Using CORBA to Create Client/Server Applications

by Marshall Brain & Christopher McGee

Comment on this article

If you are looking for a way to connect a client to a server in an object oriented way, you have two predominant choices: DCOM and CORBA. DCOM is the Microsoft solution and is extremely useful when you are writing code for the Windows desktop. DCOM is covered extensively in a series of 3 DevJournal articles (Understanding DCOM). In this article we will explore the CORBA solution to the distributed application problem.

[Download the source code for this article]

CORBA Overview

In a normal object oriented program written in C++ or Java, the program contains all of the classes that the application needs. The compiler compiles and links these objects. When you instantiate objects with the **new** statement, they are created in the application's memory space and methods execute as part of a single process.

CORBA is a technology that allows a client application to call objects that reside on a server. The server may be running on the same machine or on a machine 3,000 miles away. At its most basic level, CORBA is extremely simple - instead of instantiating an object in your process's memory space as you do in a normal program, you instantiate the object on a server machine somewhere on the network. The calls to the object's methods and the parameters passed to the methods are then packaged in the form of a network packet and sent to the server. The function actually runs on the server, and any results are sent back to the caller through the network.

The advantage of this approach is that the server can act as a high-powered, central shared resource. The server can have databases or other local services that the server-based CORBA objects can access. The disadvantage, of course, is speed. The number of function calls you can make per second is severely limited by the speed of the network. That holds true for any technology that uses a network connection for data transport, so CORBA is not unique. DCOM and even sockets suffer from the same delays.

One thing unique to the CORBA approach is a facility called the ORB (Object Request Broker). The ORB runs as part of the client and server processes and handles the network connection between the two. In this example we will be using the Inprise VisiBroker ORB. With VisiBroker, the client and server machines also use a service called the OSAGENT. The OSAGENT is contacted by a client ORB at a standard port number (generally 14000). The OSAGENT acts as a directory that helps a client ORB to find an object that it is looking for.

Setting up a simple CORBA client and server in Java is amazingly easy. In the next section we will look at several pieces of example code to show you the process. Then in the following section we'll walk through an explanation of how the code works. A brief word about setup before we get started.

Setup

To use the example code in this article, you will need Sun's 1.1.5 JDK as well as the VisiBroker ORB 3.3. You can get the JDK from http://java.sun.com/products/jdk/1.1/index.html. You can get the VisiBroker ORB from http://www.inprise.com/visibroker/. Install the JDK first and then the ORB.

Example Code

To create a CORBA server and a client to access it, you start by deciding what the server class will look like and do. For this example we will use an extremely simple server class. This class will own one private integer variable. One method on the class will increment the integer, and the other method will allow the client to access the current value of the variable. Here's the interface definition for the class:

```
// Counter.idl

module Counter {

  interface Count
  {
    void increment();
    long getCounter();
  };

};
```

The class is named **Count**. Its two methods are **increment** and **getCounter**. The class is held inside a module named **Counter**, which can contain several different classes if you like. Place this IDL (Interface Definition Language) code in a file name Counter.idl.

To use this IDL file, you need to run it through the tool called IDL2JAVA. This tool will create classes for the client and the server to use. The client-side class that IDL2JAVA generates is called a Helper and will be named **CountHelper.java**. Your client uses the helper class to call the functions on the server. The server-side class that IDL2JAVA generates is called an ImplBase (also known as a skeleton) and will be called **_CountImpleBase.java**. You will implement the methods that the server uses by extending the ImplBase.

> **Note**: While the class that you use directly on the client side is CounterHelper.java, the code that executes on the client side when you call a method on the Counter interface is _st_Counter.java. This class is called the stub. It is responsible for packaging up the data to send to the server and unpackaging the return value. It talks to the skeleton on the server side - _sk_Counter.java - which is what _CounterImplBase.java is derived from.

The client code is extremely simple. Here is about the smallest client application you can create:

```
// Client.java

public class Client
{
   public static void main(String[] args)
   {
      Counter.Count count = null;
      org.omg.CORBA.ORB orb = null;

      // Initialize the ORB
      orb = org.omg.CORBA.ORB.init();

      // Bind to the object on the server
      count = Counter.CountHelper.bind(
          orb, "test");

      // Call the server functions
      count.increment();
      System.out.println(
          "Current Count = " + count.getCounter());

      // clean up
      count = null;
      orb.shutdown();
   }
}
```

In this code you can see that the client initializes the ORB, binds to the helper object in order to get a connection to the server, and then starts calling functions. That really is all there is to it. To compile this code, save it to a file named Client.java and type:

> vbjc Client.java

Alternatively you can add the three jar files for the ORB to your class path and use javac to compile the client (the three jar files are: vbjorb.jar, vbjapp.jar and vbjtools.jar).

