice type (#CDATA)>
    <!ELEMENT parameter (PriceSequence | Price)>
        <!ATTLIST parameter ref (#CDATA)>
        <!ATTLIST parameter order (#CDATA)>
    <!ELEMENT float (#CDATA)>
        <!ATTLIST float name (#CDATA)>
    <!ELEMENT string (#CDATA)>
        <!ATTLIST string name (#CDATA)>
    <!ELEMENT element (#CDATA)>
]>
```

Figure 4.3: The DTD for the parameter passing of simulated calls

Chapter 5

Communication in Heterogeneous Distributed Environments

5.1 Objective

Nowadays, we have a trend to integration of several information systems in order to provide better services to the increasingly demanding users. In fact, integrating several distributed system is not an easy task. There are many popular environments for the development of distributed applications, such as CORBA [3], DCOM [11] or Java RMI [12] etc. They are developed by different organizations, hence they use different communication protocols. CORBA systems use IIOP, DCOM systems use DCOM protocol, and Java RMI uses Java Remote Method Protocol (JRMP). So, when we want to integrate systems with different distributed environments, it would be hard to let those distributed objects to communicate with others.

Though we have many applications that help us to achieve the communication among heterogeneous distributed environments, they are not generic enough. Take the famous OrbixCOMet [13] by Iona Technologies as an example. It is a typical bridging tool, and implements the COM/CORBA Interworking specification by enabling transparent communication between COM clients and CORBA servers. There is a COMET component located between the CORBA enclave and COM/DCOM enclave, and it acts as a bridge to provide the mappings and perform translation between CORBA and COM/DCOM.

OrbixCOMet provides very good performance in bridging CORBA and COM/DCOM applications, but it is not generic enough to give bridging to other environments, such as JavaRMI. It is because OrbixCOMet uses a middleware COMET between CORBA enclave and COM/DCOM enclave, which would directly convert the binary streams of CORBA IIOP messages to binary streams of DCOM IIOP messages. Though this approach is fast, it cannot be used with other distributed environments. Moreover, as their protocols may not be supported by many common firewalls, they may also encounter the firewall problems as mentioned before.

We try to extend the mechanism described in Chapter 3 in order to support communication in heterogeneous distributed environments. We map the interface definition languages of different environments to the same XML schema, that we have introduced in Chapter 4, hence they have the "common language" to communicate. We base on our CORBA IDL's XML schema, as CORBA IDL is very generic which can be mapped to many different programming languages, hence, it has greater flexibility to let other interface definition languages map to it.

In this chapter, we will first introduce our general principles for the extension of our mechanism. Then, we will focus on the cases of DCOM and Java RMI. For each case, we will look at their mapping schema, and how we achieve communication. Lastly, we will describe how generic our approach is and how it can also adopt to other web applications.

5.2 General Concept

In Chapter 3, we proposed a mechanism of using XML streams with HTTP to solve the firewall problem in distributed systems. This mechanism works because we can avoid the use of IIOP across firewalls, we use HTTP instead. The use of shadow objects and the Servlet components can make our conversion of IIOP to XML transparent from other CORBA objects, as these CORBA objects cannot distinguish them from the real callers or callees. More important, our add-on components are also CORBA objects and located at the same enclave as those original objects, hence there are no communication problems.

Now, we use the approach described in Chapter 3 and extend it as the communication bridge among heterogeneous distributed environments. In heterogeneous distributed environments, objects in one enclave cannot communicate with objects in another enclave. It is because they use different communication protocols. CORBA systems use IIOP, DCOM systems uses DCOM protocol, and Java RMI use Java Remote Method Protocol (JRMP) and they are incompatible to each other. The case is similar to the blocking of firewalls, one enclave cannot talk with another enclave.

To solve this problem, again, we rely on our add-on components, shadow objects and Servlets components. The add-on components are developed under the same environment as the other objects located at the same enclave, so they have no problem to communicate with the objects in the same enclave. At the same time, the shadow objects and the Servlets components can talk to the others with XML and HTTP. This can help to join different heterogeneous enclaves into a network and hence they all can communicate with each other.

To achieve the joint network it is important that all enclaves must agree to use a common XML schema, such that they have the common language to talk with the others. In Chapter 4, we have described the schema of mapping

CORBA IDL to XML format. With this, we can achieve communication of CORBA objects in different enclaves, as all CORBA objects in different enclaves use the same language. We have set the schema for IDL to XML mapping to guarantee the unity in representation.

What we have to do now is to ensure that the interface definition languages in different distributed environments can map onto the same schema. CORBA has a generic IDL which is able to map to many different programming languages. As CORBA IDL is so generic, we try to use its XML schema as the fundamental schema and map other IDLs into the same schema.

In the following sections, we are going to show how we link DCOM systems, Java RMI systems and CORBA systems together with an agreed common XML schema, and a suitable architecture. Our mechanism is so generic that not only DCOM or Java RMI systems can be integrated, but also other distributed environments, or other web applications. We will describe how they can further connect to other web applications.

5.3 Case Study 1 - Distributed Common Object Model

5.3.1 Brief Overview of Programming in DCOM

Developed by Microsoft, DCOM is COM (Common Object Model) with distributed feature (COM only allows processes in a single host to communicate). DCOM supports remote objects by running on a protocol called the Object Remote Procedure Call (ORPC). A DCOM server is a body of code that is capable to be called by objects of a particular type at runtime. Each DCOM server object can support multiple interfaces each representing a different behavior of the object. A DCOM client calls the methods of a DCOM server by acquiring a pointer to one of the server object's interfaces. DCOM server components can be written in diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic and even COBOL. As long as a platform supports COM services, DCOM can be used on that platform. DCOM is now heavily used on the Windows platform. Companies like Software AG provide COM service implementations through their EntireX product for UNIX, Linux and mainframe platforms; Digital for the Open VMS platform and Microsoft for Windows and Solaris platforms.

DCOM objects use Microsoft Interface Definition Language to define their interfaces [26, 27, 28]. Figure 5.1 shows a sample MIDL file. The MIDL compiler creates the proxy and stub code when run on the MIDL file for the static invocation to work. They are registered in the systems registry to allow greater flexibility of their use and virtual table (vtable) will be used for invoking objects. In the MIDL, COM objects would implement `IUnknown` interface for static invocation. Otherwise, COM objects have to implement an interface called `IDispatch` for dynamic invocation to work. As with CORBA or Java/RMI, to allow for dynamic invocation, some ways are needed to describe the object methods and their parameters. DCOM uses type libraries to describe the object, and it also provides interfaces, obtained through the `IDispatch` interface, to query an Object's type library. In COM, an object whose methods are dynamically invoked must be written to support `IDispatch`.

Note that in DCOM, each interface is assigned a Universally Unique IDentifier (UUID) called the Interface ID (IID). Similarly, each object class is assigned a unique UUID called a CLasS ID (CLSID). COM does not support multiple inheritance, instead, it uses the notion of an object having multiple interfaces to achieve the same purpose.

```
[
  uuid(7371a240-2e51-11d0-b4c1-444553540000),
  version(1.0)
]
library SimpleStocks
{
  importlib("stdole32.tlb");
  [
    uuid(BC4C0AB0-5A45-11d2-99C5-00A02414C655),
    dual
  ]
  interface IStockMarket : IDispatch
  {
    HRESULT get_price([in] BSTR p1, [out,retval] float* rtn);
  }

