CORBA con Java IDL

© Francisco Charte Ojeda

Sumario

La plataforma Java 2 incorpora como parte de su núcleo un ORB compatible con la especificación CORBA 2.0, así como servicios adicionales que facilitan el desarrollo de servidores y clientes CORBA.

Introducción

La última versión de la plataforma Java, conocida como Java 2, se presentó en su día como el paso definitivo que conseguirá introducir Java en entornos profesionales y de ámbito empresarial. Con esta meta Java 2 incorpora muchas novedades interesantes. Una de esas novedades es Java IDL, especialmente interesante para aquellos desarrollos que requieren trabajar en entornos distribuidos heterogéneos con CORBA como pilar central.

Java 2 cuenta con un mecanismo, heredado de versiones previas, que facilita la creación y uso de componentes en entornos distribuidos. Este mecanismo es conocido como Java RMI. Java RMI es un mecanismo más sencillo de usar que Java IDL, principalmente porque está basado completamente en Java. Éste, sin embargo, es su mayor inconveniente, ya que los componentes tan sólo pueden estar creados y ser usados desde Java.

Java IDL no es una novedad propiamente dicha, puesto que existe desde hace algún tiempo, pero sí lo es el hecho de que forme parte de la plataforma Java 2. Esto significa que es posible usar Java IDL con la tranquilidad de que funcionará en cualquier sistema que tenga disponible esta versión, sin necesidad de instalar elementos adicionales.

En este artículo se introducirán los diversos elementos que forman parte de Java IDL, principalmente el compilador de IDL a Java, los paquetes en los que encontramos los diversos objetos que componen el ORB y el servicio de nombres. Tras dicha introducción se propondrá un caso práctico de desarrollo de un servidor y un cliente Java IDL.

El compilador `idltojava`

Los componentes CORBA son objetos que exponen servicios a través de interfaces. Estas interfaces se describen usando un lenguaje estándar, conocido como IDL, cuya sintaxis es similar a la usada por lenguajes como C++ y Java. Una definición IDL es posteriormente convertida, mediante una herramienta dependiente del lenguaje, en uno o varios archivos que contienen los habituales *stub* y *skeleton*, partiendo de los cuales se codificarán clientes y servidor, respectivamente.

El compilador IDL para Java 2 es `idltojava`. Esta herramienta no se encuentra en el paquete de instalación de la plataforma Java 2, siendo preciso obtenerlo separadamente de la web de JavaSoft. Se ejecuta desde la línea de comandos y el único parámetro imprescindible es el nombre del archivo que contiene el código IDL. Adicionalmente puede tomar una o más opciones e indicadores, estando todos ellos documentados en el archivo HTML que acompaña al compilador.

Al igual que la mayoría de los compiladores IDL, `idltojava` necesita acceder a un preprocesador C++ para poder procesar el código IDL. No obstante, si en dicho código no se utiliza ninguna directiva de compilación condicional o similar, puede utilizarse la opción `-fno-cpp` para evitar la búsqueda y ejecución del preprocesador. Por defecto `idltojava` está preparado para usar el preprocesador de Microsoft Visual C++, lógicamente hablando de la plataforma Windows. Si disponemos de cualquier otro compilador será preciso asignar a las variables de entorno `CPP` y `CPPARGS` el camino y los argumentos necesarios, respectivamente, para poder encontrarlo y ejecutarlo.

Suponiendo que vamos a crear un servidor cuya finalidad será facilitar a los clientes la ejecución de consultas contra una base de datos, con la consiguiente obtención de resultados, podríamos usar el código IDL mostrado en el Listado 1. Para compilar este código introduciríamos en la línea de comandos la

sentencia `idltojava -fno-cpp Consulta.idl`, que sin ninguna respuesta directa generaría una nueva carpeta conteniendo cinco archivos.

```
module SvrConsultas {
  interface IConsulta {
    string Consulta(in string Parametros);
  };
};
```

Listado 1. Código IDL contenido en el archivo Consulta.Idl

Los módulos IDL se corresponden con los paquetes Java. Esto implica, partiendo del Listado 1, que el módulo `SvrConsultas` genere como resultado un paquete con ese mismo nombre cuyos elementos se almacenan en una carpeta también llamada `SvrConsulas`. En dicha carpeta se almacena un archivo con la interfaz en lenguaje Java, un *stub*, un *skeleton* y dos archivos con clases auxiliares. Vamos a analizar cada uno de estos elementos con algo más de detalle.

