ELSEVIER

Computer Standards & Interfaces 23 (2001) 157–168

COMPUTER STANDARDS & INTERFACES

www.elsevier.com/locate/csi

Multi-technology distributed objects and their integration

Konstantinos Raptis [a,*], Diomidis Spinellis [b], Sokratis Katsikas [a]

[a] *Department of Information and Communication Systems, University of the Aegean, GR-83200 Karlovasi, Samos, Greece*
[b] *Department of Technology and Management, Athens University of Economics and Business (AUEB), Evelpidon 47A, 11362 Athens, Greece*

Received 14 November 2000; received in revised form 8 March 2001; accepted 15 March 2001

Abstract

Research on software objects, components, middleware, and component-based applications concerns among others ActiveX controls, JavaBeans (JBs), the Microsoft Transaction Server (MTS), Enterprise JavaBeans (EJBs), and how they can interoperate with each other. Is their interoperation 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 object compatibility problems by outlining three basic middleware remoting technologies: the OMG's Common Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM), and Sun's Java Remote Method Invocation (RMI), discussing the basic incompatibility points, and overviewing the basic strategies for bridging the gap between CORBA, DCOM, and RMI. © 2001 Published by Elsevier Science B.V.

Keywords: Software objects; Components; Bridge; Midlleware; Object compatibility; Interoperation

1. Introduction

Software development is becoming increasingly complicated. Business requirements for client/server applications, support for multiple platforms, and sophisticated end-user functionality have forced developers to adopt new approaches. One of the concepts that changed the rules in software development is object-oriented programming (OOP), which is organized around objects rather than actions. According to Budd [1], "All objects are instances of a class. The method invoked by an object in response to a message is determined by the class of the receiver. All objects of a given class use the same method in response to similar messages".

The need to develop software based on existing code rather than development from scratch led to the development of component-based software. Components are typically object-oriented, or at least used as objects. Szyperski [16] defines a software component as "a unit of composition with contractually specified interfaces and explicit context dependencies only. One that can be deployed independently and is subject to third-party composition".

To be composable, components need to be identified by meaningful characteristics, namely: 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 execution context of the component.

* Corresponding author. Tel.: +30-273-25-282; fax: +30-273-25-282.

E-mail addresses: krap@aegean.gr (K. Raptis), dds@aueb.gr (D. Spinellis), ska@aegean.gr (S. Katsikas).

0920-5489/01/$ - see front matter © 2001 Published by Elsevier Science B.V.
PII: S0920-5489(01)00070-8

Moreover, a software component must meet three basic demands:

1. it must be directly usable: the component must contribute to the development process;
2. it must be a defined and discrete unit: the content of the component must be explicitly identified; and
3. it must be separable from its original context and usable in other contexts: any component must be reusable in applications other than the one it was first defined.

The rapid deployment of software components and the advantages of component-based applications against the monolithic applications drive many enterprises to deploy their network applications based on components. Although software components address many enterprise application development issues, their use unavoidably generates new problems. As we will see below, the components must comply with a specific underlying middleware architecture in order to interact with each other efficiently. This dependency on the underlying architecture creates compatibility problems between components based on different architectures. These problems become critical by changes in company information systems due to acquisitions, mergers, and infrastructure upgrades [2].

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

2. Middleware technologies

The ability to construct applications using objects from different vendors, running on different machines, and on different operating systems, it is not an easy task. The need for interaction between the software objects led to the specification of middleware models. The Object Management Group's Component Object Request Broker Architecture (CORBA), Microsoft's Distributed Component Object Model (DCOM), and the Sun Microsystems Remote Method Invocation (RMI) are three models that enable software objects from different vendors, running on different machines, and on different operating systems, to work together.

To provide our readers with a feeling of the different technologies, we have implemented three versions of a very simple client/server application, utilizing CORBA (Listing 1), DCOM (Listing 2), and Java RMI (Listing 3). The application client uses the server to sum two numbers. The client, *SumClient*, initially sends two numbers to the server, *SumServer*, through the *SetNum()* function, and then calls the *Sum()* function through which the server returns to the client the sum of the two numbers. We present the code for those applications without including the automatically generated helper files. The client/server applications are implemented using the Java language in order to provide a uniform language-layer presentation through which the differences of CORBA, DCOM and Java RMI can be easily identified.