  [
    uuid(BC4C0AB3-5A45-11d2-99C5-00A02414C655),
  ]
  coclass StockMarket
  {
    interface IStockMarket;
  };
};
```

Figure 5.1: An example of MIDL document

5.3.2 Mapping the Two Different Interface Definitions

A MIDL document (refer to Figure 5.1 as example), is usually consisted of two components: interface header and interface body. The interface header is the part bounded by a pair of square brackets, which specifies the information about the interface as a whole. It contains some attributes such as UUID, version numbers, etc. These attributes are not necessary in our XML messages formation. Below the square brackets are interface bodies. The IDL interface body contains data types used in remote procedure calls and the function prototypes for the remote procedures. The interface body can also contain imports, constant declarations, type declarations and object method declarations. So, we are only interested in the information inside the interface body for the translation of MIDL files to XML format, and we will ignore the things included in square brackets.

In the naming convention of DCOM, names started with letter `I` are the interface names. So, we need to convert the contents inside the interfaces to XML messages. For example, `IStockMarket` and `IDispatch` in Figure 5.1 are the interface names. The DCOM IDL file associates the `IStockMarket` interface with an object class `StockMarket` as shown in the coclass block. That means object class `StockMarket` is implementing interface `IStockMarket`, so we need the information provided by interface `IStockMarket` for generating XML messages.

Inside an interface definition are their method definitions. `HRESULT` is the default return type of all object methods in the MIDL file, which represents error and success notifications (such as failure, insufficient memory, invalid arguments, etc). For method parameters, the pair of square brackets in front of each parameter indicates if the parameter is for input, output or both. The last parameter may be the return value, which is indicated by the keyword `retval`. The final objects implementing the interface will return the parameter with `retval` keyword as return value.

Table 5.1 shows the mapping of basic types between MIDL and CORBA IDL. This mapping is based on OMG COM/CORBA Interworking specification which only spells out what the requirements for mapping and interworking are, but provides no implementation [3, 26, 29]. Note that in the conversion from CORBA IDL to MIDL, as MIDL does not have `long long` and `unsigned long long`, these two types will only be mapped to `long` and `unsigned long` respectively.

Table 5.1: Mapping the Basic types in MIDL to CORBA IDL/XML Schema

DCOM MIDL Type	CORBA IDL Type	Corresponding XML
short	short	`<short> </short>`
unsigned short	unsigned short	`<ushort> </ushort>`
long	long	`<long> </long>`
unsigned long	unsigned long	`<ulong> </ulong>`
float	float	`<float> </float>`
double	double	`<double> </double>`
char	char	`<char> </char>`
bool	boolean	`<boolean> </boolean>`

For other complex types and other definitions, though DCOM and CORBA have different syntax, they contain similar information for those definitions such that mapping for them would be trivial.

There are two special issues that needed discussion, they are inheritance and exceptions. CORBA IDL supports interface inheritance, while DCOM does not. Instead of supporting multiple inheritance, DCOM uses the notion of an object having multiple interfaces to achieve the same purpose. In our XML representation for interface, we will simply list all the resultant methods, attributes and exceptions inherited from or extended from all interfaces to the new interfaces. Hence, we need not handle the problem of differences between multiple interfaces extension and multiple interfaces inheritance.

Another issue is exception. One difference between CORBA (and Java/RMI) IDLs and COM IDLs is that CORBA (and Java/RMI) can specify exceptions in the IDLs while DCOM does not. There are system exceptions also in DCOM, but they need not be defined in the MIDL file, which is similar to CORBA. So, we will not include these in our XML representation. For user defined application exceptions, we need to represent them as parameters of the corresponding methods.

The last issue is the attributes. In CORBA IDL, they are declared inside the object interface as ordinary object attributes, with specification if they are read-only. But in MIDL, methods for reading and writing are defined instead. Say an attribute with the name as `<NAME>`, in Java RMI Interface, it would be presented as two methods, one is for reading `_get_<NAME>()`; another one is for writing, `_put_<NAME>()`. If the attributes are read-only, they will not have `_put_<NAME>()` methods.

5.3.3 Sample Architecture of Communicating Between DCOM and CORBA

The mechanism applied here is similar to what we have proposed in Chapter 3. In the DCOM enclave, if a client object needs to call another object outside, a `Shadow Server` object is added to the enclave to simulate the behaviour of the target server object outside. If there is a server object waiting for calls from outside, a Servlet component has to assoicate with it. They are responsible for the conversion of XML message to or from DCOM method calls. In the CORBA side, the configuration is the same as described in previous chapter. Figure 5.2 shows the details of the architecture of establishing such a communication.

For the Servlet components, besides using Java Servlets, we can also use alternative solutions, such as Active Server Pages, CGI, etc. It is because in Windows environment, where DCOM systems work the best at, those alternative solutions may work better than Java Servlets, in terms of performance and compatibility. Anyway, Java Servlets still work fine with DCOM systems.

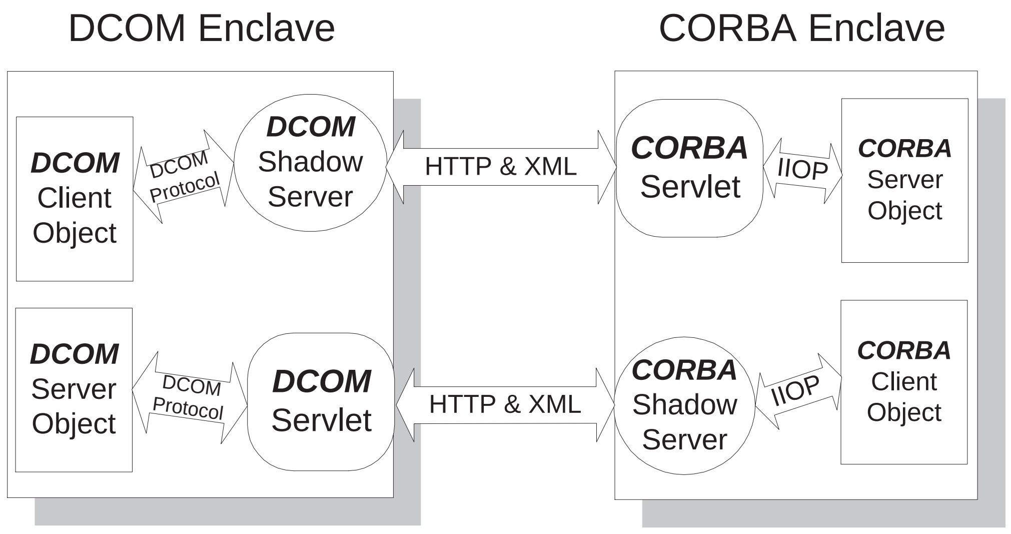


Figure 5.2: Our mechanism to support communication among DCOM and CORBA

5.4 Case Study 2 - Java Remote Methods Invocation

5.4.1 Brief Overview of Programming in Java RMI

Java RMI is similar to CORBA or DCOM which also enables the programmers to create distributed applications. Its application objects are Java-to-Java, in which the methods of remote Java objects can be invoked from other Java virtual machines, possibly on different hosts. A Java program can make a call on a remote object once it obtains a reference to the remote object, either by looking up the remote object in the bootstrap-naming service provided by RMI, or by receiving the reference as an argument or a return value. A client can call a remote object in a server, and that server can also be a client of other remote objects. RMI uses Object Serialization to marshal and unmarshal parameters and does not truncate types, supporting true object-oriented polymorphism.

Similar to CORBA and DCOM, Java RMI also has its interface definition language. As Java RMI supports only Java-to-Java communications, it uses the ordinary Java interface definition as its interface definition language. All remote interfaces extend, either directly or indirectly, the interface `java.rmi.Remote`. The Remote interface defines no methods, as shown here:

```
public interface Remote{ }
```

For example, the code fragment in Figure 5.3 defines a remote interface for a bank account that contains methods that deposit to the account, get the account balance, and withdraw from the account.

