Integrating Java and CORBA:

A Programmer's Perspective

Martin Schaaf • *University of Kaiserslautern*
Frank Maurer • *University of Calgary*

The introduction of Java's proprietary remote method invocation (RMI) with version 1.1 of the Java Development Kit simplified the challenging task of developing distributed object-based systems. RMI provides convenient integration with Java; however, it lacks interoperability with other languages. The Object Management Group's common object request broker architecture (CORBA), on the other hand, is a platform- and language-neutral specification for developing distributed object systems. CORBA provides services not covered by RMI, such as managing transactional safety and persistency.

In this issue's *Spotlight*, we use a small chat room application to describe how a programmer can combine Java RMI's ease of use with CORBA's language neutrality. We start with an implementation based on a set of distributed objects using RMI. We then adapt the example to CORBA or, more specifically, the RMI-over-IIOP (Internet inter-ORB protocol) specification developed by Sun and IBM. RMI-over-IIOP obviates the need to learn another language, namely the interface definition language (IDL), to work with CORBA; it provides nearly full RMI semantics by incorporating the Java-to-IDL mapping. The necessary Objects-by-Value CORBA enhancement is now standardized and part of the IDL language as well. Since JDK 1.3 supplies an ORB, a Java developer does not need additional software to work with CORBA.

Before presenting the example, we briefly introduce RMI and CORBA and their terminologies. Even a short introduction describing the functionality of both specifications would go far beyond the scope of this tutorial. We therefore encourage you to look at the additional literature listed in the sidebar, "Java RMI and CORBA Resources."

Before starting this tutorial, you should make sure that the new JDK 1.3 is installed on your machine (available online at http://java.sun.com/j2se/1.3/). The examples in this tutorial can be downloaded from *IC Online* (http://computer.org/internet/v5n1/rmitut.htm) or http://wwwagr.informatik.uni-kl.de/~schaaf/rmitut/. We present only a few central code fragments here.

Introduction to RMI

RMI was first included in JDK 1.1 to allow easy development of distributed applications without additional third-party software. Specifying a remotely accessible object starts with defining a Java interface that must inherit from `java.rmi.Remote`. The remote interface specifies every remotely callable method. Parameter types used in remote methods can either be primitive types (like float), or they can be serializable objects or remote interfaces. Serializable objects will be passed as copies using the Java object serialization service. A remotely enabled object will be passed as a remote reference.

After specifying a remote interface, you must provide an implementation class. Here, you can specify additional methods, but only those stated in the remote interface will be remotely accessible. At runtime, the implementation class must be exported after creation. This is done either by inheriting the implementation class from `java.rmi.server.UnicastRemoteObject` or by an explicit call to the `exportObject` static method of that class. Finally, you must create the stub and skeleton classes with the *rmic* compiler that comes with the JDK. These classes enable remote access to the implementation methods (see Figure 1).

Imagine the Caller-object holding a remote reference to the remote object `x` represented by the remote interface and the implementation class `x_Imp`. Rather than invoking a method of `x_Imp` directly, the Caller-object invokes a method of

 1089-7801/00/$10.00 ©2001 IEEE

x_Stub, which is type-compatible because it also implements the remote interface x. The stub marshals the parameters and handles the transmission to the skeleton, x_Skel, that gets created on the server side when exporting the remote object x. The protocol between stub and skeleton is the Java remote method protocol (JRMP). The skeleton unmarshals parameters, invokes the desired implementation method, and passes the results and exceptions. If the result type is a class that implements a remote interface itself (let's say, Y_Imp), the skeleton creates and passes the associated serialized stub containing the remote reference instead. Because the stub is type-compatible with the remote interface, a client awaiting a result of that type will be satisfied. It now holds another stub that redirects every invocation to the remote object Y.

