# Cómo desarrollar aplicaciones en entornos distribuidos con CORBA

**© Francisco Charte Ojeda**

## Sumario

Primera parte de una serie de artículos sobre CORBA, comenzando con una introducción a qué es CORBA y cómo funciona.

## Introducción

Para el grueso de los programadores, gran parte de los cuales se mueven en torno a Windows, las palabras CORBA y OMG suenan a algo nuevo, debido al reciente surgir de los entornos distribuidos. Lo cierto, sin embargo, es que CORBA existe casi desde antes que Windows y, por supuesto, desde antes que Internet fuese algo conocido. Ha sido no obstante la Red, concretamente el uso de sus protocolos en redes empresariales, lo que ha provocado el crecimiento de los entornos distribuidos y heterogéneos e, indirectamente, el uso de CORBA.

En estas nuevas redes conviven diferentes sistemas operativos, desde el omnipresente Windows hasta los sistemas *mainframes* pasando por Mac OS, Linux e innumerables versiones de UNIX. Asimismo los desarrollos sobre esos sistemas están hechos usando diferentes lenguajes y herramientas, lo que complica aún más una posible colaboración entre ellas. El único hilo conductor que comunica aplicaciones y sistemas tan diferentes es un protocolo de transporte, conocido como IP, y algunos protocolos de conversación como TCP y UDP. A estos protocolos se les llama genéricamente TCP/IP.

La presencia de ese hilo conductor, los protocolos TCP/IP, y la existencia de los elementos de CORBA en una treintena de sistemas diferentes es lo que hace posible crear aplicaciones distribuidas que se comunican entre sí saltando los muros que representan los sistemas y lenguajes. No importa, por lo tanto, la herramienta de desarrollo usada o el sistema operativo sobre el que se trabaje. CORBA es fácil de usar desde C++ y Java, los dos lenguajes a los que más se asocia, pero también desde Borland Delphi, COBOL o Visual Basic, por mencionar sólo los más conocidos.

CORBA es una arquitectura íntimamente ligada a las tecnologías de orientación a objetos, si bien es cierto que no es preciso un lenguaje orientado a objetos para aprovechar las posibilidades de CORBA. Las aplicaciones CORBA no son construcciones monolíticas, sino más bien elementos discretos, los podemos llamar componentes, que se ejecutan distribuidamente sobre múltiples sistemas y se comunican entre sí mediante una infraestructura de red.

Si está interesado en conocer cuáles son los elementos fundamentales de esta arquitectura y quiere saber cómo se desarrollan aplicaciones CORBA siga leyendo. Este artículo le servirá para adquirir una visión global sobre técnicas que están llamadas a tener cada vez una mayor presencia en el ámbito de la informática empresarial.

## Modelos de aplicaciones

Los departamentos de informática y empresas de desarrollo tienen que decidir, a la hora de planificar la creación de una nueva aplicación, cuál de los modelos posibles quieren utilizar. Actualmente los modelos existentes son básicamente tres: monolítico o tradicional, cliente/servidor y de múltiples capas o

niveles. Cada uno de estos modelos tiene, como cualquier otro elemento, sus ventajas y desventajas, que será preciso conocer para poder decidir.

Cualquier aplicación actual cuenta generalmente con tres partes bien diferenciadas: una interfaz de usuario, unas reglas de negocio y una gestión de datos. La interfaz de usuario es el elemento con el que interacciona el usuario de la aplicación, ejecutando acciones, introduciendo u obteniendo información. Las reglas de negocio son las que procesan la información para generar los resultados que se persiguen, siendo el elemento fundamental que diferencia a unas aplicaciones de otras. Por último está la gestión de datos, que se ocupa del almacenamiento y recuperación de la información en medios persistentes, habitualmente discos.

En una aplicación monolítica, representada en la Figura 1, las tres partes citadas forman un todo y se ejecutan en la misma máquina. Es la tradicional aplicación DOS/Windows que se encarga de solicitar/mostrar los datos, los procesa y, además, gestiona su almacenamiento/recuperación en archivos o bases de datos locales. Se trata de un modelo fácil de desarrollar, difícil de mantener, poco escalable y que precisa de una cierta potencia de proceso. Comparativamente hablando, instalar en cada puesto de operador de una empresa un sistema informático completo para ejecutar una aplicación sería como tener una centralita telefónica para cada trabajador, por el simple hecho de que necesita utilizar el teléfono. Resulta caro y los costes de mantenimiento son también altos.