```
public interface BankAccount extends java.rmi.Remote {
   public void deposit (float amount)
      throws java.rmi.RemoteException;
   public void withdraw (float amount)
      throws OverdrawnException, java.rmi.RemoteException;
   public float balance()
      throws java.rmi.RemoteException;
}
```

Figure 5.3: An example of Java RMI interface definition

For object methods, each method must declare `java.rmi.RemoteException` in its throws clause, in addition to any application-specific exceptions. Similar to the IDLs of other distributed environments, a remote object passed as an argument or return value (either directly or embedded within a local object) must be declared as the remote interface, not the implementation class.

5.4.2 Mapping the Two Different Interface Definitions

Here, we map the Java RMI Interface [12, 30] into the same XML schema derived from the CORBA IDL. In order to encourage convergence between the RMI and CORBA communities, Object Management Group (OMG) has released the specification for converting the Java Language to IDL mapping [31]. Its target is to define a solution that is both fully compatible with current RMI semantics and fully compatible with OMG IDL, IIOP, and CORBA object model. Sun Microsystems has developed Java RMI/IIOP [14] which uses this specification to do conversion in the RMI objects such that those objects will use IIOP to communicate. With the usage of RMI/IIOP, original RMI objects have to be modified in order to use IIOP, but these objects would not be able to communicate with ordinary RMI objects again.

To make RMI objects able to communicate with CORBA objects without any modification to the existing source code, we apply our mechanism again. We need to map Java Interface to CORBA IDL, and vice versa. Our conversion schema is based on the OMG specifications for Java to IDL mapping [31], and IDL to Java mapping [32].

In Table 5.2, they are the general rules for converting the basic types in Java Interface to the corresponding types in CORBA IDL and our XML schema. This conversion scheme would be useful for binding existing RMI objects towards other systems. For example, the `Shadow Servers` in the RMI enclaves need to use these rules to convert parameters in Java RMI into our common XML schema.

Table 5.3 shows the general rules for converting the basic types in CORBA IDL, or our XML schema, to Java primitive types in RMI Interface. It is used to let outside objects to communicate with Java RMI. For example, the Servlet Components in the RMI enclaves need to use these rules to convert parameters

Table 5.2: Mapping Basic Types from Java to CORBA IDL/XML schema

Java RMI Type	**CORBA IDL Type**	**Corresponding XML**
short	short	`<short> </short>`
int	long	`<long> </long>`
long	long long	`<longlong> </longlong>`
float	float	`<float> </float>`
double	double	`<double> </double>`
char	char	`<char> </char>`
boolean	boolean	`<boolean> </boolean>`

Table 5.3: Mapping Basic Types from CORBA IDL/XML schema to Java

CORBA IDL Type	**Java RMI Type**	**Corresponding XML**
short	short	`<short> </short>`
unsigned short	short	`<ushort> </ushort>`
long	int	`<long> </long>`
unsigned long	int	`<ulong> </ulong>`
long long	long	`<longlong> </longlong>`
unsigned long long	long	`<ulonglong> </ulonglong>`
float	float	`<float> </float>`
double	double	`<double> </double>`
char	char	`<char> </char>`
boolean	boolean	`<boolean> </boolean>`

in our common XML schema to Java RMI types.

For other definitions, the Java Interface does not have much difference when compared to CORBA IDL, such as struct type definition, enum type definition, object interface definition, module definition, etc. Though they have different syntax to CORBA IDL, they still have the same contents which can be directly mapped to the same XML schema.

However, there are points worth mentioning. One is the attributes. In CORBA IDL, they are declared inside the object interface as ordinary object attributes with specification if they are read-only. But in the Java Interface, methods for reading and writing are defined instead. Say an attribute with the name as `<NAME>`, in Java RMI Interface, it would be presented as two methods, one is for reading `get<NAME>()` (or if the attribute is boolean, it would be `is<NAME>()`); another one is for writing, `set<NAME>()`. If the attributes are read-only, they will not have `set<NAME>()` methods.

Another point worth noting is about exceptions. Some system exceptions may be thrown out in Java RMI, and they must be defined in the interface definition in all object methods as `RemoteException`. It is similar to CORBA, in which system exceptions may be thrown in every method, but they need not be defined in CORBA IDL. We do not mark anything in XML schema for system exceptions for CORBA, and hence for Java RMI interface, we do not mark anything in XML schema for system exceptions, too.

The last point is about struct and enum complex type. There is no direct mapping in Java, instead, Java uses a class (having no operations except constructors) to represent. So, if there exists a class in Java with no operations, we can map them to struct or enum type in our XML schema.

5.4.3 Sample Architecture of Communicating Between JavaRMI and CORBA

The mechanism applied here is similar to what we have applied in the case of DCOM before. In the Java RMI enclave, if there is a client RMI object that needs to call another object outside, a `Shadow Server` object is added to the enclave to simulate the behaviour of the target server object outside. If there is a RMI server object waiting for calls from outside, a Servlet component has to assoicate with it. They are responsible for the conversion of XML message to or from Java RMI method calls. Same configuration is set in the CORBA enclaves. Figure 5.4 shows the details of the architecture of such communication.

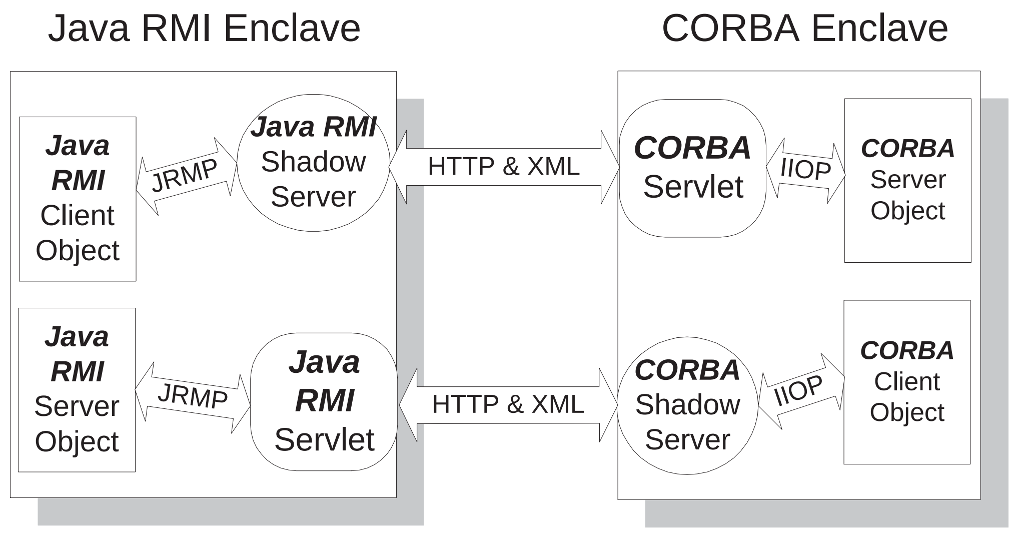


Figure 5.4: Our mechanism to support communication among Java RMI and CORBA

5.5 Be Generic: Binding with the WEB

In the previous sections, we described how we use the same XML scheme as the communication protocol such that we can connect CORBA objects, DCOM objects, and Java RMI objects together. With a common language for communication, objects from heterogeneous environments can interact with each other.

In fact, the calling sides are not limited to CORBA, DCOM and Java RMI objects, but they can be many other implementations. It is because with a right format of XML message and HTTP protocol, any application can invoke the Servlet components associated with the target object in order to call any object methods. It is not difficult to form an XML message and send it by HTTP protocol, as many implementations can achieve this. Hence, the caller can be some traditional stand-alone program (e.g. a C or C++ program, a Java application, etc.), or some web applications (e.g. a Java applets, a Perl

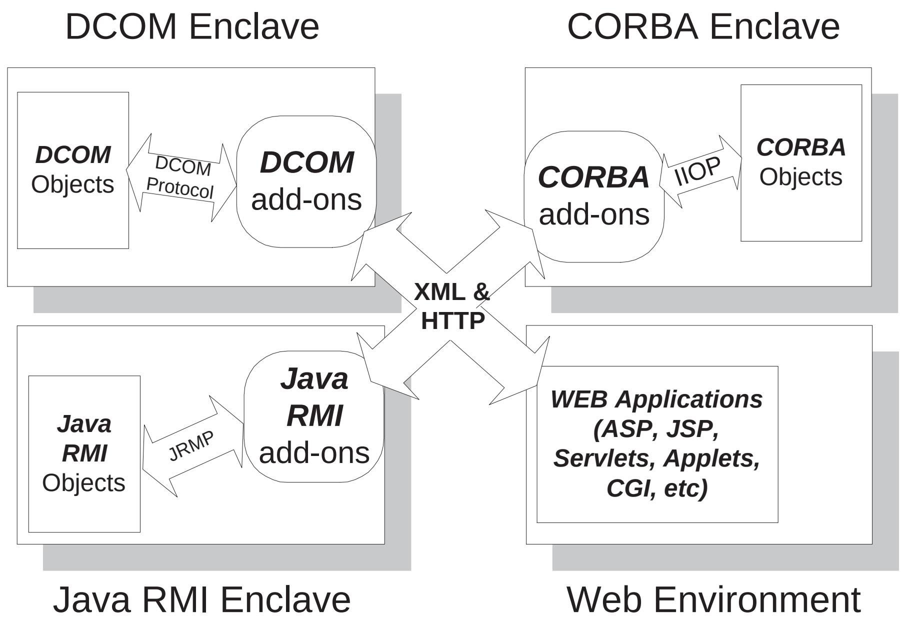


Figure 5.5: Allowing heterogeneous systems to communicate

CGI script, Java Server Pages, Active Server Pages, etc.), or even an ordinary HTML webpage with a button associated with HTTP POST method!

The Servlet components provide great flexibility of the implementation of the client side. With Servlet component and XML messages, even the server enclaves may not be designed as a web application at the beginning, they can still be integrated to the Internet environment easily.

Moreover, the sides being called are also not limited to CORBA, DCOM and Java RMI objects. All components or programs that are able to parse the XML message can be invoked by other CORBA, DCOM or Java RMI objects. `Shadow Servers` on the client side convert all client objects' requests to XML messages, which are readable to many implementations, such as ASP, JSP, etc. Hence, our mechanism can bring all CORBA, DCOM, all Java RMI objects into a completely web-based environment, that is, they can invoke web-based applications, or be invoked by web-based applications.

Chapter 6

Building a Scalable Mediator-based Query System

6.1 Objectives

In the previous chapters, we have introduced our mechanisms for supporting IIOP calls in two CORBA enclaves separated by firewalls, for supporting CORBA callbacks, and for supporting the communication among different distributed environments. In order to let you have a more detailed understanding of our proposed mechanisms, and also to show you how our mechanisms contribute in integrating different distributed systems, we would like to show you how we implement a practical example, a mediator-based query system. It demonstrates how we use our mechanisms to bypass firewalls, to use callback features, and to expand across heterogeneous systems, in order to build a scalable information systems for system integration process.

In this Internet age, people put lots of information on the Internet for others to retrieve. Though there are plentiful information ready for us, we may not be able to query for the contents we need. First, the volume of information is expanding dramatically. Even within an organization, multiple databases are usually employed to store their data. Hence, techniques for searching across a number of distributed data sources are important. Second, information may be provided by various organizations, which means we may need to search across many different sites to obtain the richer information.

In order to solve the first problem, we have established a web-based query system using mediators to search in distributed databases. The mediator is the middleware that forwards user queries to various database engines, and when the database engines searched out the results, it integrates them and returns them back to the users. We will give an introduction to mediators, and describe our system design and implementation in section 6.2. We use CORBA for our infrastructure implementation such that mediators can make queries to various data sources, or even other mediators, within the CORBA enclave.

Although the second problem, that is, making queries to other sites, can be solved by an extension to our mediator system, we need to tackle some technical problems first.

The first problem is the firewall issue. For a local system, we usually have a firewall to protect the computers inside from outside attacks. As we are using CORBA and IIOP cannot pass through firewalls for communication, we try to use Java Servlets, XML and HTTP to simulate object method calls and parameter transmissions in CORBA. By doing so, we can make our system to be more scalable in the Internet with firewalls. This will be discussed in 6.3.

We then demonstrate how we enhance our query system by using the call-back feature. We extend our mechanism to use XML and Servlets to perform some interesting features with callback. Section 6.4 will cover this part.

The second problem is that when we need to combine information among heterogeneous distributed environments, we do not have a generic method to do so. Here, we use XML and Servlets again to connect our CORBA-based system with DCOM-based system and JavaRMI-based system. This part will be covered in section 6.5

By doing all this, finally, we develop a simple and generic way to achieve a more scalable query system against firewalls and heterogeneous distributed environments.

6.2 Introduction to Our Mediator-based Query System

6.2.1 What is mediator?

The mediator is the middleware between the clients and database servers, which can solve some deficiencies of traditional client/server systems [33, 34]. The tight relation between client and server may lead to the following problems: First, a server may be dedicated to some clients only; also, clients may need to search a number of servers to obtain what they need, while those servers may be heterogeneous. Mediator is one of the architectures that can meet the need to make data widely available over a distributed environment. Mediators forward client queries to appropriate data sources, and then integrate the answers from different sources, and forward the integrated answer back to the clients. Figure 6.1 is an example.

There are several advantages of using a mediator system:

- Conceptually, all distributed data sources are integrated into a single component even though the data sources are heterogeneous. Hence, clients need not know about the location or other specific information of the data sources.

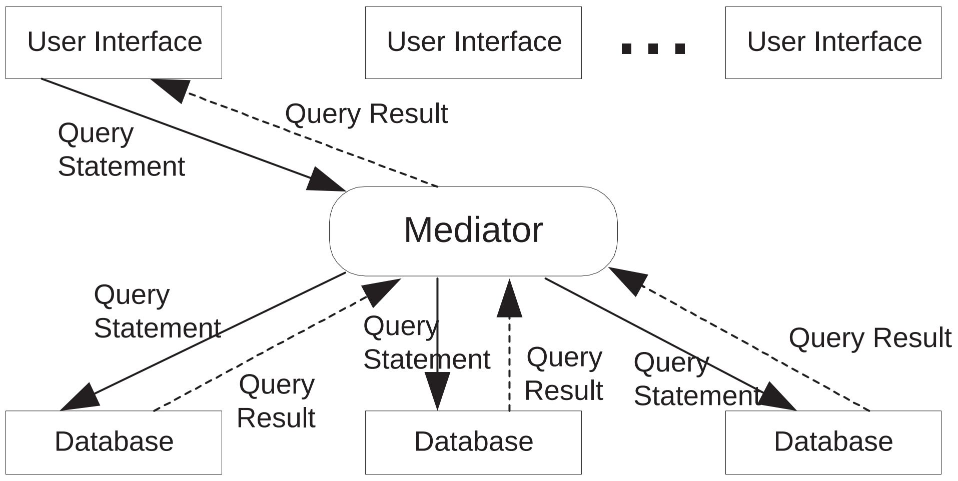


Figure 6.1: Diagram of the mediator concept

- Client programs need not care about the changing of data source locations, and the addition, deletion, or even failure of some data sources.
- The mediators can help the users to choose the most appropriate data sources, based on their queries submitted, to enrich the quality of information retrieval.

6.2.2 The Architecture of our Mediator Query System

Here, we describe the basic architecture of our mediator query system. Our mediator query system is mainly consisted of two components: Query Mediators and Database Query Engines. The design of the architecture of our query system is shown in Figure 6.2. Similar to other mediator systems, the database engines are waiting for the requests from the mediator components. Also the mediator components are waiting for requests from the user interface and upon reception will send these queries to the database engines.

Furthermore, mediators can also send queries to other mediators, which

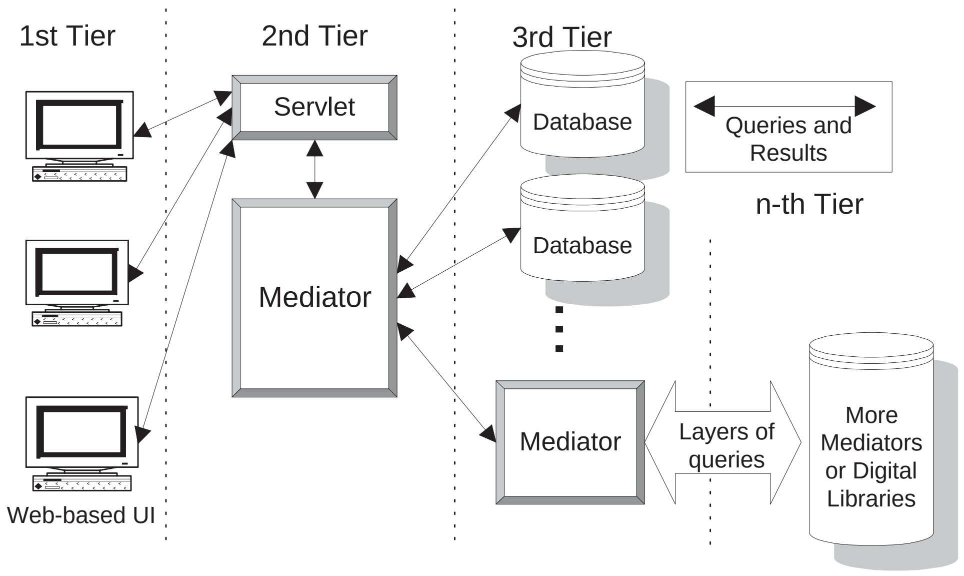


Figure 6.2: The architecture of our query system

may further forward queries again to other database engines or mediators. This mechanism forms an n-tier distributed system. As mediator components have to make queries to both database engines and other mediators, we would like those database engines and mediators to have the same generic interface.

In our system, we also use XML for the internal data representation and storage because it works well in a heterogeneous environment. Hence, we use XML-QL [35, 36] as the query language in the whole system, which is a query language dedicated for XML data developed by AT&T. We use news data obtained from local newspapers in our experiments. They are all converted to XML format.

In the practical application of using mediator architecture in a distributed environment, we need to handle some special cases. One is the infinite looping problem: as a mediator may make queries to another mediator, the queries may be transferred from one mediator to another. Eventually, there may be a case that the mediators have formed a cyclic path and the first mediator is being queried by itself. We need some methods to detect infinite looping. One possible approach is to give each query a unique ID, and all mediators keep track of all IDs of those queries that are already submitted but no replies have been received yet. In case there is an upcoming query with the same ID as any one entry in its record, we can tell that an infinite looping has occurred.

The second problem is to avoid having a long waiting time for users, which may be caused by: the connection between some objects may have been broken, or the number of layers that the queries need to traverse may be too many. For the broken connection problem, we simply use a time-out parameter to specify the maximum amount of time that we are willing to wait. For the too many layers of query traversal problem, we simply use a maximum layer parameter to specify the maximum number of layers that we want to go.

6.2.3 The IDL Design of the Mediator System

We are using CORBA for our system infrastructure. To design the interfaces of different components, we use IDL. CORBA IDL is an interface definition language structures for all concepts of the CORBA object model independent of programming languages. Both Query Mediators and Query Database Engines are implementing the same interface in order to make these two objects the same in the view of the users. In our IDL, we only define a common interface called `QueryEngine`. (See Figure 6.3)

We are supposed to provide to the `QueryEngine` a query statement, and it will return to us the answer in String format, which is a XML expression. We have defined only one simple method in `QueryEngine`, i.e. `query()`, which has a XML-QL statement as its argument, and returns a XML string as the result. This can be used in both Database Query Engines and Query Mediators, such that programmer can notice no difference between making a query on them.

