Locating CORBA objects using Java IDL

Learn how to use stringification and the COS Naming Service to find CORBA objects spread around the enterprise

Summary
Implementing CORBA objects is no trivial matter. In fact, it's a feat worthy of a round of high fives. Once the celebration subsides and everyone has left your cube suitably impressed with your coding talents, you must figure out how to publish your new object so that any computer on the enterprise can *easily* locate it. What else would you use but a naming service?

In this month's **Enterprise Java** column, Andy Krumel examines the naming facilities specified in the CORBA standard and several proprietary extensions implemented by CORBA vendors. *(4,000 words)*

Note: This article assumes you have an understanding of the CORBA standard and CORBA IDL. A link to a CORBA tutorial is provided at the end of the article.

By Andy Krumel

One of the longest weeks in my programming career was the one I spent attempting to combine a CORBA object implemented using Visigenic's VisiBroker for Java with a client using JDK 1.2 beta 2.

Obviously, this was a while ago since Java 2 was released very recently and Visigenic subsequently has been swallowed by Borland, which morphed into Inprise. Anyway, ever the one who likes to stick to standard protocols and APIs, I originally implemented the server and client applications using the Java IDL API integrated into the JDK. Playing safe, the server and client relied on the standard CORBA naming service for locating the CORBA object. All went according to schedule until the server's default Java IDL ORB was replaced with Visigenic's ORB. If I used the VisiBroker Naming Service implementation, the client could not locate the naming service, and if I used the Java IDL implementation, the server could not locate the naming service. *Pow*-- instant headache!

Had I finally been let down by open standards? Had IIOP really not made it possible for multiple ORBs to interoperate? Yes and No turned out to be the respective answers to those questions. As usual, the Templar Knight in me ran off to do battle without first stopping to read the specification's fine print or performing adequate prototyping. A little food, sleep and research quickly solved the problem once all possible hacking (and I, myself) was exhausted.

I hope this article will prevent you from experiencing one of those dreaded weeks in which your progress report simply says: "Fixed a bug by writing three lines of code."

Interoperable Object Reference (IOR)

The CORBA market provides many strategies, standards, and products for locating CORBA objects, but only one mechanism works for all IIOP-compliant CORBA implementations: Interoperable Object References (IORs). When working in a multiple-ORB environment, an IOR usually provides the only means to obtain an initial reference to an object, be it a naming service, transaction service, or customized CORBA servant.

ORBs supporting IIOP identify and publish object references using IORs. An IOR contains the information required for a client ORB to connect to a CORBA object or *servant*. Specifically, an IOR contains the following:

IIOP version -- Describes the IIOP version implemented by the ORB
Host -- Identifies the TCP/IP address of the ORB's host machine
Port -- Specifies the TCP/IP port number where the ORB is listening for client requests
Key -- Value uniquely identifies the servant to the ORB exporting the servant
Components -- A sequence that contains additional information applicable to object method invocations, such as supported ORB services and proprietary protocol support

In short, an IOR specifies the wire protocol for talking to an object as well as specifying the object's network location.

The IOR structure isn't important to programmers, since an IOR is represented through a `String` instance by a process known as stringification. *Stringification* is the process of converting a servant reference to and/or from a string representation of an IOR. Once an object reference has been stringified, it can be used by other applications to obtain a remote servant reference.

IORs are convenient because they are easy to use and are ORB-implementation-independent, but they present a small challenge: How does a client obtain a copy of the stringified IOR? Although this article won't present any "real" solutions to this problem, let me offer a few possibilities:

Distributed filesystem -- The server process writes the stringified IOR to a known mount point, or shared folder, to be read by client applications, assuming every system supports a common distributed filesystem (NFS, Novell, Windows networking)
Web publishing -- The server process writes the stringified IOR to a known location in the server's document root using a servlet or CGI script, enabling client programs to easily retrieve stringified references from the Web server using the HTTP protocol
Database -- The server process writes the IOR to a table of IORs using a symbolic name as the primary key value and client programs retrieve the IOR using any suitable database API

A future column will demonstrate how to use Java Naming and Directory Interface (JNDI) to access naming and directory services, and then provide several techniques to publish CORBA objects and IORs.