While this is typically how a client gets a remote reference, it has been assumed that the client already had a remote reference to x. For retrieving initial references, a name service associates remote references with names. RMI provides the registry—a simple transient naming service that is a remote object itself—for this purpose. Last but not least, RMI includes distributed garbage collection, which unexports remote objects safely in order to free resources on the server.

Further information is available at the Java RMI homepage (http://java.sun.com/j2se/1.3/docs/guide/rmi/) and the RMI specifications page (http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html). Several tutorials cover other aspects of RMI such as activation or the usage of socket factories.

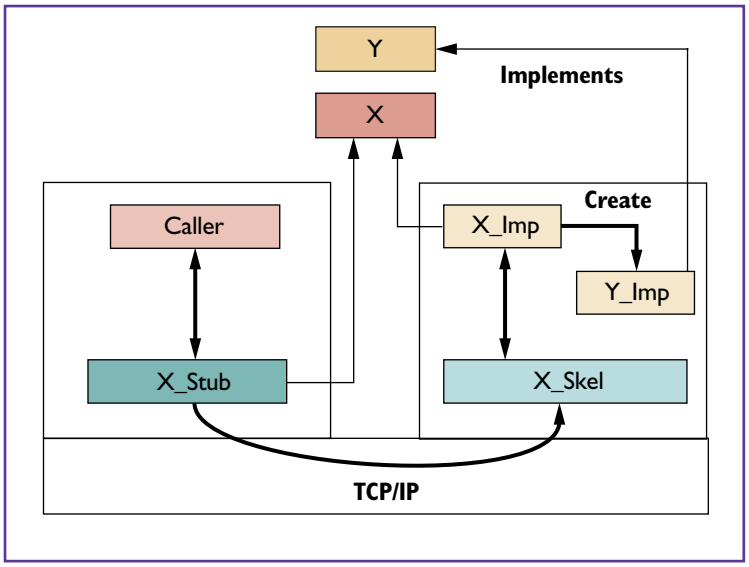

Figure 1. RMI stubs and skeletons.

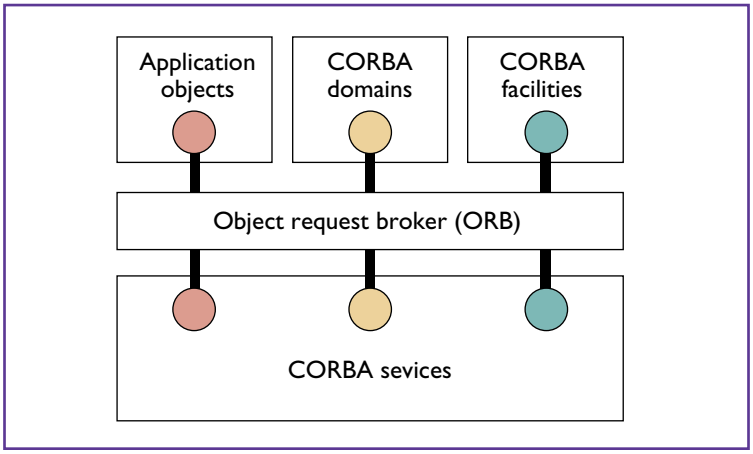

Figure 2. Object management architecture (OMA).

Introduction to CORBA

The term CORBA refers to a set of specifications maintained by the OMG and first released in July 1992. CORBA aims to provide

- access to services,
- discovery of resources and object names,
- error handling, security policies, and
- language and platform neutrality.

Figure 2 depicts CORBA's highest-level specification: the object management architecture (OMA).

The heart of the OMA is the ORB that mediates the information flow between objects. Every object participating in a CORBA network is connected to an ORB that comes, in practice, as a programming library suitable for the programming language used in a given software development project. Communication between ORBs is standardized by the general inter-ORB protocol (GIOP). IIOP is a version of this protocol, specialized for TCP/IP-based networks. These standards are the key to the interoperability CORBA provides among objects written in different languages.

