Capítulo 9

Programación Distribuida

Introducción

La evolución de los sistemas (se entiende, sistemas de software) fue exigiendo, conforme estos crecían en complejidad, en respuesta a nuevas necesidades, que las actividades que involucraran a todo el sistema fuesen distribuyéndose entre elementos de software y hardware separados, cada cual especializado en una actividad.

Existen 3 razones principales para que un sistema necesite distribuir sus actividades:

- Los datos utilizados por el sistema deben estar distribuidos.
- La computación con dichos datos debe estar distribuida.
- Los usuarios de la aplicación deben de estar distribuidos.

Por *distribuido* se entiende el hecho de que, las actividades de un sistema se realicen en dos o más nodos, cada nodo se comunica con el otro mediante una red (del tipo que sea necesario) y cada nodo está representado comúnmente por un computador. No entraremos en detalles sobre cómo estas tres razones fundamentales aparecieron, ni cuáles son sus ventajas y desventajas. Lo que sí nos interesa son las tecnologías con las que se pueden desarrollar estos sistemas y sus consideraciones respecto a lo que el desarrollo de estos sistemas implica.

Existen muchas tecnologías con las que se puede afrontar la Programación de Sistemas Distribuidos (PSD). Dichas tecnologías, en mayor o menor medida, permiten traspasar los límites del lenguaje de programación, del proceso, del sistema operativo y de la computadora individual. Dichas tecnologías se montan sobre otras preexistentes, en una arquitectura estratificada, donde con cada nuevo estrato le brinda un nivel de abstracción mayor al programador. La forma con la que se interactúa con un elemento de esta arquitectura se le conoce como protocolo de comunicación.

Una de las primeras tecnologías de gran aceptación (implementada en muchos lenguajes de programación) para la PSD fue los *sockets* (aparece por primera vez en Berkeley Unix en 1981). Los sockets son soportados por la gran mayoría de sistemas operativos:

- Los S.O. BSD Unix brindan sockets como parte del kernel.
- Los S.O. Windows brindan un API para sockets llamado WinSock.
- MS-DOS, Mac-OS, OS/2, etc., brindan librerías de sockets.
- Lenguajes POO como Java brindan clases que facilitan el trabajo con sockets.

Los sockets homogenizan el trabajo con diferentes protocolos de comunicación, a la vez que son muy eficientes. Esto simplificaba la PSD, independizándola (en cierta medida) del protocolo soportado por la plataforma subyacente (Hardware + S.O.). Sin embargo, los sockets sólo permiten trasmitir y recibir datos no estructurados (paquetes de bytes), los que deben crearse y enviarse (mediante un formato propietario) de un nodo a otro de una red. Al recibir estos datos, se deben de interpretar y en algunos casos, responder de forma similar. Adicionalmente, el programador debe lidiar con los problemas de pérdida de conexión, pérdida de paquetes de datos, problemas de inversión de los bits y sincronización del trabajo entre el que envía y el que recibe.

Un nivel algo mayor de abstracción se brindó con RPC (Remote Procedure Call, presentada por primera vez por Birrell y Nelson en 1985). RPC permite a un programa llamar/invocar/ejecutar un procedimiento/función/rutina en un sistema remoto (un nodo distinto, dentro de la misma red, del nodo donde se ejecuta el programa llamador). La ventaja de realizar un RPC con respecto a utilizar una tecnología de envío y recepción de datos, es la de simular la llamada a un procedimiento local, simplificando enormemente la lógica de programación. El siguiente diagrama muestra la secuencia de trabajo en una llamada a un procedimiento remoto.

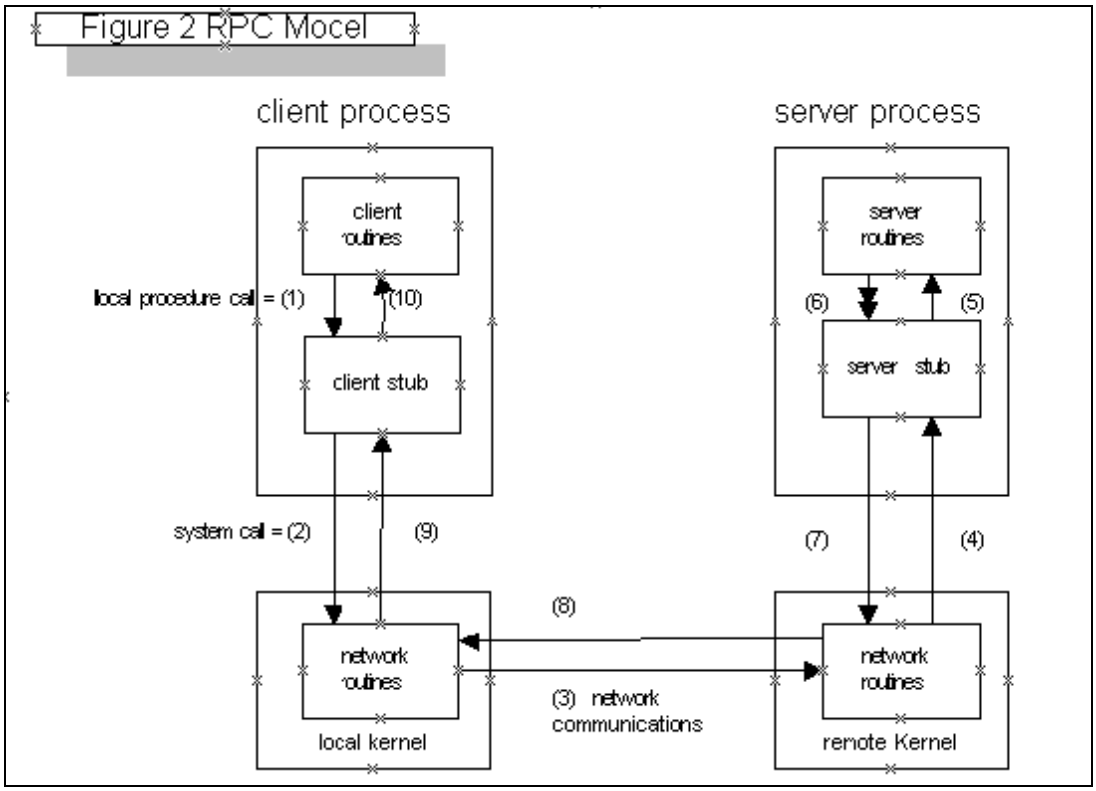

Figura 9.1. Modelo RPC

En conclusión, cuando se realiza una RPC, dicha llamada:

- Parece una llamada a un procedimiento local.
- Esconde todos los aspectos de la trasmisión de los argumentos y recepción de los resultados.
- Permite un modelo de programación más sencillo.