***Figura 1.*** *Una aplicación monolítica duplica en cada ordenador todos sus elementos: interfaz de usuario, lógica o reglas de negocio y acceso a datos.*

El modelo cliente/servidor es el más usado en la actualidad. Una aplicación cliente/servidor, también conocida como aplicación en dos capas, separa la interfaz de usuario y reglas de negocio de la gestión de los datos, que queda en manos de un servidor que suele ejecutarse en una máquina dedicada. El equipo de desarrollo se centra en la creación de la interfaz con el usuario y el proceso de los datos, mientras que el almacenamiento y recuperación se deja en productos de tipos estándar: los habituales servidores de datos (Oracle, Informix, Interbase, SQL Server, etc.)

La aplicación es relativamente más compleja en el modelo cliente/servidor que en el monolítico. Ahora hay que comunicarse con un servidor de datos remoto pero, a cambio, muchos procesos que se realizaban localmente en la aplicación son ejecutados ahora por ese servidor, que es capaz de ofrecer resultados más elaborados. La potencia de proceso necesaria ahora se reparte, siendo precisos clientes más o menos rápidos para ejecutar las reglas de negocio y un potente servidor que sea capaz de atender las solicitudes de datos de todos esos clientes, tal y como se muestra en la Figura 2.

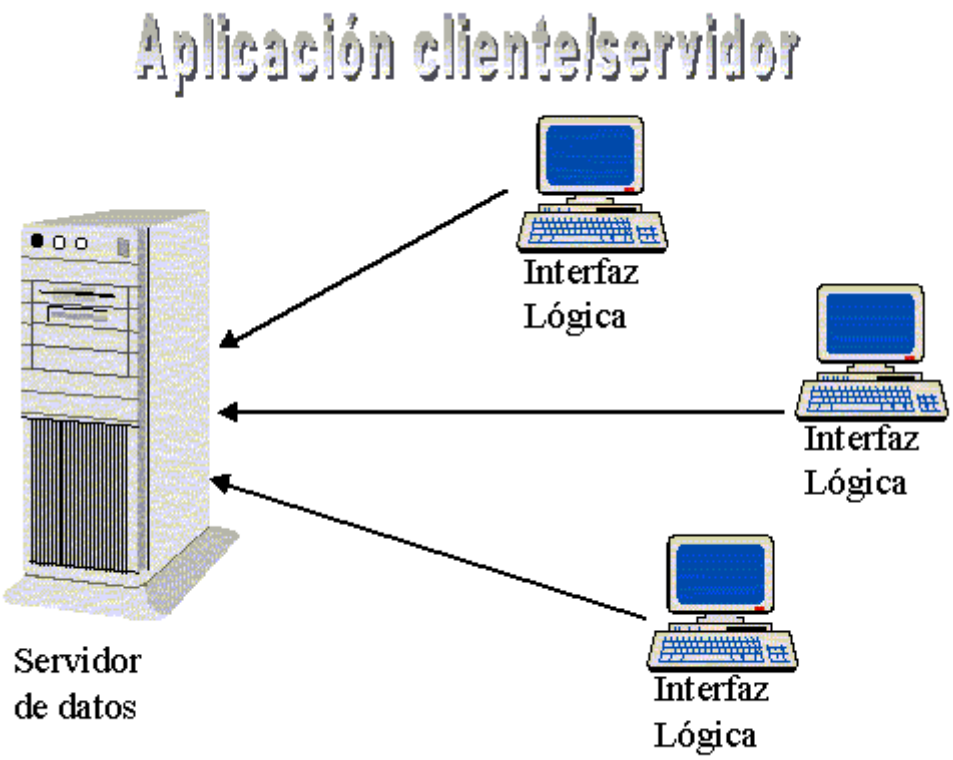


***Figura 2.*** *En una aplicación cliente/servidor existe un servidor de datos que es el único que manipula la base de datos, ejecutando las peticiones que recibe de los clientes.*

## Aplicaciones en múltiples capas

Esta extraña pero aceptada denominación hace referencia a aplicaciones que se ejecutan de forma distribuida en varias máquinas, denominándose nivel o capa a cada una de las divisiones. Cuando la aplicación se divide en dos partes se habla de arquitectura cliente/servidor, mientras que al existir tres o más niveles las aplicaciones se denominan multicapa o *multitier*.