CORBA services are standardized objects providing functionalities like naming, persistence, and transactions. CORBA domains describe some vertical market standards (for instance, financial services). Facilities like information management and systems management, which are common to all domains, are grouped within the CORBA facilities. Neither are within the scope of this tutorial.

The IDL is another important CORBA standard; it describes the interface of a remote CORBA object. IDL definitions are comparable to RMI's remote interfaces. Because CORBA is language neutral, IDL can be viewed as the lingua franca for describing services implemented in heterogeneous languages.

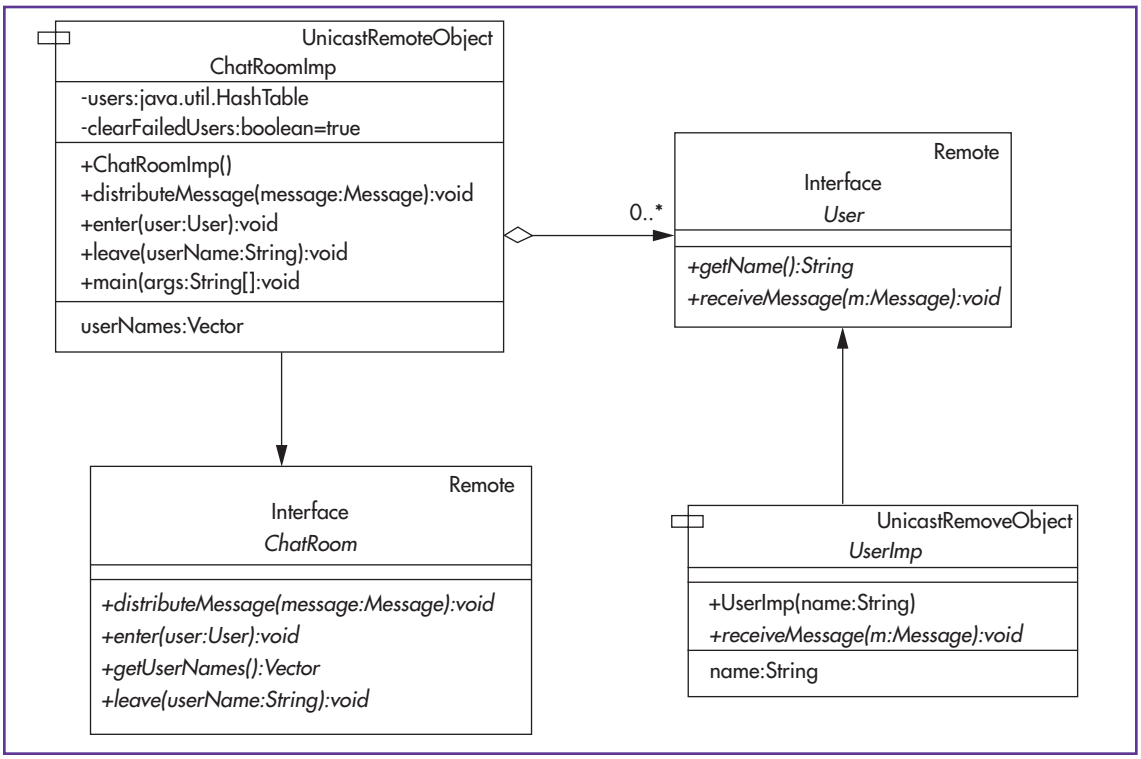

Figure 3. UML diagram for object types in example RMI-based chat application.

IDL consists of several basic data types mainly originating from C, such as `array`, `union`, or `struct`. To allow a concrete programming language to implement CORBA objects, the OMG has standardized a set of mappings from IDL to languages such as C, Cobol, Lisp, and Smalltalk. The IDL-to-Java mapping is the base for the tool *idlj*, which can be used for generating Java stubs, skeletons, and ties from IDL definitions.