La interfaz del servidor

Una interfaz escrita en IDL no puede ser utilizada directamente desde un lenguaje de programación, por lo cual es necesario realizar una traducción. El compilador `idltojava` se encarga de esta tarea, generando un archivo que contiene la misma interfaz pero escrita en Java. A partir del código del Listado 1 se genera en la carpeta `SvrConsultas` un archivo llamado `IConsulta.Java`, con el contenido que puede apreciarse en el Listado 2.

```
/*
 * File: ./SVRCONSULTAS/ICONSULTA.JAVA
 * From: CONSULTA.IDL
 * Date: Wed Feb 24 19:07:42 1999
 *   By: C:\ARCHIV~1\IDLTOJ~1\IDLTOJ~1.EXE Java IDL 1.2 Aug 18 1998
16:25:34
 */

package SvrConsultas;
public interface IConsulta
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity {
    String Consulta(String Parametros)
;
}
```

Listado 2. Archivo `IConsulta.Java` generado por `idltojava`

El módulo IDL `SvrConsultas` se ha convertido en la paquete Java `SvrConsultas`, ocurriendo lo mismo con la interfaz `IConsulta`. Ésta cuenta con un solo método, cuya sintaxis Java es prácticamente idéntica a la usada en el código IDL que habíamos escrito. Observe que el tipo IDL `string` se ha convertido al tipo Java `String`, que es su equivalente.

No hay que olvidar que esta interfaz Java se utilizará para implementar un objeto CORBA o bien para facilitar que un cliente acceda a dicho objeto. Por ello la interfaz `IConsulta` está basada en `org.omg.CORBA.Object`, una interfaz que cuenta con los métodos necesarios para mantener, comprobar y manipular la referencia a un objeto CORBA. Como veremos posteriormente, los métodos de esta interfaz se han implementado automáticamente al compilar el código IDL.

La otra interfaz de la que desciende `IConsulta`, llamada `org.omg.CORBA.portable.IDLEntity`, no aporta método alguno y sirve tan sólo como una marca utilizada para saber si un cierto objeto es una implementación IDL y, por lo tanto, dispone de una clase *Helper* que facilita su gestión. Trataremos las clases *Helper* algo más adelante en este mismo artículo.

El esqueleto de servidor

Partiendo de la interfaz Java que se acaba de comentar, el compilador IDL ha generado un esqueleto en el cual se implementan los métodos de bajo nivel y carácter genérico. Dicho esqueleto está contenido en el archivo `_IConsultaImplBase.Java`, siendo su contenido el mostrado en el Listado 3. Como puede ver, `_IConsultaImplBase` es una clase abstracta que deriva de `org.omg.CORBA.DynamicImplementation` e implementa la interfaz `SvrConsultas.IConsulta`.

```
/*
 * File: ./SVRCONSULTAS/_ICONSULTAIMPLBASE.JAVA
 * From: CONSULTA.IDL
 * Date: Wed Feb 24 19:07:42 1999
 *   By: C:\ARCHIV~1\IDLTOJ~1\IDLTOJ~1.EXE Java IDL 1.2
                                          Aug 18 1998 16:25:34
 */
package SvrConsultas;
public abstract class _IConsultaImplBase extends
             org.omg.CORBA.DynamicImplementation
             implements SvrConsultas.IConsulta {
    // Constructor
    public _IConsultaImplBase() {
         super();
    }
    // Type strings for this class and its superclases
    private static final String _type_ids[] = {
        "IDL:SvrConsultas/IConsulta:1.0"
    };
    public String[] _ids() { return (String[]) _type_ids.clone(); }

    private static java.util.Dictionary _methods =
                                       new java.util.Hashtable();
    static {
      _methods.put("Consulta", new java.lang.Integer(0));
     }
    // DSI Dispatch call
    public void invoke(org.omg.CORBA.ServerRequest r) {
       switch (((java.lang.Integer) _methods.get(
                                    r.op_name())).intValue()) {
           case 0: // SvrConsultas.IConsulta.Consulta
             {
             org.omg.CORBA.NVList _list = _orb().create_list(0);
             org.omg.CORBA.Any _Parametros = _orb().create_any();
             _Parametros.type(org.omg.CORBA.ORB.init

().get_primitive_tc(org.omg.CORBA.TCKind.tk_string));
             _list.add_value("Parametros", _Parametros,
                org.omg.CORBA.ARG_IN.value);
             r.params(_list);
             String Parametros;
             Parametros = _Parametros.extract_string();
             String ___result;
                          ___result = this.Consulta(Parametros);
             org.omg.CORBA.Any __result = _orb().create_any();
             __result.insert_string(___result);
             r.result(__result);              }
             break;
            default:
             throw new org.omg.CORBA.BAD_OPERATION(0,
                   org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE);
       }
   }
  }
```

Listado 3. Archivo _IConsultaImplBase.Java generado por idltojava

Este esqueleto de servidor es abstracto puesto que su finalidad es únicamente servir como clase base de otra, escrita por nosotros, en la que realmente se implementen los métodos de la interfaz `IConsulta`. `_IConsultaImplBase` incorpora la implementación de los métodos necesarios para identificar a un objeto de esta clase, así como para realizar llamadas dinámicas a los métodos de la interfaz `IConsulta`.

