Distributed Object Bridges and Java-based Object Mediator

Konstantinos Raptis, Diomidis Spinellis, Sokratis Katsikas

An important aspect of research on software objects, components, and component-based applications concerns their interoperation. Is their interoperation technically possible? Which elements are responsible for the software objects' incompatibility? Is compatibility a responsibility of the objects or of their underlying architectures? In this article we discuss the object compatibility problems, 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.

1 Introduction

The need to develop software based on existing code rather than development from scratch, led to the emergence of component-based software. Components are typically object-oriented, or at least used as objects. Szyperski [Szyperski 98] defines a software component as: "A unit of composition with contractually specified interfaces and explicit context dependencies only that can be deployed independently and is subject to third-party composition".

Moreover, the need for interaction between the software components led to the specification of middleware remoting models. The Object Management Group's Component Object Request Broker Architecture (CORBA) [Object Management Group 96], Microsoft's Distributed Component Object Model (DCOM) [Microsoft Corporation 98], and Sun Microsystems' Remote Method Invocation (RMI) [Sun Microsystems 96] are three models that enable software components from different vendors, running on different machines, and on different operating systems, to work together.

In order for developers to use a component as part of an application, they must be able to distinguish its attributes. That is, the component must be identified by meaningful characteristics, which are: the component name, which provides the developer with the ability to identify it, the component interface, which identifies the operations fulfilled by the component, and the component model which specifies the semantics and context of the component.

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 behaviour. This task may not always be possible due to technical or organizational constraints. An exemplar case involves two Java objects, each using the RMI and DCOM technologies respectively; the interoperation between them brings out the conflicts between the Java virtual machine and Microsoft Java virtual machine. In the last section of this article we present our research work on a generic architecture that allows bridging an RMI-based client application with a COM-based server application using the Java programming language as the basis for building an operational mediator.

As a component's instance is typically an object and anything applying to objects has also to apply to components, in the next paragraphs our discussion will focus on software objects. In the following section we present the compatibility problems between the software objects and representative attempts for bridging CORBA, DCOM and RMI technologies.

Konstantinos Raptis is a PhD student in the Department of Information and Communication Systems at the University of the Aegean. His research interests include distributed applications, software component models and distributed component interoperation technologies. Contact him at krap@aegean.gr.

Diomidis Spinellis is an assistant professor at the Department of Information and Communication Systems at the University of the Aegean. Contact him at dspin@aegean.gr.

Sokratis Katsikas is a professor at the Department of Mathematics at the University of the Aegean. Contact him at ska@aegean.gr.

2 Object Incompatibility

For software objects to be able to interact with each other they must comply with the rules of their underlying technology. However, it is difficult, if not impossible, for two objects, hosted on different object architectures, to interact with each other. The incompatibility reasons stem from the differences of the underlying models and the way they present and use the software objects. We discern three basic incompatibility points.

- Different Interface Approaches and Implementations: One of the basic elements of an object is its interfaces. Through their interfaces objects expose their functionality. An interface consists of a description of a group of possible operations which a client can ask from an object. A client interacts

only with the interfaces of an object, never with the object itself. Interfaces allow objects to appear as black boxes. Different approaches and implementations of objects' interfaces make them invisible to clients of other technologies.

- Different Object References and Storage: When a client wishes to interact with an object it must first retrieve information on the object's interface. A client's underlying technology must recognize an object's name, it must know where to look, and how to retrieve its information, i.e. it must know how the required object's technology stores its information. If a client's technology does not have that kind of ability then it is impossible for the necessary information of the needed object to be found.
- Different Protocols: Another basic element in distributed object interactions is the protocols used for the data transmission. By the term protocol we do not mean only the transport-level protocol, such as TCP/IP but include the presentation and session level protocol which the Request Brokers (RBs) support. The transport-level protocol is responsible for the transmission of the data to the end point. The presentation and session level protocols are responsible for the formatting of the data transmitted from a client to an object, and vice versa, between different RBs. According to Geraghty et al. [Geraghty et al. 99]: "Although the client and server may speak the same protocol, it is critical that they speak the same language, or higher-level protocol".