La RPC es actualmente la base de muchos sistemas distribuidos. Sin embargo, si bien una vez implementado su uso es relativamente sencillo, dicha implementación dista mucho de serlo. Existen adicionalmente muchas

limitaciones con respecto al tipo de datos que pueden trasmitirse, por lo que el uso de una RPC no es del todo transparente para el programador.

El siguiente nivel de abstracción lo ofrece el enfoque de los Sistemas de Objetos Distribuidos y las tecnologías que permiten su programación. Dado que en un sistema distribuido lo que se distribuye son datos y funcionalidad, los objetos se prestan como una forma muy clara y natural de modelar dicha distribución, y de compartirla entre los diferentes nodos de una red. Luego, el diseño de sistemas orientados a objetos se amolda con gran elegancia al diseño de sistemas distribuidos. COM, .NET Remoting, RMI y CORBA son ejemplos de tecnologías para la Programación de Sistemas de Objetos Distribuidos. Al igual que la RPC, los objetos distribuidos tienen muchas ventajas pero no aíslan del todo al programador de los problemas inherentes a la PSD, como veremos en la siguiente sección.

En las tecnologías descritas, es importante no perder de vista el hecho de que las primeras han servido como base para el desarrollo de las siguientes. Por ejemplo, en la implementación de la trasmisión de datos para un RPC se puede (y de hecho, se hace) utilizar internamente sockets. Igualmente, muchos de los elementos conceptuales del trabajo con objetos distribuidos se basan en RPC, e internamente las llamadas a los métodos de un objeto distribuido pueden programarse como una RPC (y ciertamente, COM y CORBA lo hacen).

Programación de Sistemas de Objetos Distribuidos

En la *Programación de Sistemas de Objetos Distribuidos* (PSOD), el esquema de trabajo para llamar a un método de un objeto distribuido es similar al de una RPC. Los beneficios adicionales que se tienen con la PSOD frente a RPC equivalen a los existentes entre la POO y la programación estructurada.

En la PSOD, un objeto distribuido puede o bien:

- Ser accedido de manera distribuida.
- Ser transmitido.

En el primer caso, el objeto distribuido es creado en un nodo de la red y sus métodos son llamados remotamente desde otros nodos, probablemente desde otros objetos. Los datos del objeto existen y sus métodos se ejecutan en el nodo donde el objeto es creado. El resto de objetos en otros nodos de la red acceden a él mediante una referencia distribuida, como veremos más adelante. En el segundo caso, una copia del objeto es trasmitida, por lo que los datos y métodos de esta copia existen en otro nodo de la red. Esta diferencia es similar al *paso por referencia* y *paso por valor* utilizado en las llamadas a métodos. De forma similar a Java, sólo un conjunto de tipos de datos (nativos a la tecnología de PSOD utilizada) puede ser transmitido (pasado por valor), mientras que el resto de objetos son sólo accedidos (pasados por referencia). Al igual que en la programación con objetos co-locales (no distribuidos), estas diferencias deben tomarse en cuenta.

Para los objetos distribuidos mediante una referencia remota, el control de su ciclo de vida difiere de los objetos co-locales. Un objeto distribuido puede:

- Preexistir (en un programa en ejecución) en un nodo y ser accedido desde otro objeto en otro nodo.
- Ser creado en un nodo a solicitud de un objeto en otro nodo.
- Permanecer vivo hasta que el último objeto que lo refería descarte dicha referencia. Esto puede ser realizado por el programador de manera explícita o por algún mecanismo automático.

- Permanecer vivo hasta finalizar el programa donde fue creado.

Existen diferentes estrategias para controlar el ciclo de vida de un objeto distribuido. Sin embargo, lo importante es notar el hecho de que la creación y destrucción de dicho objeto ya no depende necesariamente del programa que lo accede remotamente. Por ello, se definen dos nuevos conceptos en PSOD:

- **Activación**. Un objeto distribuido se activa cuando está listo para ser accedido de manera remota. Dicha activación puede realizarse a solicitud de un objeto en otro nodo (activación desde el cliente o activación a demanda), o previamente (activación desde el servidor). El objeto activado pudo haberse creado como respuesta a una solicitud de activación o antes de ésta. Puede crearse un objeto por cada solicitud de activación o bien utilizarse un único objeto para atender todas las solicitudes.

- **Desactivación**. Un objeto distribuido se desactiva cuando deja de estar listo para ser accedido remotamente. Dicha desactivación puede realizarse cuando todos los objetos que lo referenciaban remotamente lo dejan de hacer, o posteriormente. El objeto desactivado puede destruirse como consecuencia de la desactivación o puede mantenerse con vida para posteriores activaciones.

La PSOD debe tomar en cuenta las diferencias que existen con respecto a la programación con objetos no-distribuidos o *co-Locales*. El siguiente cuadro muestra algunas de las diferencias más saltantes:

Tabla 9.1. Diferencias entre programación con objetos distribuidos o con co-locales

	Objetos Co-Locales	**Objetos Distribuidos**
Comunicación	Rápida	Lenta
Fallas	Los objetos fallan juntos	Los objetos fallan separadamente. El enlace de red puede romperse.
Acceso concurrente	Sólo con múltiples hilos	Sí
Seguridad	Sí	No

Es importante no perder de vista que los objetos distribuidos existen en procesos y lugares distintos, por lo que el acceso a sus datos es inherentemente compartido entre los diferentes hilos y la sincronización para su acceso, dado que son hilos de distintos procesos, es más complicada. Éstas y otras consideraciones necesariamente deben de tomarse en cuenta en la PSOD.