Lo primero que encontramos en `_IConsultaImplBase` es un constructor por defecto, que no necesita parámetro alguno y se limita a ejecutar el constructor de la clase base. A continuación encontramos una cadena con la identificación de la clase, cadena que es devuelta por el método `_ids()` implementado poco más abajo.

Un cliente que dispone del *stub* apropiado puede efectuar llamadas estáticas a los métodos expuestos en la interfaz de un servidor CORBA. Esto, de hecho, es lo habitual. Es posible, no obstante, realizar llamadas dinámicas prescindiendo del *stub* y sirviéndose de un repositorio de interfaces. En cualquier caso, el método `invoke()` es el encargado de recibir cualquier invocación a un método de la interfaz `IConsulta`, que se traducirá en una llamada al método que corresponda, para lo cual se apoya en una tabla de búsqueda generada poco antes. Dicha tabla contiene el nombre de cada método y un índice, como puede ver en el Listado 3. En este caso tan sólo existe un elemento, el método `Consulta()`, al que corresponde el índice 0.

Como veremos en un momento, la clase `_IConsultaImplBase` realmente nos servirá para crear la implementación propiamente dicha de nuestro servidor. No es necesario conocer el funcionamiento de `_IConsultaImplBase` para poder desarrollar un servidor CORBA, pero siempre resulta interesante tener una idea de lo que está ocurriendo.

El *stub* para el cliente

Un componente CORBA que actúa como servidor puede estar ejecutándose en cualquier máquina conectada a una red, recibiendo peticiones por parte de clientes que pueden ser tanto locales como remotos. Indistintamente de ello, el cliente siempre tendrá la ilusión de que la llamada que está realizando se ejecuta localmente. Esto es así gracias a la clase *stub* generada por el compilador `idltojava`. Al compilar nuestra interfaz IDL `IConsulta` el código generado en el archivo `_IConsultaStub.Java` es el que puede verse en el Listado 4.

La finalidad del *stub* es, por lo tanto, evitar que el programador tenga que ocuparse de dar todos los pasos que serían necesarios para poder ejecutar un método en un objeto remoto. Para ello la clase `_IConsultaStub` implementa la interfaz `IConsulta`, de tal forma que es posible llamar directamente al método `Consulta()` facilitando el parámetro necesario y obteniendo el resultado correspondiente.

Como puede ver en el Listado 4, la implementación del método `Consulta()` que hay en el *stub* no realiza realmente ninguna operación con los parámetros. Seremos nosotros los que tengamos que escribir el código del método `Consulta()`, cuando implementemos el servidor. En el *stub* lo que se hace es utilizar las clases del ORB para preparar y ejecutar la llamada remota, básicamente creando un objeto `org.omg.CORBA.Request` y usando el método `invoke()` para llamar a `Consulta()`.

Observe que tras crear el mencionado objeto se establece el tipo del valor de retorno, en este caso una cadena, procediéndose a continuación a preparar una lista con los parámetros de entrada. Dados estos pasos se ejecuta la llamada y se obtiene el valor de retorno, que es lo que finalmente se devuelve.