2.1. OMG CORBA

The Common Object Request Broker Architecture (CORBA) [10] is an open standard specifying a framework for transparent communication between applications and application objects. It is defined and supported by the Object Management Group (OMG), a nonprofit, trade association of over 700 software developers, vendors, and end-users. According to CORBA, a client which asks for some services from an object makes a request which is transferred to the object request broker. The ORB is responsible for forwarding the request to the right object implementation. This request contains all the information required to satisfy the request: target object, operations, zero or more parameters, and an optional request context.

The client's request to the ORB and the forwarding of the request from the ORB to the object implementation must be in a form that will be understandable independently from the specific ORB. The client invocation style depends on the component's interface details available to the client at compilation time: if the client has access to the appropriate component and its interface then it can use the static interface, otherwise the client has to deposit its request to the ORB through the dynamic interface. To allow ORB-neutral invocation, the request is made to the ORB through an interface called "IDL Stub", if there is a static invocation, or through the "Dynamic Invocation", if there is a dynamic invoca-

tion. In the server end, the promotion of the request to the object implementation is also made through interfaces called “Static IDL Skeleton” and “Dynamic Skeleton”.

We stress that clients never come in direct contact with the objects, but always with the interfaces of these objects. The interfaces are determined through OMG’s IDL (Interface Definition Language). The clients are not “written” in OMG’s IDL, but in a language for which there is a mapping to OMG’s IDL.

In addition, the communication of a client with an object running as a different process or on a different machine uses a communication protocol to portably render the data format independently from the ORB. For this reason, the OMG has designed the General Inter-ORB Protocol (GIOP), a protocol which ensures the connection of the ORBs no matter where they came from. The Internet Inter-ORB Protocol (IIOP) is a specific mapping of the GIOP to TCP/IP connections, the most popular protocol for network connectivity.

CORBA Interface

```
// SumInt.idl

interface SumInt {
  void SetNum(in double x, in
double y);
  double Sum();
};
```

CORBA Server

```
// SumServer.java

package addnum;
import org.omg.CORBA.*;
import java.io.*;
import java.util.*;

public class SumServer extends
_SumIntImplBase {
  double first;
  double second;
  SumServer() {
    super();
  }
  public void SetNum(double x, double y) {
    first=x;
    second=y;
  }
  public double Sum() {
    return first+second;
  }

  public static void main(String args[]) {
    ORB orb = ORB.init(args, new Properties());
    BOA boa =
((com.ooc.CORBA.ORB)orb).BOA_init(args,
new Properties());
    SumServer server=new SumServer();
    try {
      String ref = orb.object_to_string(server);
      String refFile = "Sum.ref";
      java.io.PrintWriter out = new
java.io.PrintWriter(
        new java.io.FileOutputStream(refFile));
      out.println(ref);
      out.flush();
    }
    catch(java.io.IOException e) {
      System.out.println(e);
      System.exit(1);
    }
    boa.impl_is_ready(null);
  }
}
```

CORBA Client

```
// SumClient.java

package addnum;

import org.omg.CORBA.*;
import java.io.*;
import java.util.*;

public class SumClient {

  public static void main(String args[]) {

    ORB orb = ORB.init(args, new
Properties());

    String ref=null;

    try {
      String refFile="Sum.ref";
      java.io.BufferedReader in=
        new java.io.BufferedReader(new
FileReader(refFile));
      ref=in.readLine();
    }

    catch(IOException e) {
      System.out.println(e);
      System.exit(1);
    }

    org.omg.CORBA.Object obj =
orb.string_to_object(ref);
    SumInt look_sum =
SumIntHelper.narrow(obj);
    look_sum.SetNum(15,20);
    double result=look_sum.Sum();
    System.out.println(result);

  }
}
```

Listing 1: CORBA client-server example

2.2. Microsoft DCOM

Microsoft's Distributed Component Object Model (DCOM) [7] is an object-oriented model designed to promote interoperability of software objects in a distributed, heterogeneous environment. It's an extension of Microsoft's Component Object Model (COM) [8]. COM's target is to allow two or more applications or objects to easily cooperate with one another, even if they have been written by different vendors at different times, in different programming languages, or if they are running on different machines, and under different operating systems.

