



Getting started with CORBA

12.1	CORBA implementations	12.3	CORBA vs. RMI
12.2	Anatomy of a CORBA-based application	12.4	Summary

CORBA is merely a specification that does nothing if there is no software that implements it. There are a number of Java-based CORBA implementations; these are known as ORBs. Developing distributed applications in CORBA is similar to that of RMI in the sense that an interface must be defined first. However, unlike RMI, where interfaces are defined in Java, CORBA interfaces are defined in IDL.

This chapter shows you the anatomy of a CORBA-based application by walking you through the development and running of an example application. We'll also discuss the major differences between CORBA and RMI.

12.1 CORBA IMPLEMENTATIONS

The CORBA specification wouldn't be useful if there wasn't software to implement it. As you've already learned, the software that implements the CORBA specification is called an ORB. A number of ORBs exist today. Here are some of the most popular implementations that have Java support:

- VisiBroker from Inprise Corporation (<http://www.inprise.com>).

- OrbixWeb from IONA Technologies (<http://www.iona.ie>).
- JavaIDL from JavaSoft (<http://www.java.sun.com/products/jdk/idl>).

Other implementations of CORBA include PowerBroker from Expersoft (<http://www.expersoft.com>), DAIS from ICL (<http://www.icl.co.uk>), ObjectBroker from BEA Systems, (<http://www.beasys.com>) and ComponentBroker from IBM (<http://www.software.ibm.com/ad/cb>).

The CORBA programming examples in this book use the VisiBroker ORB for Java, version 3.1. To give you a brief history of the product, VisiBroker started as PostModern's Black Widow ORB. It was the first Java implementation of CORBA. In 1996, PostModern was acquired by Visigenic, a vendor for database middleware. Visigenic changed the name Black Widow to VisiBroker for Java. In 1997, Visigenic was acquired by Borland. In 1998, Borland changed its name to Inprise Corporation, but the ORB it acquired from Visigenic is still known as VisiBroker.

12.2 ANATOMY OF A CORBA-BASED APPLICATION

There are a number of steps for developing a CORBA-based application. Figure 12.1 illustrates the CORBA application development paradigm.

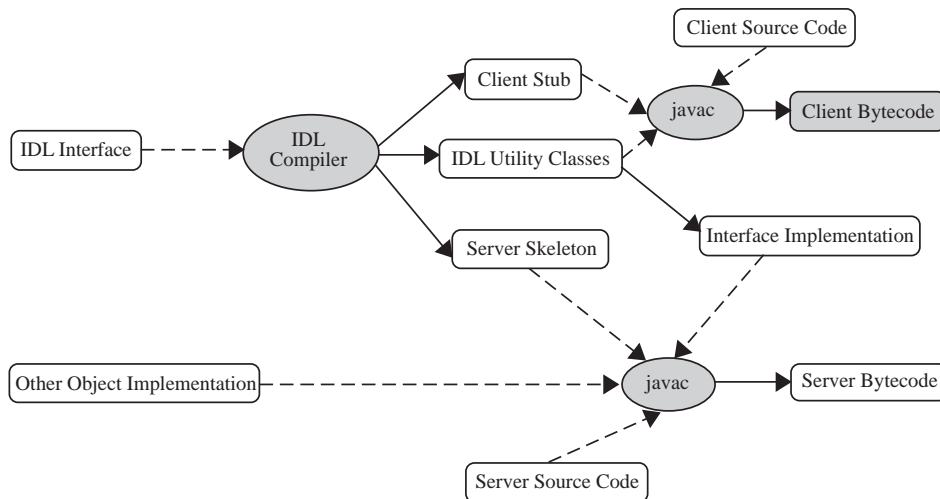


Figure 12.1 The CORBA application development paradigm

This section will outline the steps for developing a CORBA-based application using the VisiBroker ORB for Java. The following list is an overview of the steps.

- 1 Define interfaces using IDL.
- 2 Implement the CORBA classes.

- 3 Develop the server program.
- 4 Develop the client program(s).
- 5 Start the Smart Agent, the server, and the client(s).

I will explain each step separately by walking you through the development of the arithmetic server that we developed using sockets and RMI in chapters 3 and 8, respectively. The first step in developing the arithmetic server is to define an interface in IDL.

12.2.1 Defining an IDL interface

When defining an interface, keep in mind what types of operations the server will support. In the case of the arithmetic server, the operations that will be performed are of a mathematical nature—adding arrays, subtracting arrays, and so on. However, for the sake of simplicity, we will only consider the `sum_arrays` operation. As an exercise, you can modify the code and add more operations on your own.

Example 12.1 shows the IDL interface for the `Add` object. Note that the `sum_arrays` operation has three parameters. The first two are the input of the two arrays to be added, and the third parameter is the output holder (the sum of the two arrays). The first two parameters are declared `in`, and the third parameter is declared `out`. IDL defines three parameter-passing modes: `in`, `out`, and `inout`. As the names suggest, the `in` parameter is used for input, the `out` parameter is used for output, and the `inout` parameter is used for both input and output.