Si obviamos los detalles aislados en la clase que actúa como *stub*, usando este módulo sin preocuparnos de su contenido, a la hora de crear el cliente veremos que básicamente es como si obtuviésemos una referencia a un objeto y realizásemos una llamada local a uno de sus métodos. La realidad, sin embargo, será mucho más compleja, tal y como se muestra de forma simplificada en la Figura 1.

```
/*
 * File: ./SVRCONSULTAS/_ICONSULTASTUB.JAVA
 * From: CONSULTA.IDL
 * Date: Wed Feb 24 19:07:42 1999
 *   By: C:\ARCHIV~1\IDLTOJ~1\IDLTOJ~1.EXE Java IDL 1.2
                  Aug 18 1998 16:25:34
 */

package SvrConsultas;
public class _IConsultaStub
    extends org.omg.CORBA.portable.ObjectImpl
    implements SvrConsultas.IConsulta {

    public _IConsultaStub(org.omg.CORBA.portable.Delegate d) {
          super();
          _set_delegate(d);
    }

    private static final String _type_ids[] = {
        "IDL:SvrConsultas/IConsulta:1.0"
    };

    public String[] _ids() { return (String[]) _type_ids.clone(); }

    //      IDL operations
    //          Implementation of ::SvrConsultas::IConsulta::Consulta
    public String Consulta(String Parametros)
  {
          org.omg.CORBA.Request r = _request("Consulta");

r.set_return_type(org.omg.CORBA.ORB.init().get_primitive_tc(
                org.omg.CORBA.TCKind.tk_string));
          org.omg.CORBA.Any _Parametros = r.add_in_arg();
          _Parametros.insert_string(Parametros);
          r.invoke();
          String __result;
          __result = r.return_value().extract_string();
          return __result;
    }
};
```

Listado 4. Archivo `_IConsultaStub.Java` **generado por** `idltojava`

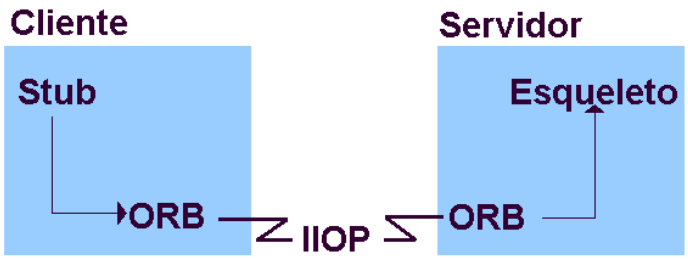

Figura 1. Esquema simplificado que muestra la estructura de un cliente y un servidor, con sus correspondientes ORB usando el protocolo IIOP para comunicarse por una red IP.

Clases auxiliares

Además de los archivos comentados hasta ahora, el compilador `idltojava` genera dos más que contienen clases auxiliares. Se trata de clases cuya finalidad es contener básicamente métodos estáticos, es decir, que pueden ser ejecutados sin necesidad de crear un objeto de esa clase. Las dos clases en cuestión son `IConsultaHolder` e `IConsultaHelper`.

Las referencias genéricas a objetos Java pueden ser convertidas en referencias de un cierto tipo mediante la habitual técnica de moldeado, existente también en otros lenguajes como C/C++ y Pascal. Una referencia a un objeto genérico CORBA, por el contrario, no puede convertirse mediante un moldeado de tipo en Java, siendo necesario usar métodos alternativos.

El método más interesante de la clase `IConsultaHelper`, llamado `narrow()`, tiene la finalidad de facilitar la conversión de una referencia genérica a una del tipo que nos interesa lo que, en definitiva, permitirá usar esa referencia para llamar a los métodos establecidos originalmente en la interfaz. Lógicamente cada interfaz que se defina en un módulo IDL contará con su propia clase `IXXXHelper`, con un método `narrow()` específico para ella.

Por regla general no nos preocuparemos del resto de los métodos de las clases auxiliares, el único útil en la mayoría de los casos es `narrow()` cuyo uso veremos posteriormente, a la hora de implementar un cliente que use el método `Consulta()` de la interfaz `IConsulta`.

Inicialización del ORB

Tanto servidor como cliente deben inicializar y usar los servicios de su ORB para facilitar la comunicación con la otra parte. El ORB es un objeto de la clase `ORB`, definida en el paquete `org.omg.CORBA` que, lógicamente, tendremos que incluir en los módulos de código que escribamos a la hora de implementar servidor y cliente.

La creación e inicialización del ORB se realizan en un solo paso: la llamada al método estático `init()`. Existen varias versiones de este método en la clase `ORB`, según deseemos usar un ORB en una aplicación corriente o en un *applet* Java. Los parámetros necesarios son dos: los parámetros de entrada al método `main()` y una matriz de propiedades. Si se trata de un *applet* el primer parámetro será una referencia a sí mismo, es decir, se usará `this` como primer parámetro.

Típicamente el segundo parámetro al llamar al método `init()` será nulo, a menos que se precise realizar alguna operación especial como seleccionar un ORB que no sea el implementado por defecto por Java IDL. Para ello hay que crear un objeto `java.util.Properties`, utilizar el método `put()` para añadir los parámetros que procedan y después pasar el mencionado objeto como segundo parámetro al método `init()`.