A client requests services from an object via the object interface represented as a pointer. Therefore, the programming language in which the client is implemented must have the ability to create pointers and call functions through these pointers and the client must have the right pointer to that interface. The interfaces are determined through Microsoft's Interface Description Language (different from OMG's IDL) which allows the developers to construct the object interfaces.

In case the client does not have the right pointer to the appropriate interface, it addresses COM giving as input the class identification, CLSID, of the object and the server's type of object class: "in-process", "local", or "remote". COM then uses the Service Control Manager (SCM) to search and find the applicable server and give back to the client the requested pointer. The laying down of the client's request and its promotion to the appropriate object implementor is done via proxy and stub interfaces, respectively. These interfaces are responsible to marshal and unpack the transmitted data. Similarly, the stub marshals the object's response and the proxy unpacks and promotes the response back to client.

Obviously, every client must provide at least the class identification (CLSID) of the object it needs and the type of the server. The client may be implemented in any programming language as long as the language supports the construction and management of pointers. Inter-process communication is performed using Remote Procedure Calls (RPCs). The remote procedure call mechanism is based on the Distributed Computing Environment Remote Procedure Call mechanism (DCE RPC).

DCOM Interface

```
//SumInt.odl

[uuid(351420C2-B26C-11D3-8EE4-
0050048ADE88), version(1.0)]
library SumIntlib {
  importlib("stdole32.tlb");

  [uuid(351420C3-B26C-11D3-8EE4-
0050048ADE88)]
  dispinterface ISumInt {
    properties:
      [id(1)] double first;
      [id(2)] double second;
    methods:
      [id(3)] void SetNum(double x,
double y);
      [id(4)] double Sum();
  };

  [uuid(351420C4-B26C-11D3-8EE4-
0050048ADE88)]
  coclass SumServer {
    dispinterface ISumInt;
  };

};
```

DCOM Server

```
// SumServer.java

import com.ms.com.*;
import sumintlib.*;

public class SumServer implements
ISumInt {

  double first;
  double second;

  public void SetNum(double x, double y) {
    first=x;
    second  y;
  }

  public double Sum() {
    return first+second;
  }
}
```

DCOM Client

```
// SumClient.java

import sumintlib.*;

public class SumClient {

  public static void main(String args[]) {

    try {
      ISumInt look_sum = (ISumInt) new
sumintlib.SumServer();
      look_sum.SetNum(15,20);
      double result=look_sum.Sum();
      System.out.println(result);
    }

    catch(com.ms.com.ComFailException
e) {
      System.out.println("SumClient err: "
+ e.getMessage());
      System.out.println(e.getHResult());
  }
}
```

Listing 2: DCOM client-server example

On July 2000, Microsoft announced its .net platform as the new base for developing Web-based applications. The .net platform as a programming model represents the next evolution of Microsoft's Component Object Model. The .net platform consists of a set of application level frameworks, a set of base frameworks, and a common language runtime. At the center of the .net framework is an object model called the Virtual Object System providing a middleware infrastructure for components based on different technologies across the Web [6]. The interoperability among the different technologies across the Web is made possible using open Web protocols such as XML and SOAP.

2.3. Java / RMI

The Remote Method Invocation (RMI) [14] is also an object-oriented model designed to promote interoperability of Java objects in a distributed, heterogeneous environment. RMI is the mechanism for the transparent communication exclusively between Java objects.

