Java as Distributed Object Glue[1]

Konstantinos Raptis, Diomidis Spinellis, Sokratis Katsikas
Department of Information and Communication Systems
University of the Aegean
GR-83200, Karlovassi
Samos Island, Greece
{krap, dspin, ska}@aegean.gr
Fax No: +30 - 273 - 82009

Abstract: An important aspect of research on software objects, components, and component-based applications concerns their interoperation. When there is a need for two or more software components, based on different technologies, to interoperate the mission target is to make the components hide the fact that the other components are functioning under a different technology without changing their characteristics and behavior. In this paper we describe basic strategies for bridging the gap between the three basic middleware remoting technologies (CORBA, DCOM, and RMI) and present our approach for a Java-based Object Mediator architecture.

Keywords: software objects, components, middleware, bridge, mediator.

Introduction

The need for interaction between software components from different vendors, running on different machines, and on different operating systems led to the specification of middleware remoting models. The Object Management Group's Component Object Request Broker Architecture (CORBA)[1], Microsoft's Distributed Component Object Model (DCOM)[2], and Sun Microsystems' Remote Method Invocation (RMI)[3] are three models that enable software components with different descent to work together.

For software components to be able to interact with each other they must comply with the rules of their underlying middleware technology[4]. However, it is difficult, if not impossible, for two components, hosted on different component architectures, to interact with each other. The incompatibility problems stem from the differences of the underlying models and the way they present and use the software components.

When there is a need for two or more software components, based on different technologies, to interoperate the mission target is to make the components hide the fact that the other components are functioning under a different technology without changing their characteristics and behavior.

Our research work concerns the exploitation of the Java language as a tool for the creation of a mediation mechanism for bridging together the three most widespread commercial middleware remoting technologies; CORBA, DCOM, and RMI. We are using the Java language as a "general purpose object glue". Our target is to allow a server object, which may be CORBA or DCOM or RMI compliant, to expose its methods to CORBA-, DCOM-, and RMI-based clients.

As a component's instance is typically an object and anything applying to objects has also apply on components, in the next paragraphs our discussion will focus on software objects. Before the presentation

[1] In World Computer Congress 2000, Beijing, China, August 2000. International Federation for Information Processing.

of our research findings we present the main attempts for bridging CORBA, DCOM and RMI technologies.

Bridging Distributed Objects

Nowadays, discourse about software objects, components, and component-based applications is about ActiveX controls, JavaBeans (JBs), Microsoft Transaction Server (MTS), and Enterprise JavaBeans (EJBs) and how they can interoperate each other. All the above are not independent models; they all depend on the underlying architecture that each has as a basis for its construction.

In the next paragraphs of this section we provide some of the attempts that have been done for bridging CORBA, DCOM, and RMI, the most widespread commercial middleware remoting technologies.

CORBA-DCOM Bridge

CORBA and DCOM, as extension of COM, are the two most important middleware remoting technologies. Their importance stems from their ancestry. CORBA is child of the Object Management Group an association including Sun Microsystems, Compaq, Hewlett-Packard, IONA, Microsoft and others, while DCOM comes from Microsoft which has the highest share in the desktop operating system market. Although COM and its extension DCOM are built-in in Microsoft's OSs, the widespread adoption of Microsoft's OSs and the development of programming languages which support rich COM/DCOM frameworks, led to the production of many components based on Microsoft's architecture. On the other hand, the fact that the OMG provides CORBA as specifications for ORBs instead of a product led many companies to create their own CORBA compliant request brokers providing the developers and the users with a range of ORBs capable to satisfy various demands.

The OMG understanding the need for bridging their differences decided to include as part of its updated revision 2.0 of CORBA architecture and specification the Interworking Architecture which is the specification for bridging OLE/COM and CORBA. The Interworking Architecture addresses three points:

- Interface Mapping. As both models use IDLs to define the interfaces and as any object is exposed by its interface, there must be a mapping between them in order for a CORBA object to viewed as a COM object and vice versa.
- Interface Composition Mapping. While CORBA supports multiple interface inheritance, COM provides single inheritance. In order for the bridge to be successful there must be a map from CORBA's multiple inheritance to COM's single inheritance and vice versa.
- Identity Mapping. This specification is concerned with the mapping between the different Interface IDs that are used by CORBA and COM.

The OMG does not provide an implementation of a COM/CORBA bridge but only specifications. The implementation belongs to commercial companies which have released many bridge tools, compliant with OMG's specification. Some of these products are PeerLogic's COM2CORBA, IONA's OrbixCOMet Desktop, and Visual Edge's ObjectBridge.