Cabe anotar que algunos autores diferencian entre objetos distribuidos locales y objetos distribuidos remotos. Los primeros son objetos existentes en distintos procesos en un mismo nodo, mientras que los segundos existen en procesos en distintos nodos de una red.

Estudiaremos dos de las tecnologías actuales de PSOD: CORBA y .NET Remoting.

CORBA

Las siguientes secciones describirán la arquitectura de CORBA y su implementación desde Java.

La Arquitectura de CORBA

CORBA (Common Object Request Broker Arquitecture, lo que podría traducirse como Arquitectura de Agentes Intermediarios de Despacho de Solicitudes de Objetos Comunes) es un estándar de objetos distribuidos remotos, más que un modelo de componentes. Este estándar permite la interoperación de una colección de objetos heterogéneos.

CORBA es desarrollado por la OMG (Object Management Group), organización que comprende a más de 700 empresas de desarrolladores de software y hardware.

CORBA define un estándar de objetos distribuidos, basando la colaboración entre estos, en la resolución de solicitudes de servicio que dichos objetos brindan. Todo en CORBA gira en torno a esta solicitud-despacho de servicios brindados por objetos remotos.

CORBA forma parte fundamental de la arquitectura OMA (Object Management Arquitecture) desarrollada por la OMG. OMA es una visión de alto nivel de un entorno distribuido completo. Consiste en cuatro componentes que pueden ser, a groso modo, divididos en dos partes:

- Componentes orientados a los sistemas (Object Request Brokers y Object Services).
- Componentes orientados a las aplicaciones (Application Objects y Common Facilities).

La figura 9.2 muestra la forma en que se interrelacionan estas partes.

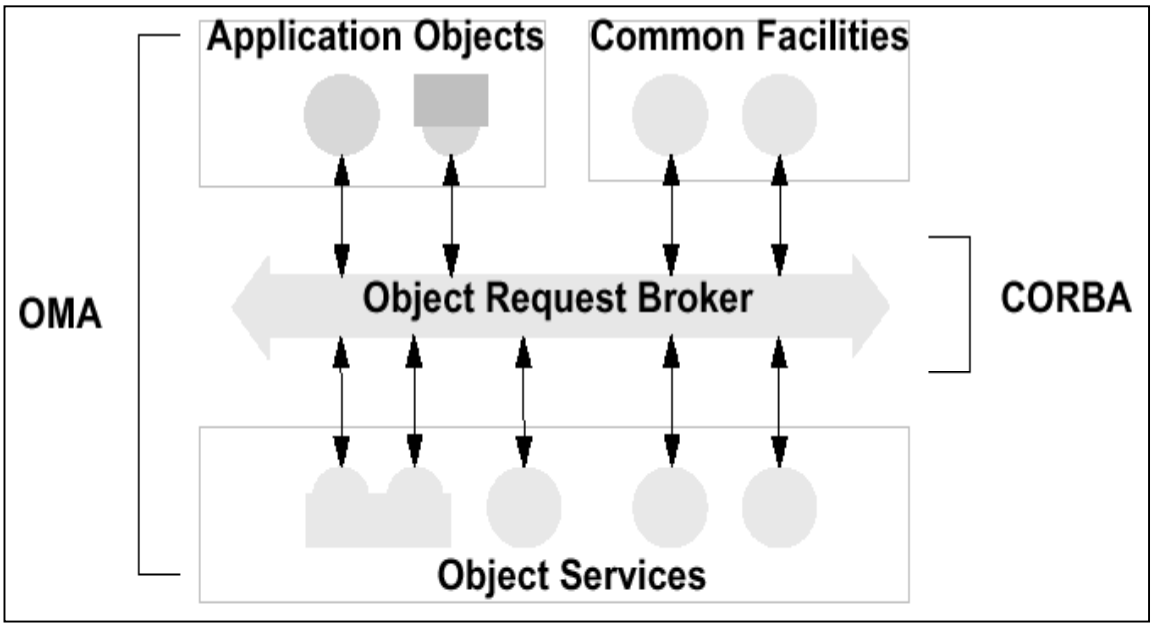

Figura 9.2. CORBA

Los cuatro componentes se comunican entre sí mediante *CORBA*, el cual está formado básicamente por elementos llamados *Object Request Brokers*, o *ORB's*, los que se encargan de administrar toda la comunicación entre estos componentes. Los *ORB's* permiten interactuar con objetos heterogéneos, en entornos distintos, en plataformas distintas donde residen e independientemente de las técnicas utilizadas para desarrollarlos. El diseño de *CORBA* se basa en el modelo de objetos de la OMG.

Este modelo de objetos establece básicamente una forma común en que los objetos distribuidos, manejados por CORBA, exponen su funcionalidad de manera independiente a su implementación, es decir, una interfaz.

Las interfaces de CORBA se especifican utilizando un lenguaje IDL, el que es un lenguaje neutro de definición de interfaces.

La figura 9.3 muestra como funciona el mecanismo básico de CORBA de solicitud de un servicio.