Example 12.1: `Arith.idl`

```
// Arithmetic IDL interface.
module Arith {
    interface Add {
        const unsigned short SIZE = 10;
        typedef long array[SIZE];
        void sum_arrays(in array a, in array b, out array c);
    };
};
```

Once we finish defining the IDL interface, we are ready to compile it. `VisiBroker` comes with an IDL compiler, `idl2java`, which is used to map IDL definitions into Java declarations and statements. We run the `idl2java` compiler from the command line and, as an argument, we feed it the `idl` module we want to compile, as follows:

```
% idl2java Arith.idl
Creating: Arith
Creating: Arith/AddPackage
Creating: Arith/AddPackage/SIZE.java
Creating: Arith/AddPackage/arrayHolder.java
Creating: Arith/AddPackage/arrayHelper.java
Creating: Arith/Add.java
Creating: Arith/AddHolder.java
Creating: Arith/AddHelper.java
Creating: Arith/_st_Add.java
Creating: Arith/_sk_Add.java
Creating: Arith/_AddImplBase.java
```

```
Creating: Arith/AddOperations.java
Creating: Arith/_tie_Add.java
Creating: Arith/_example_Add.java
%
```

As you can see, the `idl2java` compiler has generated a number of files; they are all stored in a subdirectory named `Arith`, which is the module name specified in the IDL file. As you will see in chapter 14, IDL modules are mapped to Java packages.

For this simple example, the generated files which are important to us are shown in table 12.1.

Table 12.1 Generated file list

Filename	Description
<code>Add.java</code>	The <code>Arith</code> interface declaration.
<code>AddOperations.java</code>	Declares the <code>sum_arrays</code> method.
<code>_st_Add.java</code>	Stub code for the <code>Arith</code> object on the client side.
<code>_sk_Add.java</code>	Stub code for the <code>Arith</code> object implementation on the server side.
<code>_example_Add.java</code>	Code you can fill in to implement the <code>Arith</code> object on the server side.

The `Add.java` file contains the Java code that was generated from the `Arith.idl` interface definition. The generated code in `Add.java` (without the comments) from the `Arith.idl` interface is shown in example 12.2. This segment of code will help implement the `sum_arrays` method.

Example 12.2: `Add.java`

```
/**
 * Generated by the idl2java compiler.
 */
public interface Add extends org.omg.CORBA.Object {
    final public static short SIZE = (short) 10;
    public void sum_arrays(
        int a[],
        int b[],
        Arith.AddPackage.arrayHolder c
    );
}
```

At this point, we can implement the `sum_arrays` method.

12.2.2 Implementing the CORBA classes

Implementing the `sum_arrays` method is easy. As shown in the previous section, the `idl2java` compiler has generated a file named `_example_Add.java`. This file actually contains

some constructors as well as the definition of the `sum_arrays` method. I copied the file into `AddImpl.java` and implemented the `sum_arrays` method as shown in example 12.3.

Example 12.3: `AddImpl.java`

```
/**
 * @(#)AddImpl.java
 */
public class AddImpl extends Arith._AddImplBase {
    /** Construct a persistently named object. */
    public AddImpl(java.lang.String name) {
        super(name);
    }
    /** Construct a transient object. */
    public AddImpl() {
        super();
    }
    public void sum_arrays(
        int[] a,
        int[] b,
        Arith.AddPackage.arrayHolder c
    ) {
        c.value = new int[10];
        for (int i = 0; i < Arith.AddPackage.SIZE.value; i++) {
            c.value[i] = a[i] + b[i];
        }
    }
}
```

Now we can compile it:

```
% javac AddImpl.java
```

We are now ready to develop our arithmetic server program.

12.2.3 Developing the server program

Example 12.4 shows an implementation of the `Server` class for the server side of our arithmetic server. The `Server` class does the following:

- Initializes the ORB.
- Initializes the BOA.
- Creates an `AddImpl` object.
- Activates the newly created object.
- Prints out a status message.
- Waits for incoming client requests.

As you can see from example 12.4, the `Server` class is small and fairly easy to follow and understand.

Example 12.4: `Server.java`

```
import org.omg.CORBA.*;
/**
 * @(#)Server.java
 */
public class Server {
    public static void main(String argv[]) {
        try {
            // Initialize the ORB.
            ORB orb = ORB.init();
            // Initialize the BOA.
            BOA boa = orb.BOA_init();
            // Create the AddImpl object.
            AddImpl arr = new AddImpl("Arithmetic Server");
            // Export the newly created object.
            boa.obj_is_ready(arr);
            System.out.println(arr + " is ready. ");
            // Wait for incoming requests.
            boa.impl_is_ready();
        } catch (SystemException se) {
            se.printStackTrace();
        }
    }
}
```

Once we finish implementing the server program, we can compile it:

```
% javac Server.java
```

The next step is to implement a client program that uses one or more of the services offered by the server. In our application, adding arrays is the only service the server program offers.