RMI-CORBA Bridge

The widespread deployment of Java language and its use in the development of Web-based applications in combination with the presence of CORBA as a mature middleware technology quickly led to the combination of these two. Although Sun provided its own model for remote java-object interactions, the Java Remote Method Protocol (RMI), the effective combination of Java language with the CORBA architecture led OMG and Sun to think for the marriage of RMI with CORBA. According to Sun[5] the Java developers would be able to use RMI-based Java objects and interoperate with CORBA-based remote objects. In June of 1999, Sun and IBM announced the release of the RMI architecture over IIOP protocol. According to RMI-IIOP any RMI-based object can be accessed by a CORBA one and vice versa. In order for this goal to be achieved, OMG has adopted two standards for Object By Value and the Java-to-IDL mapping. Moreover Sun made some changes in RMI to work under the new requirements.

Apart from the adoption of IIOP as RMI's alternative protocol, a new version of the rmic compiler has been developed in order to generate IIOP stubs/ties and IDL interfaces. Furthermore, the use of new commands and tools, for example for naming and storing in registry the RMI-objects and for ORB activation, is required in order for the RMI-IIOP-based objects to be accessed by CORBA-based.

DCOM-RMI Bridge
In this field the attention is focused on the attempts for integrating Java language and COM and on the bridging of JavaBeans with ActiveX.

Until recently, Microsoft supported COM/DCOM with its own edition of Java language, Visual J++. In order for users of the native Java language to use the COM technology, Microsoft supports the Microsoft Visual Machine (MSVM). According to Microsoft[6], the MSVM provides all the mechanisms that are required for a Java object to be viewed like a COM object and for a COM object to be accessible like a Java object.

As for JavaBeans – ActiveX bridging, a number of companies, including Microsoft and Sun, provide bridges for JavaBeans and ActiveX components to interoperate with each other taking advantage of the JavaBeans architecture flexibility in connection with protocol usage. Moreover, a lot of the work concerns the possibility of a JavaBean component to be used in ActiveX-component based environments like Microsoft Office or Visual Basic applications.

Java as Tool for Object Bridging

In the previous section we presented the basic attempts for bridging the CORBA, DCOM, and RMI middleware remoting technologies. All three of them have a common point; they support the Java language for building distributed objects. This common point led us to examine how we could exploit the Java language in order to bridge distributed objects that were compliant with these three technologies.

From the view of the software developer any software object is constructed following the rules of its native language and the rules of the technology that it conforms to. The basic points that make the objects comply with one of the three technologies are the object interfaces, the object naming, the object storage, and the object reference. These characteristics are not collide i.e. an object may has two interfaces which expose the same methods but in different technologies or it can be stored with two different ways.

Suppose we have two objects based on different technologies, a server object which exposes some methods and a client object which needs the exposed methods. For these objects to communicate a mediation mechanism is needed to transform one technology to the other and promote the client's request to the server and vice versa. For the interaction to succeed the mediator must have a double role. From the client's point of view the mediator must operate as if it was a client's technology server object and from the server's point of view it must operate as if it was a server's technology client object. Moreover, the mediator must be constructed in a common form understandable from the different technologies.

The programming language that could be used for the construction of the mediator must be supported from all underlying technologies. The fact that the Java programming language is supported by all the three technologies led us to use the Java as the programming tool for the mediator's construction. Besides, using the Java language we have the advantage that our mediator can function over any operating system. Moreover, the acceptance of Java as a suitable language for network applications ensures its support from various middleware remoting technologies.

The mediator's architecture must support its double role as the client's technology server object and as the server's technology client object. For this goal to be achieved the mediator must include all the necessary attributes of bridged technologies. Our work on this goal targets the construction of a virtual server which operates under the rules of the client's technology. Therefore, the mediator must implement a client's technology interface through which it exposes the real server's methods to the client. Moreover, the mediator must be exposed and stored through client's technology appropriate rules.

Up to this point, our architecture follows the structure of a distributed client/server application. The difference stems from the fact that our mediator does not really implement the exposed methods. In reality, our mediator forwards the client's request to the real server object. For our mediator to be able to forward the request to the server object it must act as a server's technology native client which requests the server's methods. That is, our mediator must support all the appropriate server's technology attributes for a client object. It must be able to understand the methods exposed by the server's technology interface, to declare the server object using its technology naming and storage methods and to request the methods using the technological appropriate object reference. All these attributes are included in the mediator through which it has the ability to forward the request like a native client.

Using the Java programming language all the above client and server attributes are declared in our mediator by using common language expressions. Moreover, the fact that there are mappings between the Java language and the different interface definition languages help us to declare the method's attributes in a common form.