El caso más típico es el mostrado en la Figura 3, la aplicación en tres capas: un cliente que ejecuta la interfaz, un servidor de aplicaciones que procesa las reglas de negocio y un servidor de datos. Existe, no obstante, la posibilidad de aumentar el número de capas añadiendo nuevos servidores de aplicaciones y datos, de tal forma que la carga de trabajo se distribuya entre varias máquinas.

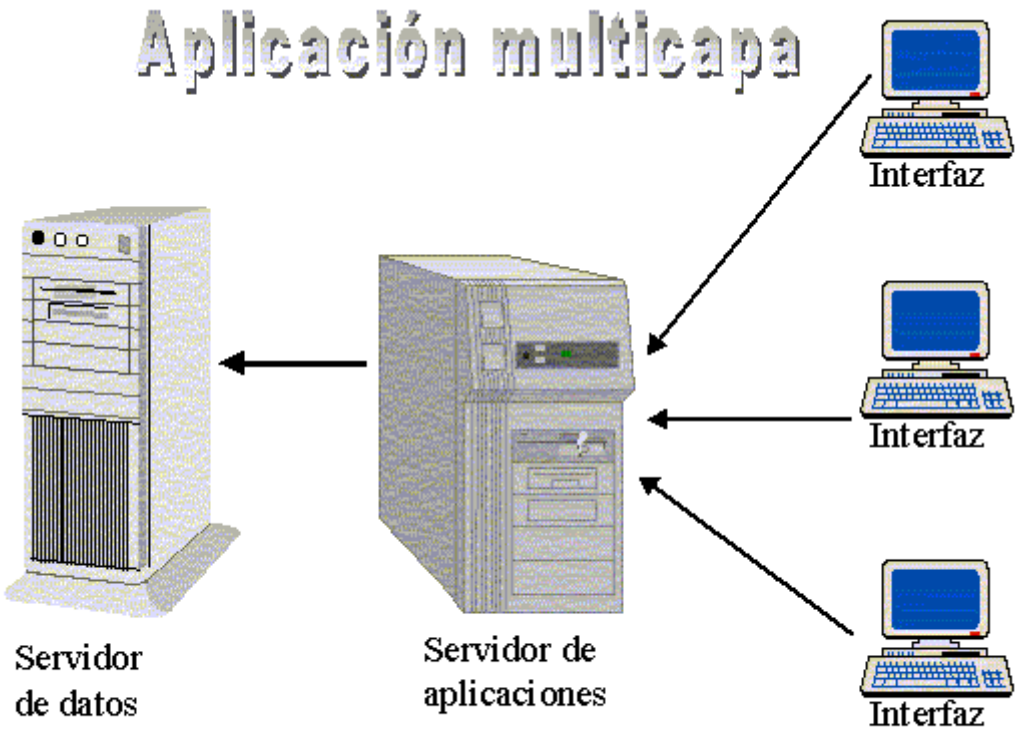


***Figura 3.*** *Ésta sería una típica aplicación en tres niveles. Los clientes sólo cuentan con una interfaz, la lógica se encuentra en un servidor de aplicaciones y los datos en un servidor de datos.*

Un servidor de aplicaciones puede ser un servidor web que ejecuta programas ante determinadas solicitudes de los clientes. Los servidores más conocidos disponen de conjuntos de funciones, como ISAPI/NSAPI, que permiten esas construcciones. En este caso está claro que el cliente, que ejecuta la interfaz, sería una aplicación como Netscape Communicator, Internet Explorer o similar.

Alternativamente es posible utilizar tecnologías de objetos distribuidos, como DCOM, CORBA o Java RMI, que permiten ejecutar objetos remotos sin necesidad de contar con un servidor web. En cualquier caso, siempre es preciso contar con unos servicios mínimos que permiten localizar los objetos, gestionar las llamadas entre clientes y servidor, etc. A estos servicios se les conoce habitualmente como *middleware*.