Table 1 presents the basic differences of the three models in relation with the above incompatibility points.

The differences presented in table 1, are not the only ones between these three architectures and the only reasons for objects' incompatibility. If we made a detailed comparison between these models we would see many more differences and we could find many more reasons. The differences, we described, are the prime causes. As we will see in the next paragraphs all attempts that have been made for bridging these object middleware architectures focus their attention on these points.

Incompatibility Points	CORBA	DCOM	RMI
Interface Approaches & Implementations	IDL	MIDL	Java
Object Identification	Identification through Object and Interface Names	Identification through a GUID (CLSID & IID)	Identification through a URL-based Object Name and Interface Name
Object Reference	Reference through an Object Reference (OR)	Reference through an Interface Pointer	Reference through a URL-based Object Reference
Object Storage	Storage in Implementation Repository	Storage in Registry	Storage in rmiregistry
Protocols	GIOP/IIOP/ESIOP	Object RPC (ORPC)	JRMP/IIOP

Table 1: CORBA/DCOM/RMI basic differences in relation with incompatibility points.

3 Bridging the Gap

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 with each other. The common point is that all the above are software component models, i.e. not all of them are 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 an extension of COM, are the two most important middleware remoting technologies. Their importance stems from their ancestry. CORBA is a child of the Object Management Group an association including Sun Microsystems, Compaq, Hewlett-Packard, IONA, Microsoft and others, while DCOM comes from Microsoft which enjoys the highest share in the desktop operating system market. Although COM and its extension DCOM are built-in in Microsoft's OSs and there are no other providers of these technologies, 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 side, 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 of satisfying various demands.

The OMG, understanding the need for bridging their differences, and after the first OLE/CORBA bridge from IONA Technologies in 1995, 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 be viewed as a COM object and vice versa. In particular, the OMG specifies four distinct mappings: CORBA/COM, CORBA/OLE Automation, COM/CORBA, and OLE Automation/CORBA mapping.
- Interface Composition Mapping. One of the basic differences between CORBA and COM interfaces is the characteristic of inheritance. While CORBA supports a

multiple interface inheritance, COM provides only 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 task has been left 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. All the above products realize one of the interface mappings that OMG specifies. Their main goal is to provide a two-way interworking between COM and CORBA applications.

RMI-CORBA Bridge

The widespread deployment of the 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. As a first step Java was included in the languages with mappings to OMG IDL. 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 consider the marriage of RMI with CORBA. According to Sun [Sun Microsystems 97] 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 the 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 the registry the RMI-objects and for ORB activation, is required in order for the RMI-IIOP-based objects to be accessed by the corresponding CORBA-based ones.

DCOM-RMI Bridge

No special work has been done for bridging COM/DCOM with RMI. In this field the attention is focused on the attempts at integrating Java language and COM and on the bridging of JavaBeans with ActiveX.

Microsoft supports COM/DCOM under its own edition of the Java language. In order for users of the native Java language to use the COM technology, Microsoft supports the Microsoft Virtual Machine (MSVM). According to Microsoft [Microsoft Corporation 99], the MSVM provides all the mechanisms 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 bridging JavaBeans and ActiveX, 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's flexibility in connection with protocol usage. Moreover, a lot of the work concerns the possibility of using a JavaBean component in an ActiveX-component based environment like Microsoft Office or Visual Basic.

4 Java-based Object Mediator

We have approached the bridging of the three middleware remoting technologies under a different view from those we have described. Our intention was to exploit the Java language, its capability to run irrespective of the operating environment and its acceptance by all the three technologies, as a tool for the creation of a mediator mechanism for bridging all the three technologies together. We are using the Java language as a "general-purpose object glue". Our target was to allow a server object, which may be CORBA, DCOM, or RMI compliant, to expose its methods to CORBA-, DCOM-, and RMI-based clients. In the next paragraphs we will present an example of our approach involving bridging an RMI-client with a COM-server.