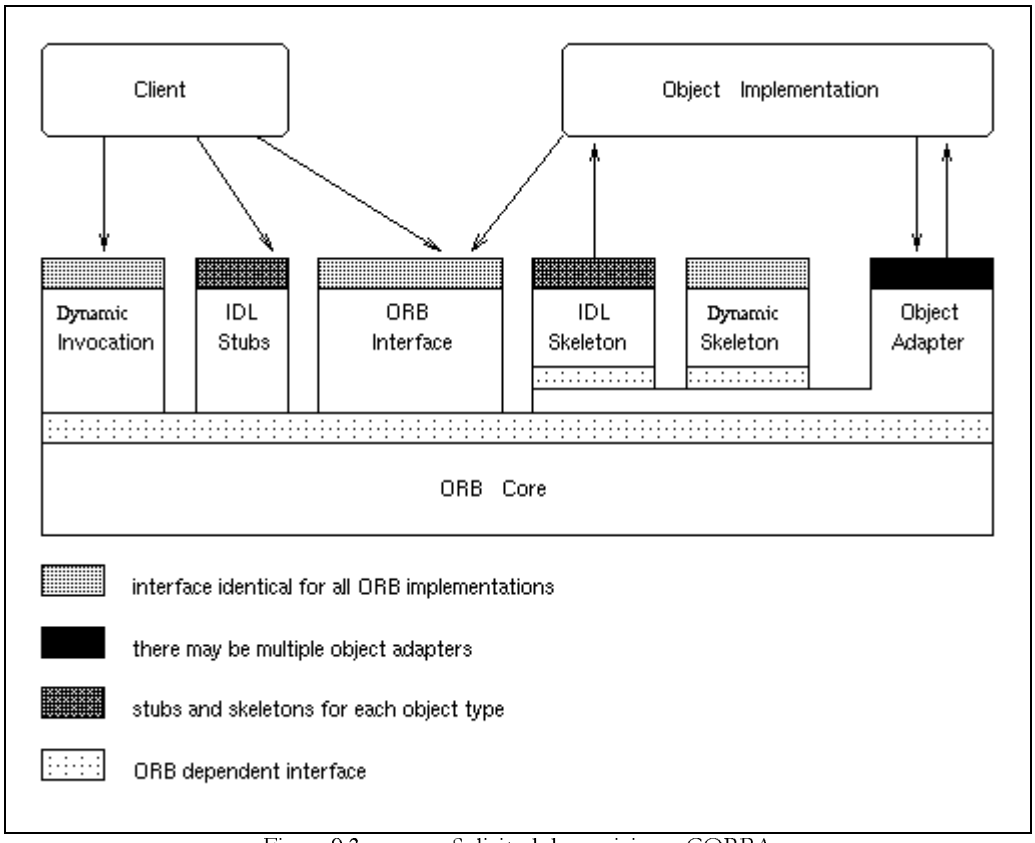

Figura 9.3. Solicitud de servicio en CORBA

Como se aprecia en el diagrama, los IDL permiten crear un elemento *Stub* y *Skeleton* que hacen el trabajo de la comunicación en sí, entre el cliente y el servidor, donde reside el objeto que implementa dicho servicio.

El *Stub* y el *Skeleton* realizan internamente la comunicación entre las partes y el *ORB* instalado. Los *ORB's* se comunican luego entre sí mediante un protocolo especialmente definido para esto, el IIOP (*Internet Inter-ORB Protocol*). El IIOP se monta sobre *TCP/IP*, y define básicamente un protocolo de comunicación entre ORB's. El *Stub* y el *Skeleton* implementan el *marshaling* y *unmarshaling* de los datos pasados, y establecen una forma de mapeo entre la implementación original del cliente y el objeto en el servidor, de forma que pueda realizarse la comunicación de una forma transparente.

El componente central de CORBA es el *ORB*. Un *ORB* implementa toda la infraestructura de comunicación necesaria para identificar un objeto, localizarlo y realizar las solicitudes a sus servicios. Luego, tanto en el nodo del cliente como en el del servidor deberá existir un *ORB* instalado.

Los *ORB's* pueden ser desarrollados por distintos proveedores, para distintas plataformas, siempre que respeten los estándares exigidos por CORBA. De hecho, existe actualmente en el mercado multitud de *ORB's* que implementan en mayor o menor grado, todos los servicios delineados por el estándar CORBA.

Los Servicios de CORBA

El estándar CORBA establece un conjunto de servicios para soportar la integración e interoperación de objetos distribuidos.

Tabla 9.2. Listado de servicios CORBA

Servicio	Descripción
Object Life Cicle	Define cómo los objetos en CORBA son creados, removidos, movidos y copiados.
Naming	Define la forma en que los objetos en CORBA pueden tener nombres simbólicos para ser reconocidos.
Events	Permite la comunicación entre objetos distribuidos.
Relationships	Permite relacionar arbitrariamente diversos objetos distribuidos.
Externalization	Coordina la transformación de un objeto CORBA hacia un medio externo, y al revés.
Transactions	Coordina los accesos atómicos a los objetos.
Concurrency Control	Provee un servicio de bloqueo de objetos para asegurar un acceso serializable.
Property	Permite crear asociaciones nombre – pares de valores con objetos CORBA.
Trader	Permite buscar objetos en base a los servicios que brinda.
Query	Permite realizar consultas a objetos.

Implementación desde Java

Desarrollaremos un sistema en Java que implemente un objeto CORBA, el cual implementará un método que permita retornar un valor. La figura 9.4 muestra la forma en que se utilizará el objeto.

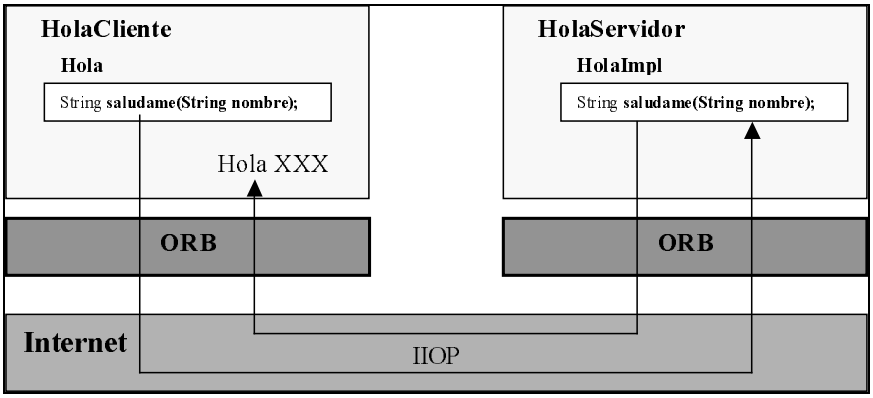

Figura 9.4. Esquema de implementación

La secuencia de pasos a seguir y que corresponde a la secuencia típica de desarrollo de un objeto CORBA es:

- Definición de la interfaz: Archivo IDL.
- Compilación de la interfaz.

- Implementación del servidor.
- Implementación del cliente.
- Ejecución de la aplicación.

Definición de la interfaz

El siguiente código corresponde al archivo *Hola.IDL* que utilizaremos para nuestra aplicación.

```
module HolaApp {
    interface Hola {
        string saludame(in string nombre);
    };
};
```

La palabra *module* permite agrupar un conjunto de interfaces, de la misma forma como la palabra reservada *package* de Java, permite agrupar un conjunto de clases.