Desarrollar una aplicación en múltiples capas es una tarea más compleja, sobre todo si se compara con la construcción de un típico programa monolítico. A cambio, sin embargo, se obtienen muchos beneficios, como la simplicidad de mantenimiento y distribución o la gran escalabilidad. Al ser los clientes una simple interfaz, que no ejecuta ningún código de proceso de datos, cualquier necesidad de cambio en dicho proceso afectaría tan sólo al servidor de aplicaciones. Esto implica que las modificaciones en la lógica de la aplicación no conllevarían una redistribución e instalación en todos los clientes, sino tan sólo en el servidor, lo cual es un considerable ahorro de recursos en el mantenimiento de una aplicación.

El tema de la escalabilidad es otro de los puntos fuertes de las aplicaciones en varias capas. Si en un momento determinado el servidor de aplicaciones se ve saturado por la carga de trabajo, al crecer el número de clientes, no es preciso invertir gran cantidad de dinero en comprar servidores cada vez mayores, basta con instalar un nuevo servidor que comparta el trabajo con el ya existente. De esta forma varios servidores relativamente baratos pueden efectuar el mismo trabajo que un caro *mainframe*.

## Construcción de una aplicación distribuida

A la hora de desarrollar una solución de este tipo surge un lógico problema: la comunicación entre clientes y servidor de aplicaciones. Obviamente debe contarse con una infraestructura de red, este es el

punto de partida sin el cual nada más tiene sentido. Dicha infraestructura conecta a los diversos clientes con el servidor de aplicaciones y a éste con el servidor de datos. La comunicación entre ambos servidores es algo que no ha de preocupar, ya que existen estándares para ello y bastará con realizar una correcta configuración.

Asumiendo que se cuenta con la infraestructura necesaria y una serie de protocolos de conexión, como TCP/IP o similar, los clientes pueden comunicarse con el servidor básicamente utilizando tres técnicas diferentes: mensajes, llamadas remotas y objetos distribuidos. El servidor de aplicaciones se comunicará con el de datos generalmente usando mensajes.

La técnica de más bajo nivel consiste en enviar mensajes entre los dos puntos, cliente y servidor. Con este fin es posible usar los habituales servicios de *sockets*, existentes en prácticamente todos los sistemas, o bien recursos más específicos como los *mailslots* y *pipes*. En cualquier caso, el desarrollador de la aplicación tendrá que preocuparse de todos los detalles de comunicación: apertura del canal, envío y recepción, interpretación y proceso.

En un nivel intermedio se encuentran las llamadas a procedimientos remotos, conocidas como RPC (*Remote Procedure Call*). Aunque existen muchas implementaciones RPC, la más ampliamente aceptada es la conocida como DCE RPC (*Distributed Computing Environment RPC*) de la OSF (*Open Software Foundation*). Descrito básicamente, podría decirse que mediante RPC un determinado proceso puede realizar una llamada a un procedimiento que se ejecuta en otro proceso que, lógicamente, puede encontrarse en otra máquina de la red. Para ofrecer esta facilidad RPC cuenta con una serie de servicios y unas normas a la hora de definir los procedimientos.

El escalón superior de esta lista de posibilidades lo ocupan los modelos de objetos distribuidos, cuyos exponentes más representativos son DCOM, CORBA y Java RMI. Éstos están basados en mecanismos de tipo DCE RPC, por lo que aprovechan la experiencia e infraestructura ya existente. Su aportación, no obstante, es muy significativa: el desarrollador no gestiona mensajes o llama a procedimientos individuales sino que obtiene referencias a objetos, pudiendo llamar a métodos, obtener y establecer propiedades.

# Modelos de objetos distribuidos

Para elegir uno de los modelos de objetos distribuidos actualmente disponibles es preciso conocer sus características. Si bien no es la finalidad concreta de este artículo, que se centra más en las propiedades de un modelo concreto: CORBA, veamos brevemente en qué se diferencian, o en qué coinciden, los tres modelos ya mencionados.

