Java and .Net interoperability using CORBA

A recently released product takes the pain out of implementing object-level calls between Java and .Net

By Christian Donner

While the need for interoperability between Java and .Net has become a common problem in larger organizations, CORBA is often not the first choice of techniques for building a bridge between the two worlds. The lack of a commercially available CORBA implementation for .Net required a high up-front investment into this technology, resulting in significant increases in cost and time to market. Since the release of MiddTec's MiddCor earlier this year, this barrier has vanished. In this article, Christian Donner explores the fundamental concepts of using CORBA in a heterogeneous environment consisting of Java and .Net. The article features a simple application that implements object-level calls from Java to .Net in less than 50 lines of source code. (*1,600 words;* **May 17, 2004**)

For many corporations, today's heterogeneous software environments require interoperability between platforms that extends beyond the loose coupling provided by Web services. As an alternative, CORBA is readily available on the Java side, but until recently, writing sa custom object broker for .Net was necessary for C# or Visual Basic .Net (VB.Net) objects to act as a CORBA server or client. (Unfamiliar with CORBA? Read the sidebar "What Is Corba?") With the release of MiddCor, a commercial CORBA implementation for .Net, a complete CORBA infrastructure for Java and .Net interoperability is now readily available, and the lead-time for such integration has virtually disappeared.

The Hello CORBA application

In this article, you learn how to write a simple CORBA application of the "Hello World" type. The client calls the server and passes a parameter named `sendMsg`, containing the value `Spring`. The server writes the string `"Hello " + sendMsg` to the console and returns the value "`Good Bye Winter`" to the client. The client writes the returned string to the console as well.

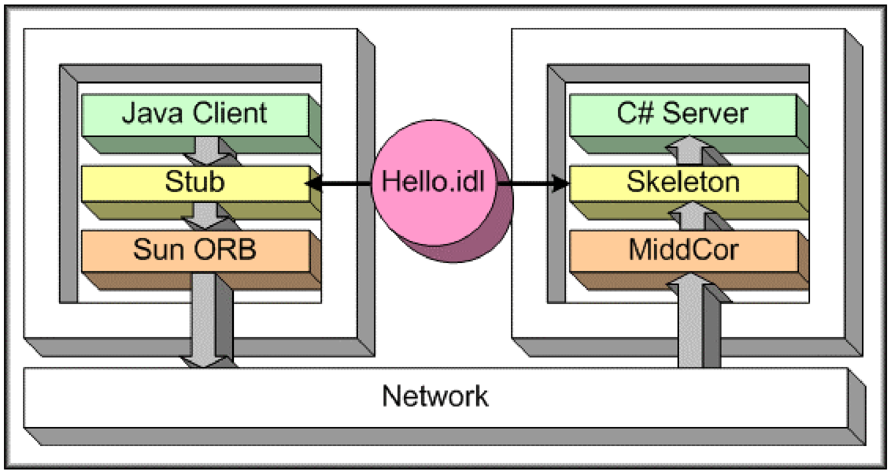
The server application is based on the `hello sample` that ships with the MiddCor trial version. You can quickly get the server up and running by modifying the sample source files.

You need three software products to build this application:

- Microsoft's *.Net Framework SDK*, which can be downloaded for free but does not provide a GUI, or *Visual Studio .Net 2003.*
- Either the J2EE or the J2SE edition of Sun's *Java 1.4 SDK*. Both can be downloaded from Sun.
- CORBA for .Net. The only commercially available product is currently *MiddCor* from MiddTec. A 30-day evaluation copy can be downloaded from the manufacturer's Website.

Write the IDL

The IDL (interface definition language) defines the namespace, class names, and method signatures for the exposed calling interfaces.



CORBA and the IDL in a heterogeneous environment.

Its constructs resemble Java interfaces. The sample application's IDL is rather simple:

```
module HelloCorba
{
    interface Greetings
    {
        string hello( in string sendMsg);
    };
};
```

The `module` section defines the namespace. We will publish the namespace in the IOR file (interoperable object reference), so that the client can use it to find the referenced service. In the C# server code, you embed your class implementation in a *namespace* section.

`interface` is the name of the servant (i.e., server) object. We will see that for the actual implementation, the suffix `Impl` is appended, a widely used convention.

The class `Greetings` has one exposed method, `hello()`, which expects a string parameter and returns also a string. Note that the example uses only the IDL data type `string`. More information about the IDL can be found in Resources.

Save the file in a directory of your choice and call it `hello.idl`.