12.2.4 Developing the client program

Most of the files needed to implement the client program are contained in the `Arith` package generated by the `idl2java` compiler. The `Client` class shown in example 12.5 performs the following actions:

- Initializes the ORB.
- Binds to an `Add` object.
- Requests the `Add` to sum two arrays with the specified input values.

- Gets the sum of the two arrays using the object reference returned by the `sum_arrays` method.
- Prints out the sum of the two arrays.

Example 12.5: Client.java

```
import org.omg.CORBA.*;
/**
 * @(#)Client.java
 */
public class Client {
    public static void main(String argv[] ) {
        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int b[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        Arith.AddPackage.arrayHolder result = new Arith.AddPackage.array-
Holder();
        try {
            // Initialize the ORB.
            ORB orb = ORB.init();
            // Locate an Add object.
            Arith.Add add = Arith.AddHelper.bind(orb, "ArithmeticServer");
            add.sum_arrays(a, b, result);
            System.out.print("The sum is: ");
            for (int i = 0; i < Arith.AddPackage.SIZE.value; i++) {
                System.out.print(result.value[i]+"  ");
            }
            System.out.println();
        } catch (SystemException se) {
            se.printStackTrace();
        }
    }
}
```

We'll now move on to the final step in the development lifecycle of a CORBA-based application: starting the smart agent, the server, and the client.

12.2.5 Starting the smart agent, the server, and the client

The VisiBroker smart agent, `osagent`, provides a fault-tolerant object location service and runtime licensing of VisiBroker applications. We start the smart agent as a background process on a Unix-based system with this command:

```
% osagent &
```

If you are running Windows NT, you can start the smart agent as an NT service through the Services Control Panel, or you can use this command:

```
prompt> osagent -c (or osagent -C)
```

The `-c` or `-C` option allows the `osagent` to run in console mode.

Once the `osagent` is running, we can start the server program. We will do this using the `vbj` command, which invokes the JVM and offers other special services such as setting paths.

```
% vbj Server
```

```
AddImpl[Server,oid=PersistentID[repID=IDL:Arith/Add:1.0,objectName=Arithmetic Server]]is ready.
```

Finally, we can run the client program:

```
% vbj Client
```

```
The sum is: 2 4 6 8 10 12 14 16 18 20
```

In running the application in this chapter, we assumed that both the client and server are running on the same host. It is more common however, for the client and server to run on different hosts. One way to do this is by having an osagent run on each host. However, VisiBroker makes this unnecessary by providing the `OSAGENT_ADDR` property. To start a client or a server on a host that is not running the osagent, use the `OSAGENT_ADDR` property to specify the host that is running an osagent. For example, if there is an osagent running on the host “veda,” the following command can be used to start a client residing on a different host:

```
% java -DOSAGENT_ADDR=veda Client
```

12.3 CORBA vs. RMI

The arithmetic server developed in this chapter was also implemented using sockets (in chapter 3), and RMI (in chapter 8). In chapter 8, I briefly compared RMI with sockets. In this section, I will briefly compare CORBA with RMI.

Looking at the arithmetic server, the amount of code we had to write for each was almost the same. However, in CORBA’s case, the code is more complex and the programmer has to be familiar with IDL for describing interfaces as well as with the IDL-to-Java mapping. You can see how RMI is a simpler system for developing distributed applications. In general, however, CORBA differs from RMI in the following areas:

- CORBA interfaces are defined in IDL, while RMI interfaces are defined in Java.
- CORBA was designed with language independence in mind, where objects run in a heterogeneous environment. On the other hand, RMI was designed for a single language where objects run in a homogenous language environment.
- CORBA objects are not garbage collected. They are language independent, and they therefore have to be consistent with languages that do not support garbage collection. Once it is created, a CORBA object exists until you get rid of it, and deciding when to get rid of objects can be a hard decision when designing CORBA-based applications. RMI objects, on the other hand, are garbage-collected automatically.
- RMI does not support `out` and `inout` parameters (CORBA does) since local objects are passed by copy and remote objects are passed by reference to a stub.

12.4 SUMMARY

- The software that implements the CORBA specification is called an ORB. Some of the most widely used ORBs are ORBIX from IONA Technologies, VisiBroker from Inprise Corporation, and JavaIDL from JavaSoft.

- Developing a CORBA-based application is similar to that of RMI in that the first step is to define an interface. However, unlike RMI, where an interface is defined in Java, CORBA interfaces are defined in IDL. Following the interface definition, we implement the interface, develop the server and client programs, start VisiBroker's Smart Agent, and finally start the server and client programs.
- There are a number of differences between RMI and CORBA:
 - 1 CORBA is a language-independent distributed environment. RMI is designed for a single language environment—Java.
 - 2 RMI objects are garbage-collected, but CORBA objects are not.
 - 3 Unlike CORBA, RMI does not support out and inout parameters since local objects are passed by copy and remote objects are passed by reference to a stub.

