Java distributed objects: Using RMI and CORBA

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. About this tutorial

What is this tutorial about?

As computers become increasingly connected, there is a need to leverage distributed computing platforms to allow data sharing, improve performance through parallelism, and place function where it is best performed.

Distributed computing requires a programming model that makes it easy for application programmers to create and maintain these distributed applications. The Java platform is a great vehicle for the distributed computing architecture.

When the Java language exists on both ends of a communication line, the Remote Method Invocation (RMI) API and infrastructure, part of J2SE, provides support for this programming model. In the case where Java coding exists only on one side and another language rules on the other end, the Java language maintains a function designed to operate with the industry standard Common Object Request Broker Architecture (CORBA).

In this tutorial, we will examine distributed object concepts and determine how these concepts are implemented in Java programming using both RMI and CORBA technologies.

Should I take this tutorial?

This is an intermediate-level tutorial; it assumes you know how to read and write basic Java programs, both applications and applets.

If you are already a Java programmer and have been curious about distributed objects (including topics like RMI and CORBA) and the Java technologies that support them (for example, Java IDL and RMI-IIOP), then this tutorial is for you.

This tutorial introduces the basic concepts of distributed objects, then walks you through examples that highlight both RMI (using applications and applets) and CORBA technologies; the examples will use the Java IDL capability. The Java RMI-IIOP function will also play a role in the discussed material. The same example is executed using multiple technologies so you can compare and contrast how the programming is handled in each technology.

Tools, code samples, and installation requirements

To complete this tutorial you will need the following:

- The *Java 2 platform, Standard Edition, version 1.4*
- A standard Java language-compatible editor
- The tutorial *source code* so that you can follow the examples as we go along

About the author

Brad Rubin is principal of Brad Rubin & Associates Inc., a computer-security consulting company specializing in wireless network and Java application security and education. Brad spent 14 years with IBM in Rochester, MN, working on all facets of the AS/400 hardware and software development, starting with its first release. He was a key player in IBM's move to embrace the Java platform, and was lead architect of IBM's largest Java application, a business application framework product called SanFrancisco (now part of WebSphere). He was also Chief Technology Officer for the Data Storage Division of Imation Corp., as well as the leader of its R&D organization.

Brad has degrees in Computer and Electrical Engineering, and a Doctorate in Computer Science from the University of Wisconsin, Madison. He currently teaches the Senior Design course in Electrical and Computer Engineering at the University of Minnesota, and developed and teaches the University's Computer Security course. Contact Brad at *BradRubin@BradRubin.com*.

Section 2. Distributed objects 101

How non-distributed programming works

For most conventional (non-distributed) Java programming, all the objects that make up a program are present on the same machine and in the same process or JVM.

When one object calls a method on another object, the *calling* object knows the memory address of the *called* object and can just change the machine program counter to start running in the new object. The called object's memory address is saved on the machine stack, so when the called method returns, the address on the stack is reloaded into the machine program counter and execution continues back in the calling object. This sequence is common to all programming languages that execute in a single process, on a single machine.

So how does distributed programming differ from non-distributed? What mechanisms support distributed programming functions?

How distributed programming is different

In distributed computing, program modules can exist in different processes on the same machine or in different machines. We would like to preserve the same semantics described on the previous panel, but conventional CPU architectures do not support memory addresses that exist in other machine address spaces.

In addition, the different machines can have differences in fundamental architecture, such as integer size, floating point format, and endian order. This further complicates the interaction between program models on different platforms.

There are other thorny issues that arise in distributed programming, such as remote garbage collection. In a non-distributed Java application, the JVM knows when an object is no longer referenced, so it can reuse the memory the object is using. We want a distributed object infrastructure that supports this function even when the objects are spread across multiple virtual machines.

To accomplish this, we need some intermediate machinery to make distributed method calls look, to the application programmer, just like local method calls. Ideally, this machinery should take care of the resolution of both method calls and returns in different address spaces, converting data if the architecture differences warrant it.