Originally, CORBA only allowed primitive data types or remote references to be passed by remote invocations of methods. The current IDL definition incorporates the Objects-by-Value suggestions from Sun and IBM, so it is now possible to transfer object state and behavior as with RMI. Relying heavily on this feature, the Java-to-IDL mapping lets a Java programmer develop CORBA-based applications without having to learn IDL. The developer can specify interfaces within Java using the interface construct. The new *rmic*, which is included in JDK 1.3, can create CORBA stubs, skeletons, and ties directly from interface definitions in Java. It is also possible to generate an IDL specification automatically and use it as input for other IDL compilers to create services or clients in other programming languages.

RMI Example

Our example RMI-based chat application uses peer-to-peer communication: the client talks to the server, and the server actively sends messages to the client. Figure 3 shows the Unified Modeling Language diagram introducing the object types involved.

A *user* consists of the interface `User` and the implementation class `UserImp`. A *chat room* consists of the interface `ChatRoom` and the implementation class `ChatRoomImp`. A chat room aggregates remote interfaces of type `User`.

To enter a chat room, a user presents a previously created `UserImp` object as argument to the chat room's `enter` method. Because `UserImp` is an implementation class of a remote object and `enter` is a remote method of the chat room, a stub to `UserImp` will be created and passed to the chat room. The chat room then possesses a remote reference of the `user`, which is collected in a hash table. To send a message, a user invokes the `distributeMessage` method, resulting in the execution of the code shown at *1* in Figure 4.

Note that invocation of remote methods might result in a remote exception. In our example, this likely indicates that a user has terminated the client without leaving the chat room first. To stabilize the chat room, the remote references to these users should be deleted because invoking methods of unreachable remote objects can be very time consuming. Whenever the chat room observes a failed user, it informs all other users by invoking the leave method so they can update their view.

```
package example.rmi;
import java.rmi.Naming;
import java.rmi.server.UnicastRemoteObject;
…

public class ChatRoomImp extends UnicastRemoteObject implements ChatRoom {
   private java.util.Hashtable users = null;
   private boolean clearFailedUsers = true;
   …

   public synchronized void distributeMessage(Message message) { *1*
    Vector failedUsers = new Vector();
    for (Enumeration e = users.keys(); e.hasMoreElements(); ) {
        String userName = (String)e.nextElement();
      User user = (User)users.get(userName);
      try {
        user.receiveMessage(message);
      } catch (RemoteException ex) {
        failedUsers.addElement(userName);
      }
    }
    if (!clearFailedUsers) return;
    clearFailedUsers = false;
    for (Enumeration e=failedUsers.elements(); e.hasMoreElements(); ) {
      leave((String)e.nextElement());
    }
    clearFailedUsers = true;
  }

  public synchronized void leave(String userName) {
    Message message = new UserLeftMessage(userName);
    distributeMessage(message);
    users.remove(userName);
  }
  …

  public static void main(String[] args) {
    if (args.length < 1) {
      System.out.println("Required argument: <name to register>");
      return;
    }
    try {
      Naming.rebind("rmi://localhost/" + args[0],
                  new ChatRoomImp());
      *2*
    } catch (java.net.MalformedURLException e) {
      e.printStackTrace();
    } catch (RemoteException e) {
      e.printStackTrace();
    }
  }
}
```

Figure 4. Chat room implementation class.

```
public void connect()throws ChatException,NotBoundException,
MalformedURLException,RemoteException,
UnknownHostException {
…
ChatRoom chatRoom =(ChatRoom)java.rmi.Naming.lookup(getURLTextField().getText());
chatRoom.enter(getUser());
…
}
```

Figure 5. Retrieving a remote reference from the RMI naming service.

Figure 6. Chat application client.

The `clearFailedUsers` flag avoids multiple invocations of `leave` in case another user fails while multicasting the leave message. All chat room methods are synchronized to prevent users from entering or leaving during message distribution.

In the main method, a reference to a chat room implementation object is registered to a name you provide. It is assumed that a registry is running on the same host as the implementation object before you start the server.