Using stringification

A CORBA object is uniquely identified by an object reference. The IIOP specification defines an IOR as an ORB-independent object reference. Given a stringified object reference, a client can create a proxy that enables requests to be forwarded to the remote CORBA object. Java IDL, along with the underlying OMG CORBA standard, defines two `org.omg.CORBA.ORB` methods used for stringification:

- `String object_to_string(org.omg.CORBA.Object o)` -- Converts the given CORBA object reference to a stringified IOR. **Note:** the string returned by this method is not designed to be human readable, as is the case with the `Object toString()` method.
- `org.omg.CORBA.Object string_to_object(String ior)` -- Converts a stringified IOR produced by the method `object_to_string()` back to a CORBA object reference.

The stringification process is illustrated in Figure 1.

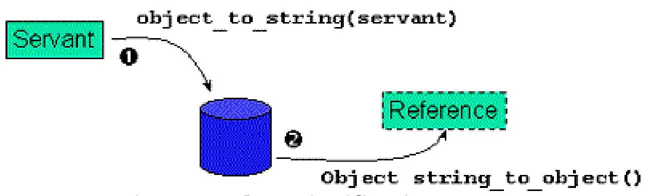

Figure 1. The stringification process

Example code

The following code sample creates a `SimpleObject` implementation, generates a stringified IOR, and writes it to a file called `simple.ior`.

```
package ior;
```

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

public class IorServer {
   public static void main(String args[]) throws IOException {
      //create object and register with ORB
      SimpleObjectImpl simple = new SimpleObjectImpl();

      //initialize ORB and stringify object
      ORB orb = ORB.init(args, null);
      String ior = orb.object_to_string(simple);
      System.out.println("IOR: " + ior);

      //write stringified object to file
      FileWriter fw = new FileWriter("simple.ior");
      fw.write(ior);
      fw.close();

      //block to prevent application from terminating
      //CORBA does not create any user threads to service clients
      System.out.println("Ready for client requests to simple object...");
      try {
         Thread.currentThread().join();
      } catch(InterruptedException ex) {}
   }
}
```

After using the command-line parameters to initialize the ORB, a stringified IOR is generated using the `object_to_string()`. The IOR is written to a file which the client program will read to obtain the IOR. Java IDL does not create any user threads to service client requests, so the program deadlocks the interpreter's thread to prevent the application from terminating. Running the application yields the following output:

```
> java ior.IorServer

IOR:000000000000001949444c3a696f722f53696d706c654f626a656374
3a312e30000000000000000100000000000000030000100000000000a737
465656c7261696e00079e00000018afabcafe000000023bd4cf8d00000008
0000000000000000
Ready for client requests to simple object...
```

The IOR may be a `String`, but it certainly is not human readable.

For this simple example, I am going to run the client application from the same directory as the `IorServer` application. The client application initializes the ORB, reads in the stringified IOR from the `simple.ior` file, and converts the IOR to a `SimpleObject` CORBA reference (proxy).

```
package ior;

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

public class IorClient {
   public static void main(String args[]) throws IOException {
      //initialize ORB and stringify object
      ORB orb = ORB.init(args, null);

      //read stringified object to file
      FileReader fr = new FileReader("simple.ior");
      BufferedReader br = new BufferedReader(fr);

      String ior = br.readLine();

      org.omg.CORBA.Object obj = orb.string_to_object(ior);
      SimpleObject proxy = SimpleObjectHelper.narrow(obj);
```