Remote Method Invocation (RMI) and Common Object Request Broker Architecture (CORBA) are two examples of APIs and middleware that support distributed method calls.

Why the Java platform works well for distributed object programming

The Java platform is ideal for distributed programming for several reasons:

- The Java platform defines a consistent size and format for basic types, such as integer

and float, as well as endian assumptions for all the platform implementations. This means that the transport mechanism can be significantly simplified because it doesn't have to do any translation or adjustment when talking to the other side. This has performance advantages as well.

- The main Java distributed-programming API, RMI, is part of the standard edition of the language and is therefore available on all Java-supported systems.
- The Java RMI interface is fairly easy to use.
- The Java classloader can automatically download client code for thick client-server or peer-to-peer applications.
- Java parameters can be passed by reference (remote) or by value (serialized).
- If you can not assume the Java language exists on all sides of a distributed application, you can still use the language to call objects on another system that are written in another programming language, such as C++, by using the CORBA support.
- The pervasiveness of the Java platform, and its use as an instructional language in many programming courses, means that there is an available pool of programming skill for the environment.
- Lastly, the Java platform also supports many of the other techniques for distributed programming, like the socket programming model.

Distributed architectures

So why would we want to distribute computing?

- Distributed computing is used when there is a central resource -- like a database -- that we want to share with multiple users or clients.
- Distributed computing is used to leverage the combined computing power of multiple systems to solve a problem more efficiently or quickly than can be done with a single system.

Multiple computer systems can be configured in several ways to share processing, including sharing memory, sharing disk, or sharing nothing but a common communication channel. The latter technique allows systems that are physically far apart to cooperate in solving computing problems.

On the topic of leveraging computing power, the rise of the Internet and the accompanying communications protocol TCP/IP has led to unprecedented connectivity of millions of computer systems. For some applications, it is desirable to leverage this wealth of computing power to solve problems. Even more compelling, most computer systems have ample idle

time and might as well help solve other problems. In the future, Grid computing can harness this distributed computing power to be sold, much like an electric utility sells electrical power.

Distributed architecture models

Let's examine three models used to leverage distributed computing:

- **Thin client-server.** This model is extremely pervasive today, with the popularity of the Web browser. In this model, there is no application-specific code that runs on the client -- the client makes requests of and receives responses from the server through a Web browser; the server executes the requests and returns responses to the client. The nicest attribute of this model is that a user can execute arbitrary applications with no setup or installation, and the user can execute these programs from any connection point, whether inside a company or through any Internet-accessible vehicle. This model uses little distributed computing, except for the user interface and some basic format checking and light function through languages like JavaScript. RMI can be, but is generally not, used as a protocol in these environments. HTTP is the common protocol in this model.

- **Thick client-server.** This model is used when more client-side processing is either necessary (from a functional standpoint) or desirable (from a performance standpoint). Thick clients can provide a user with a richer application experience. The trade-off, however, is that there must be an install step to get the client code for the application on the client machine. Java applets can ease this installation by automatically downloading the client code. RMI is a common API for this model.

- **Peer-to-peer.** if we put both ends of a client-server relationship on an equal footing, allowing either end to initiate a request to the other, we have a peer-to-peer model. This model is growing in importance and popularity; many of the MP3-sharing programs use this model. RMI can be used as a protocol in these environments.

Techniques for distributed object programming

There are a wide variety of techniques available for communicating information, remotely executing code, and coordinating processing between processes, whether on the same machine or between distant machines.

One of the earliest, and still common, techniques is to open a communication link between processes. Once this link is established, one process can send a stream of data, which encodes certain actions and information, to another process, which decodes this stream and performs an action, often returning results back to the initiating process using the same technique. Socket programming is an example of this technique; the HTTP protocol that moves HMTL-encoded information between client and server is another example.