Los tipos de datos de los métodos de una interfaz son propios de CORBA, y serán mapeados a sus correspondientes en Java a la hora de compilar el archivo con un compilador especialmente diseñado para Java.

Aunque este ejemplo no lo muestre, el IDL de CORBA permite la herencia múltiple entre interfaces. En contraparte, un objeto de CORBA sólo puede implementar una interfaz.

Compilación de la interfaz:

Para compilar la interfaz utilizamos el programa utilitario idlj.exe, incluido en el JDK de Java. Para versiones anteriores del JDK, el programa equivalente es idltojava.exe. Otros entornos de desarrollo de Java pueden proveer sus propios programas para este fin.

Compilamos el archivo IDL pasándolo como parámetro al programa idlj.exe ejecutado desde una línea de comandos, por ejemplo:

```
C:\>idlj.exe -fall hola.idl
```

Esto provocará la generación de los siguientes archivos:

- **Hola.java**: Contiene la representación de la interfaz definida en el archivo IDL, pero en lenguaje Java.
- **HolaOperations.java**: Contiene la definición de las operaciones de la interfaz Hola.
- **HolaPOA.java**: Representa el elemento *Skeleton* el cual provee la funcionalidad básica para el servidor.
- **_HolaStub.java**: Representa el elemento *Stub* y provee la funcionalidad básica para el cliente.
- **HolaHelper.java**: Provee funcionalidad adicional para el uso, por parte del cliente, del objeto CORBA.
- **HolaHolder.java**: Contiene un miembro que es una instancia publica de tipo *Hola*. Esta clase es la encargada de realizar las operaciones de entrada y salida de argumentos.

La figura 9.5 muestra la relación de estas clases con las clases de soporte de CORBA para Java y las clases que el programador debe desarrollar *manualmente* una vez generadas las clases arriba descritas. Los nombres para estas últimas clases son referenciales, el programador puede darles cualquier nombre que desee.

Examinemos el archivo *Hola.java* y *HolaOperations.java* para ver cómo es que se mapea una interfaz CORBA al lenguaje Java.

```
/* Archivo Hola.java */
package HolaApp;
public interface Hola extends HolaOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity {
}
/* Archivo HolaOperations.java */
package HolaApp;
public interface HolaOperations {
  String saludame (String nombre);
}
```

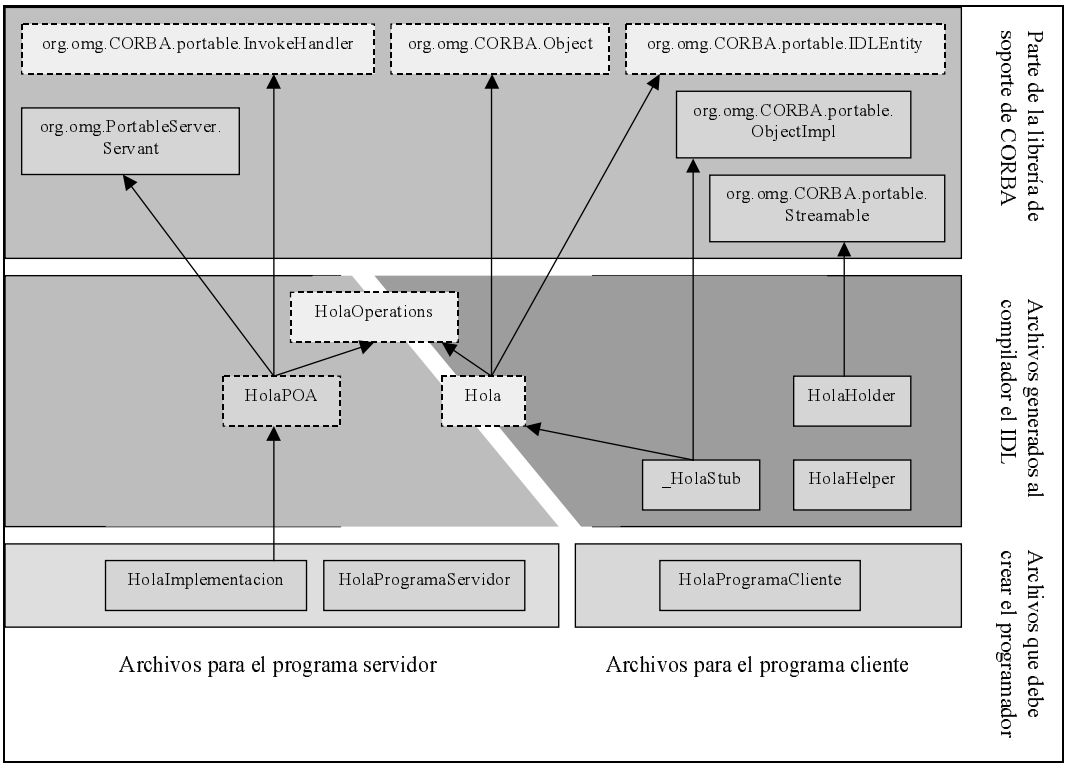

Figura 9.5. Archivos generados

Como se puede apreciar, la conversión es bastante directa. Mediante la interfaz *Hola* el cliente se comunicará con un objeto *_HolaStub*, el cual a su vez se comunicará (a través del *ORB*) con la implementación real, un objeto remoto que implemente la clase abstracta *HolaPOA*. En la jerga de CORBA, al objeto distribuido real se le llama *sirviente*, para poder diferenciarlo del programa en donde es creado, al que se le llama *servidor*.

Note que la clase se define como perteneciente al paquete *HolaApp*. Ésta es la forma en que se traduce a Java la palabra *module* de un IDL de CORBA. Cuando se compila el archivo IDL, se crea la carpeta *HolaApp*, dentro de la cual se crean todos los archivos explicados. Luego, tanto en el programa servidor como en el cliente se utilizará la sentencia:

```
import HolaApp.*;
```

De forma que se acceda a las clases de este paquete.