Write the server

Start by running the MiddCor IDL compiler, *MiddCorIdl.exe*:

```
> middcoridl hello.idl
```

The IDL compiler creates one C# output file called `hello.cs`. This file contains the client stub as well as the server skeleton code for the `Greetings` class introduced in the IDL definition above. The actual implementation class `GreetingsImpl` is derived from the skeleton class `GreetingsPOA` (derived from the CORBA servant class). The client stub code is not used since our client will be a Java program.

The only steps left are to code the implementation of the `GreetingsImpl` class and to code a `Server` class that provides a runtime process for the ORB and the server object.

First, we make available the classes in the `System` and the `MiddTec` namespaces. `System` includes, for example, classes for all the data types and is required for almost every .Net application:

```
using System;
using MiddTec;
```

In the `HelloCorba` namespace, we implement the two classes described above. `GreetingsImpl` is the class exposed through CORBA. It is derived from `GreetingsPOA` and contains the sample application's simple business logic (Note: The `a_` prefix is a convention used by MiddTec to make code more readable. The `a_` prefix is used for argument parameters. You are free to name parameters as you like.):

```
namespace HelloCorba
{
  public class GreetingsImpl: GreetingsPOA
  {
    public override string hello( string a_sendMsg )
    {
      System.Console.WriteLine("Client says: Hello {0}\n",
          a_sendMsg);
      return "Good Bye Winter";
    }
  }
```

The `Server` class's `main()` method initializes the ORB, finds the root POA (Portable Object Adapter), activates its POA manager, and creates the IOR file for the client's use. It then runs the ORB and cleans up after an exception:

```
  class Server
  {
    static void Main(string[] args)
    {
      MiddTec.CORBA.ORB oOrb = MiddTec.CORBA._ORB.init( args, null);
```

Our newly initialized ORB has a root POA process in a holding state. We must get a reference to it to activate it. The return of `resolve_initial_references` is typed as `CORBA.Object` and must be "narrowed" to get the desired type, `PortableServer.POA`:

```
      MiddTec.PortableServer.POA oRootPOA =
        MiddTec.PortableServer.POAHelper.narrow(
            oOrb.resolve_initial_references( "RootPOA" ));
      oRootPOA.the_POAManager.activate();
```

The next step is to create an instance of `GreetingsImpl` that handles client requests:

```
      GreetingsImpl oGreetings = new GreetingsImpl();
```

Then we must connect the servant implementation with the POA. The result is a CORBA object reference. We use this reference to write the IOR file:

```
      MiddTec.CORBA.Object obj =
        oRootPOA.servant_to_reference( oGreetings);

      MiddTec.CORBA._ORB.wrIORtoFile( "c:\\hello.ior", obj );
```

The rest of the work on the server side is trivial—run the ORB, inform the user that it is running, and destroy it in case of an exception:

```
      System.Console.WriteLine("Server is running ...\n" );

      try
      {
        oOrb.run();
      }
      catch( System.Exception )
      {
        oOrb.destroy();
      }
    }
  }
}
```

Save this code to a file named `helloSrv.cs`, compile it, and run the resulting executable (don't forget to include `hello.cs` in your Visual Studio project). A DOS window should appear and display the message "Server is running ...". Our server now waits for requests.

Write the client

The Java client code is even simpler than what you already accomplished. For the sake of clarity, the example does not include any code for exception handling.

To start, create a copy of the file `hello.idl` from above in a directory of your choice and run the Java IDL compiler *idlj*. The switch `-f client` will omit the generation of the server skeleton code, which we don't need:

```
idlj -f client hello.idl
```

The IDL compiler creates a subdirectory called `HelloCorba`, which is the name of the module or namespace. You will find five source files in this directory, which I will not further explain. Feel free to look at the generated source code. Compile the generated code and create a module jar file:

```
javac HelloCorba\*.java

jar -cvf HelloCorba.jar HelloCorba\*.class
```

Now you are ready to code the client application. Create a Java source file in the directory where your `HelloCorba.jar` file resides. First, import the required Java namespaces:

```
import HelloCorba.*;
import org.omg.CORBA.ORB;
import java.io.*;
```

Next, write the `GreetingsClient` class implementation. It contains only a `main()` method:

```
public class GreetingsClient
{
  public static void main(String args[]) throws IOException
  {
```

The first line initializes the ORB, analogous to the C# `Server` class implementation above:

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

The client then reads the IOR file created by the server. If the client runs on a different host, the file has to be deployed prior to its invocation:

```
    FileReader fr = new
    FileReader("c:\\hello.ior");
    BufferedReader br = new
    BufferedReader(fr);
    String ior = br.readLine();
```

The IOR can then be converted to an object reference, using the ORB's `string_to_object` method:

```
    org.omg.CORBA.Object obj = orb.string_to_object(ior);
```

The reference must be narrowed to the desired type `Greetings`:

```
    Greetings proxy = GreetingsHelper.narrow(obj);
```

Now you can safely call the remote object and write the returned message to the console:

```
    String msg = proxy.hello("Spring");
    System.out.println("Server says: " + msg);
  }
}
```

Compile and run the client application:

```
javac GreetingsClient.java

java GreetingsClient
```

The line "Server says: Good Bye Winter" should appear in the DOS window. At the same time, the line "Client says: Hello Spring" should appear in the server window that should still be open.

Instead of exchanging IORs, you can use a naming service like *orbd* or *tnameserv* that ships with the Java SDK.

Conclusion

This project demonstrates that CORBA is a powerful solution for .Net and Java interoperability. CORBA is complex and cannot be mastered in one afternoon. It is now possible, however, to develop a working CORBA application for .Net with little prior knowledge, because most of the complexity is buried within the implementation of a commercially available object request broker.

About the author

Christian Donner is a senior consultant and software architect at Molecular. His interests include enterprise application and data integration, and strategies for introducing new technologies in large corporations. Christian is a Sun Certified Enterprise Architect for the Java 2 Platform. He currently lives in Boston.

Resources

- Download the source code that accompanies this article: http://www.javaworld.com/javaworld/jw-05-2004/java.net/jw-0517-java.net.zip
- Microsoft Visual Studio .Net 2003: http://msdn.microsoft.com/vstudio/
- You can also use the free .Net Framework SDK 1.1: http://www.microsoft.com/downloads/details.aspx?FamilyID=9b3a2ca6-3647-4070-9f41-a333c6b9181d&displaylang=en
- J2EE 1.4: http://java.sun.com/j2ee/1.4/download.html
- MiddTec MiddCor Trial Version: http://www.middtec.com
- A comprehensive overview of the J2SE 1.4 CORBA implementation can be found in the series of articles "J2SE 1.4 Breathes New Life into the CORBA Community" by Tarak Modi (*JavaWorld*):
    - Part 1: Get started developing enterprise CORBA applications with the latest version of J2SE (August 2002)
    - Part 2: Gain code portability with the Portable Object Adapter (September 2002)
    - Part 3: Create enterprise-level apps with the POA (October 2002)
    - Part 4: Portable interceptors and the Interoperable Naming Service (November 2002)
- IDL data types and Java mapping: http://java.sun.com/j2se/1.3/docs/guide/idl/mapping/jidlMapping.html
- Browse the **CORBA** section of *JavaWorld*'s Topical Index: http://www.javaworld.com/channel_content/jw-corba-index.shtml

What is CORBA?

CORBA is the acronym for Common Object Request Broker Architecture, an open standard for the middleware that provides the mechanism for exposing an object's methods to remote callers (to act as a **server**) and for discovering such an exposed server object within the CORBA infrastructure (to invoke it as a **client**).

CORBA has been around for about 10 years. It replaced the older remote procedure call (RPC) standard, which proved unsuitable for the object-oriented programming paradigm. Both standards share the concept of a platform-independent interface definition language (IDL). The IDL serves as the common denominator. It is used for the definition of the calling interfaces and their signatures. An IDL compiler is a tool that a platform vendor must provide. It compiles the IDL file into platform-specific stub code and maps the parameter types to platform-specific types. An IDL compiler can generate both the client stubs and the server skeleton code. By the way, CORBA objects can act as servers and clients simultaneously.

For a server resource to be accessible, the client must have a reference to it. In essence, such a reference contains the host providing the service, the object name, the version, and the network protocol. The server can generate this information and convert it to a string. The string can then be stored in a file or in a database. It is called IOR for interoperable object reference. The client must access the IOR by reading the string from the filesystem, a Web server, or the database and convert the string back to an object reference. The article's example demonstrates how to do that.

A CORBA ORB transparently handles object location, object activation, parameter marshalling, fault recovery, and security. Note that the Portable Object Adapter (POA) concept was an add-on to the initial CORBA specification. Prior to POAs, server implementations were not "portable" because they contained code specific to a vendor's ORB. The code in this article is portable and should run with other ORBs than Middtec's MiddCor, once they become available.