A&QT-R 2002 (THETA 13)
2002 IEEE-TTTC International Conference on Automation, Quality and Testing, Robotics
May 23 – 25, 2002, Cluj-Napoca, Romania

A comparison of distributed object-programming techniques

Honoriu Vălean, Tiberiu Leția, Adina Aştilean, Mihai Hulea, Camelia Avram

Technical University of Cluj-Napoca, Daicoviciu 15 st., Cluj-Napoca

ABSTRATCT

The explosive growth of the Web, the increasing popularity of PCs and the advances in high-speed network access have brought distributed computing into the main stream. To simplify network programming and to realize component-based software architecture, three distributed object models have emerged as standards, namely, JAVA/RMI (Remote Method Invocation), CORBA (Common Object Request Broker Architecture) and DCOM (Distributed Component Object Model). In the paper, a comparison of the three technologies, using the same program example is done.

KEYWORDS: Distributed objects, JAVA/RMI, CORBA, DCOM

1. INTRODUCTION

The implementation of control systems, usually involves the use of distributed computing, over heterogeneous networks. Distributed-object computing implement the object-oriented programming technique by allowing objects to be distributed across heterogeneous networks. Thus, each of the distributed object components interoperate as a unified whole. The objects may be distributed on different computers throughout a network, living within their own address space outside of an application, and yet appear as though they were local to an application.

There are three main paradigms for implementing distributed objects: Java/RMI, CORBA and DCOM. Each of these paradigms involves a client-server architecture. In the paper, a comparative study of these three technologies is done.

2. REMOTE METHOD INVOCATION

Java/RMI, developed by Java Soft, relies on a protocol called the Java Remote Method Protocol (JRMP). Java is strongly related to Java Object Serialization, which allows objects to be marshaled (or transmitted) as a stream. A drawback of the method arises from the necessity to write both the Java/RMI server object and the client object in Java. The access to the server object outside of the current Java Virtual Machine (JVM) or on another machine's JVM is done through an interface, implemented by the server. The interface exposes a set of methods, which are specific to the services offered by the server object. A client locates a server object for the first time, using a naming mechanism called a RMIRegistry, that runs on the Server machine and holds information about available server objects. The Java/RMI client executes a lookup for a server object reference for acquiring an object reference to a Java/RMI server object and invokes methods exposed by the server object, as if the Java/RMI server object is residing in the same address space as the client. For acquiring a server object reference, the client should specify the URL of the server object.

The RMI system uses a special compiler, called RMIC, for allowing the transparent access to the remote objects. The compiler generates the classes for linking the local object representation, and the remote object, implemented by the server.

If a client needs some service from a remote distributed object, it invokes a method implemented by the remote object. The service provided by the remote distributed server object is encapsulated as an object and exposed by an interface. The interface will also ensure type consistency between the Java/RMI client and the Java/RMI server object. Every remotable server object as to extend the *java.rmi.Remote* class. Similarly, any method that can be remotely invoked in Java/RMI may throw a *java.rmi.RemoteException*.

As example, let's consider a server which implement as attribute an integer array, and exposes two access methods: *getData()* and *putData()*. The client programs access trough the exposed methods the variables stored in the server attribute (figure 1).

The implementation of the interface is:

```
import java.rmi.*;
public interface ExampleInterface extends remote {
  int getData(int index) throws RemoteException;
  void getData(int index, int data) throws RemoteException; }
```

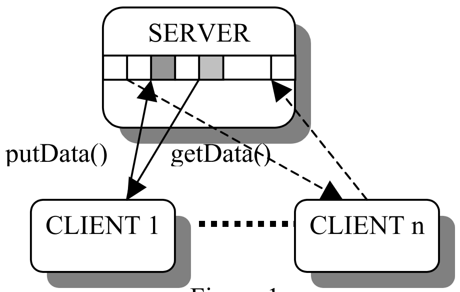

Figure 1

The proposed implementation for the server is the following:

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;
import java.rmi.net.*;
public class ServerRmi extends UnicastRemoteObject implements ExampleInterface {
  public int[] dat=new int[10];
  public int getData(int index) throws RemoteException {return dat[index]);}
  void getData(int index, int data) throws RemoteException {dat[index]=data;}
  public ServerRmi() throws RemoteException {}
  public static void main(String[] args) throws Exception {
    ServerRmi pt=new ServerRmi();
    try { Naming.rebind("//ServerName:portNumber/FarObj",pt);}
    catch (RemoteException re) {System.err.println("Rebind "+re);}
    catch (MalformedURLException me) { System.err.println("RebindURL "+me);}
    catch (Exception ex) { System.err.println("Exception "+ex);}}}
```

Usually, the Java/RMI client first calls *System.setSecurityManager()*, to install a security manager before doing any remote calls. An implicit RMISecurityManager is provided by JavaSoft, but programmer has the opportunity to write his own implementation. Then, the Java/RMI client instantiates a Java/RMI server object by binding to a server object's remote reference through the call to *Naming.Lookup()*. After acquiring a valid object reference to the Java/RMI server object, the client can call into the server object's methods as if the server object resided in the client's address space.

The client program implementation could be:

```
import java.rmi.*;
import java.rmi.registry.*;
public class ClientRmi {
  public static void main(String[] args) throws Exception {
    int value=1;
    if(System.getSecurityManager()==null)
      {System.setSecurityManager(new RMISecurityManager());}
    try {
      ExampleInterface rO=(ExampleInterface)
         Naming.lookup((“//ServerName:portNumber/FarObj”,pt);
      for (int k=0;k<10;k++) {
        rO.putData(3,value);
        value=rO.getData(5); }}
    catch(Exception ex) { System.out.println(“Exception “+ex);}}}
```

3. COMMON OBJECT REQUEST BROKER

CORBA, developed by Object Management Group, is based on a protocol called the Internet Inter-ORB Protocol (IIOP) for remoting objects. All operations in the CORBA architecture are related on an Object Request Broker (ORB**)**. The ORB acts as a central Object Bus over which each CORBA object interacts transparently with other CORBA objects located either locally or remotely. Each CORBA server object has an interface and exposes a set of methods. To request a service, a CORBA client acquires an object reference to a CORBA server object. Then, the client can make method calls on the object reference as if the CORBA server and the client are residing in the same address space. The ORB is responsible for finding a CORBA object's implementation, preparing it to receive requests, communicate requests to it and carry the reply back to the client. There are two methods to allow a CORBA object to interact with the ORB: through the ORB interface or through an Object Adapter – such as Basic Object Adapter (BOA) or Portable Object Adapter (POA).

CORBA uses an IDL interface for expose remote methods. The IDL code can be translated in Java using the IDLJ compiler.

CORBA supports multiple inheritance at the IDL or interface level and can specify exceptions in the IDLs. In CORBA, the IDL compiler generates type information for each method in an interface and stores it in the Interface Repository (IR). For getting run-time information about a particular interface, a client has to query the IR and then, it can use that information to create and invoke a method on the remote CORBA server object dynamically, through the Dynamic Invocation Interface (DII). On the server, the Dynamic Skeleton Interface (DSI) allows a client to invoke an operation of a remote CORBA server object that has no compile time knowledge of the type of object it is implementing. To invoke a remote method, the client makes a call to the client proxy. The client proxy packs the call parameters into a request message and invokes the IIOP wire protocol to ship the message to the server. On the server, the wire protocol delivers the message to the server’s stub. The then unpacks the message and calls the actual method on the object. In CORBA, the client stub is called the stub or proxy and the server stub is called skeleton. A specific CORBA architecture is presented in figure 2.

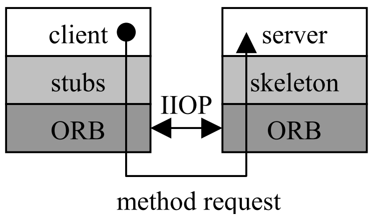

Figure 2

The IDL interface is a module which acts as container for the related methods. An implementation for the IDL interface is:

```
module ExchangeData {
  interface Exchange {
    long getData(in long index);
    void putData(in ling index, in long data); }; };
```

This module is compiled with JIDL. After compilation, the java code for the interface will result as follows:

```
package ExchangeData;
public interface ExchangeOperations {
  int getData(int index);
  void putData(int index, int data);}
```

The implementation for the interface, generated in Java is:

```
package ExchangeData;
public interface Exchange extends ExchangeOperations;
org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity{}
```

Thus, the server implementation will be:

```
import ExchangeData.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import java.util.*;
import java.text.*;
class ExchangeServer extends _ExchangeDataImplBase{
  int dat[]=new int[10];
  public int getData(int index) { return dat[index]);}
  public void putData(int index, int data) { dat[index]=data;} }
public class ObjectServer {
  public static void main(String[] args) throws Exception {
    try {
      ORB orb=ORB.init(args, null);
      ExchangeServer excObjRef=new ExchangeServer();
      orb.connect(excObjRef);
      org.omg.CORBA.Object objRef=orb.resolve_initial_references(“NameSevice”);
      NamingContext ncRef=NamingContextHelper.narrow(objRef);
      NameComponent nc=new NameComponent(“Exchange”,””);
      NameComponent[] path={nc};
      NcRef..rebind(path, excObjRef);
      java.lang.Object sync=new java.lang.Object();
      synchronized(sync);
      sync.wait(); }}
  catch (Exception ex) { System.out.println(“Exception”+ex);}}
```

The implementation for the client program is the following:

```
import ExchangeData.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
public class ExchangeClient {
  public static void main(String[] args) throws Exception{
    int value=1;
    try {
      ORB orb=ORB.init(args,null);
      org.omg.CORBA.Object objRef=orb.resolve_initial_references(“NameSevice”);
      NamingContext ncRef=NamingContextHelper.narrow(objRef);
      NameComponent nc=new NameComponent(“Exchange”,””);
      NameComponent[] path={nc};
      Exchange excObjRef=ExchangeHelper.narrow(ncRef.resolve(path));
      for (int k=0;k<10;k++) {
        schObjRef.putData(3,value);
        value=schObjRef.getData(5); }}
    catch (Exception ex) {System.out.println(“Exceptir”+ex);}}}
```

4. DISTRIBUTED COMPONENT OBJECT MODEL

DCOM, developed by Microsoft, implements remote objects architecture by using a protocol called the Object Remote Procedure Call (ORPC). The ORPC layer is built on top of DCE RPC and interacts with COM's run-time services. A DCOM server is a body of binary code that is capable to instantiate objects of a particular type at runtime. DCOM server object can support multiple interfaces, which represent a different behavior of the object. A DCOM client calls the exposed methods of a DCOM server by acquiring a pointer (*pVtbl*) to one of the server object's interfaces. The client object then starts calling the server object's exposed methods through the acquired interface pointer as if the server object resided in the client's address space. As specified by COM, a server object's memory layout conforms to the C++ *vtable* layout.

COM supports both static and dynamic invocation of objects. In the static invocation method, the code for the proxy and stub is created by the Microsoft IDL (MIDL) compiler and then is registered in the systems registry. This is the *vtable* method of invoking objects. In the dynamic invocation case, COM objects implement the *Idispatch* interface. Then, type libraries, which are binary files that describe the object, are used to "understand" the object behaviour. COM 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*. The DCOM IDL file also associates the *IExchange* interface with an object class Exchange, as shown in the coclass block. Each interface has assigned a Universally Unique IDentifier (UUID), which is called the Interface ID (IID). Similarly, each object class has assigned a unique UUID called a CLasS ID (CLSID). COM gives up on multiple inheritance to provide a binary standard for object implementations. Instead of supporting multiple inheritance, COM uses the notion of an object having multiple interfaces to achieve the same purpose. This also allows for some flexible forms of programming.

The IDL is generated as follows:

```
[
uuid(0af84600-2f51-1670-b521-00ac12b6500),
version(1.0)
]
library ExchangeData
importlib("stdole32.tlb");
[
uuid(EFC034D0-5B45-1BD2-19C5-0B05614C6550),
dual
]
interface IExchange : IDispatch
{
HRESULT getData([in] int index, [out, retval] int * rtn);
HRESULT putData([in] int index, [in] int value);
}
[
uuid(EFC034D0-5B45-1BD2-19C5-0B05614C6550),
]
coclass Exchange
{
interface IExchange;
};
};
```

The implementation of the server could be:

```
import com.ms.com.*;
class ExchangeServer implements IExchange{
  private static final String CLSID = " EFC034D0-5B45-1BD2-19C5-0B05614C6550";
  int dat[]=new int[10];
  public int getData(int index) { return dat[index]);}
  public void putData(int index, int data) { dat[index]=data;} }
}
```

The client program is the following:

```
public class ExchangeClient {
  public static void main(String[] args) ) throws Exception {
    int value=1;
    try {
       IExchange exc = (IExchange) new exchangedata.Exchange();
       for (int k=0;k<10;k++) {
         exc.putData(3,value);
         value=exc.getData(5);  }}
     catch (com.ms.com.ComFailException ex) {
       System.out.println( "COM Exception:" );
       System.out.println( ex.getHResult() );
       System.out.println( e.getMessage() ); }}}
```

5. CONCLUSIONS

In the paper, three paradigms of manipulating distribute objects over heterogeneous networks were presented.

Java/RMI can be used on diverse operating system platforms, UNIX, Windows etc., to handheld devices as long as there is a Java Virtual Machine (JVM) implementation for that platform. But, the programs can be implemented only in Java.

CORBA is a specification, it also can be used on diverse operating system to handheld devices as long as there is an ORB implementation for that platform. It is not necessary to implement CORBA servers and clients in Java.

COM specification is at the binary level and it allows DCOM server components to be written in diverse programming languages like C++, Java, Object Pascal (Delphi), Visual Basic, etc. As long as a platform supports COM services, DCOM can be used on that platform. DCOM is heavily used on the Windows platform, but as well on UNIX-based platforms.

6. REFERENCES

[1] Gopalan, S.R. A Detailed Comparison of CORBA, DCOM and Java/RMI. www.execpc.com/~gopalan/misc/

[2] Ashbury, S, Weiner, S.R. Developing Java Enterprise Applications. Wiley Computer Publishing, ISBN 047-1327565, 1999.

[3] Eckel, B. Thinking in Java, 2nd Edition, Prentice-Hall, ISBN 13-127363-5, London, 2000.

[4] Nogorotham, M. Java Networking & AWT API SuperBible. Macmillan Computer Publishing, ISBN 157-169031x, 1996.