Una vez inicializado, es posible usar los métodos del ORB para realizar tareas comunes como la resolución de referencias a elementos generales o la conversión de referencias a objetos CORBA en cadenas y viceversa. A la hora de implementar el servidor y cliente de ejemplo veremos algunos de estos métodos.

El servicio de nombres

Los ORB CORBA usan un protocolo común, normalmente IIOP, que les permite comunicarse entre sí a pesar de que hayan sido implementados por diferentes fabricantes. Hasta llegar a ese punto en el que se establece la comunicación, sin embargo, es preciso que el cliente localice de alguna forma al servidor, que puede encontrarse virtualmente en cualquier punto de la red.

Existen diversos métodos cuyo fin es facilitar la búsqueda de objetos CORBA, siendo uno de los más habituales usar el servicio de nombres descrito en los servicios comunes CORBA. Imagine que tiene una aplicación cliente, un programa cualquiera, que necesita acceder a los datos almacenados en un archivo, que haría las veces de servidor. Para ello precisaría conocer no sólo el nombre del archivo, sino también el camino completo en el que se encuentra. Esta referencia, que sería una cadena de caracteres, se facilitaría al sistema de archivos del sistema operativo quien, en último término, sería el que resolvería la búsqueda.

El servicio de nombres CORBA es similar al sistema de archivos, con la diferencia de que localiza objetos, no archivos, y de que la búsqueda no se realiza en un soporte de información sino a través de una

red. Lógicamente cada fabricante tiene su particular implementación del servicio de nombres CORBA pero, en teoría, todos ellos deberían interactuar sin problemas entre sí puesto que han de seguir una especificación muy clara.

A diferencia de un sistema de archivos, el servicio de nombres CORBA no está siempre disponible. Esto es, es preciso iniciar dicho servicio, generalmente ejecutando algún agente que queda a la escucha en un cierto puerto del sistema. Otra diferencia fundamental es que no es preciso poner en marcha el servicio de nombres en todas las máquinas de la red, se trata de un servicio distribuido que es suficiente con que esté disponible en un puesto.

Un sistema de archivos es persistente, mientras que el servicio de nombres CORBA no lo es. Esto significa que cada vez que se detiene y reinicia se pierden todos los objetos que se habían registrado hasta ese momento, siendo preciso reiniciar también los servidores. Quedan fuera de esta regla las referencias a ciertos objetos genéricos que siempre están disponibles, desde el mismo momento en que se inicia el servicio de nombres.

Inicio del sistema de nombres

El servicio de nombres CORBA es una especificación, de la cual cada fabricante realiza su implementación. La implementación Java IDL es un programa, llamado `tnameserv`, que es necesario ejecutar en un puesto de la red. Este servicio sí forma parte de la plataforma Java 2, por lo que a diferencia del compilador `idltojava` no es necesario obtenerlo separadamente.

Al ejecutar el servicio de nombres obtendremos una salida similar a la mostrada en la Figura 2. La larga secuencia de dígitos hexadecimales es la IOR del objeto CORBA que actúa como servicio de nombres. Una IOR es una referencia a un objeto CORBA independiente de ORB, lo cual significa que puede ser usada para facilitar la comunicación entre objetos que usan ORB de distintos fabricantes. En este caso el IOR del servicio de nombres podría servir para obtener una referencia a él sin usar el habitual método de resolución que se expondrá en el siguiente punto.

Otro de los datos que podemos apreciar en la Figura 1 es el número de puerto que deberá utilizar cualquier servicio o cliente que necesite acceder al servicio de nombres. Por defecto el puerto usado es el 900. Si se desea usar otro puerto basta con indicarlo al ejecutar `tnameserv`, usando la opción `-ORBInitialPort` tal y como se hace en la siguiente sentencia. En este caso el puerto a usar es el `1025`.

```
tnameserv -ORBInitialPort 1025
```