From a general point of view our mediator's operation must fulfil four basic demands:

- The mediator must comply with the client's and server's side architectures. Our mediator does comply with this demand by including all the necessary attributes in order to act like a client's technology virtual server and like a server's technology virtual client.
- The mediator must hide the technological differences on interfaces. Our mediator uses his virtual client attributes to recognize and understand the interface through which the server object exposes its methods. Moreover the construction of a new interface, following the principles of client's technology, and the implementation of this interface by the mediator provides the client with the illusion that it interacts with a server object of the same technology.
- The mediator must hide the fact that the client looks for the server using different object reference and storage methods than the server does. In the same way the fact that the mediator supports all the appropriate attributes of the bridged technologies and its double role as a virtual server and virtual client let us to use simultaneously different methods for naming and storing the mediator as if it was a client's technology server and to recognize and call the real server as if it was a server's technology client.
- The mediator must communicate with the client and server objects using their technology communication protocols. In our client-mediator-server interaction the client and the mediator are interacting using the client's technology protocol and the mediator interacts with the server using the server's technology protocol.

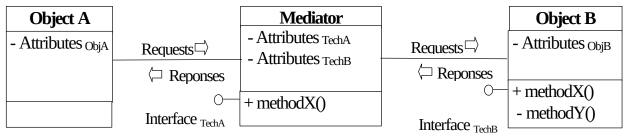

Figure 1: Class diagram of ObjectA-ObjectB interaction.

There is a fifth basic demand concerning the environment where the mediator is hosted. It is obvious that the mediator's environment must support all the bridged technologies in order for the mediator to function properly at run time. In figure 1 we present a class diagram of an interaction between two different technology objects.

As we present in the previous section, up to now we can bridge:

- CORBA-DCOM: using implementations of OMG's Interworking Architecture.
- RMI-CORBA: using the IIOP as RMI's alternative protocol.
- DCOM-RMI: there is not any bridging tool. We can bridge only JavaBeans and ActiveX components.

Following our architecture we have succeed to bridge an RMI-client application with a CORBA-server application and a CORBA-client with an RMI-server without using the IIOP as RMI's protocol, a CORBA-client with a COM-server and, an RMI-client with a COM-server. Moreover, we can extend our mediator in order for an RMI-client to requests some services from a COM-server simultaneously with a CORBA-client. Using the Java programming language and this architecture we have the ability to construct a mediator mechanism in order to bridge any, of those three middleware remoting technologies compliant, objects. Any limitations stem from unsolved incompatibilities on mappings between OMG's and Microsoft's IDLs and the Java programming language.

Conclusions

Although many attempts have been undertaken to bridge the gap between the objects’ underlying architectures, they are not enough at the time to provide true vendor-, language-, and technology-independent interoperation between different software objects. Unfortunately, until now the use of a single middleware product is the most reliable solution.

Compatibility problems between different vendors' products persist even if the products are compliant with the same technology[7]. Even for the available bridge tools their "fully compliant" statements many times refer to a single vendor's products selection which does not support the vendor's independence theory.

Our research concerns the development of an architecture suitable to provide an independent context for building mediators able to integrate multi-technology distributed objects. We are using the Java language as the programming tool for the creation of a mediator mechanism, as a "general purpose object glue". The results of our work have proven that the integration of different middleware remoting technologies is possible.

Up to now our architecture allows the interoperation between an RMI-client and a CORBA-server, a CORBA-client and an RMI-server, a CORBA-client and a COM-server, and an RMI-client and a COM-server. Our interoperation between RMI and CORBA does not depend on the support of the IIOP protocol in the RMI architecture. In the future we plan to complete the circle of the interoperations between the RMI, the CORBA and the COM technologies. Moreover, we plan to create a tool through which a developer will automatically create the needed mediator mechanism. The integration and the application of the Java Object Mediator can provide truly vendor-, language-, and technology-independent interoperation between different software objects.

References

1. Object Management Group, Inc., "The Common Object Request Broker: Architecture and Specification", Revision 2.3, Object Management Group, Inc., June 1999.

2. Microsoft Corporation, "DCOM Architecture, White Paper", Microsoft Corporation, Redmond WA USA, 1998.

3. Sun Microsystems, Inc., "Java Remote Method Invocation Specification", Revision 1.50, Sun Microsystems, Inc., Mountain View, California USA, October 1998.

4. Guijun Wang, Liz Ungar, Dan Klawitter, "Component Assembly for OO Distributed Systems", IEEE Computer, Vol. 32, No 7, pp. 71-78, July 1999.

5. Sun Microsystems, Inc., "Java-Based Distributed Computing, RMI and IIOP in Java", Sun Microsystems, Inc., Mountain View, California USA, June 26, 1997. Available online: http://www.javasoft.com/pr/1997/june/statement970626-01.html, January 2000.

6. Microsoft Corporation, "Integrating Java and COM, A Technology Overview", Microsoft Corporation, Redmond WA USA, January 1999.

7. John Charles, "Middleware Moves to the Forefront", IEEE Computer, Vol. 32, No 5, pp. 17-19, May 1999.