For an RMI client to use an object's services, it must submit a request to the RMI which contains an object reference, the desired methods of that object, and the necessary parameters for the implementation of the request. RMI is then responsible to find and promote the request to the right object. The submission of the client's request is made to a stub. The stub is the top level of the RMI which makes the object appear as if it is in the same process as the client. The stub presents the request in the right form to be transmitted to the server-side RMI part. After the stub, the request passes through the remote reference layer to the transport layer which is responsible for transmitting the request. At the server side, the object collects the request in a reverse way through the RMI layers and responds back to the client via RMI in the same way that the client did.

To deposit a request, the client must have an object reference to the needed object. In the case where the client does not have the object reference it can look for it through the appropriate service provided by the Java RMI namely the *java.rmi.Naming* class. Like the other two technologies the RMI clients come in contact only with the interfaces of the objects. Those interfaces are defined using the Java language and not via a special IDL.

The client/object communication procedure through RMI is the same irrespective of whether the object resides locally or remotely. In the case where the client communicates with an object residing on a different machine the use of the Java Remote Method Protocol (JRMP) is necessary.

Java RMI Interface	Java RMI Server	Java RMI Client

// SumInt.java
 	
import java.rmi.*;
 	
public interface SumInt extends Remote {
void SetNum(double x, double y) throws RemoteException;
double Sum() throws RemoteException;
}
	// SumServer.java
	
	import java.rmi.*;
	import java.rmi.server.*;
	
	public class SumServer extends UnicastRemoteObject
	implements SumInt {
	double first;
	double second;
	
	public SumServer() throws RemoteException {
	super();
	}
	public void SetNum(double x, double y) {
	first=x;
	second=y;
	}
	public double Sum() {
	return first+second;
	}
		// SumClient.java
		
		import java.rmi.*;
		
		public class SumClient {
		
		public static void main(String args[]) {
		
		try {
		String name = "//Localhost/SumServer";
		SumInt look_sum = (SumInt)Naming.lookup(name);
		look_sum.SetNum(15,20);
		double result=look_sum.Sum();
		System.out.println(result);
		}
		
		catch (Exception e) {
		System.out.println("SumClient err: " + e);
		System.exit(1);

```
  public static void main(String args[]) {

    try {
      SumServer server=new SumServer();

Naming.rebind("//Localhost/SumServer
", server);
      System.out.println("SumServer
ready.....");
    }
    catch (Exception e) {
      System.out.println("SumServer err: " +
e);
      System.exit(1);
    }
  }
}
```

```
    }

  }
}
```

Listing 3: Java RMI client-server example

3. Object incompatibility

For software objects to interact with each other, they must comply with the rules of at least one of the above models. However, it is difficult if not impossible for two objects conforming to dissimilar technologies 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 can identify three basic incompatibility points [12].

3.1. Different interface approaches and implementations

One of the basic elements of an object are its interfaces. Through their interfaces objects expose their functionality. An interface consists of a description of a group of possible operations that 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 object interfaces make them invisible to clients of other technologies.

Both CORBA and DCOM use special Interface Definition Languages (IDLs) for the interface specification. Java RMI uses the Java language to define the interfaces. In DCOM, every interface has a Universally Unique Identifier (UUID), called the Interface Identifier (IID), and every object class has its own UUID, called Class Identifier (CLSID). Moreover, every object must implement the IUnknown interface. When using the Java language to specify DCOM objects every Java class implements that interface behind the scenes through the Microsoft Java Virtual Machine (MSJVM). In Java RMI the interface must be declared as public, it must extends the interface *java.rmi.Remote* and each method must declare *java.rmi.RemoteException* in its *throws* clause.

3.2. 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 and disseminates 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.

In CORBA (Listing 1), the IDL compiler generates the appropriate client stubs and server skeletons for a client to deposit a static invocation to the requested object. Moreover, all the necessary information is stored in the Interface Repository through which a client can get run-time information for a

dynamic invocation. The client is searching for the needed methods using the object's name reference and invokes it statically, through the client stub interface, or dynamically, through the dynamic invocation interface, depending on its run-time knowledge. For the interaction to be possible the CORBA server program must bind the server object using the CORBA Naming Service. Prior to the above interaction, the CORBA server and the CORBA client program must first initialize the CORBA ORB through the *ORB.init()* method.