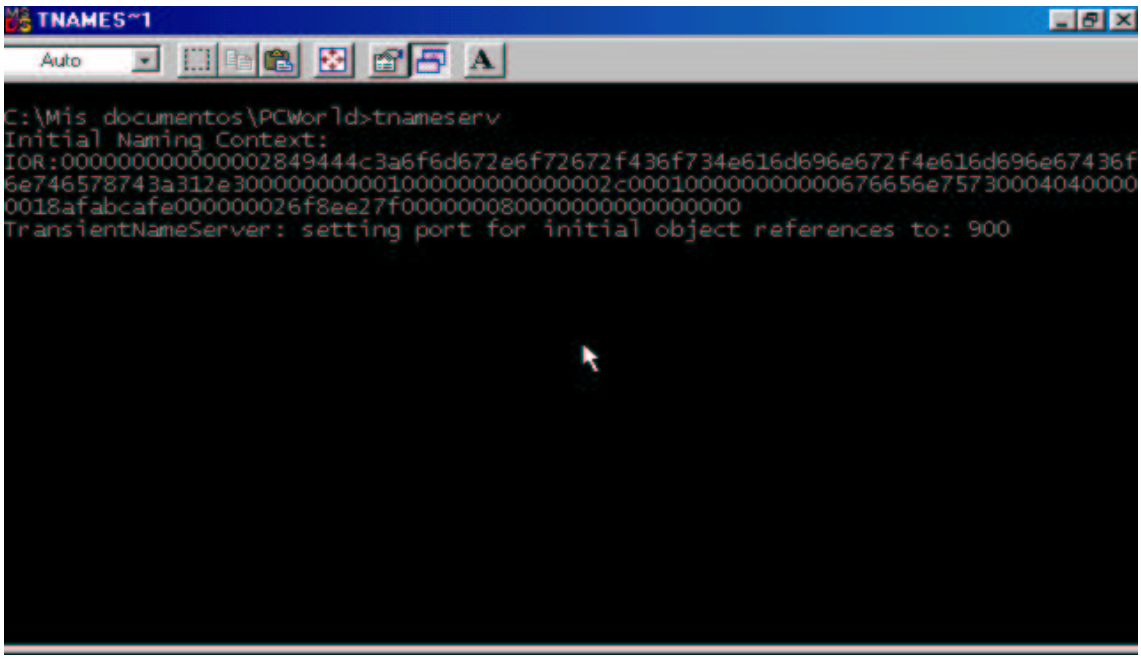

Figura 2. Al poner en marcha el servicio de nombres de Java IDL obtenemos el IOR correspondiente y el número de puerto por el que se queda a la escucha.

Para poder acceder al servicio de nombres desde cualquier aplicación es preciso saber el nombre del servidor y número de puerto en que está funcionando. Generalmente `tnameserv` se ejecutará en un ordenador de la red, un servidor, usando un puerto bien conocido por servidores y clientes CORBA. Un método alternativo consiste en usar la IOR devuelta por el servicio de nombres cuando es puesto en marcha, de tal forma que podría estar ejecutándose en un punto u otro de la red de forma indistinta. En cualquier caso, esa IOR debería estar accesible para las aplicaciones en algún punto bien conocido, como un servidor web o una base de datos.

Uso del servicio de nombres

Una vez puesto en marcha, cualquier aplicación que lo precise puede acceder al servicio de nombres. Las operaciones habituales son básicamente dos: registrar una nueva referencia a un objeto, en el caso de los servidores, y resolver una referencia, por parte de los clientes. Independientemente de ello, el primer paso que habrá que dar será obtener la referencia al propio servicio de nombres, que no es más que otro objeto CORBA.

El ORB, que ya teníamos inicializado, cuenta con un método llamado `resolve_initial_references()` que se utiliza para obtener referencias a objetos de los servicios comunes, como el propio servicio de nombres. Dicho método tan sólo precisa como parámetro el nombre del objeto cuya referencia se desea obtener, devolviendo como resultado una referencia de tipo genérico que será preciso convertir al tipo adecuado.

La siguiente sentencia usa el método `resolve_initial_references()` del ORB para buscar el objeto `NameService`. La referencia obtenida es de tipo genérico, concretamente `org..omg.CORBA.Object`. No es posible usar una referencia de este tipo para acceder a los métodos del servicio de nombres, que es un objeto de la clase `NamingContext`, por lo que es necesario realizar la conversión apropiada usando el método `narrow()` de la clase `NamingContextHelper`. Ésta es una clase auxiliar similar a la generada por `idltojava` para nuestras clases.

```
NamingContext ServNombres =
  NamingContextHelper.narrow(
   Orb.resolve_initial_references("NameService"));
```

Teniendo disponible una referencia al servicio de nombres es posible realizar básicamente dos tareas diferentes: registrar un objeto, facilitando un camino y una referencia, o bien resolver una referencia partiendo del camino. En el primer caso se usa el método `rebind()`, facilitando el camino que identificará al objeto y una referencia, mientras que en el segundo nos serviremos del método `resolve()` entregando como único parámetro el citado camino.