Implementación del Servidor

El siguiente es el código corresponde al archivo *HolaServidor.java* que hará las veces de nuestro programa servidor.

```
// Todas las aplicaciones de CORBA requieren de las clases en este paquete
import org.omg.CORBA.*;

// Clases relacionadas al servicio POA
import org.omg.PortableServer.*;

// Clases relacionadas al servicio POA
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

// Este paquete que contiene la implementación del Skeleton: HolaPOA.class
import HolaApp.*;

class HolaSirviente extends HolaPOA {
    public String saludame(String nombre) {
        return "Hola " + nombre;
    }
}

class HolaServidor {
    public static void main(String args[]) {
        System.out.println("Iniciando el programa servidor 'HolaServidor' ...");
        try{
            // Obtengo una referencia a un objeto ORB
            ORB orb = ORB.init(args, null);

            // Obtengo una referencia al servicio POA
            org.omg.CORBA.Object obj = orb.resolve_initial_references("RootPOA");
            POA servPOA = POAHelper.narrow(obj);

            // Activo el administrador del servicio POA
            POAManager admPOA = servPOA.the_POAManager();
            admPOA.activate();

            // Obtengo una referencia al servicio de nombres NamingContext
            obj = orb.resolve_initial_references("NameService");
            NamingContextExt servNomb = NamingContextExtHelper.narrow(obj);

            // Creo el objeto distribuido y lo registro en los servicios
            HolaSirviente hola = new HolaSirviente();
            byte[] holaID = servPOA.activate_object(hola);
            NameComponent[] nc = servNomb.to_name("Hola");
            servNomb.rebind(nc, servPOA.servant_to_reference(hola));

            // Evito que el programa finalice, de forma que el objeto distribuido
            // creado permanezca vivo para poder dar servicio a clientes
            System.out.println("Inicializacion completa. Esperando clientes...");
            orb.run();

            // desregistro el objeto distribuido
            servPOA.deactivate_object(holaID);
        }
        catch(Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
        System.out.println("Programa finalizado");
    }
}
```

Note que la clase *HolaSirviente* se basa en la clase *HolaPOA* para implementar el objeto distribuido que será accedido remotamente.

El programa servidor realiza las siguientes acciones:

1. Obtiene una referencia a un objeto de la clase ORB. Mediante esta referencia se accederá a los servicios de CORBA.

2. Obtiene una referencia a un objeto de la clase POA. Mediante esta referencia se accede a los servicios POA, mediante la cual, se puede manipular las políticas de manejo de los objetos distribuidos registrados.

3. Obtiene una referencia a un objeto de la clase POA Mediante esta referencia se accede al servicio de nombres.

4. Creo el objeto sirviente y lo registro en los servicios.

5. Ejecuto el método *run* de *ORB*, el cual deja al programa en estado de espera, de forma que éste no termine y de oportunidad a que los programas clientes puedan acceder al objeto distribuido registrado.

Note también la forma en que se crea y se accede a la funcionalidad de un objeto CORBA. Todas las referencias iniciales a los objetos CORBA se obtienen en base a la interfaz base *org.omg.CORBA.Object*. Una vez que se obtiene una referencia de dicho tipo, se puede acceder a la interfaz correspondiente de la implementación real del objeto (la interfaz *Hola*), mediante el método *narrow* de la clase *Helper* correspondiente. Éste es uno de los usos más importantes de esta clase. ¿Por qué no realizar una operación *cast* directa? Dado que las operaciones *cast* de un tipo de referencia a otra son controladas por el lenguaje y que al objeto al que referenciamos puede estar implementado en cualquier otro lenguaje, no es seguro realizar este tipo de operaciones sobre referencias a objetos CORBA. La clase *helper* del objeto CORBA utilizado nos ayuda a realizar una verificación, en tiempo de ejecución, de si una referencia a un objeto CORBA implementa o no la interfaz correspondiente a la clase *helper* utilizada.

Implementación del Cliente

El siguiente es el código corresponde al archivo *HolaCliente.java* que hará las veces de nuestro cliente.

```
// Todas las aplicaciones de CORBA requieren de las clases en este paquete
import org.omg.CORBA.*;

// Clases relacionadas al servicio POA
import org.omg.PortableServer.*;

// Clases relacionadas al servicio POA
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

// Este paquete que contiene la implementación del Stub: HolaPOA.class
import HolaApp.*;


class HolaCliente {
    public static void main(String args[]) {
        try{
            // Obtengo una referencia a un objeto ORB
            ORB orb = ORB.init(args, null);

            // Obtengo una referencia al servicio de nombres NamingContext
            org.omg.CORBA.Object obj =
orb.resolve_initial_references("NameService");
            NamingContextExt servNomb = NamingContextExtHelper.narrow(obj);
```

```
            // Obtengo el objeto registrado
            NameComponent[] nc = servNomb.to_name("Hola");
            Hola hola = HolaHelper.narrow(servNomb.resolve(nc));

            // Utilizo la referencia
            String respuesta = hola.saludame("Otto");
            System.out.println("respuesta = " + respuesta);
        }
        catch(Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}
```

Note que la forma en que se inicializa el trabajo es similar al servidor, se inicializa una referencia al ORB instalado para luego acceder al servicio de nombres y con él se obtiene una referencia a un objeto remoto registrado con un nombre en particular. Nuevamente, se requiere de una operación *cast* mediante la clase *Helper* correspondiente. Una vez que se obtiene la referencia deseada, se puede llamar a sus métodos como si correspondiera a un objeto co-local.

Ejecución de la Aplicación

Una vez compilados los programas cliente y servidor, se procede a la ejecución. Para esto levantamos primero el servicio de nombres con la instrucción:

```
tnameserv -ORBInitialPort nameserverport
```

El parámetro *nameserverport* representa el puerto por el que se realizará las comunicaciones. Si se está trabajando en dos máquinas distintas, se deberá hacer esto en las dos máquinas.

Luego de levantado el servicio de nombres, se corre el programa servidor con la siguiente instrucción:

```
java HolaServidor -ORBInitialHost nameserverhost -ORBInitialPort nameserverport
```

Los parámetros representan el nombre del equipo y el puerto donde el servicio de nombre se encuentra ejecutándose. Si el servidor está en la misma máquina que el servicio de nombres, el parámetro *nameserverhost* no será necesario.