The tools we were using in our research include:
- The Sun Microsystems' Java Development Kit, version 1.2 (JDK 1.2, Java 2 Platform [Java 2 Platform 97]).
- The Object-Oriented Concepts' CORBA compliant ORB, ORBacus for C++ and Java, version 3.1.1 [ORBacus 99].
- Microsoft's Software Development Kit for Java, version 3.2 (SDK for Java 3.2 [Microsoft Software Development 99]).
- Microsoft's Virtual Machine (Microsoft VM) build 3188 [Microsoft Virtual Machine 99].

Moreover, the resulting programs can operate under the Microsoft's Windows 98 operating system.

Our intention was to enable an RMI-client program to request methods exposed by a COM-server. Before proceeding with the bridge of the RMI-client with the COM-server we had successfully bridged an RMI-client with a CORBA-server and a CORBA-client with a COM-server by developing a mediator program using the Java language. We will briefly discuss the way we have bridged the RMI-client with the CORBA-server. The bridge between CORBA-client and COM-server follows the same architecture.

We first developed in Java a CORBA-server application, which exposed some methods through its IDL interface. When running the server application, it can receive calls from a CORBA-client application. We then created an RMI-based client/server application where the server exposed, through an RMI-interface, the same methods as the CORBA-server.

In the RMI-server application we added all the necessary attributes to make it act like a CORBA-client application in parallel with its action as an RMI-server. When the RMI-server receives the RMI-client's request, instead of implementing the requested methods, it actually forwards the request, as a CORBA-client, to the CORBA-server application. The

CORBA-server application then responds to the virtual CORBA-client which then acts like an RMI-server and forwards the response to the RMI-client application. Figure 1 presents the class diagram of this RMI-client / CORBA-server interaction.

In the same way we developed another Java mediator allowing a CORBA-client application to request methods exposed by a COM-server application. In this interaction our mediator had to satisfy the demands of a CORBA-server application in parallel with the demands of a COM-client application.

When one tries bridge an RMI-client application directly with a COM-server application it is impossible for a mediator to function as an RMI-server application and as a COM-client simultaneously. Although we had no errors during compilations, we could not get our mediator, at the run time, to implement simultaneously the appropriate classes of JDK's java.rmi package and Microsoft's VM @com directives through which a Java object may be presented as a COM object.

For the interaction between the RMI-client application and the COM-server application to succeed, we used the previous two mediators for bridging RMI-client with CORBA-server (mediator A) and CORBA-client with COM-server (mediator B). Thus, the RMI-client's request was forwarded to the COM-server through mediator A and mediator B. Similarly, the RMI-client receives the COM-server's response through mediator B and mediator A.

From the above discussion it is obvious that in order for the mediator to function properly two basic rules must be followed:

1. The mediator must comply with the client- and server-side architectures. That is, the mediator must have a double role. It must act like one architecture's server application and like the other architecture's client application.
2. The environment where the mediator is hosted must support all the necessary technologies. That is, for the mediator to operate properly in its double role its environment must support simultaneously, at run time, the different architectures.

When trying to bridge directly the RMI-client with the COM-server the second rule could not be satisfied because of conflicts between the Sun and Microsoft Java editions in relation to the support of the RMI and COM technologies.

Except for the above two rules our mediator does not have to support the three incompatibility points we presented in the second section. That is, the different interface approaches and implementations, the different object references and storage, and the different protocols. This goal was achieved by using the Java programming language and our mediator architecture in order to construct our system. We overcame these points of incompatibility by using the Java programming language, which is supported by all the three technologies, and by the mediator's architecture, which includes the attributes of all the interacting technologies.

The fact that we were using the Java language to construct our mediator provided us the advantage that the attributes of our mediators would be understandable through the mapping between the Java language and the different interface definition languages. The architecture of our mediators gave us the ability to name and store the mediator object according to the principles of the client's technology, in parallel with searching and retrieving the server object according to the principles of the server's technology. In the same way the mediator's architecture allowed us to use the client's technology protocol in the client-mediator interaction and the server's technology protocol in the mediator-server interaction.