In the DCOM client/server interaction (Listing 2), the MSJVM hides many of the code details needed in the previous CORBA example. The server object binding is performed (using the *javareg* utility) through the system registry where the COM clients search for the needed COM components. In the DCOM client, the instantiation of the DCOM object is done using the *new* keyword. Although it seems that the client is referring to the needed remote object by its name, the system is looking for that object based on its CLSID. All the necessary calls to the *IUnknown* and *IDispatch* interfaces used by the client to acquire the appropriate pointer to the server object and for the management of server's object life cycle are accomplished transparently through the MJVM.

Looking at the Java RMI example (Listing 3), in the server side program one creates the server object and binds it to the RMIRegistry using the *Naming.rebind()* method by assigning a URL-based name. On the client side, the Java RMI client gets a reference from the server's registry using the URL-based object's name through the *Naming.lookup()* method.

3.3. Different protocols

Another basic element in distributed object interactions is the protocol used for the data transmission. In our case, a protocol does not denote only the transport-level protocol, such as TCP/IP but includes the presentation and session level protocols supported by the Request Brokers (RBs). 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 between different RBs from a client to an object, and vice versa. According to Geraghty et al. [5]: "Although the client and server may speak the same protocol, it is critical that they speak the same language, or higher-level protocol".

In Table 1 we present the basic differences of the three models according to the above incompatibility points. These differences 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 find many additional reasons; the differences we described are however the prime causes of incompatibilities. As we will see in the next paragraphs, all attempts for bridging these object middleware architectures focus their attention on these points.

4. Bridging the gap

When two or more objects, based on different technologies must to interoperate the mission target

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

Incompatibility points	CORBA	DCOM	RMI
Interface approaches and implementations	IDL	MIDL	Java
Object identification	Identification through Object and Interface Names	Identification through GUID (CLSID and IID)	Identification through URL-based Object Name and Interface Name
Object reference	Reference through Object Reference (OR)	Reference through Interface Pointer	Reference through URL-based Object Reference
Object storage	Storage in Implementation Repository	Storage pointers in the System Registry	Storage in rmiregistry
Protocols	GIOP/IIOP/ESIOP	Object RPC (ORPC)	JRMP/IIOP

is to make the objects hide the fact that the other objects are functioning under a different technology without changing their characteristics and behavior. According to object technology the "shop-window" of an object is its interface. When an object wishes to contact another it must be able to view, understand, and work out with the other object's interface in order to request the needed methods. Each technology has its own way to create the objects' interfaces using its own IDL. Therefore, for two technologies to interoperate any object must be able to understand the other technology's IDL. Suppose a client deposits to its RB a request for some methods. The RB must be able to look for the appropriate object which exposes these methods. The RB must therefore know the way the other RB names and stores the information of its objects in order to be able to find them and forward the request. For the above operations to be feasible, the requests and the responses to be transmitted must be formatted in a mutually understandable way i.e. different RBs must communicate using the same protocols.

The goals we outlined can be achieved by using a proxy bridge object [11]. This object is used as a mechanism to translate the requests and the responses in an understandable form and maintain the main characteristics and behavior of the real object. Moreover, the bridge object is provided with all the necessary attributes so that it can be viewed from the different technology's object as if it was part of the same technology. Figs. 1 and 2 are examples of class and activity diagrams of the interaction of different technology objects.

We can distinguish three cases depending on where the bridge object resides. It could reside in the client's machine, in the server's machine, or on a third machine. If the bridge object resides in the client's machine, the client's environment must support two or more different middleware technologies. Moreover, if the server object changes its state, this must be propagated to every bridge object, i.e. to the machine of every client. When the bridge object resides on the server's or an entirely different machine, then the above problems are not relevant, but performance problems are likely to occur. In the next paragraphs, we outline some of the attempts for bridging CORBA, DCOM, and RMI.

4.1. CORBA–DCOM bridge

CORBA and DCOM, as an extension of COM, are the two most widespread middleware technologies. Their importance stems from their "parents". 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 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 to satisfy different demands.