The problem with this technique is that a programmer must invent a protocol and command-encoding scheme for each application. This technique also results in two different programming paradigms that make programming more difficult -- a procedure-calling/parameter-passing model native to the programming language and a command model for the socket programming. Also, there may be differences in data format and endian-ordering assumptions between the two machines, further complicating the code.

Another common model is Remote Procedure Calls (RPCs). Here, a programmer can call a remote procedure and pass parameters back and forth just like they were making a local procedure call. This allows the same programming style to be used for both local and distributed programming.

Still, the data format and endian-ordering issues exist, and RPCs do not follow the object-oriented model.

Let's look at RMI as a solution.

RMI as a technique for distributed programming

RMI is the Java solution for the "object-RPC" objective we had in the previous panel.

RMI allows for the same (well, almost the same, as we will see) programming model for both local and distributed object-oriented method calls, assuming the Java language is used for both the caller and the called method.

RMI allows parameters to be passed in one of two ways. With parameter passing by value, a new object is created in another process and a copy of the parameter is made (using a process called *serialization*) into the new object. With parameter passing by reference, the object is not created in another process, but remains in place and is operated on remotely.

The processes of serialization and pass-by-value are outside the scope of this tutorial. An introduction to Java serialization can be found in Resources on page 18 .

Distinctions between RMI and CORBA

RMI is part of the JSDK and is an API that supports remote method calls and returns for distributed computing applications. It assumes that the Java programming language is used by both the caller and the called method.

CORBA is an industry-wide standard for allowing remote method calls and returns, but unlike RMI, it can be used when the caller and the called method use different programming languages, including the case where neither side uses the Java language.

RMI is a simpler API because knowing that both sides support the same language and machine architecture makes the remote method call problem an easier one to solve.

The JSDK provides support for RMI-CORBA, which allows a Java object to call a CORBA object using two different approaches. We will discuss this topic later in the tutorial.

Section 3. RMI concepts and examples

RMI overview

In a non-distributed Java application, code in an object can call a method in another object and the JVM resolves the addresses and passes any parameters from the caller to the called method; it also passes any return parameters back to the caller when execution resumes in the caller.

In a distributed Java application, although the method programming code looks the same as in the non-distributed case, a completely different mechanism is used to hook up these objects. When the object wants to call a method, it instead calls a companion client object -- known as a *stub* -- that represents the called server object.

This stub invokes the RMI infrastructure on the client and moves data across the network to the RMI infrastructure on the server, which in turn invokes a companion server object called a *skeleton*.

This skeleton object then calls the actual server object's method. Upon return, the same mechanism is invoked in reverse, so the return parameters are passed to the server skeleton object, then passed across the network using the RMI infrastructure, calling the client stub object, and then returned to the actual calling object.

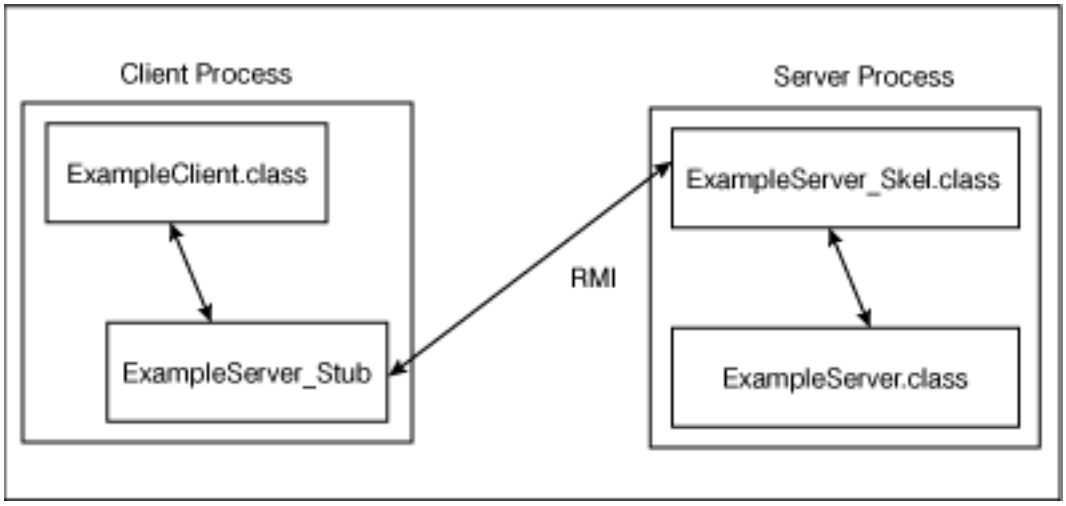
The beauty of the RMI distributed-object infrastructure is that the programmer just programs method calls as if the called object were present in its JVM.