DCOM (*Distributed COM*) es una evolución del modelo de componentes COM (*Component Object Model*) diseñado por Microsoft para su plataforma Windows. Dada la presencia masiva de Windows en la mayoría de los sistemas informáticos, podemos decir que DCOM está disponible en un gran porcentaje de los ordenadores que se usan actualmente. Es posible usar DCOM prácticamente desde cualquier lenguaje, creando tanto servidores como clientes. Si bien en un principio DCOM estaba sólo presente en Windows 95/98 y NT, actualmente existen implementaciones en sistemas UNIX como Solaris, esperándose en un futuro que empresas colaboradoras de Microsoft, como Software AG, trabajen en implementaciones en los sistemas más usados.

Si bien es cierto que Windows 95/98/NT suman la mayor parte de los usuarios de sistemas Microsoft, no es menos cierto que aún hay un número importante de ellos que siguen trabajando con Windows 3.1/WFW y DOS. Estos usuarios no cuentan con la posibilidad de usar DCOM, ni siquiera como clientes.

RMI (*Remote Method Invocation*) es un mecanismo Java que facilita la creación de objetos remotos y la invocación a sus métodos. Al ser una tecnología 100% Java, teóricamente puede utilizarse en cualquier sistema en el que exista una máquina virtual y su entorno de ejecución JRE (*Java Runtime Environment*), lo cual garantiza una transportabilidad muy alta. Un mismo servidor o cliente, sin necesidad de recompilación, puede llevarse desde Windows a Linux, por poner un ejemplo, y seguir funcionando sin problemas. El mayor inconveniente de Java RMI es que, como su propio nombre indica, tan sólo puede ser usado por los desarrolladores que usan el lenguaje Java, en contraposición a DCOM, cuyos componentes pueden crearse y utilizarse con la mayoría de los lenguajes.

Las ventajas de CORBA (*Common Object Request Broker Architecture*) frente a sus dos más claros competidores son bastantes. Los servidores y clientes CORBA pueden desarrollarse usando prácticamente cualquier lenguaje, desde los más usados en la actualidad, como ocurre con DCOM, hasta los veteranos pero aún muy presentes, como es el caso de COBOL. Existen servicios CORBA disponibles para una treintena de plataformas, desde DOS o Linux hasta los *mainframes* pasando, como es lógico, por todas las versiones de Windows. Se trata, además, de una tecnología que lleva utilizándose, depurándose y

evolucionando desde hace una década. Todos estos elementos hacen que CORBA sea la opción de muchas grandes empresas, tanto en nuestro país como fuera de él.

Seleccionar uno de estos modelos para un desarrollo concreto no implica siempre excluir a los demás modelos, siendo posible establecer *puentes* que facilitan la colaboración. Existen, por ejemplo, productos capaces de actuar como intérpretes comunicándose por una parte con DCOM y por otra con CORBA, lo cual amplia aún más las posibilidades a nuestro alcance.

# ¿Qué es CORBA?

Podríamos definir CORBA como un conjunto de especificaciones, gestionadas por el OMG (*Object Management Group*) cuya finalidad es facilitar la interoperabilidad entre componentes *software* implementados en cualquier lenguaje y que se ejecutan en cualquier sistema y plataforma *hardware*. Conseguir este objetivo implica, lógicamente, una cierta complejidad de la arquitectura, aunque no lo es tanto cuando se conoce.

El OMG está actualmente formado por más de ochocientas empresas entre las cuales se encuentran las más importantes del sector: Hewlett-Packard, Sun Microsystems, Inprise, Lucent, Iona, Compaq y Microsoft, por mencionar sólo algunas conocidas. Los objetivos del OMG se plasman documentalmente en la OMA (*Object Management Architecture*), una arquitectura en la cual CORBA es una pieza más, quizá la más importante. Las mencionadas empresas, denominados miembros *contribuyentes*, son las que aportan su conocimiento en el campo del *software* y realizan las especificaciones, teniendo derecho a opinar y votar sobre las propuestas que se realicen. Además también forman parte del OMG otras muchas empresas y organismos, desde infinidad de universidades hasta empresas como Boeing, Ford o France Telecom, que representan los intereses de ciertos sectores en esa tecnología.