```
module QueryEngineApp
{
  struct SysPara
  {
    long qid;
    long timeout;
    short maxlayer;
  };

  interface QueryEngine
  {
    string query(in SysPara para, in string QueryStatement);
  };
};
```

Figure 6.3: The IDL design of our system

Though they only share the same interface, the implementation of `query()` method would be different.

6.2.4 Components in the Query Mediator System

We rely on CORBA technology for building the system infrastructure because CORBA provides a very good infrastructure for designing and implementing applications in a distributed environment. In order to integrate our system into the web environment, we also use Java Servlet technology. Java is used for our implementation, because of its portability. As we have mentioned before, both the Query Database Engine class and the Query Mediator class are implementing the `QueryEngine` interface. We have named these two classes as `QueryDB` and `QueryMed` respectively.

A `QueryDB` object is directly connected to the data source. A caller can call the method `query()`, and this method will take the query statements (XML-QL statements in our implementation) as the argument and search for the XML document specified, then it will return the result to the caller in a stream of

XML string. We have the `QueryDBServer` object as the server for creating a `QueryDB` object, and registering it to the CORBA name service. The server is also ready to set up multiple threads to support multiple requests on a `QueryDB` object at a time. This server should be started at command prompt.

`QueryMed` object is the Query Mediator which forwards query statements to other mediators or database engines. Its implementation is more complicated than `QueryDB`. Other than the `QueryEngine` interface, `QueryMed` also implements another interface, `QueryMediator`, shown in Figure 6.4. Methods of this `QueryMediator` interface cannot be called by other distributed objects, but can only be called by Query Mediator Server objects, which contain the `QueryMed` objects and located at the same host with them.

```
public interface QueryMediator
{
     public QueryEngineApp.QueryEngine[] qelist();
     public void qelist(QueryEngineApp.QueryEngine[] arg);
     public void append_result(String res);
}
```

Figure 6.4: `QueryMediator`, another interface that `QueryMed` Class implemented

In a `QueryMed` object, the attribute `qelist` would store all the `QueryDB` objects and `QueryMed` objects which it will further search for. And `query()` will start a thread for each `QueryDB` or `QueryMed` object and the thread will take the XML-QL query statement as argument and pass it to its corresponding object in qelist by calling their `query()` method. Then, when all these objects have returned the XML result back to the threads, they will call the `append_result()` method of the parent `QueryMed` object, `query()` will further organize and integrate the results into a single XML file stream and then return it to the caller.

`QueryMedServer` object is similar to `QueryDBServer` object, which will create a `QueryMed` object to handle queries. It will also bind the list of query engines (`QueryDB` and `QueryMed` objects) from CORBA services and can set up multiple threads to support multiple requests at the same time.

Both the database and mediator need to use a configuration file to configure the objects before start up. The configuration file would contain the following attributes: CORBA name server location, CORBA name server port, Object name used for registering in CORBA name server, log file name, and for `QueryMed` object, it also needs the list of `QueryMed` and `QueryDB` objects for distributing the queries.

With `SysPara` object as the parameter of `query()`, we can detect infinite loops and avoid long waiting. The `qid` in `SysPara` is a unique number to identify a query. This number consists of the system time when the user generates the query, the IP address of the user's machine, plus a four-byte random number. As described before, when a mediator needs to call other mediators or database engines, it has to pass this parameter to them by using the newly modified `query()` method interface. The mediator itself will keep track of all IDs of those queries that are already submitted but no replies yet. In case there is an upcoming query with the same ID as any one entry in its record, we can tell an infinite loop has occurred. When an infinite loop is detected, that query mediator will simply do nothing and return an empty string to the caller.

`maxlayer` states the maximum layer that the query can travel onwards. When that value is passed from one mediator to another mediator or database engine, the value will decrease by one. The query will stop being forwarded when the `maxlayer` value becomes zero. `timeout` states the maximum time in milliseconds that a mediator or database engine can wait. When that value is passed from one mediator to another mediator or database engine, the value will be decreased by the estimated processing time of that mediator itself. The estimated time is calculated by the statistic of previous connections and queries. The query will stop being forwarded when that value becomes zero.

6.3 Helping the Mediator System to Expand Across the Firewalls

We use CORBA to implement our mediator query system. Though CORBA is a very good architecture for distributed systems, we still meet some difficulties in achieving a real scalable query system, because the common use of firewalls will block CORBA IIOP communication. Here, we apply our mechanism with using XML and Java Servlets to expand our system across firewalls.

6.3.1 Implementation

We now have two mediator query systems as above, and there is a firewall separating them. To enable their communication, the `QueryMed` object must be able to be called by an object (say, another mediator object) from another enclave outside the firewall.

In our implementation, the `QueryMed` object that would be called by outside is associated with a Servlet component. The Servlet component forwards the requests from outside to the `QueryMed` object immediately, thus the `QueryMed` object can accept HTTP requests from outside. We use TOMCAT Servlet engine [37] in our implementation.

On the client side (caller side), we have created a new class, `HttpGateway`, which is the Shadow Mediator object and is used to connect to the Servlet component of the target mediator. `HttpGateway` class implements the same interface, i.e. `QueryEngine` interface, as the `QueryMed` mediator object does. Besides, `HttpGateway` also implements another interface, `HttpQueryGateway`, for its special need. This interface is shown in Figure 6.5.