Implementación del servidor

Del servidor hasta ahora tan sólo contamos con un esqueleto, la clase `_IConsultaImplBase`, que servirá como base de la clase en la que se implementen los métodos inicialmente establecidos en la interfaz IDL. Aunque en este caso tan sólo tenemos una, en la práctica existirá una clase de implementación por cada interfaz que se haya definido. Esto no conlleva, sin embargo, que tengan que existir varios servidores de objetos, ya que un mismo servidor puede preparar y registrar todos los objetos que sean precisos.

El módulo ServidorConsulta.`java`, que puede ver en el Listado 5, contiene la definición de dos clases: `ConsultaServant` y `ServidorConsulta`. La primera deriva de `_IConsultaImplBase` y es la clase que realmente implementa la funcionalidad de la interfaz `IConsulta`, en este caso mostrar los parámetros recibidos y devolver una cadena indicando que la consulta se ha ejecutado. En la práctica esta clase podría realizar cualquier función, lógicamente más útil que la mostrada en este ejemplo.

La segunda clase es realmente la aplicación que actúa como servidor, tal y como denota la existencia del método `main()`. En éste se comienza inicializando el ORB, creando un objeto de la clase `ConsultaServant` y conectando dicho objeto con el ORB, de tal forma que éste pueda resolver adecuadamente las peticiones. En este momento tenemos un objeto CORBA preparado para ser usado.

Para que el objeto anterior pueda estar accesible desde un cliente, ya sea en la misma máquina o en cualquier otra conectada a la red, es necesario informar al servicio de nombres de cuál es el nombre y la

referencia. Tras obtener una referencia al servicio de nombres preparamos el camino con que se identificará el objeto, en este caso el nombre `IConsulta`, y usamos el método `rebind()` para realizar el registro.

```
// Este módulo contiene el servidor, con el correspondiente
  // método main(), y la clase que servirá a las peticiones
  // de los clientes CORBA

import SvrConsultas.*; // Importamos el esqueleto y clases auxiliares

  // Paquetes para acceder al ORB y al servicio de nombres
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

  // Esta clase implementa la interfaz IConsulta derivando del
  // esqueleto generado por idltojava
class ConsultaServant extends _IConsultaImplBase {
     // simplemente implementamos el método
    public String Consulta(String Parametros) {
          // que se limitará a mostrar los parámetros recibidos
        System.out.println(Parametros);
          // y devolverá una cadena como resultado
     return "Ejecutada la consulta\n" + Parametros;
    }
}

  // Esta clase es el servidor propiamente dicho
public class ServidorConsulta {
    // El método main() es el punto de entrada al servidor
  public static void main(String args[]) {
    try {
        // Obtenemos una referencia al ORB, inicializándolo
      ORB Orb = ORB.init(args, null);

        // Creamos el objeto que implementa la interfaz IConsulta
      ConsultaServant Servidor = new ConsultaServant();
        // Conectamos el objeto CORBA con el ORB
      Orb.connect(Servidor);

        // Obtenemos una referencia al servicio de nombres
      NamingContext ServNombres = NamingContextHelper.narrow(
         Orb.resolve_initial_references("NameService"));

        // Preparamos el camino con el nombre del objeto
      NameComponent NombreInterfaz =
                    new NameComponent("IConsulta", "");
      NameComponent CaminoObjeto[] = {NombreInterfaz};

        // y lo registramos en el espacio de nombres
        // facilitando el camino y la referencia al objeto
      ServNombres.rebind(CaminoObjeto, Servidor);

      System.out.println(
          "Servidor esperando solicitudes de consultas");

        // Esperamos a recibir peticiones de clientes
      java.lang.Object RS = new java.lang.Object();
      synchronized(RS) {
        RS.wait();
      }

    } catch (Exception X) { // en caso de fallo
       System.err.println(X); // mostrar toda la información
       X.printStackTrace(System.out); // disponible
    }
  }
}
```

Listado 5. Implementación del servidor `ServidorConsulta`

Si la aplicación servidor, que es la que ha creado el objeto que finalmente atenderá a las solicitudes de los clientes, termina sin más, dicho objeto será destruido y el servicio de nombres contará con una referencia a algo inexistente. Por eso es preciso esperar indefinidamente solicitudes de clientes.

Observe que toda la operación de inicialización del ORB, conexión y registro en el servicio de nombres se protege en un bloque contra posibles excepciones. La más habitual es que no pueda accederse al servicio de nombres, bien porque no se haya iniciado o se encuentre escuchando en otro servidor o puerto.