The client can retrieve the registered remote reference using the code fragment shown in Figure 5. Here, the URL will be taken from a text field the user inputs.

Compiling and Running the Example

We are now ready to compile and start the example. We assume that you have successfully downloaded and unzipped the complete source code into a directory named `<top>` (included in your CLASSPATH) and preserved the file structure of the zip-file. On Windows machines, you can compile and create RMI stubs and skeletons by invoking the following commands:

```
cd <top>
javac example\rmi\*.java
javac example\rmi\ui\*.java
rmic example.rmi.ChatRoomImp
rmic example.rmi.UserImp
```

Before running the chat room, start up the registry. Make sure that all class definitions of objects you want to register are present in the registry's CLASSPATH.

```
rmiregistry.exe
```

Now you can start the server and client with:

```
java example.rmi.ChatRoomImp
    <name to register>
java example.rmi.ui.ChatClient
```

You should see a small window, similar to the one in Figure 6, where you have to provide the chat room's URL and your desired username.

RMI-over-IIOP Example

We will now move on to RMI-over-IIOP by migrating the example. The programmer does not have to change much; providing some additional arguments to the *rmic* stub compiler coming with JDK 1.3 mainly does the job. JDK 1.3 adopts the new RMI-over-IIOP specification. Rather than specifying an IDL interface, we can reuse our Java interface definitions from the RMI-based example. Other than that we just have to make some minor changes to the implementation code. Since Java now contains an ORB, no additional software from third-party vendors is required.

The `ChatRoomImp` implementation class must inherit from `PortableRemoteObject` of the package `javax.rmi.*` instead of `UnicastRemoteObject` (see *1* in the code fragment in Figure 7). Again, we could alternatively choose to export the object directly by calling `PortableRemoteObject.export`. Both alternatives implicitly initialize the ORB. We have to apply the same changes to the `UserImp` class.

The next change (see *2* in Figure 7) concerns registration of the chat room at the name service. We use the generic Java naming and directory interface (JNDI) for this. JNDI defines a set of operations

for a naming service; an actual implementation can be a CORBA naming service that we indicate at the command line on startup. JNDI will also be used when retrieving the reference to the chat room service object *3* shown in Figure 8. In that code fragment it is assumed that the user has provided a name we can retrieve with the `getNameText` method.

The last change in our implementation code is to use the static `narrow` method of `PortableRemoteObject` to type-cast the remote reference, when receiving a `type-unknown` reference from the name service. This ensures type-safety.

Compiling and Running the Example

After completing these changes, we are ready to compile and start the example. Compilation and creation of CORBA stubs, ties, and skeletons is done by invoking the following commands:

```
cd <top>
javac example\rmiiiop\*.java
javac example\rmiiiop\ui\*.java
rmic -iiop -always
example.rmiiiop.ChatRoomImp
rmic -iiop -always
example.rmiiiop.UserImp
```

Invoking the *rmic* with the –iiop parameter creates the stubs/skeletons/ties directly. If you want to preserve the IDL file generated from the interface specifications, you can use the –idl parameter.

Now we start the transient name server that comes with Java. You can think of it as a CORBA equivalent for the RMI registry.

```
tnameserv.exe
```

Create and bind a chat room remote object to a given name.

```
java -
Djava.naming.factory.initial=com.sun.j
ndi.cosnaming.CNCtxFactory
-
Djava.naming.provider.url=iiop://local
host:900
    example.rmiiiop.ChatRoomImp <name
to register>
```

As you can see here, we have defined some additional properties. With the first property, we specify the CORBA naming service factory. Remember that the JNDI interface is generic in the sense that it can interface to arbitrary naming services. Finally, we provide the same properties when invoking the client.

```
package example.rmiiiop;
import javax.naming.*;
import javax.rmi.PortableRemoteObject;
…

public class ChatRoomImp extends
PortableRemoteObject implements ChatRoom { *1*

…

  public static void main(String[] args) {
    if (args.length < 1) {
      System.out.println("Required argument:
                        <name to register>");
      return;
    }
    try {
      Context nc = new InitialContext();    *2*
      nc.rebind(args[0], new ChatRoomImp());
    } catch (NamingException e) {
      e.printStackTrace();
    } catch (RemoteException e) {
      e.printStackTrace();
    }
  }
}
```