Luego se corre el programa cliente:

```
java HolaCliente -ORBInitialHost nameserverhost -ORBInitialPort nameserverport
```

Los parámetros tienen el mismo significado que en el caso anterior.

El nombre del *host* puede ser especificado de cualquiera de las formas en que la red actual permita la identificación de los equipos en ella. Una forma común es utilizar la dirección IP. El número de puerto comúnmente utilizado es 1050, aunque se recomienda utilizar cualquier número sobre 1024, dado que los números menores a éste son por lo general, utilizados por los servicios del sistema operativo.

Diferencias entre versiones

El estándar de CORBA ha ido evolucionando y con ello, sus implementaciones en los diferentes lenguajes de programación que lo soportan. En Java, esta diferencia es notoria entre las versiones del JDK 1.4 y las anteriores. En las versiones anteriores al JDK 1.4, se manejaba la comunicación entre la clase que implementaba el *Skeleton* del objeto sirviente y la interfaz del ORB en ejecución mediante un objeto adaptador, llamado BOA (Basic

Object Adapter). El estándar de BOA no fue completo, por lo que hubo diferencias entre sus implementaciones para ORB's de distintos proveedores. Debido a esto, se desarrollo un nuevo estándar, POA (Portable Object Adapter). La versión del JDK 1.4 maneja el nuevo estándar POA.

Otros Recursos

Para más información acerca de CORBA, la siguiente lista de direcciones de Internet puede ser de utilidad.

Curso de introducción a CORBA:

`http://developer.java.sun.com/developer/onlineTraining/corba/`

Temas relativos a CORBA, como el uso del compilador IDL:

`http://java.sun.com/j2ee/corba/`

Ejemplo completo de desarrollo de un objeto CORBA en Java, correspondiente a las versiones previas al JDK 1.4:

`http://java.sun.com/docs/books/tutorial/idl/`

Ejemplo completo de desarrollo de un objeto CORBA en Java, correspondiente a la versión del JDK 1.4:

`http://developer.java.sun.com/developer/technicalArticles/releases/corba/`

.NET Remoting

La respuesta de Microsoft a las necesidades de PSOD fue COM. A diferencia de CORBA, en donde el enfoque está orientado a los objetos distribuidos remotos, COM se orienta a los componentes.

La noción de componente es más general que la de objeto. Un objeto encapsula datos y su comportamiento relacionado, mientras que un componente encapsula cualquier abstracción de software útil. Debido a que no todas las abstracciones de software son necesariamente objetos, se pierde muchas posibilidades de reutilización de software por enfocarse demasiado en los objetos. En la programación de sistemas orientados a componentes y basados en componentes, un programa se visualiza como una composición de componentes configurables y reutilizables, de manera que se espera satisfacer la evolución de requerimientos mediante el desenchufado, reconfiguración y reenchufado de los componentes afectados por los cambios.

En particular, esta visión de un sistema distribuido fue enfocada a lograr componentes visuales reutilizables, los que pudieran ser desarrollados en cualquiera de los lenguajes para los que Microsoft brinda soporte. Estos componentes, inicialmente llamados componentes OLE y luego ActiveX, pueden ejecutarse en un mismo proceso o en procesos separados, pueden haberse desarrollado en distintos lenguajes (principalmente Visual Basic y C++ con MFC), pueden estar desarrollados para procesadores distintos (un componente para un procesador de 16-bits puede interactuar con uno para 32-bits). Con DCOM y luego COM+, los componentes podían estar en procesos en nodos distintos de una red. En esta evolución, .NET Remoting es el escalón actual, con el que se busca superar muchas de las limitaciones de sus predecesores.

Es importante recalcar que .NET Remoting, como sus predecesores, tienen un fuerte enfoque a la *Programación de Sistemas Basados en Componentes*, más que PSD. Muchos de los servicios para PSD que CORBA ofrece ya maduros, o no existen o aún están en desarrollo en .NET Remoting (por ejemplo, CORBA es inherentemente multiplataforma, mientras que .NET Remoting esta amarrado al sistemas operativo Windows, incluso sólo a partir de ciertas versiones). Como contraparte, el soporte para componentes de CORBA es aún pobre frente al

nivel alcanzado en .NET Remoting (por ejemplo, un objeto de CORBA está aún muy lejos de brindar las facilidades con las que, ya hace años, un control ActiveX puede simplemente arrastrarse y soltarse dentro de una ventana, configurar sus propiedades y completar una interfaz gráfica en muy poco tiempo).

La Arquitectura de .NET Remoting

NET Remoting abarca comunicación entre objetos de distintos dominios (un concepto similar al de hilo) del mismo proceso, de diferentes procesos o de diferentes computadores conectados en red.

Los objetos se distribuyen mediante ensamblajes. Los ensamblajes remotos pueden ser configurados para trabajar localmente en el dominio del mismo proceso, o como parte de otro proceso. Si el ensamblaje es parte de otro proceso, el cliente recibe un objeto proxy (similar al Stub de CORBA) en lugar de una referencia al objeto real. El proxy envía los mensajes a través de un canal.

La siguiente lista enumera los elementos que intervienen en la distribución de un objeto:

- **El objeto remoto**. Este hereda de MarshallByRefObjet, la clase base de todos los objetos distribuidos.
- **Un canal**. El concepto de canal involucra un protocolo de comunicación. Actualmente la librería de .NET brinda clases para dos tipos de canales: TCP y HTTP. También se puede desarrollar nuevos tipos.
- **Los mensajes**. Estos son enviados a través de los canales. Estos mensajes contienen la información acerca del objeto remoto, el método que es invocado de éste, así como los argumentos que se le pasan.
- **El formateador**. Define cómo los mensajes son transferidos dentro del canal, o extraídos de él. La librería de .NET ofrece dos formateadores, para SOAP y para datos binarios. También se puede desarrollar nuevos tipos.
- **El proveedor de formateadores**. Permite asociar un formateador con un canal. Al crear un canal, se puede seleccionar que proveedor usar.
- **El objeto proxy**. El objeto cliente conversa con este objeto en lugar del objeto real. Existen dos tipos de objetos proxy: Transparente y real.
- **El Sumidero de Mensajes**. Es un objeto interceptador de mensajes, utilizados para actividades como filtrado de datos. Estos interceptores están tanto en el lado del servidor como en el del cliente. Un sumidero está asociado a un canal. El proxy real usa un sumidero para pasar los mensajes dentro del canal, de forma que este sumidero puede hacer alguna interceptación antes de pasarlo al canal.
- **El activador**. El cliente puede utilizar un activador para crear un objeto remoto en el servidor o para obtener un proxy de un objeto activado en el servidor.
- **La clase RemotingConfiguration**. Utilizada para configurar servidores y cliente remotos. Esta clase puede ser utilizada para leer archivos de configuración o para configurar dinámicamente objetos remotos.
- **La clase ChannelServices**. Utilizada para registrar canales y para enviar mensajes dentro de éstos.