```
        //invoke methods
        proxy.invoke();
        System.out.println("Invoked method on simple object");
    }
}
```
In CORBA, the `org.omg.CORBA.Object` interface defines a CORBA proxy (object reference). It is analogous to `java.rmi.Remote` in RMI. The choice of the interface name is unfortunate, since it collides with the `java.lang.Object` class name if the `org.omg.CORBA` package is imported. But I digress.

The `string_to_object()` method returns a CORBA `Object` implementation, which refers to a `SimpleObject` servant. In Java, a program would typically *down-cast* the reference to a `SimpleObject` and invoke the reference's methods.

When using CORBA, however, a *narrow* must be performed. Narrowing is performed using the IDL-to-Java-compiler-generated `XXXHelper` class -- `SimpleObjectHelper` in this case. Attempting a simple cast will cause a `ClassCastException`. Strange, but true.

COS Naming Service

The COS Naming Service is an OMG-specified extension to the core CORBA standard, where COS stands for Common Object Service. This naming service allows an object to be published using a symbolic name, and it allows client applications to obtain references to the object using a standard API. The COS Naming Service can reside on any host accessible within the network and enables applications to publish, lookup and list CORBA object references from any network host.

A *namespace* is the collection of all names bound to a naming service. A name space may contain naming context bindings to contexts located in another server. In such a case, the name space is said to be a *federated name space* since it is a collection of name spaces from multiple servers. An interesting point is that the location of each context is transparent to the client applications; they will have no knowledge that multiple servers are handling the resolution requests for an object.

Java 2 ships with a compliant implementation of the COS Naming Service, called `tnameserv`. The command-line syntax for running `tnameserv` is:
```
> tnameserv [-ORBInitialPort ####]
```
The `tnameserv` runs on port 900 unless specified otherwise using the `-ORBInitialPort` command-line parameter.

The Java IDL COS Naming Service implementation supports transient bindings only, which means objects must be reregistered each time the naming service is restarted. The COS Naming Service implementations for Iona and Inprise are much more sophisticated and scalable, since they support persistent bindings, load balancing, and customization. The JDK `tnameserv` naming service is more than adequate to get developers started on a project. When rolling out your applications, you will want to purchase a commercial implementation to gain persistent naming and load balancing capabilities.

A COS Naming Service stores object references in a hierarchical name space; much like a filesystem uses a directory structure to store files. Specifically, *bindings* maintain the association between a symbolic name and an object reference, while *naming contexts* are objects that organize the bindings into a naming hierarchy. Like the root directory in a filesystem, the *initial naming context* provides a known point to build binding hierarchies. To complete the filesystem analogy, a binding maps to a file and a naming context maps to a directory. Thus, a naming context can contain multiple bindings and other naming contexts.

Consider a company that maintains an administrative database of devices on the network. These devices implement an administrative CORBA interface that defines attributes for querying current state and performing basic configuration tasks. Figure 2 depicts a possible naming hierarchy and illustrates that the same object may occur in multiple places within a naming hierarchy. For example, each printer is listed under a building and under printers.

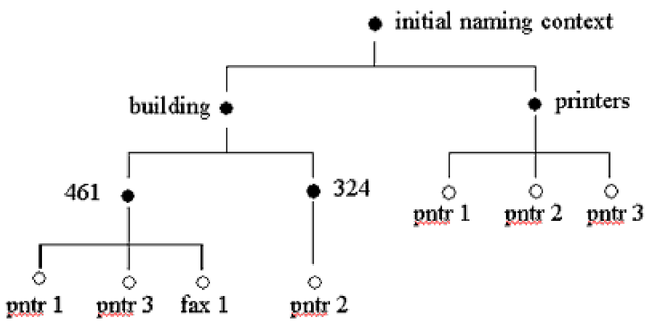

Figure 2. Example naming hierarchy

In this article, a dash (-) is used to separate each naming component. For example, `building-461-fax 1` would denote the `fax 1` node located under the `461` context, which happens to be bound to the `building` context. The name `building-461-fax 1` is known as a *compound name* because it consists of more than a single binding.

Getting the initial naming context

Before performing any tasks using the COS Naming Service, you must obtain a reference to the initial naming context. There are two standard ways to get the initial naming context:

- **Initialization service** -- The ORB class provides the `org.omg.CORBA.Object resolve_initial_references(String name)` method to obtain well-known services, where the `name` parameter specifies the requested service. Example services include the interface repository and the COS Naming Service. The name `NameService` requests a reference to the naming service. The following code demonstrates how to obtain a reference to the COS Naming Service:

```
package ior;

import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.AlreadyBound;
import org.omg.CosNaming.NamingContextPackage.CannotProceed;
import org.omg.CosNaming.NamingContextPackage.NotFound;
import org.omg.CORBA.*;

public class NamingServer {
   public static void main(String args[])
   throws NotFound, CannotProceed,
org.omg.CORBA.ORBPackage.InvalidName,
          org.omg.CosNaming.NamingContextPackage.InvalidName
   {
      //initialize ORB
      ORB orb = ORB.init(args, null);

      //obtain reference to the naming service
      org.omg.CORBA.Object objRef;
      objRef = orb.resolve_initial_references(NS);

      //narrow reference to a NamingContext
      NamingContext ncRef;
      ncRef = NamingContextHelper.narrow(objRef);
      . . .
   }
}
```

  The ORB uses the parameters passed into `init()` to locate the COS Naming Service on the network. The Java IDL ORB looks for the following parameters:

  - `org.omg.CORBA.ORBInitialHost` -- The host where the naming service is running
  - `org.omg.CORBA.ORBInitialPort` -- The port where the naming service is listening

  When specifying these properties on the command line, you can omit the `org.omg.CORBA` portion of the property name, but the full name should be specified if setting applet parameters. The following invocation requests the ORB to locate the naming service on host `steelrain` at port 2345:

  ```
     C:>java ior.NamingClient -ORBInitialHost steelrain -ORBInitialPort
  2345
  ```

  By default, the Java IDL ORB attempts to locate the naming service on the localhost for applications and the codebase host for applets at port 900.

  As convenient as this mechanism is to locate the COS Naming Service, it is a proprietary solution. All ORBs will support the initialization service call but they locate services using proprietary techniques. The techniques discussed above for locating the Java IDL COS Naming Service using the Java IDL ORB do not work when using another vendor's ORB or naming service. If your applications must run in a multivendor environment, the next technique provides a mechanism for obtaining the initial naming context.

- **Naming context IOR** -- Use the IOR that is printed out when the naming service is created to obtain a reference to the naming service using the stringification process described earlier.

  The `tnameserv` process prints its IOR and the TCP port on which it is listening when it is started:

  ```
  >tnameserv
  Initial Naming Context:
  IOR:000000000000002849444c3a6f6d672e6f72672f436f734e616d696e672f4e616d6
  96e
  67436f6e746578743a312e300000000000010000000000000030000100000000000a73746
  5
  656c7261696e0007f700000018afabcafe000000023be598160000000080000000000000
  00
  TransientNameServer: setting port for initial object references to: 900
  ```

```
Once the initial naming context is obtained, it must be narrowed to the
org.omg.CosNaming.NamingContext type. The following
snippet uses the IOR passed in as the first command-line parameter to
locate the COS Naming Service:

package ior;

import org.omg.CORBA.ORB;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.InvalidName;
import org.omg.CosNaming.NamingContextPackage.CannotProceed;
import org.omg.CosNaming.NamingContextPackage.NotFound;

public class IorNSClient {
   public static void main(String args[])
   throws NotFound, CannotProceed, InvalidName
   {
      //initialize the ORB
      ORB orb = ORB.init(args, null);

      org.omg.CORBA.Object ref;
      NamingContext ncRef;

      //get naming context reference using IOR
```

```
      String ior = args[0];
      ref = orb.string_to_object(ior);

      //narrow reference to a NamingContext
      ncRef = NamingContextHelper.narrow(ref);
      . . .
   }
}
```

Unlike the initialization service, using the IOR to locate the COS Naming service works with any combination of vendors' ORB and COS Naming Service implementations.

Resolving an object reference

Once the initial naming context is obtained, it can be used to locate registered objects. The `NamingContext` CORBA interface defines the methods for creating new subcontexts, registering CORBA objects, and locating CORBA objects:

- `bind()` -- Creates a binding of a name and an object relative within a naming context
- `rebind()` -- Unbinds the given name from an existing object (if such a binding is present) and binds it to a new object
- resolve() -- Retrieves the object reference bound to the given name
- new_context() -- Creates a new, unbound naming context
- bind_context() -- Binds a new subcontext to the naming service; equivalent to creating a new subdirectory inside a directory

The following samples demonstrate how to use each method, and the following client-side code obtains a reference to the object using the compound name simple-object (where simple is a naming context and object is the binding of the CORBA object):