La misión del OMG es crear especificaciones e ir adaptándolas a las necesidades evolutivas de la informática, definiendo claramente qué servicios, protocolos y estructuras deben implementarse para que un producto pueda considerarse compatible CORBA. El OMG, sin embargo, no crea ni vende producto alguno, es un organismo independiente en sí, aunque las empresas que lo forman son libres de crear implementaciones de esas especificaciones como de hecho ocurre. Ésta es una diferencia fundamental respecto a los otros modelos, que han sido definidos e implementados por un determinado fabricante que, hasta ahora, ejerce un control total sobre él.

Para conseguir los objetivos que se persiguen, CORBA se apoya en tres pilares fundamentales: el lenguaje de definición de interfaces IDL (*Interface Definition Language*), los ORB (*Object Request Broker*) y el protocolo GIOP (*General Inter-ORB Protocol*) y sus derivados. Algunos de estos elementos, como el lenguaje IDL, existían en especificaciones previas del DCE RPC de la OSF, otros como los protocolos de comunicación entre ORBs no aparecieron hasta la especificación CORBA 2.0.

Llegados a este punto, en el que ya sabemos qué es CORBA, para qué sirve y el lugar que ocupa dentro del desarrollo de aplicaciones distribuidas, vamos a adentrarnos en el estudio de los tres citados pilares: IDL, ORB y GIOP. También trataremos algunos de los servicios de CORBA, sin los cuales no será posible la cooperación entre aplicaciones.

# El lenguaje IDL

¿Cómo facilitar la cooperación entre componentes implementados con diferentes lenguajes? ¿Pueden Java, COBOL, C++, Visual Basic y Delphi entenderse entre ellos? La respuesta es un sí condicional: es posible siempre que se disponga de un punto en común. En el ámbito CORBA ese punto de encuentro es el lenguaje IDL.

IDL no es un lenguaje de programación, sino un lenguaje descriptivo. Como su propio nombre indica se usa para describir interfaces, entendiendo como tales tablas de métodos asociados en las que se especifica el nombre de cada uno de ellos, los parámetros que precisan y el tipo del valor de retorno que devolverán. IDL es un lenguaje neutro, aunque su sintaxis es parecida a la de C/C++ y Java.

Una interfaz IDL es, por lo tanto, una especie de contrato entre dos partes: el cliente y el servidor. En dicho contrato se especifica claramente qué interfaces puede encontrar el cliente en el servidor, qué métodos implementa cada interfaz, cuáles son sus parámetros, etc. Conceptualmente hablando podríamos decir que una interfaz IDL es muy similar a construcciones habituales como la definición de una clase abstracta, en C++, o una interfaz, en Delphi o Java.

Las interfaces IDL se agrupan formando módulos. El módulo es la construcción de primer nivel en IDL, existiendo generalmente uno por cada archivo de definición. En el Listado 1 puede ver un ejemplo

sencillo. Se trata de un módulo que tan sólo contiene una interfaz, ésta cuenta con cinco métodos cuya finalidad teórica es facilitar la ejecución de operaciones estadísticas.

Un módulo IDL, como el mostrado en el Listado 1, es tan sólo una enumeración de los elementos que existirán en dicho módulo, pero el código en sí no es útil directamente para construir una aplicación. Por eso existen herramientas conocidas como *compiladores IDL*, aunque en realidad no son compiladores puesto que no generan código ejecutable. Lo que hacen es traducir la descripción IDL a un determinado lenguaje, generando los módulos, clases o interfaces que procedan y a partir de las cuales se comenzará a trabajar.

```
module Estadisticas {
  interface Basicas {
    void Anade(in float Valor);
    void Elimina();
    float Media();
    float Maximo();
    float Minimo();
  };
};
```

***Listado 1.*** *Módulo IDL con la definición de una interfaz que dispone de cinco métodos*

El compilador IDL sí es dependiente del lenguaje, es decir, existen compiladores que traducen IDL a Java, IDL a C++, IDL a COBOL, etc. La definición original es independiente del lenguaje, pero la implementación del servidor o el cliente se realizan en un determinado lenguaje. Es la descripción original, no obstante, la que permite que dos implementaciones en lenguajes diferentes se realicen de forma compatible, traduciendo las construcciones y tipos de IDL a los elementos apropiados de dichos lenguajes.