```
public interface HttpQueryGateway
{
    public String medURL();
    public void medURL(String U);
}
```

Figure 6.5: `HttpQueryGateway`, another interface that `HttpGateway` Class implemented

The `medURL()` method in the interface is used to specify the URL, or the IP address of the target mediator, which is located in another CORBA enclave. This methods should be invoked by its server only, which contains the `HttpGateway` at the same host.

If a mediator wants to call another mediator located at another CORBA enclave, it only needs to call the corresponding `HttpGateway` object. (Actually, that mediator can treat that `HttpGateway` object as the real target mediator object.) The `HttpGateway` object will convert all the necessary parameters into XML format, and then send the request message to the target mediator by HTTP. The target mediator has a Servlet component and will receive the HTTP calls. It then converts the XML parameters back to their original format.

We can summarize the procedures for communication by referring to the scenario shown in 6.6. The scenario is that Mediator `M1` wants to make a query to Servlet component `SC` of the mediator `M2` in another CORBA enclave. The procedures are:

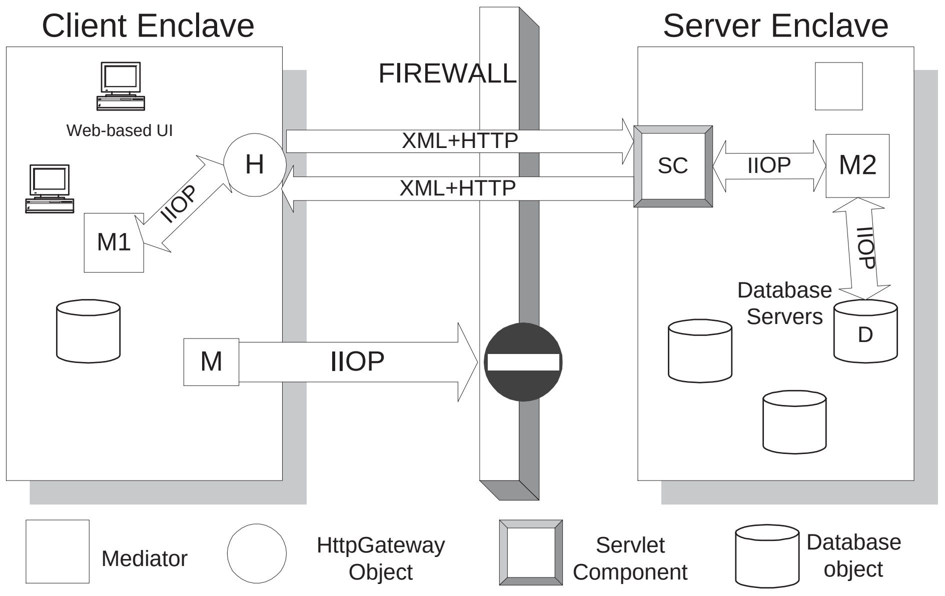


Figure 6.6: The architecture of our query system

1. Mediator `M1` calls `HttpGateway` object `H` with ordinary IIOP connection.
2. `H` converts the IIOP calls to HTTP calls with parameters converted into XML format.
3. The Servlet component, `SC`, of the target mediator gets the HTTP calls from `H` and converts them back to ordinary calling to the target mediator, `M2`.
4. `M2` keeps on calling other Database object `D`, the result is returned to `M2`, and `M2` further returns it to `SC`.
5. `SC` converts the result in XML format, and returns it with `HTTP` calls to `H`.
6. `H` returns result back to Mediator `M1` by using ordinary IIOP return method.

```
<request>
    <QueryEngine type="interface">
        <query type="operation">
            <parameter ref="in" order="1">
                <SysPara>
                    <long name="qid">3984982418240339</long>
                    <long name="timeout">2000</long>
                    <short name="maxlayer">3</short>
                </SysPara>
            </parameter>
            <parameter ref="in" order="2">
                <string name="QueryStatement">
                     where <news>$B</news> in "database.xml"
                     <keyword>satellite</keyword> in $B
                     construct <result> $B </result>
                </string>
            </parameter>
        </query>
    </QueryEngine>
</request>
```

Figure 6.7: An sample request message in XML for calling a mediator object

We have described that parameters are converted to XML format for transmission. Here shown in Figure 6.7 is a sample of such XML request messages with parameters embedded. Figure 6.8 shows a typical response message in XML format. We use tags to state the objects that are being called, the method being invoked, the required parameters and their types, and the values of those parameters.

We can see that both simple data types (like `String` type variable of XML Query Statement) and complicated class objects (like the `SysPara` class of other enhancement parameters) can be well represented by XML. Basically, it is believed to be able to handle all kinds of data structures because of XML's semi-structured nature.

```
<response>
   <QueryEngine type="interface">
      <query type="operation">
         <return>
            <string>
               <news> <source>South China Morning Post
               </source> <date> <day>15</day><month>4
               </month> <year>2000</year> </date> <title>
               Press warning appro priate, says Beijing
               </title> <content>Beijing yesterday defended
               remarks made by senior SAR-based official
               Wang Fengchao that local media should avoid
               reporting separatist views.</content> </news>
            </string>
         </return>
      </query>
   </QueryEngine>
</response>
```

Figure 6.8: An sample response message in XML returns from a mediator object

6.3.2 Across Heterogeneous Systems with DTD

To achieve a scalable system, we need to deal with the heterogeneity of different local systems. We set up some standard formats for different systems to follow in order to communicate with other systems. We need two standards, one is structure of data, and another one is the interface of the system components. If the structures of data cannot be compromised, we will have confusion of communication. If the interfaces cannot be compromised, we even cannot invoke other components of the system. Both important information can be obtained from CORBA IDL files.

To reach a compromise on a standard for data, we use DTD as the grammar book for XML data. This DTD is obtained from the corresponding IDL file by our conversion schema. IDL gives an interface for programmer to develop objects that have the same interface. But IDL itself is not enough, as for parameters passing with using XML and HTTP, we also need to define the parameter format in XML by DTD. The DTD for parameters is shown in Figure 6.9. Hence, different systems can follow the DTD and understand the parameter formats. By following all those mentioned, we can achieve a scalable query without any firewalls or heterogeneous systems problems.

```
<!-- For Request Messages -->
<!DOCTYPE request [
    <!ELEMENT QueryEngine (query)>
        <!ATTLIST QueryEngine type (#CDATA)>
    <!ELEMENT query (parameter*)>
        <!ATTLIST query type (#CDATA)>
    <!ELEMENT parameter (SysPara | string)>
        <!ATTLIST parameter ref (#CDATA)>
        <!ATTLIST parameter order (#CDATA)>
    <!ELEMENT SysPara (long,long,short)>
        <!ATTLIST SysPara name (#CDATA)>
    <!ELEMENT long (#CDATA)>
        <!ATTLIST long name (#CDATA)>
    <!ELEMENT short(#CDATA)>
        <!ATTLIST short name (#CDATA)>
    <!ELEMENT string (#CDATA)>
        <!ATTLIST string name (#CDATA)>
]>


<!-- For Response Messages -->
<!DOCTYPE response [
    <!ELEMENT QueryEngine (query)>
        <!ATTLIST QueryEngine type (#CDATA)>
    <!ELEMENT query (return)>
        <!ATTLIST query type (#CDATA)>
    <!ELEMENT return (string)>
    <!ELEMENT string (#CDATA)>
]>
```

Figure 6.9: The DTD for the parameter passing of simulated calls

```
module QueryEngineApp
{
  struct SysPara
  {
    long qid;
    long timeout;
    short maxlayer;
  };