```
package ior;

import org.omg.CosNaming.*;
import org.omg.CORBA.ORB;
import org.omg.CosNaming.NamingContextPackage.CannotProceed;
import org.omg.CosNaming.NamingContextPackage.NotFound;
import ior.*;

public class NamingClient {
   public static void main(String args[])
   throws NotFound, CannotProceed, org.omg.CORBA.ORBPackage.InvalidName,
          org.omg.CosNaming.NamingContextPackage.InvalidName
   {
      NamingContext ncRef = //get initial context here

      //create the naming path
      NameComponent name[] = {
         new NameComponent("simple", ""),
         new NameComponent("object", "")
      };

      //resolve the path to a reference
      org.omg.CORBA.Object ref;
      SimpleObject obj;

      ref = ncRef.resolve(name);
      obj = SimpleObjectHelper.narrow(ref);

      //invoke methods
      obj.invoke();
   }
}
```

In a filesystem, concatenating the parent directory names and the filename together using a slash (this picks your direction) forms the path to a file.

When using the naming service, the path to the object is formed using a sequence of NameComponent instances, where each NameComponent represents a binding within the naming hierarchy. Each NameComponent contains two strings: id and kind. The naming service does not use these strings except to ensure that each id is unique within the specified context. The id specifies the key value for the binding, while the kind provides a description (usually an empty string).

Binding an object

The following server-side code publishes the object retrieved in the previous example. Thus, it publishes a `ior.SimpleObject` implementation to the `simple-object` name within the naming service:

```
NamingContext ncRef = //get initial context here


NamingContext simpleCxt;
NameComponent simpleName[] = { new NameComponent("simple", "") };

simpleCxt = ncRef.new_context();
try {
   ncRef.bind_context(simpleName, simpleCxt);
} catch(AlreadyBound ex) {
   //already bound so resolve to it instead
   objRef = ncRef.resolve(simpleName);
   simpleCxt = NamingContextHelper.narrow(objRef);
}

//export the object to the naming service
NameComponent objName[] = { new NameComponent("object", "") };
simpleCxt.rebind(objName, obj);