Figure 7. RMI-over-IIOP adapted implementation class of a chat room.

```
public void connect() throws ChatException,
NamingException,
                            RemoteException {
  …
  Context nc = new InitialContext();
  Object obj = nc.lookup(getNameText());   *3*
  ChatRoom chatRoom = (ChatRoom)
PortableRemoteObject.narrow(obj,

ChatRoom.class);
  *4*
  chatRoom.enter(getUserInternal());
  …
}
```

Figure 8. Retrieving a remote reference from a JNDI naming service.

Java RMI and CORBA Resources

Further information on Java RMI and CORBA is available online at a number of places.

Java Remote Method Invocation (RMI)homepage • http://java.sun.com/j2se/1.3/docs/guide/rmi/.

RMI specifications • http://java.sun.com/j2se/1.3/docs/guide/rmi/spec/rmiTOC.html.

RMI tutorials • http://java.sun.com/j2se/1.3/docs/guide/rmi/getstart.doc.html, http://java.sun.com/j2se/1.3/docs/guide/rmi/rmisocketfactory.doc.html, and http://java.sun.com/j2se/1.3/docs/guide/rmi/activation.html.

RMI-over-IIOP at IBM • http://www.ibm.com/java/jdk/rmi-iiop/.

RMI-over-IIOP at Sun • http://java.sun.com/products/rmi-iiop/.

RMI-over-IIOP programmer 's guide • http://www.ibm.com/java/jdk/rmi-iiop/docs/aix130/rmi_iiop_pg.html.

Java IDL • http://java.sun.com/j2se/1.3/docs/guide/idl/.

Object Management Group homepage • http://www.omg.org.

CORBA mapping specifications • http://www.omg.org/technology/documents/formal/corba_language_mapping_specifica.htm.

CORBA Technology and the Java platform • http://java.sun.com/j2ee/corba/.

Objects-by-Value specification • ftp://ftp.omg.org/pub/docs/orbos/98-01-18.pdf.

```
java -
Djava.naming.factory.initial=com.sun.j
ndi.cosnaming.CNCtxFactory
-
Djava.naming.provider.url=iiop://local
host:900
example.rmiiiop.ui.ChatClient
```

If everything works well, you should now see a window like the one in Figure 6.

Conclusion

Within this tutorial, we adapted an example RMI-based application to the new RMI-over-IIOP standard. While CORBA for Java has been available for a fairly long time, it has never been so easy to develop distributed applications based on this standard. Instead of specifying remote interfaces with CORBA IDL, it is now possible to use Java interfaces. With the new standardized Java-to-IDL mapping, it is even possible to pass objects by value in remote method invocations. This extends the original CORBA semantics of remote method calls to RMI semantics. Because the Java-to-IDL standard is relatively new, we cannot expected all existing CORBA systems to support it. With RMI-over-IIOP, however, your Java applications can now connect easily to arbitrary CORBA object services provided by software vendors like GemStone or Bea because the applications can now talk IIOP.

Frank Maurer is codirector of the Alberta Software Engineering Research Consortium (ASERC) and an associate professor at the University of Calgary. He has a PhD in computer science from the University of Kaiserslautern, Germany. He is a member of the *IEEE Internet Computing* editorial board.

Martin Schaaf is working on a PhD at the University of Kaiserslautern. His research interests include knowledge management techniques and distributed systems. He received the Diplom from the University of Kaiserslautern.

Readers can contact the authors at maurer@cpsc.ucalgary.ca or schaaf@informatik.uni-kl.de.