After the first OLE/CORBA bridge from IONA Technologies in 1995, OMG 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

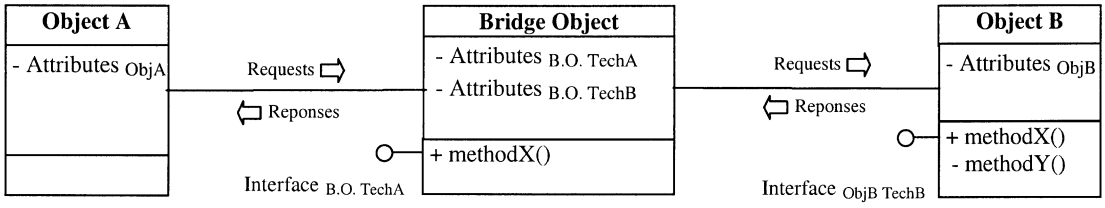

Fig. 1. Class diagram of Object A–Object B interaction.

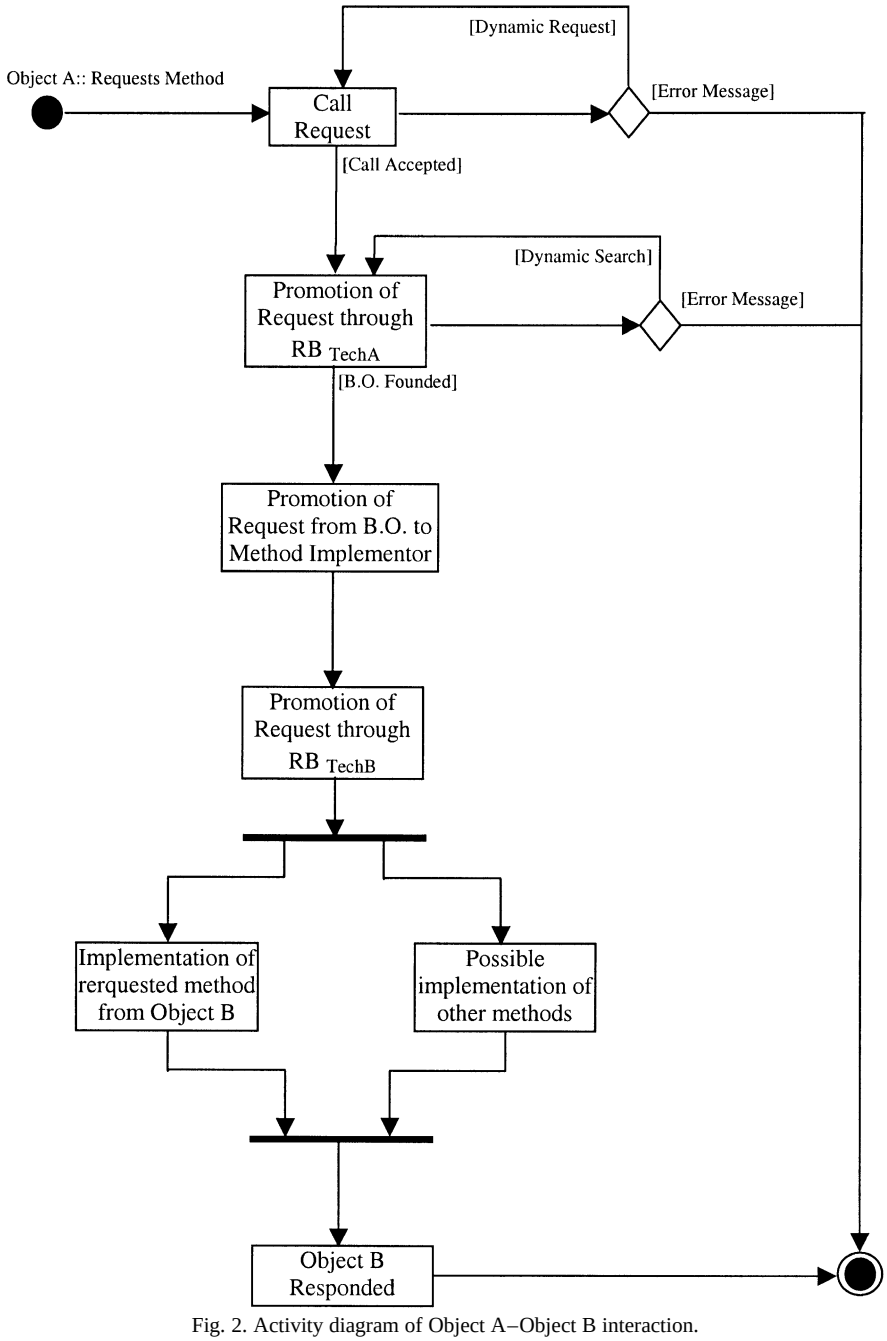

Fig. 2. Activity diagram of Object A–Object B interaction.

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. Particularly, OMG specifies four distinct mappings: CORBA/COM,

CORBA/OLE Automation, COM/CORBA, and OLE Automation/CORBA.

- Interface Composition Mapping. One of the basic differences between the CORBA and COM interfaces is the characteristic of inheritance. While CORBA supports multiple interface inheritance, COM supports multiple interfaces for objects and therefore provides single inheritance. For a bridge to work, 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 used by CORBA and COM.

As we saw, OMG provides the specifications regarding the mappings between COM and CORBA IDLs and interfaces. One point that the Interworking Architecture does not specify concerns the approach that should be taken to bridge COM and CORBA. We can distinguish two basic approaches for bridging, the static bridging, and the dynamic bridging [5].

Under static bridging, the creation of an intermediate code to make the calls between the different systems is required. That intermediate code would be the client's side proxy which could be different in order to receive an object system's call, transform it, and forward it to another object system. The main advantage of static bridging is that it can be easily implemented because it has to deal with object interfaces which contain known code. The disadvantage of the static bridge is that any changes on the interfaces require a change in the bridge.

In dynamic bridging, there is no code dependent on the types of calls, i.e. the interfaces that must be generated. The operation of a dynamic bridge is based on the existence of a dynamic mechanism which can manage any call in spite of the interfaces. The dynamic bridging appears as an enhanced solution by which many problems of static bridging can be avoided. The ability of CORBA to handle dynamic invocation calls through the Dynamic Invocation Interface (DII) and the similar ability of DCOM through dynamic OLE Automation, makes the use of dynamic bridging quite versatile especially in an application environment where a large number of interfaces are involved.