//block to prevent program from ending and wait for client requests
System.out.println("Ready for client simple requests...");
```

This example uses a compound name that requires a little care. A real naming service implementation will support persistent names, which means a context can remain in existence long after the registered object has been removed from the name space. Your object registration code needs to handle the cases in which a parent naming context may or may not currently exist, and preserve all bindings if it does exist.

This example uses exception handling to efficiently handle both cases. The code assumes the `simple` context is not currently bound to the initial naming context and proceeds as follows:

1. The application creates a one-element `NameComponent` array for the `simple` naming context binding.
2. The application creates a new, unbound naming context by invoking the `new_context()` method on the root naming context. A `NamingContext` instance can only be created using another `NamingContext`.
3. The new `NamingContext` is bound to the root naming context under the name `simple` using the `bind_context()` method.
4. If a context is already bound to the `simple` name, the `NamingContext bind_context()` method generates an `org.omg.CosNaming.CosNamingPackage.AlreadBound` exception. Anticipating this problem, the program catches this exception (if it occurs) and resolves the existing context using `resolve()`, just like resolving a normal object.

Once the `simple` naming context is obtained from the naming service, a NameComponent array is built and bound using the `NamingContext rebind()` method.

Notice how `NamingContext` objects are published using the `bind_context()` method while all other CORBA objects are published using the `bind()` or `rebind()` methods. A `NamingContext` can be published using a `bind()` method, but it will not be usable as a naming context binding with the naming service.

Other naming approaches

If using an IOR is too crude and the COS Naming Service is too cumbersome, you may want to investigate the Java Naming and Directory Service (JNDI) or the Inprise Web Naming Service. JNDI provides a generic API for directory and naming services. The current incarnation of JNDI (as of mid-January) contains a beta implementation of a COS Naming Service provider. This provider enables you to bind, list, and resolve bindings using the standard JNDI API, which can be much more simple and elegant than using the standard `org.omg.CosNaming` package.

The Inprise Web Naming Service uses a Web server to manage and publish CORBA object IORs. This makes obtaining references as easy as specifying a URL and could potentially solve the problem of obtaining the initial naming context reference.

Both Inprise and Iona provide proprietary naming facilities through the use of binding an object to a name at the time of object construction. The name bindings are broadcast across the network through a proprietary communication scheme to all properly configured ORBs. The end result is a very easy-to-use naming service implementation in which the client application uses a single, lightweight `bind()` method to resolve to a specific object.

The limitations of this type of naming scheme are:

1. Client hosts must be properly configured to receive binding broadcast notifications.
2. An object can be bound to a single name only.
3. Proprietary schemes will not work when using multiple-ORB implementations. This situation can occur when using ORBs built into commercial products such as Web browsers, database servers, and application servers along with your custom applications written using your company's standard ORB -- not to mention the combinations that result when integrating with your suppliers and customers.

Conclusion

CORBA provides two standard mechanisms for enabling client programs to obtain servant references: IORs and the COS Naming Service. As is normally the case with open standards, software vendors have developed additional naming service implementations. Choosing a naming service for your applications will depend on ORB vendor(s), network topology, existing company software architecture standards, and your level of aversion to "proprietary standards."

This article discussed four object reference resolution mechanisms:

IOR -- The only mechanism that is truly ORB-independent

COS Naming Service -- Provides scalability and robustness

Inprise Web Naming Service -- Uses a familiar naming convention (URLs) to publish and resolve references; potentially ORB-independent

Proprietary ORB naming services -- Provides a simple-to-program, robust naming solution

Good luck locating the object of your dreams.■

About the author

andy.krumel Andy Krumel graduated from the US Naval Academy and started his career as a naval aviator. However, after a series of engine fires, radio failures, and assorted other mishaps, a career in computer programming looked more long-lived.

Today, Andy is the founder of K&A Software, a rapidly growing, three-year-old Silicon Valley training and consulting firm specializing in Java and distributed applications. Andy and his cohorts will do just about anything to spread the word about Java's fantastic capabilities, from speaking at conferences, to teaching corporate training classes, to even writing an occasional article. When not pontificating, Andy is squirreled away creating solutions for one of K&A's corporate clients.

Resources

- Download the complete source in zip format. You will need Java 2 to run the code
  http://www.javaworld.com/jw-02-1999/enterprise/jw-02-enterprise.zip
- Sun's Java IDL documentation, which includes documentation covering the Java IDL API, the naming service, creating and using CORBA object references, and an introductory tutorial
  http://java.sun.com/products/jdk/1.2/docs/guide/idl/index.html
- Java 2 includes support for running CORBA server and client processes, including an implementation of a CORBA-compliant transient naming service
  http://java.sun.com/products/jdk/1.2/
- Java 2 does not ship with an IDL-to-Java compiler so you must download it separately if you intend to develop and implement ORB-based services. Accessing this page requires you possess a Developer Connection account (joining is free)
  http://developer.java.sun.com/developer/earlyAccess/jdk12/idltojava.html
- An excellent introductory tutorial to Java IDL is provided online in Sun's *The Java Tutorial*
  http://java.sun.com/docs/books/tutorial/idl/index.html
- The Java Naming and Directory Interface (JNDI) is a standard Java extension and provides applications with a unified interface to multiple naming and directory services
  http://java.sun.com/products/jndi/index.html
- The CORBA/IIOP 2.2 Specification is available online
  http://www.omg.org/corba/corbaiiop.html
- The Iona OrbixNames, Iona's implementation of the COS Naming Service is documented online
  http://www.iona.com/products/sysman/orbixnames/fordevelopers.html
- The VisiBroker Naming and Event Services, Inprise's implementation of the COS Naming Service, is

documented online
http://www.inprise.com/techpubs/books/vbnes/vbnes33/index.html

- Read the first-ever **Enterprise Java** column,"Revolutionary RMI: Dynamic class loading and behavior objects," in the December issue of *JavaWorld*
http://www.javaworld.com/jw-12-1998/jw-12-enterprise.html