Stubs and skeletons

Stubs and skeletons are generated from the calling and called objects using an RMI compiler tool called *rmic*. The application packager must ensure that stubs are included in the client code or JAR file, and the skeletons are included in the server code or JAR file.

Stubs can be automatically downloaded from a Web server on demand. (There is a way to run RMI without skeletons, which helps ease some of the deployment tasks, but it is beyond the scope of this tutorial.)

The following diagram illustrates the data flow between RMI stubs and skeletons:



Let's build some objects and look at the subtle differences between local and distributed programming requirements.

Building the objects

There are almost no differences between local and distributed programming with RMI. Distributed programming has just a few programming requirements.

All classes that can be called remotely must have both an *interface* and an *implementation*. The interface must extend from a Java class called `Remote`. The implementation not only implements the interface, but it must also extend the implementation of the Java class called `UnicastRemoteObject`.

There is a communication network that exists under the remote method call mechanism, and this network can have errors or disconnect completely. Or, the server machine can disappear too. So, the client program has some errors that it must handle that occur in distributed programming. These errors manifest themselves as a `RemoteException`, so all methods on server objects that are designed to be called remotely must declare that they throw `RemoteException`. Client code should handle these errors if they arise.

Here is a simple interface for a class that sets and gets a `String`:

```
// Example.java
//
// Interface for remote object
//
import java.rmi.*;

public interface Example extends Remote  {

  public void setString( String s ) throws RemoteException;

  public String getString() throws RemoteException;
}
```

And here is the implementation:

```
// ExampleServer.java
//
// Remote Object implementation for Example interface
//
import java.rmi.*;
import java.rmi.server.*;

public class ExampleServer extends UnicastRemoteObject
  implements Example {

    private String stringState;

    public ExampleServer() throws RemoteException{}

    public void setString( String s ) throws RemoteException{
      stringState = s;
    }

    public String getString() throws RemoteException{
      return stringState;
    }
}
```

Next, we need to create a stub and skeleton for the `ExampleServer` class:

```
D:\RMICORBA> rmic ExampleServer
```

On the next panel we'll see how to locate the remote object.

The RMI registry

We must be able to locate the remote object, so RMI uses a name server called the *RMI registry* to perform this function.

An object has a name (in this case it is "Example") and a machine location (in this case it is "localhost"). In the following code, we create an `ExampleServer` object and make an entry for it in the name server located on localhost and give it a name of Example.

We use the `rebind()` method here to avoid an error if it already exists; otherwise we could use the `bind` method. We only need to run this code one time, as long as the registry process stays active. The registry is not persistent.

```
// Server.java
//
// Server program to create the "Example" remote object and enter it
// into the RMI registry
//
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;

public class Server {

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

```
    RemoteException, java.net.MalformedURLException {
  //
  // Create a new example object and enter it into the RMI registry
  // located on "localhost" under the alias "Example"
  ExampleServer es = new ExampleServer();
  Naming.rebind( "rmi://localhost/Example", es );
  }
}
```

Next, we'll see how to build a client program to make remote method calls.

Building a client program

Finally, we create a client program that makes remote method calls.

We first must locate the server object, so we look up the `Example` object in the RMI registry and then call a method to set the string and then read it back and print it to the screen.