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 Orbix-COMet 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.

4.2. 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. Although Sun provided its own model for remote Java-object interactions-the Java Remote Method Protocol (RMI)-the effective combination of the Java language with the CORBA architecture led OMG and Sun to contemplate the interoperation of RMI and CORBA. According to Sun Microsystems [15], the Java developers would be able to use RMI-based Java objects and interoperate with CORBA-based remote objects. In June 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. For this goal to be achieved, OMG has adopted two standards for "Object By Value" and "Java-to-IDL" mapping.

Apart from the adoption of IIOP as RMI's alternative protocol, a new version of the *rmic* compiler has been developed 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 so that RMI-IIOP-based objects can be accessed by CORBA-based ones.

4.3. 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 for integrating the Java language and COM and on the bridging of JavaBeans with ActiveX.

Until recently, Microsoft supported COM/DCOM in its own edition of the Java language, Visual J++. To provide Java users access to COM technology, Microsoft supported the Microsoft Java Visual Machine. According to Microsoft [9], the MSJVM provided 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. With the release of Visual Studio .net Microsoft has stopped the active support of Java in favor of C#.

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 conjunction with the underlying protocols. 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.

5. Conclusions

Most of the work in the area we surveyed concerns bridging CORBA and DCOM. This is expected considering the widespread deployment of Microsoft's operating systems and the acceptance of CORBA as the most mature middleware architecture. Moreover, the early presence of a variety of COM components and ORB products from commercial companies led developers to use those products. As a result the bridging between CORBA and DCOM was an urgent need.