  interface QueryEngine
  {
    string query(in SysPara para, in string QueryStatement);
    void subscribe(in QueryEngine qe, in string topic);
    void notify(in string newContent);
  };
};
```

Figure 6.10: The IDL design of our system

6.4 Adding the Callback Feature to the Mediator System

To better help the users in obtaining the information they need, one important feature of modern information systems is allowing users to specify some topics of information they want to subscribe. Whenever there is an update of the specified information, the digital library can inform the subscribed users immediately. This feature requires callbacks.

To allow callbacks, we add two methods to the `QueryEngine` interface. One is `subscribe()`, which takes a `string` as parameter to specify the topic of information that the caller wants to subscribe; and an object with `QueryEngine` interface as another parameter to specify the object requests for callback. To be generic, all user interface objects, mediator objects, shadow objects, and database objects would implement this interface. Figure 6.10 shows the new IDL file.

A conceptual diagram of our system mechanism for callbacks is shown in Figure 6.11. And below is the step-by-step desciption of the procedures:

1. Mediator `M1` calls `HttpGateway` object `H1` with ordinary IIOP connection. `M1` also puts itself as one of the parameter in `subscribe()` method. (Same invocation method as calling the target mediator for normal callback)
2. When `H1` observes that it is a callback invocation, it generates a Servlet component (`SC1`), which is assoicated with `M1`, immediately.
3. `H1` sends the IIOP calls to HTTP calls with parameters converted into XML format. The information of `SC1` will also be sent to the server side. These information are embedded into the parameter tag as attributes.
4. When `SC2` observes that it is a callback invocation, it generates a shadow client object, `H2` (shadow of `M1`), immediately. `H2` is initialized by the information of `SC1` (such as IP address, port number).
5. `SC2` will invoke `M2`'s `subscribe()` method substituting `M1` by `H2` in the parameter position, such that `M2` will invoke `H2` when callback is needed.
6. Whenever there is a callback, `M2` calls `H2` `notify()` and H2 will send the request to `SC1`. Finally, `M1` `notify()` method will be invoked by `SC1`.

6.5 Connecting our CORBA System with Other Environments

Merging only CORBA systems would be a great limitation for system integration. Here, we demostrate how we apply our mechanism to allow CORBA objects, DCOM objects and Java RMI objects to be able to call each other.

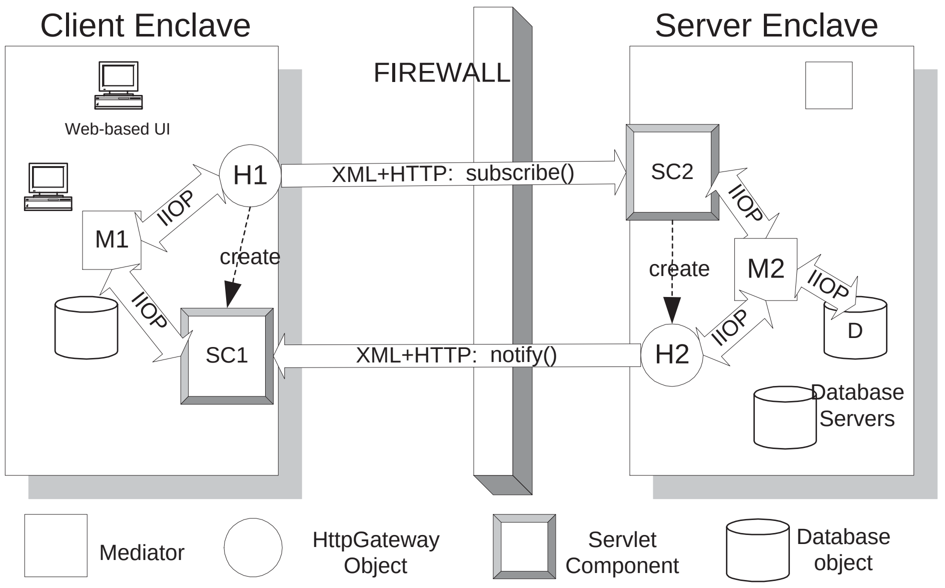


Figure 6.11: Mechanism for supporting callbacks in our query system

Our target is to expand our system across heterogeneous distributed environments. To make the whole system to be more generic, we carefully design the MIDL of the DCOM system and interfaces of RMI components to be very similar to our existing CORBA system, such that calling the DCOM mediators or Java RMI mediators would have no difference as calling the CORBA mediator objects. For simplicity of the example, we use the CORBA IDL in Figure 6.3 to develop our DCOM system and Java RMI system.

6.5.1 Our Query System in DCOM

Our DCOM system is developed on Windows 2000 operating systems, with using Microsoft Visual J++ for implementation. Our implementation is based on the MIDL file shown in Figure 6.12. From the MIDL, we can find out that it is basically the same as the IDL of CORBA. One thing worth to point out is `query()`, the return value is specified in the parenthesis with marking as `retval`. It is because the default return type in DCOM object is `HRESULT`, hence the real return value is defined inside the parenthesis.

```
import "oaidl.idl";
import "ocidl.idl";

typedef struct SysPara
{
    long qid;
    long timeout;
    short maxlayer;
}SysPara;

[ uuid(AC6EDE04-ADF2-4324-BB8C-B350295BFD5E) ]
interface ICOMQueryEngine : IDispatch
{
    HRESULT query([in] SysPara para,
                  [in] char * queryStmt
                  [out, retval] char ** rtn);
};

[ uuid(AC6EDE03-ADF2-4324-BB8C-B350295BFD5E), version(1.0) ]
library QuerySystemLib
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
    [
        uuid(AC6EDE02-ADF2-4324-BB8C-B350295BFD5E),
    ]
    coclass QueryEngine
    {
        [default] interface ICOMQueryEngine;
    };
};
```

Figure 6.12: The MIDL file for the query system in DCOM enclave

6.5.2 Our Query System in Java RMI

Our Java RMI system is developed in the Unix environment, but it can be run in any operating systems. Our implementation is based on the Java interface definition files shown in Figure 6.13. They are basically the same as the IDL of CORBA system.

One special thing to point out is the struct type of `SysPara` in IDL. As Java interface definition does not support struct type, a new class of `SysPara` is defined instead. But it is mapping to the same XML schema as struct type in XML.

```
/* SysPara.java */
public class SysPara implements java.io.Serializable{
    public long qid;
    public long timeout;
    public short maxlayer;

    public SysPara() {
        qid=-1;
        timeout=-1;
        maxlayer=-1;
    }
}
```