```
// ExampleClient.java
//
// Client program calling Example remote object to set a string and then
// get it.
//
import java.rmi.*;

public class ExampleClient {

  public static void main ( String[] args ) {
    try {
      // Find remote "Example" object in the RMI registry
      Example example = (Example) Naming.lookup( "Example" );
      //
      // Set string and then get it again to display
      example.setString( "Success!" );
      System.out.println( example.getString() );

    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

To execute the example, we have to start up the RMI registry, run the server program once, and then we can run the client:

```
D:\RMICORBA> start rmiregistry
D:\RMICORBA> start java Server
D:\RMICORBA> java ExampleClient
```

You should see the phrase, "Success!".

RMI and applets

Java applets can also use RMI. Here is the code for an applet client that calls our simple string setter and getter remote object:

```
// ExampleClientApplet.java
//
// Client for applet version of client calling Example remote object
//
import java.applet.Applet;
import java.awt.Graphics;
import java.rmi.*;

public class ExampleClientApplet extends Applet {
  public void paint(Graphics g) {
    try {
      //
      // Find remote "Example" object in RMI registry
      Example example = (Example) Naming.lookup( "Example" );
      //
      // Set string and then get it again to display
      example.setString( "Applet Success!" );
      g.drawString( example.getString(), 50, 25);

    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Here is the corresponding HTML for the applet:

```
<HTML>
<HEAD>
<TITLE> Example Client Program </TITLE>
</HEAD>
<BODY>
<APPLET CODE="ExampleClientApplet.class" WIDTH=150 HEIGHT=25>
</APPLET>
</BODY>
</HTML>
```

Running an RMI applet

To run the applet client, we start the RMI registry and the server program as usual, but instead of running `ExampleClient.java`, we point a Web browser at the ExampleClient.html file.

You should see the phrase "Applet Success!".

Now that we've looked at RMI in single-machine conditions, let's see what it's like to have to split the function between two machines.

Deploying RMI on two machines

So far, we have looked at how RMI can be used to run a program in one process, or JVM, and have it communicate with another process in which both processes are on the same machine. When we split the function between two machines, we have to consider where to deploy the various pieces of code.

If we want to run our example on two machines, here is the breakout of code:

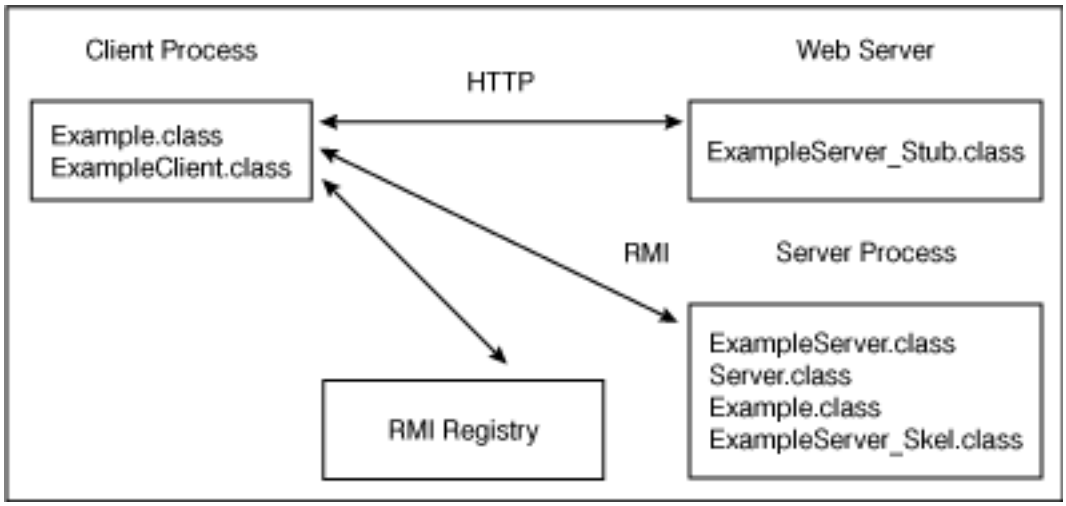
- **Client:** Example.class, ExampleClient.class, ExampleServer_Stub.class
- **Server:** ExampleServer.class, Server.class, Example.class, ExampleServer_Skel.class

As a variation, the ExampleServer_Stub.class file can reside on a Web server to be downloaded on demand if the `-Djava.rmi.server.codebase` option is used on the command line that starts the client. For example:

```
java -Djava.rmi.server.codebase=http://www.mywebserver.com/ ExampleClient
```

For the applet client, the Web server, in addition to ExampleServer_Stub.class, can contain ExampleClient.html, Example.class, and ExampleClientApplet.class. If the Web server is a different machine than the RMI server, then the applets will have to be digitally signed because the Java security model only allows communication with a single machine by default. More information on the Java security model and code signing can be found in Resources on page 18 .

The following diagram details a typical RMI application deployment:



Section 4. CORBA concepts and examples

CORBA overview

As previously mentioned, CORBA defines several services, one of which performs a function similar to RMI, but it works with a variety of different programming languages and not just with the Java platform.

CORBA was defined by a consortium called the Object Management Group (OMG), which consists of about 800 member companies.

In general, CORBA defines two things:

- The entity that allows communication between two processes, called an ORB (Object Request Broker)
- A protocol that the ORB uses for communication between processes, called IIOP (Internet Interoperability Protocol)

The Java platform contains a CORBA ORB. CORBA uses the same stub/skeleton technique as RMI, but unlike RMI, CORBA generates the stubs and skeletons from a language-independent interface description called IDL (Interface Description Language) instead of the source code language.

IDL specifies method names, as well as calling and return parameters, in a language-neutral manner.

Java IDL and RMI-IIOP overview

Java IDL and RMI-IIOP provide two ways to use Java client code with a CORBA server object.

With Java IDL, the development steps begin by defining the remote interface in IDL, compiling the IDL to the target-server language, writing the remote-server object using that target language, and deploying the pieces.

Because there is no RMI registry in the CORBA environment, the CORBA Naming Service (COSNaming) provides the server object-location lookup function.

With RMI-IIOP, the stub and skeletons are generated right from the Java object definition. Instead of the proprietary protocol used in RMI to communicate between two processes, RMI-IIOP uses the CORBA IIOP protocol, so it can call objects in non-Java languages. The development steps begin with the Java implementation class instead of the IDL. This class is compiled with `rmic` using the `-iiop -d` flags, and a server class is written that uses the COSNaming, accessed through the Java Naming and Directory Interface (JNDI).

The next few panels demonstrate the Java IDL technique so you can get a more general picture of how CORBA works in other languages. You can find more details on the RMI-IIOP technique in Resources on page 18 .

Writing the IDL

Let's look at how to write our example application using CORBA. We first define the IDL for the remote object:

```
interface ExampleCORBA {
  attribute string s;
};
```

The attribute `string` actually triggers an implicit getter and setter method implementation generation.

We can now compile the IDL using the `idlj` command. The `-fall` switch generates both the client and server code (all).

```
D:\RMICORBA> idlj -fall ExampleCORBA.idl
```

Defining the remote-object IDL and compiling the IDL generates the following files:

- `_ExampleCORBAStub.java` is the stub code.
- `ExampleCORBA.java` is the remote interface.
- `ExampleCORBAHolder.java` is support code.
- `ExampleCORBAHelper.java` is support code.
- `ExampleCORBAOperations.java` is support code.
- `ExampleCORBAPOA.java` is the skeleton code and the superclass we extend from when implementing the server code.

Writing the remote object

Next, we implement the server class, called (by convention) `Servant`, and extend from the base class generated by the IDL compilation:

```
// ExampleCORBAServant.java
//
// The remote implementation for ExampleCORBAPOA
//
import org.omg.CORBA.*;

public class ExampleCORBAServant extends ExampleCORBAPOA {
  private String stringState;
  private ORB orb;

  public void setORB( ORB o ) {
    orb = o;
  }

  public String s() {
    return stringState;
  }
```

```
  public void s( String s ) {
    stringState = s;
  }

}
```

The CORBA server code

Next, we implement the server code that takes care of the remote object instantiation and name server registration:

```
// CORBAServer.java
//
// Server to make a new "Example" remote object and put a reference to
// it in the name service
//
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

public class CORBAServer {

  public static void main ( String[] args ) {
    try {
      //
      // Initialize the ORB
      ORB orb = ORB.init( args, null );
      POA rootpoa =
        POAHelper.narrow( orb.resolve_initial_references("RootPOA"));
      rootpoa.the_POAManager().activate();
      //
      // Create a new remote Example object, set the ORB in it
      // enter it into the name server
      ExampleCORBAServant e = new ExampleCORBAServant();
      e.setORB(orb);
      //
      // Make a reference to the object
      org.omg.CORBA.Object ref = rootpoa.servant_to_reference(e);
      ExampleCORBA eRef = ExampleCORBAHelper.narrow(ref);
      //
      // Find the name service
      org.omg.CORBA.Object oRef =
        orb.resolve_initial_references("NameService");
      NamingContextExt ncRef = NamingContextExtHelper.narrow( oRef );
      //
      // Enter the reference into the name service
      String name = "Example";
      NameComponent path[] = ncRef.to_name( name );
      ncRef.rebind(path, eRef);
      //
      // Wait and listen for requests
      orb.run();

    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Java CORBA client program

Finally, we create a client program that makes remote method calls:

```
// ExampleCORBAClient.java
//
// CORBA client for calling remote "Example" object
//
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class ExampleCORBAClient {

  public static void main ( String[] args ) {
    try {
      //
      // Initialize the ORB
      ORB orb = ORB.init( args, null );
      //
      // Find the nameserver and downcast it from
      // Object to NamingContext
      org.omg.CORBA.Object o =
        orb.resolve_initial_references( "NameService" );
      NamingContext ns = NamingContextHelper.narrow(o);
      //
      // Find the "Example" alias in the root path and downcast it to
      // ExampleCORBA
      NameComponent nc = new NameComponent( "Example", "");
      NameComponent path[] = {nc};
      o = ns.resolve( path );
      ExampleCORBA example = ExampleCORBAHelper.narrow( o );
      //
      // Call the Example remote object to set the string and read and
      // print it
      example.s( "CORBA Success!" );
      System.out.println( example.s() );

    } catch (Exception e) { e.printStackTrace(); }
  }
}
```

Now we're ready to run the CORBA application.

Running a CORBA application

To execute the code example from the previous panel, we have to start up the RMI registry, run the server program once, and then we can run the client:

```
D:\RMICORBA> start tnameserv
D:\RMICORBA> start java CORBAServer
D:\RMICORBA> java ExampleCORBAClient -ORBInitialHost localhost
```

You should see the phrase "CORBA Success!".

Section 5. Wrapup and resources

RMI or CORBA, which do you choose?

You have now seen how both RMI and some of the Java CORBA support are used to perform the same function of having a client object call methods in a server object. So which do you choose, RMI or CORBA?

Here is a general rule of thumb. If you know you can use the Java platform on each end of the distributed environment, choose RMI for its simpler programming and deployment model. If you need to have a Java client object talk to a non-Java-based server object, then consider CORBA.

If you need to use CORBA, you should first consider RMI-IIOP because it works off the native Java interface and does not require IDL. If the rest of your CORBA environment uses IDL, then you can leverage this with Java IDL.

But if you go the CORBA route, be prepared to deal with the increased complexity of the API and the more complex development and deployment requirements. There may be additional cost involved as well; while everything you need for RMI comes with the JSDK, some of the CORBA pieces may require a license fee.