Codificación de un cliente

Asumiendo que ya hemos creado nuestro servidor, estamos en disposición de crear el cliente o clientes que utilizarán sus servicios. En el Listado 6 se muestra una simple aplicación que usa el primer parámetro facilitado en la línea de comandos para realizar una consulta, mostrando en pantalla el resultado obtenido y finalizando.

```
// Este es el programa cliente, que obtendrá una referencia
  // al objeto sirviéndose del servicio de nombres, tras lo
  // cual ejecutará la consulta que se haya introducido como
  // parámetro de entrada

import SvrConsultas.*; // Importamos el stub y las clases auxiliares

  // Paquetes para acceder al ORB y al servicio de nombres
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class Cliente {
    // Punto de entrada al cliente
  public static void main(String args[]) {
    try {
        // Inicializamos el ORB y obtenemos la referencia
      ORB Orb = ORB.init(args, null);

        // Obtenemos una referencia al servicio de nombres
      NamingContext ServNombres = NamingContextHelper.narrow(
         Orb.resolve_initial_references("NameService"));

        // Preparamos el camino con el nombre del objeto
      NameComponent NombreInterfaz =
                    new NameComponent("IConsulta", "");
      NameComponent CaminoObjeto[] = {NombreInterfaz};

        // Solicitamos al servicio de nombres la resolución
        // de la referencia
      IConsulta RefServ = IConsultaHelper.narrow(
            ServNombres.resolve(CaminoObjeto));

        // y la usamos para ejecutar la consulta
      String Resultado = RefServ.Consulta(args[0]);
      System.out.println(Resultado); // y mostrarla

    } catch (Exception X) { // si hay algún fallo
      System.out.println(X); // mostrar toda la información
      X.printStackTrace(System.out); // disponible
    }
  }
}
```

Listado 6. Implementación del cliente

Los primeros pasos dados en el método `main()` del cliente son similares a los anteriormente descritos en el servidor: se inicializa el ORB, se obtiene una referencia al servicio de nombres y se

prepara el camino con el nombre del objeto al que quiere accederse. Observe que no se usa el método `connect()` del ORB, puesto que el cliente no dispone de un objeto que pondrá a disposición de otros programas.

Usando el método `resolve()` del servicio de nombres se obtiene una referencia al objeto servidor, siempre asumiendo que dicho servidor esté en funcionamiento y sea posible acceder a él mediante el citado servicio de nombres. La referencia devuelta es genérica, por lo que es preciso realizar una conversión usando el método `narrow()` de la clase auxiliar `IConsultaHelper` generada previamente por `idltojava`.

Llegados a este punto ya tenemos la referencia a la interfaz `IConsulta`, sin importar el punto en el que realmente se está ejecutando el objeto servidor. Aparentemente el objeto se ejecuta de forma local, por lo que no tenemos mas que llamar al método `Consulta()` entregando los parámetros de la consulta y obteniendo el correspondiente resultado, que mostramos seguidamente en la salida estándar. Dados estos pasos el programa termina y se devuelve a la línea de comandos.

Al igual que ocurre con el servidor, la mayor parte del código del cliente se encuentra en el interior de un bloque `try` que nos permite interceptar cualquier excepción que pueda producirse. En este caso simplemente se da salida a toda la información disponible. En un cliente real sería más apropiado controlar excepciones concretas informando al usuario de la imposibilidad de acceder al servicio de nombres, la inexistencia del objeto que se solicita, etc.

Concluyendo

Como ha podido verse en este artículo, crear servidores y clientes CORBA no es una tarea compleja usando Java IDL, una de las novedades más interesantes de la plataforma Java 2. En principio, sobre todo si sólo es usuario del lenguaje Java, puede no apreciar ninguna ventaja de CORBA respecto a Java RMI, mecanismo este último que permite la misma funcionalidad.

Las ventajas de Java IDL, sin embargo, son muchas. Seguramente la más importante es la interoperabilidad con cualquier cliente o servidor CORBA, indistintamente del sistema en que esté funcionando y el lenguaje en que se haya desarrollado. Siempre que los objetos estén usando un ORB compatible con la especificación CORBA 2.0 y se apoyen en IIOP como protocolo de comunicación, no existirá problema alguno en ello.

Es posible, por ejemplo, crear un cliente ligero en Java para usar servicios CORBA creados en COBOL en sistemas *mainframe*. De esta forma dichos servicios estarían accesibles desde cualquier cliente de red a través de un navegador web, aplicaciones a la que están acostumbrados la mayoría de los usuarios. Es simplemente un ejemplo de cómo renovar y mejorar un sistema sin necesidad de descartar el código ya existente, algo que sí habría que hacer con Java RMI puesto que servidor y clientes deberían estar escritos en Java.