5 Conclusions

The interoperation between different technology objects is in practice much more complex and difficult than in theory. 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 has been the most reliable solution. Compatibility problems between different vendors' products persist even if the products are compliant with the same technology [Charles 99]. Even for the bridge tools available their "fully compliant" statements often refer to a selection of a single vendor's which does not support the vendor's independence theory.

Our research concerns the development of an architecture capable of providing an independent context for building mediators capable of integrating multi-technology distributed objects. We are using the Java language as the programming tool for the creation of a mediator mechanism. We are using the language as a "general-purpose object glue". The results of our work until now, 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

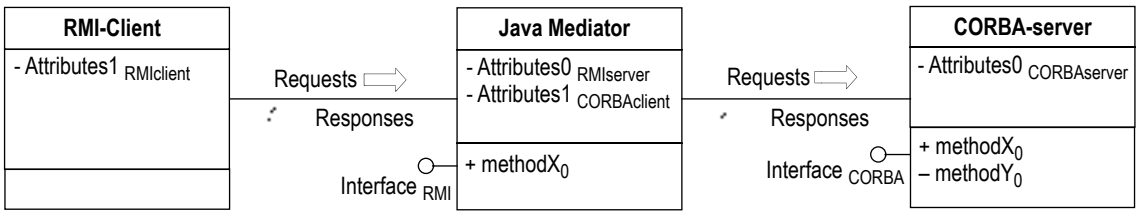

Fig. 1: Class diagram of RMI-client / CORBA-server interaction.

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 necessary 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

[Charles 99]
John Charles, "Middleware Moves to the Forefront", IEEE Computer, Vol. 32, No 5, pp. 17-19, May 1999.

[Geraghty et al. 99]
Ronan Geraghty, Sean Joyce, Tom Moriarty, and Gary Noone, "COM-CORBA Interoperability", Prentice-Hall, Inc., 1999.

[Java 2 Platform 97]
"Java 2 Platform, Standard Edition", Sun Microsystems, Inc., Mountain View, California USA, June 26, 1997.
Available on-line: http://java.sun.com/j2se/, November 1999.

[Microsoft Corporation 98]
Microsoft Corporation, "DCOM Architecture, White Paper", Microsoft Corporation, Redmond WA USA, 1998.

[Microsoft Corporation 99]
Microsoft Corporation, "Integrating Java and COM, A Technology Overview", Microsoft Corporation, Redmond WA USA, January 1999.

[Microsoft Software Development 99]
"Microsoft Software Development Kit for Java 3.2", Microsoft Corporation, Redmond WA USA.
Available on-line: http://www.microsoft.com/java/sdk/32/, November 1999.

[Microsoft Virtual Machine 99]
"Microsoft Virtual Machine build 3188", Microsoft Corporation, Redmond WA USA.
Available on-line: http://www.microsoft.com/java/vm/dl_vm32.htm, November 1999.

[Object Management Group 96]
Object Management Group, Inc., "The Common Object Request Broker: Architecture and Specification", Revision 2.0 (Updated), Object Management Group, Inc., July 1996.

[ORBacus 99]
"ORBacus for C++ and Java", Object Oriented Concepts, Inc., Billerica Ma USA.
Available on-line: http://www.ooc.com/ob/, November 1999.

[Sun Microsystems 96]
Sun Microsystems, Inc., "Java Remote Method Invocation Specification", Beta Draft Revision 1.2, Sun Microsystems, Inc., Mountain View, California USA, December, 1996.

[Sun Microsystems 97]
Sun Microsystems, Inc., "Java-Based Distributed Computing, RMI and IIOP in Java", Sun Microsystems, Inc., Mountain View, California USA, June 26, 1997.
Available on-line: http://www.javasoft.com/pr/1997/june/statement 970626-01.html, September 1999.

[Szyperski 98]
Clemens Szyperski, "Component Software: Beyond Object-Oriented Programming", Addison-Wesley Publishing Company, Inc., 1998.