If you want to instrument the client code so it detects problems and lets you know about them, this is how you would do it:

```
// Client.java

public class Client
{
   public static void main(String[] args)
   {
      Counter.Count count = null;
      org.omg.CORBA.ORB orb = null;

      // Initialize the ORB
      try
      {
          orb = org.omg.CORBA.ORB.init();
      }
      catch (org.omg.CORBA.SystemException se)
      {
         System.err.println(
             "initializtion problem in the ORB "
            + se);
         System.exit(1);
      }

      // Bind to the object on the server
      try
      {
         count = Counter.CountHelper.bind(orb,
             "test");
      }
      catch (org.omg.CORBA.SystemException se)
      {
         System.err.println(
             "Binding problem in the ORB " + se);
         System.exit(1);
      }

      try
      {
        count.increment();
        System.out.println(
            "Current Count = " + count.getCounter());
      }
      catch (org.omg.CORBA.SystemException se)
      {
        System.err.println(
            "Increment failure " + se);
        System.exit(1);
      }

      // clean up
      try
      {
        count = null;
        orb.shutdown();
      }
      catch (org.omg.CORBA.SystemException se)
      {
        System.err.println(
            "Problem with cleanup " + se);
        System.exit(1);
      }
   }
}
```

You have several options when you initialize the ORB and bind to the server:

- The ORB can accept and parse options from the command line
- You can set up binding options
- You can directly connect to a specific machine

The following code fragment shows you these three options in action:

```
orb = org.omg.CORBA.ORB.init(args, null);

org.omg.CORBA.BindOptions bindOptions = new
   org.omg.CORBA.BindOptions();
bindOptions.defer_bind = false;
bindOptions.enable_rebind = true;

count = Counter.CountHelper.bind(
    orb, "test",
    "marshall.iticentral.com",
    bindOptions);
```

Here the initialization function accepts arguments from the command line. The bind function accepts a specific machine name and binding options.

On the server side you need to create 2 pieces: a piece of server code for the client to bind to and an implementation for the Count class. The implementation for the Count class inherits from ImplBase and looks like this:

```
// CountImpl.java

public class CountImpl extends Counter._CountImplBase
{
   private int c = 0;

   public CountImpl(String name) {
      super(name);
   }

   public CountImpl() {
      super();
   }

   public void increment() {
      c = c + 1;

      System.out.println(c);
   }

   public int getCounter()
   {
      return c;
   }
}
```

It's hard for a class to get any simpler than that - all it does is implement the methods of the class, and there is absolutely nothing special about it. The server code to handle this class looks like this:

```
// Server.java

public class Server
{
   public static void main(String[] args)
   {
      org.omg.CORBA.ORB orb = null;
      org.omg.CORBA.BOA boa = null;
      Counter.Count count = null;

      try
      {
         orb = org.omg.CORBA.ORB.init(args, null);
      }
      catch (org.omg.CORBA.SystemException se)
      {
         System.err.println(
             "Initialization problem in the ORB "
            + se);
         System.exit(1);
      }
      try
      {
         boa = orb.BOA_init();
      }
      catch (org.omg.CORBA.SystemException se)
      {
         System.err.println(
             "Initialization problem in the BOA "
            + se);
         System.exit(1);
      }

      try
      {
         count = new CountImpl("test");
         boa.obj_is_ready(count);
         System.out.println(count + " ready.");
         boa.impl_is_ready();
      }
      catch (org.omg.CORBA.SystemException se)
      {
         System.err.println("Ready problem " + se);
         System.exit(1);
      }
   }
}
```

Save the class implementation in a file named CountImpl.java and the server code in Server.java. Compile the server with:

```
> vbjc Server.java
```

Now that everything is compiled you can try it out. In one command shell type:

```
> osagent
> vbj Server
```

This will start the server. In a second command shell type:

```
> osfind
> vbj Client
```

The OSFIND command will confirm that the server is running - OSFIND lists the running servers it finds and the named objects they are capable of serving. The second command executes the client. You should see both the client and server windows print the current value of the counter. You can run the client multiple times to increment the counter.

The server code starts by initializing the server's ORB. It then initializes the server's BOA (Basic Object Adapter). The BOA's job is the tell the server's ORB when the server class is ready. The server then instantiates a single instance of the count object, giving it the name "test" and then tells the ORB that both the count object and then the server is ready. The imple_is_ready method allows the server to instantiate multiple objects and perform other initializations before allowing the server to be accessed from the outside.