```
/* QueryEngine.java */
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface QueryEngine extends Remote
{
    String query(SysPara para, String queryStmt)
          throws RemoteException;
}
```

Figure 6.13: The DTD for the parameter passing of simulated calls

6.5.3 Binding Heterogeneous Systems

With the interface definition files of DCOM system and Java RMI, the same XML schema can be mapped from those interfaces. Hence, the mediator objects of all systems would have the same interface for calling, hence the scalability of the binded system is greatly increased. Figure 6.14 shows how the mediator objects in heterogenous distributed environments communicate with objects in other enclaves. A common XML schema is the key part to achieve this communication.

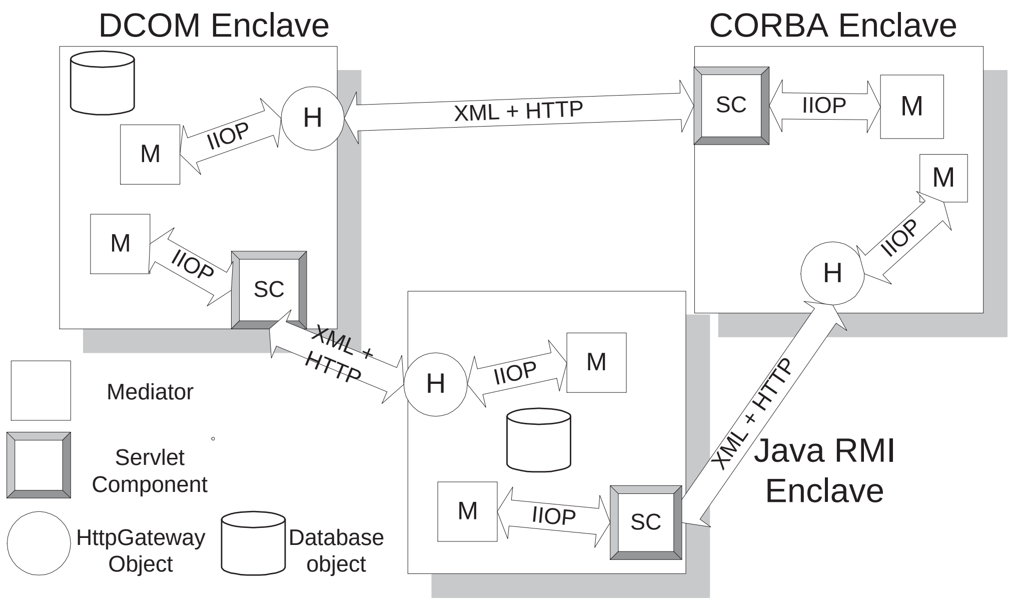


Figure 6.14: Query system in heterogeneous environments with our mechanism

In fact, this is for the demostration of a generic query system across heterogeneous distributed systems with applying our mechanism in it. By matching the newly designed interface definition with the existing XML schema, a highly generic and scalable mediator-based query system is achieved. In normal way of system integration, we use the interface definition files to generate the XML schema for data transmission, but not using the XML schema to design the interface definition.

Chapter 7

Evaluation

7.1 Performance Statistics

In Chapter 6, we have described our implementation of a mediator-based query system and demonstrated how we applied our mechanism on a practical application. Now, we are going to evaluate its performance in this chapter. Based on the original architectures of CORBA, DCOM or Java RMI, we have provided some add-on components in them for connecting to other enclaves across firewalls and beyond heterogeneous environments. As some add-on components are added in it, it would be important to measure if those components are the burdens of the system.

We tested our system in an environment with general workload such that we can ensure that our results would not be influenced by other factors, like network congestion. The query system was installed in a number of personal computers with Pentium III 500MHz CPU and 10 Mbps network connection. For the objects in CORBA enclave, they are implemented in Java, and tested with Linux platform; for the objects in DCOM enclave, they are implemented in C++ and tested with Windows 2000 platform; for the objects in Java RMI enclave, they are located in Linux platform. For each query in our tests, the

Table 7.1: Performance Statistics of the Query System Described in Chapter 6

Effective Process Time	**Milliseconds**
Mediator Objects (excluding waiting time for the return of query results and connection setup time)	20 - 80
Database Objects	180 - 800
IIOP Communications with CORBA enclave (connection within LAN)	10 - 100
Shadow Client or Server (excluding waiting time for the return of query results and connection setup time)	20 - 100
Servlet Components with Tomcat Servlet Engine [37] (excluding waiting time for the return of query results)	120 - 250
HTTP communications towards other enclaves (connection in WAN)	240 - 2200

system would return a few hundred bytes of text stream information. As we wanted to see the overhead of using our approach, we kept track of the time used in each event in different objects. With similar composition of components in each enclave, different enclaves would have similar performance. We now focus on the information gathered in CORBA enclave for analysis. The results are shown in Table 7.1.

From the statistics, we can figure out the following characteristics of our system:

- Mediator objects are light-weighted objects when compared to Database objects, as they need not perform complicated computation but only forwarding queries and merging the results.
- The performance of our add-on components are somehow similar to those light-weighted objects. This is because our add-on components are only converting the method calls into XML messages, or vice versa, which do not involve complicated computation.

- The most time-consuming part of the whole process is the Internet connection, which is unavoidable in the communications in a worldwide area.
- When compared to the time for Internet connection, time spent on our add-on components would not be significant.

We can conclude that our add-on components are light-weighted and would not be the burdens of our overall system. Though they need some time for processing, the time they used would be neglible when compared to the long Internet transmission time. So, our add-on components would not make a great influence on the whole system performance.

7.1.1 Overhead in other methods

Though our add-on components bring overhead to the systems, the overhead is light-weighted. Moreover, other existing methods also bring overhead.

Our mechanism can substitute for the use of DCOM/CORBA bridging appications. We want to compare the overhead of these applications and our mechanism, so we use OrbixCOMet as an example to evaluate its overhead. Fatoohi et al have done some experiments to evaluate the performance of OrbixCOMet [38], Table 7.2 shows the result.

Table 7.2: Performance Evaluation of OrbixCOMet

	CORBA Server	**DCOM Server**
CORBA Client	2.6 msec (without OrbixCOMet)	250 msec (with OrbixCOMet)
DCOM Client	3.8 msec (with OrbixCOMet)	1.2 msec (without OrbixCOMet)

In their experiments, the server was always located in a different host as the client. We can see that the use of the bridging application always give overhead for protocol conversion. When a DCOM Server and a CORBA Client are used, the overhead is extremely large (250 msec) in their current implementation. But for another configuration, the overhead is still around 50%. It is not a direct comparison to our apporach, as the object components being tested were different. But we can see that other bridging solutions also post overhead to the overall system, so the overhead of our mechanism is still acceptable.

7.2 Means for Enhancement

7.2.1 Connection Performance of HTTP

In order to provide better performance when applying our mechanism, we are using HTTP 1.1 [39, 40] instead of the HTTP 1.0 standard for the HTTP connections. One of the problems with the standard HTTP 1.0 is that a new TCP connection is required for each resource requested by the client, e.g. each time the client wants to invoke a server method, one TCP request is needed. This is inefficient as the initialization of each TCP request would require some overhead for connection establishment, because TCP is connection oriented. So, if this overhead is repeated for every request made, the system would be very inefficient, especially when requests are frequently called.

HTTP 1.1 allows persistent TCP connections. Once the connection is established, it will not terminate immediately when the request is finished. The connection is still maintained after one request/respond communication is ended. Hence no more overhead for connection establishment is needed for the ongoing requests. Thus, the time for overhead is reduced and we can have better performance of the overall system.

7.2.2 Transmission Data Compression

Our XML messages used in transmission are text streams embedding in HTTP calls, which are much longer than ordinary binary-based IIOP, or DCOM calls in message sizes. That means longer time is needed for transmission of those XML messages.

If the XML messages are compressed before transmission, the transmission time would be greatly reduced. The processors are getting faster nowadays, on-the-fly compression and decompression at the client and server sides should not pose too much overhead. In general, the need to compress XML data is great as all the transmission contents are serialized to text-based data and there are messages sent through the network for every request call. Moreover, XML compresses extremely well due to the repetitive nature of the tags used to describe the structure of the data.

As mentioned before, we are using HTTP 1.1 for HTTP communication. In HTTP 1.1, compression is standard for servers and clients, and XML automatically benefits from this. Currently in HTTP 1.1 standard, `gzip` is used for compression, which can provide compression rate of around 5% to 30% in some XML testing data. Thus the time for transmission can be reduced greatly.

7.2.3 Security Concern

As now we are establishing connections between the objects in two separated enclaves, we would like to have a guarantee in three security issues. First, can all objects in a single enclave be invoked by others outside the enclave, or only some dedicated objects can be invoked? Second, is it possible to verify which outside object is calling the objects inside the enclave? Third, can the transmission messages be encrypted such that others cannot steal and read the messages?

For the first problem, the use of Servlet components can help to ensure that only those objects which are prepared for being called can be invoked by outsiders. It is because only XML messages are used in communication between two enclaves, and only those objects associated with Servlets components can understand and can be invoked by those XML messages. So, it is quite safe that other objects in the same enclave are protected from being called by outsiders.

For the second and third problems, as we are applying some popular technologies, such as Servlets and HTTP, there are many good methods developed for dealing with those security problems. We can take advantage of HTTP authentication mechanisms as well as Secure Sockets Layer (SSL) [41] for secure channel communications (using secure HTTP connections via HTTPS(Secure Hypertext Transfer Protocol) [42]) to communicate in a way that can prevent eavesdropping, tampering, or message forgery. By using SSL in the communication between the add-on components (Shadow objects and Servlet components), encryption is used after an initial handshake to define a secret key. Symmetric cryptography is used for data encryption, such that the peer's identity can be authenticated using asymmetric, or public key, cryptography. So, basically, we can ensure communication security by HTTPS.

Recently, there are some stronger mechanisms to handle security issue which are dedicated for SOAP. SOAP has some similarities when compared to our mechanism. Therefore, we can also apply their security methods to ours. For example, Damiani et al [43] have suggested a simple, yet powerful and general, technique to enforce access restrictions to SOAP invocations in order to support fine-grained authorizations at the level of individual XML elements and attributes. Moreover, many security work about XML data [44, 45, 46] are also worth referencing for the improvement of our mechanism. Yet, as security is not the main concern of our research, we have not included them in our implementation.

7.3 Advantages of Using Our Mechanism

Our mechanism for communication between distributed systems using XML, Servlets and HTTP calls described in this thesis has certain strengths and weaknesses. We are discussing its pros and cons in this and the coming section. Generally speaking, our mechanism enjoys the following advantages in integrating distributed systems:

- It can solve the incompatible firewall problems of some communication protocols in distributed environments. It provides vendor-independent support. With our mechanism using HTTP, common normal firewalls cannot block the communication between distributed objects in different enclaves, and hence the scalability of system design and construction can be greatly increased.
- It can also solve the incompatible problem of heterogeneous distributed environments. XML can be used as the bridge in connecting heterogeneous distributed protocols, such as connecting CORBA, DCOM and Java RMI systems. Moreover, even if heterogeneous systems are separated by firewalls, they can still communicate with objects in other systems.
- Our mechanism can be applied to a system without modifying the originally existing objects. The newly-added components to the system are transparent to the original objects. Internal objects would not notice the difference between the real target object and the shadow object, thus no special modification or implementation is needed for ordinary internal objects, hence increasing the system transparency properly.

- Systems can maintain good security, as external objects outside the enclave can only call the objects integrated with the Servlet components, hence we can protect other internal objects from being called externally. Moreover, we are exploiting some very common products like Java Servlet or HTTP calls, whose security properties are well developed, such as HTTPS.

- No information loss or distortion, as using XML can represent the information in the transmitted messages well, even when the parameter structures of the invoking calls are complicated. This properly enhances the system interoperability.

- Our mechanism can also be used as a gateway to inter-cooperate with other Web-based applications. As long as the DTD of transmission messages is defined and agreed between both clients and servers, we can include any kind of implementations in the server and the client sides.