Summary

This tutorial serves as an introduction to distributed objects and how they can implement distributed computing models. The Java platform has a rich set of easy-to-use distributed object function in RMI, and interoperability with CORBA via two approaches. We demonstrated how a simple example can be implemented in RMI, first with a Java application and then an applet. Then, we showed how the same example would work with Java IDL in a CORBA environment.

By this point, you should find yourself well poised for continuing exploration of the RMI and CORBA distributed object APIs to develop sophisticated distributed object applications. To further your learning, you should continue to play with examples. A good next step is to try to distribute the client and server on different machines.

Resources

Downloads

- Download the complete *source code and classes* used in this tutorial.
- The *Java 2 platform, Standard Edition* is available from Sun Microsystems.

Articles, tutorials, and other online resources

- Damian Hagge's "*RMI-IIOP in the enterprise: An introduction to running RMI over IIOP* " (*developerWorks*, March 2002) offers a brief introduction to RMI-IIOP, then shows you how

to build and run a simple, Java technology-based RMI-IIOP client/server application.

- The "*RMI-IIOP Programmer's Guide* " will teach you how to write Java RMI programs that can access remote objects by using the IIOP. There is also an *RMI-IIOP frequently asked questions* to facilitate your understanding.

- Dave Bartlett's two-part article, "IDL-to-Java mapping" *Part 1* (*developerWorks*, October 2000) and "*Part 2* (*developerWorks*, November 2000), describes how discrete component interface definitions translate to Java elements.

- The *developerWorks* library offers a concise roundup of *alternative client/server implementation techniques* and their definitions (June 2001).

- The *Java RMI* page contains links to additional tutorials and specifications.

- For additional information on Java CORBA capabilities, see the *Java IDL* page.

- For more information on using Java with the IIOP protocol, see the *RMI-IIOP* page.

- The *Object Management Group* site is home to a wealth of information on this tutorial's CORBA topic, primarily focused in the *CORBA section* of this site.

- For more on the Java security model and Java digital code signing, see Brad Rubin's tutorial, "*Java security, Part 1: Crypto basics*" (*developerWorks*, July 2002)

- For additional information on Java Serialization, see the excerpted *Chapter 10* from *Java RMI* by William Grosso, an O'Reilly publication.

Books

- For an in-depth discussion of Java RMI, with a bit of CORBA coverage as well, see *Java RMI* by William Grosso; O'Reilly, 2002.

- Another real-world reference for RMI is *Java.rmi: The Remote Method Invocation Guide* by Esmond Pitt and Kathleen McNiff; Addison-Wesley, 2001.

- Arranged to be both a tutorial and a general reference, a good CORBA guide is *Advanced CORBA Programming with C++* by Michi Henning and Steve Vinoski; Addison-Wesley, 1999.

- A strong, specific reference for Java and CORBA is *Java Programming with CORBA: Advanced Techniques for Building Distributed Applications* 3rd Ed., by Gerald Brose, Andreas Vogel, and Keith Duddy; John Wiley and Sons, 2001.

Feedback

Please let us know whether this tutorial was helpful to you and how we could make it better. We'd also like to hear about other tutorial topics you'd like to see covered. Thanks!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The open source Toot-O-Matic tool is an XSLT stylesheet and several XSLT extension functions that convert an XML file into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML. (It also saves our production team a great deal of time and effort.)

You can get the source code for the Toot-O-Matic at www6.software.ibm.com/dl/devworks/dw-tootomatic-p. The tutorial Building tutorials with the Toot-O-Matic demonstrates how to use the Toot-O-Matic to create your own tutorials. developerWorks also hosts a forum devoted to the Toot-O-Matic; it's available at www-105.ibm.com/developerworks/xml_df.nsf/AllViewTemplate?OpenForm&RestrictToCategory=11. We'd love to know what you think about the tool.