You should also be able to run the server on another machine and it should work in one of two ways. If the client and server machines are on the same network segment, then the OSAGENT will find the server by broadcasting a request on that segment. If they are not on the same segment, you can modify the client code to connect directly to the server machine.

It is also possible to use CORBA from JAVA applications. The following code shows a very simple applet that calls the example server shown above:

```
// ButtonTest.java

import java.awt.*;
import java.applet.*;

public class ButtonTest extends Applet
{
   Button b;
   Counter.Count count = null;
   org.omg.CORBA.ORB orb = null;
   public ButtonTest()
   {
      setLayout(new BorderLayout());

      b = new Button("Apply");
      add("North", b);

      // ---------
      // Bind to the counter

      String args[] = new String[10];
      args[0] = "test";

      try {
          orb = org.omg.CORBA.ORB.init();
      }
      catch (org.omg.CORBA.SystemException se) {
         System.err.println(
             "ORB init failure " + se);
         System.exit(1);
      }

      try {
         count = Counter.CountHelper.bind(
             orb, "test");
      }
      catch (org.omg.CORBA.SystemException se) {
         System.err.println(
             "ORB bind failure " + se);
         System.exit(1);
      }

   }

   public void paint(Graphics g)
   {
            long i = count.getCounter();
            g.drawString(" " + i, 50, 50);
   }

   public boolean action(Event ev, Object arg)
   {
      if (ev.target instanceof Button)
      {

        try {
          count.increment();
        }
        catch (org.omg.CORBA.SystemException se) {
          System.err.println(
              "Increment failure " + se);
          System.exit(1);
        }

        repaint();
        return true;
      }
      return false;
   }
}
```

Save this code to ButtonTest.java and compile this code with the following command:

> vbjc ButtonTest.java

Now create a simple web page to load the applet, like this:

```
<html>
<body>
<applet
    code=ButtonTest.class
    width="200"
    height="200">
```

```
</applet>
</body>
</html>
```

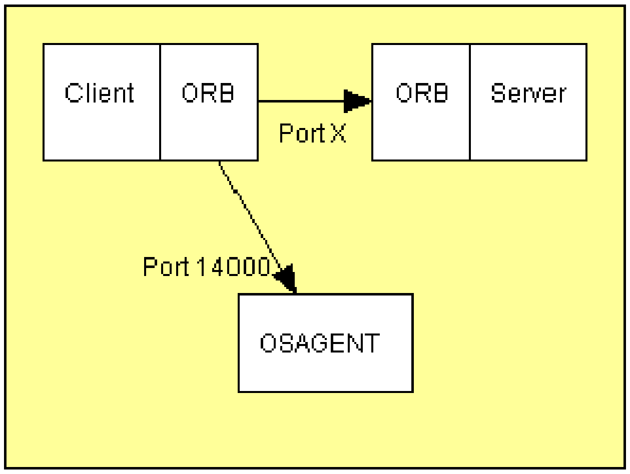
Save this HTML code to a file named first.htm. Run the applet in the applet viewer by saying:

```
> appletviewer -J-DORBagentPort=14919 first.htm
```

Any Java application can use this approach to access CORBA.

Behind the Scenes

Both the client and the server run an ORB in their process spaces. The ORB helps the client and server bind to one another and then facilitates parameter transport and function activation between the client and server through the network.



In CORBA, the server must be running before the client can connect. When the client tries to bind to the server, its ORB starts by talking to the OSAGENT. The OSAGENT acts as a directory of named server objects, and the default port number for communicating with it is 14000. The OSAGENT knows that the server is running and is exposing certain objects. The ORB for the server has picked a random port when the server started running and the OSAGENT knows this port number. The OSAGENT therefore returns to the client the machine name and port number for the server's ORB. The client's ORB then makes a direct connection to that port on the server and uses it for all method calls.

In the case where the client and server are on different machines but share the same network segment, there must be one OSAGENT running somewhere on the segment. The OSAGENT, client and server use broadcast packets to talk to one another.

In the case where the server is on a different segment than the client, the client will directly connect to the server machine. The server must run an OSAGENT. When the client connects to the server machine, it first talks to the OSAGENT to learn the port number for the server's ORB, and then directly connects to the server.

Conclusion

From this simple example, you can see that the basics of CORBA are extremely simple. Only two or three lines of code must be added to the client and server to activate CORBA. Once connected, the client and server talk using what looks like standard code. This simplicity makes it extremely easy to create distributed client/server applications with CORBA.

For more information about CORBA, and for links to Web sites that provide CORBA resources, you might begin with this:

CORBA FAQ – http://www.omg.org/gettingstarted/corbafaq.htm

Developed Under: JDK 1.1.5 and Visigenic 3.3