These advantages are very important in the integration of distributed systems. The use of firewalls and the heterogeneity of different system environments are the major obstacles of system integration, while our mechanism can provide a solution for that. Providing great transparency and ensuring no data distortion are also very important, as changes to the existing systems may lead to some potential hazards. Security concerns in our mechanism can also be answered by traditional security methods which have been proved to be safe in many real life applications.

7.4 Disadvantages of Using Our Mechanism

Though our mechanism has many advantages for integrating distributed systems, it also has some drawbacks, however. Here listed below are the disadvantages of our mechanism.

- For each request or response to a remote object, we have to use one more Servlet component and one more shadow object between the server side and the client side in our mechanism and thus the system requires extra workload and time for running them. However, these components are light-weighted and would not greatly affect the overall performance. The time for this overhead is also negligible when compared with the average Internet access delay.
- As XML messages are used in communication, the highly-readable XML messages would greatly increase the danger of eavesdropping, tampering, or message forgery. High security level is required for using in the Internet if critical data are used. Fortunately, many traditional security methods can be applied to our mechanism, such as HTTPS.

Actually, these disadvantages are the tradeoffs of some good features. Though Servlet components and shadow objects add extra workload to the systems, they provide great transparency to the existing objects for invoking objects in other enclaves. Also, the use of understandable XML can easily provide an protocol interface to other web-based application by DTD, hence increasing the system interoperability.

Chapter 8

Conclusion

Nowadays, it is frequently required to integrate several information systems to work together in order to provide more information to the increasingly demanding users. Integrating heterogeneous systems is not an easy task, and the situation would be more complicated if we want to integrate systems in distributed environments. There are many major obstacles in integration, such as common use of firewalls, or heterogeneity of distributed environments for different components. In this thesis, we suggest to use XML, Servlets, and HTTP to handle these obstacles and increase the scalability of distributed systems.

The first problem we focused on is the firewall issue. We used CORBA as an example to introduce our mechanism. Between two CORBA enclaves, if they are separated by firewalls, objects are unable to communicate with objects in another enclave as IIOP cannot pass through the firewall. To support IIOP communication to objects in other enclaves, we use HTTP carrying XML messages which contain the information for method calling. XML is semistructured and is flexible to represent the calling parameters and other relevant information.

In our mechanism, we have a `Shadow Server` in the client side, which behaves the same as the target server object. This `Shadow Server` is an ordinary

CORBA object and its responsibility is to convert the method calls to XML messages. On the server side, we have a the Servlet component which parses the received XML messages to an ordinary IIOP method calls. When we receive responses from the server object, Servlet component converts them into XML message and sends them to client side by HTTP. `Shadow Server` on the client side will parse the messages and return the results to the client object.

The mechanism described above can only handle general calls. We then extend this mechanism to support the callback feature. In callback, the server can notify the clients whenever there is an update on the server side, hence the client programs can react to changes with a faster response. This requires both sides to be able to initiate a call, which many CORBA dedicated firewalls cannot handle properly. We can simply handle this problem by using another pair of add-on components in our mechanism: one `Shadow Client` on the server side, and one Servlet component on the client side.

Then we have addressed how we can generate XML messages from Interface Definition Languages in CORBA, and briefly described how we can generate the related source code and components automatically and in a generic way by engaging the interface design (IDL) of a system.

The second problem we focused on is heterogeneity of distributed environments. If we integrate two systems in a heterogeneous environment, they cannot communicate as they have different communication protocols. There are many bridging tools available on the market, but they are not generic, as they use binary streams for bridging, which usually allows bridging between two dedicated environments only. We extend the above mechanism to allow communication among heterogeneous distributed environments by mapping different interface definition languages to the same XML schema, such that they can have a common language to communicate.

We have also presented a real example of applying our mechanism to implement a scalable mediator-based query system. This helps to make our query system to be more scalable across firewalls and across heterogeneous distributed environments.

Lastly, we evaluated the performance of the mediator-based query system in order to measure the overhead of our mechanism. We showed that the light-weighted add-on components used in our mechanism gave extra workload to the system, but the overhead is acceptable as the latency is very small when compared to the Internet latency. Enhancements on performance and security, the advantages and disadvantages of our mechanism were also presented.

We conclude our contributions in the following ways:

- A generic mechanism for distributed objects to communication across firewalls has been proposed;
- An extension of the mechanism to support callback feature has been proposed;
- A schema for mapping CORBA IDL to XML format has been proposed, and a translator for that has also been implemented;
- An extension of the mechanism to support generic remote object calling in heterogeneous environment has been implemented;
- A mediator-based query system has been implemented to demonstrate our work.

Our mechanism is a generic and simple tool for the integration of distributed systems in heterogeneous platforms and across firewalls. With consideration to the overhead, our mechanism is still very suitable to be applied to Internet platforms, which as the workplace for next generation applications.

Bibliography

[1] Wolfgang Emmerich. *Engineering Distributed Objects.* John Wiley & Sons, Ltd, New York, USA, 2000.

[2] A. Leon-Garcia and I. Widjaja. *Communication Networks.* McGraw-hill International Editions, 2000.

[3] Object Management Group, ftp://ftp.omg.org/pub/docs/formal/97-09-01.pdf. *The Common Object Request Broker: Architecture and Specification, Revision 2.1*, August 1997.

[4] Rudolf Schreiner. Corba firewalls: An introduction and analysis. Technical report, Object Security Ltd., http://www.objectsecurity.com/whitepapers/corba/fw/main.html, 1999.

[5] IONA Technologies. *Orbix Wonderwall Administrator's Guide*, June 1999.

[6] Visigenic Software, Inc. *Visigenic Gatekeeper Guide*, February 1998.

[7] Mark Elenko and Mike Reinertsen. Xml & corba. *Application Development Trends*, September 1999.

[8] D. Box. *Simple Object Access Protocol (SOAP) 1.1.* Wide Wide Web Consortium, http://www.w3.org/TR/SOAP/, May 2000.

[9] UserLand Software, Inc, http://www.xmlrpc.com/spec. *XML-RPC Specification*, October 1999.

[10] Financial Toolsmiths AB, http://xiop.sourceforge.net/. *XIOP Homepage.*

[11] N. Brown and C. Kindel. *Distributed Component Object Model Protocol - DCOM/1.0.* http://www.globecom.net/ietf/draft/draft-brown-dcom-v1-spec-03.html, January 1998.

[12] Sun Microsystems, http://java.sun.com/j2se/1.3/docs/guide/rmi/index.html. *Java Remote Method Invocation (RMI).*

[13] IONA Technologies, http://www.iona.com/docs/orbix2000/1.2/comet/html/. *OrbixCOMet Desktop Programmer's Guide and Reference*, 2000.

[14] Sun Microsystems, http://java.sun.com/products/rmi-iiop/index.html. *RMI over IIOP.*

[15] Wide Wide Web Consortium, http://www.w3.org/TR/2000/REC-xml-20001006. *Extensible Markup Language (XML) 1.0 (Second Edition)*, 2000.

[16] D. Martin, M. Birbeck, M. Kay, and B. Loesgen et al. *XML.* Wrox Press, USA, 2000.

[17] F. Boumphrey, O. Direnzo, J. Duckett, and J. Graf et al. *XML Applications.* Wrox Press, USA, 1998.

[18] J. Widom. Data management for xml: Research directions. *IEEE Data Engineering Bulletin 22(3)*, July 1999.

[19] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web: from relations to semistructured data and XML.* Morgan Kaufmann Publishers, San Franciso, USA, 1999.

[20] Sun Microsystems. *Java Servlet Specification Version 2.3*, October 2000.

[21] Sun Microsystems, http://java.sun.com/products/servlet. *Java Servlet Technology.*

[22] Alexander Nakhimovsky and Tom Myers. *Professional Java XML Programming with servlets and JSP*. Wrox Press, December 1999.

[23] Reuven M. Lerner. At the forge: Introducing soap. *Linux Journal*, 2001(11), 2001.

[24] Ron Ben-Natan. *CORBA: A Guide to Common Object Request Broker Architecture*. McGraw-Hill, 1995.

[25] Randy Otte, Paul Patrick, and Mark Roy. *Understanding CORBA*. Prentice Hall, 1996.

[26] Michael Rosen, David Curtis, and Dan Foody. *Integrating Corba and Com Applications*. John Wiley and Sons, October 1998.

[27] Rubin Brain. *Understanding DCOM*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[28] Sing Li and Panos Economopoulos. *Visual C++ 5 ActiveX COM Control Programming*. Wrox Press, Canada, 1997.

[29] Gopalan Suresh Raj. A detailed comparison of corba, dcom and java/rmi. Technical report, Web Cornucopia, http://www.execpc.com/ gopalan/misc/compare.html, 1998.

[30] Sun Microsystems, http://java.sun.com/docs/books/tutorial/rmi/index.html. *Java RMI Tutorial*.

[31] Object Management Group. *Java to IDL Language Mapping Specification, Version1.1*, June 1999.

[32] Object Management Group. *IDL to Java Language Mapping Specification, New Edition*, June 2001.

[33] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer Vol 25 No 3*, March 1992.

[34] Hui Lin, Tore Risch, and Timour Katchaounov. Object-oriented mediator queries to xml data. In *Proc. of 1st Intl. Conf. on Web Information Systems Engineering, (Vol 2)*, pages 38–45, Hong Kong, China, June 2000. ACM Conference.

[35] D. Florescu, A. Deutsch, A. Levy, D. Suciu, and M. Fernandez. A query language for xml. In *Proceeding of Eighth International World Wide Web Conference*. W3C, 1999.

[36] AT&T, http://www.research.att.com/ mff/xmlql/. *XML-QL: A Query Language for XML.*

[37] The Apache Software Foundation, http://jakarta.apache.org/tomcat. *Jakarta Project Subprojects: Tomcat.*

[38] R. Fatoohi, V. Gunwani, Q. Wang, and C. Zheng. Performance evaluation of middleware bridging technologies. In *Proc. of 2000 IEEE Int. Symp. on Performance Analysis of Systems and Software (ISPASS2000)*, pages 34–39, Austin, TX, USA, April 2000.

[39] J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1 (RFC).* The Internet Society, June 1999.

[40] H. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. Lie, and C. Lilley. *Network Performance Effects of HTTP/1.1, CSS1, and PNG.* W3 Consortium, http://www.w3.org/Protocols/HTTP/Performance/Pipeline.html, June 1997.

[41] A.O. Freier, P. Karlton, and P.C. Kocher. *The SSL Protocol - Version 3.0.* http://ftp.nectec.or.th/CIE/Topics/ssl-draft/INDEX.HTM, March 1996.

[42] Sun Microsystems. *HTTPS support in Java Plug-in through JSSE*, May 2001.

[43] E. Damini, S. Vimercati, S. Paraboschi, and P. Samarati. Fine grained access control for soap e-services. In *The Tenth International World Wide Web Conference*, pages 504–513, Hong Kong, May 2001. International World Wide Web Conference Committee.

[44] E. Damini, S. Vimercati, S. Paraboschi, and P. Samarati. Xml access control systems: A component-based approach. In *Proc. of the 14th IFIP 11.3 Working Conference in Database Security*, Amsterdam, The Netherlands, August 2000. International World Wide Web Conference Committee.

[45] M. Kudo and S. Hada. Xml document security based on provisional authorization. In *Proc. of the 7th ACM Conference on Computer and Communication Security*, pages 87–96, Athens, Greece, November 2000. International World Wide Web Conference Committee.

[46] J. Paajarvi. *XML Encoding of SPKI Certificates*. Internet Draft.

Publications

1. Wing Hang Cheung, Michael R. Lyu, Kam Wing Ng. Integrating Digital Libraries by CORBA, XML and Servlet In *Proc. of The First ACM+IEEE Joint Conference on Digital Libraries (JCDL'01)*, Roanoke, VA, USA, June, 2001.

2. Wing Hang Cheung, Michael R. Lyu, Kam Wing Ng. Tunneling Across Firewalls by Using XML and Servlet: An Experiment on CORBA. In *Proc. of The 2nd International Conference on Internet Computing (IC'2001)*, Las Vegas, NV, USA, June, 2001.

3. Wing Hang Cheung, Michael R. Lyu, Kam Wing Ng. A Scalable Mediator System beyond Firewalls using CORBA, XML, and Java Servlets. In *Proc. of the joint meeting of the 5th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2001) and the 7th International Conference on Information Systems Analysis and Synthesis (ISAS 2001)*, Orlando, FL, USA, July, 2001.