Los siguientes diagramas muestran cómo estos elementos se comunican para trabajar con objetos distribuidos.

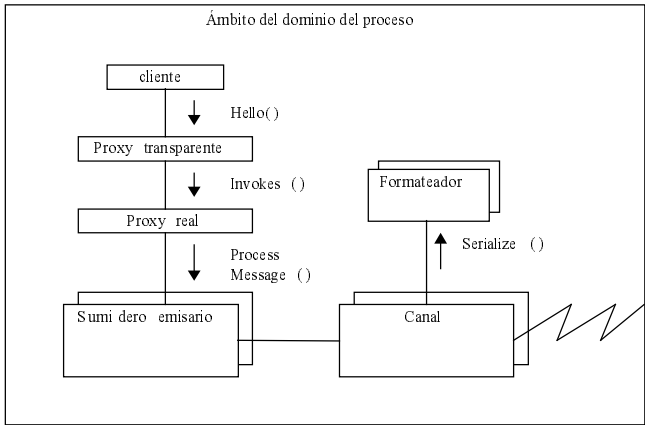

Figura 9.6. Lado del cliente.

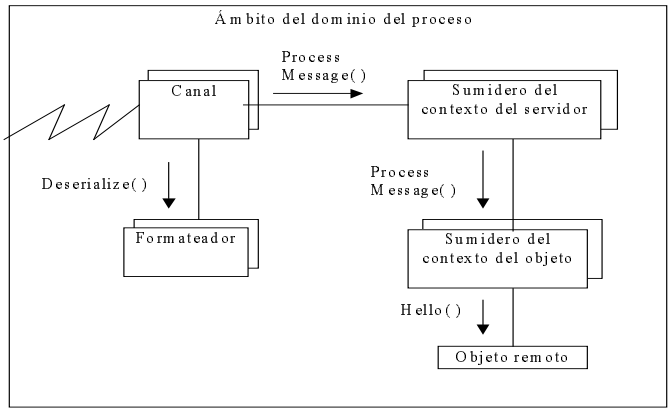

Figura 9.7. Lado del servidor.

Como puede observarse, los sumideros y los formateadores trabajan conjuntamente con el canal a ambos lados, el programa cliente y el servidor.

Implementación del Objeto Distribuido

El siguiente código muestra la implementación del objeto distribuido.

```
using System;

public class Hola : System.MarshalByRefObject {
    public Hola() {
        Console.WriteLine("Hola.Constructor llamado");
    }
    ~Hola() {
        Console.WriteLine("Hola.Destructor llamado");
    }
    public string Saludame(string nombre) {
        return "Hola " + nombre;
    }
}
```

El archivo con la implementación del objeto distribuido debe compilarse como una librería. La línea de comando para compilar este archivo es:

```
csc /target:library HolaApp.cs
```

Este comando crea el archivo HolaApp.dll, el cual deberá distribuirse tanto al programa servidor como al cliente. La manera más sencilla de hacer esto es manteniendo una copia de este archivo en la misma carpeta del programa servidor y cliente.

El siguiente código muestra la implementación del programa servidor.

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

class HolaServidor {
    public static void Main(string[] args) {
        TcpServerChannel channel = new TcpServerChannel(8086);
        ChannelServices.RegisterChannel(channel);
        RemotingConfiguration.RegisterWellKnownServiceType(
            typeof(Hola), "Hola", WellKnownObjectMode.SingleCall);
        Console.WriteLine("Presione ENTER para finalizar");
        Console.ReadLine();
    }
}
```

El programa servidor registra un canal de escucha (protocolo TCP, conectado al puerto 8086), el tipo de objeto que se distribuye (mediante la sentencia *typeof*) así como la forma en que ésta operará (*SingleCall*, que indica que se creará un nuevo objeto *Hola* por cada llamada a uno de sus métodos). A partir de este momento, la librería de .NET remoting se encargará de crear y manejar los objetos distribuidos registrados.

La línea de comando para compilar este archivo es:

```
csc /reference:HolaApp.dll HolaServidor.cs
```

Se creará el archivo *HolaServidor.exe*, el programa servidor. El siguiente código muestra la implementación del programa cliente.

```
using System;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;

class MainClass {
    public static void Main(string[] args) {
        TcpClientChannel channel = new TcpClientChannel();
        ChannelServices.RegisterChannel(channel);

        string servidor = args[0];
        string puerto = args[1];
        string uri = "tcp://" + servidor + ":" + puerto + "/Hola";
        Hola obj = (Hola)Activator.GetObject(typeof(Hola),     uri);
        if(obj == null) {
            Console.WriteLine("No se pudo localizar al objeto");
            return;
        }

        string resp = obj.Saludame("Otto");
        Console.WriteLine("Respuesta = " + resp);
    }
}
```

El programa registra el canal que utilizará (para el protocolo TCP, el puerto del cliente no es necesario fijarlo, por lo que no se pasa ningún valor). Para obtener una referencia al objeto remoto, se utiliza el método *GetObject* de la clase *Activator*, al que se le pasa el tipo de objeto que se busca y su URI (Universal Resource Identifier).

La línea de comando para compilar este archivo es:

```
csc /reference:HolaApp.dll HolaCliente.cs
```

Se creará el archivo *HolaCliente.exe*, el programa cliente.

Para ejecutar el programa, se realizan los siguientes pasos:

1. Se corre el programa servidor.
2. Se corre el programa cliente.