The attempts to bridge CORBA and RMI indicate that although Sun states that it will continue to support JRMP concurrently with IIOP as the RMI's communication protocol, the CORBA architecture will prevail over RMI. Besides, OMG's intention to support Enterprise JavaBeans confirms that notion. On the other hand, Microsoft's and Sun's work on bridging ActiveX and JavaBeans apparently focus more on the interoperation between their component models than between their middleware remoting technologies.

In the latest versions of CORBA and COM, which are CORBA 3 and COM+, there is no further contribution on the aspect of interoperability. CORBA 3 adds three new features to the previous specifications concerned with Java and Internet integration, quality of service control, and the CORBA component architecture [13]. On the other hand, COM+ enriches its ancestor with new features and services like just-in-time activation, object pooling, load balancing, in-memory databases, queued components, automatic transactions, role-based security, and events [4]. Moreover, Microsoft's promotion of its .net platform for next-generation Internet applications caused confusion about the future of technologies such as DCOM [3].

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 underlying object architectures, these are currently not providing 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 products of different vendors persist even if the products are compliant with the same technology [2]. Even for the available bridge tools their "fully compliant" statements many times refer to a single vendor's products selection that does not support the vendor's independence theory. In the future, we hope that middleware implementations using a common XML-based protocol will provide a new opportunity for truly interoperable objects.

References

[1] T. Budd, An Introduction to Object-Oriented Programming. Addison-Wesley Professional, Boston, USA, 1991.
[2] J. Charles, Middleware Moves to the Forefront. IEEE Computer 32 (5) (1999) 17–19, May.
[3] D. Deckmyn, Uncertainty surrounds Microsoft's .net plans. Computerworld 34 (27) (2000) 12, July.
[4] G. Eddon, COM+: the evolution of component services. IEEE Computer 32 (7) (1999) 104–106, July.
[5] R. Geraghty, S. Joyce, T. Moriarty, G. Noone, COM–CORBA Interoperability. Prentice-Hall Inc., New Jersey, USA, 1999.
[6] B. Meyer, The significance of "dot-Net". Software Development 8 (14) (2000) 51–60, November.
[7] Microsoft, DCOM Architecture, White Paper. Microsoft, Redmond, WA, USA, 1998.
[8] Microsoft, The Component Object Model Specification, Version 0.9. Microsoft, Redmond, WA, USA, 1995 October.

[9] Microsoft, Integrating Java and COM, A Technology Overview. Microsoft, Redmond, WA, USA, 1999 January.
[10] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.0 (Updated). Object Management Group, Needham, USA, 1996 July.
[11] K. Raptis, D. Spinellis, S. Katsikas, Java as distributed object glue. World Computer Congress 2000, Beijing, China, August. International Federation for Information Processing, 2000.
[12] K. Raptis, D. Spinellis, S. Katsikas, Distributed object bridges and Java-based object mediator. Informatik/Informatique 2 (2000) 4–8, April.
[13] J. Siegel, A preview of CORBA 3. IEEE Computer 32 (5) (1999) 114–116, May.
[14] Sun Microsystems, Java Remote Method Invocation Specification, Beta Draft Revision 1.2. Sun Microsystems, Mountain View, CA, USA, 1996 December.
[15] Sun Microsystems. Java-Based Distributed Computing, RMI and IIOP in Java, Sun Microsystems, Mountain View, CA, USA, June 26, 1997. Online, Sun Microsystems. Available online: http://www.javasoft.com/pr/1997/june/statement970626-01.html, February 2001.
[16] C. Szyperski, Component Software: Beyond Object-Oriented Programming. Addison-Wesley, 1998.

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 in the Department of Technology and Management at the Athens University of Economics and Business. Contact him at dds@aueb.gr.

Sokratis Katsikas is vice rector at the University of the Aegean. He is also a professor in the Department of Information and Communication Systems at the University of the Aegean. Contact him at ska@aegean.gr.