# El gestor de solicitudes a objetos

Asumiendo que a partir de un módulo IDL, usando los correspondientes compiladores, se ha obtenido el código necesario para implementar tanto el cliente como el servidor, surge un nuevo interrogante: ¿cómo puede el cliente llamar a los métodos del servidor? Llegados a este punto entra en escena otro de los elementos fundamentales: el ORB.

Tanto el cliente como el servidor cuentan con un ORB adaptado a sus necesidades, es decir, un ORB específico para el lenguaje y la plataforma de la aplicación concreta que se desarrolla. La finalidad del ORB es traducir la llamada del cliente, realizada en un determinado lenguaje y sobre una cierta plataforma, a un formato neutro, totalmente independiente, que puede transportarse sobre cualquier medio para el que exista un protocolo de comunicación. Lógicamente el ORB del servidor actúa en sentido inverso, recibiendo la llamada en formato neutro y convirtiéndola al lenguaje y la plataforma destino.

La función del ORB es, por lo tanto, conectar a dos partes: cliente y servidor. Presumiblemente estas partes están desarrolladas con distintos lenguajes, se ejecutan sobre plataformas distintas y funcionan con diferentes sistemas operativos. Esto significa que pueden existir diferencias en tipos de datos, el orden de los parámetros en una llamada, el orden de los bytes en una palabra según el tipo de procesador, etc. Es misión del ORB efectuar los procesos conocidos como *marshaling*: la citada traducción a un formato neutro, y *unmarshaling*: el paso inverso.

En caso de que el método invocado devuelva un valor de retorno, la función de los ORB del cliente y servidor se invierte. El servidor realiza el *marshaling* de dicho valor y lo envía al ORB del cliente, que será el que realice el *unmarshaling* y finalmente facilite el valor en formato nativo.

A pesar de la aparente complejidad que implica todo el proceso descrito, para el programador que implementa un cliente CORBA realizar una llamada a un método del servidor es tan simple como hacerlo a un método local, sin diferencias aparentes. El servidor, por su parte, recibe una invocación sin distinguir en ningún momento de dónde procede. Lógicamente es el ORB quien se encarga de todo el trabajo de bajo nivel.

Actualmente existen ORBs de diversos fabricantes, como Iona o Inprise, que pueden utilizarse con múltiples lenguajes y plataformas. El conocido Visibroker de Inprise, uno de los ORB más usados en empresas actualmente, está disponible para los lenguajes C++ y Java en plataformas Windows y UNIX. Un ORB independiente de este tipo podría utilizarse, por ejemplo, desde Microsoft Visual C++. Algunas plataformas, como Java 2, ya integran un ORB y algunos servicios básicos de CORBA, lo cual hace

innecesaria la adquisición de un ORB de terceros. También hay herramientas, como Borland C++ Builder o Borland JBuilder, que integran un ORB y diversos asistentes que facilitan el desarrollo de aplicaciones CORBA.

# GIOP y sus derivados

En un principio los dos principales puntos de apoyo de la tecnología CORBA eran el lenguaje IDL y los ORB. Con estos elementos la colaboración entre aplicaciones era posible siempre y cuando los ORB usados fuesen de un mismo fabricante ya que, en la mayoría de las ocasiones, el protocolo de comunicación que usaban para comunicarse entre ellos era propietario.

Es con la aparición de la especificación CORBA 2.0 cuando se establece un protocolo estándar que facilite la comunicación entre distintos ORB, ya sean éstos del mismo fabricante o no. Realmente son varios los protocolos propuestos, si bien existen dos de vital importancia: GIOP e IIOP (*Internet Inter-ORB Protocol*).

GIOP es una especificación genérica de cómo deben comunicarse entre sí dos ORB, al tiempo que mantiene una independencia total sobre el protocolo de transporte que se use para dicha comunicación. Utilizando un símil con el mundo de la programación, podríamos decir que GIOP es algo así como una interfaz o clase abstracta: indica qué servicios existen y cuales son las conexiones, pero en la práctica no es posible usar GIOP para nada.

Derivando de GIOP se definen varios protocolos específicos para usar como capa de transporte a TCP/IP, OSF DCE, IPX/SPX, etc. De todos ellos el más importante es el mencionado IIOP que, como puede suponer, usa los protocolos TCP/IP para comunicar a un ORB con otro. Esto significa que es un protocolo aplicable a la mayoría de redes actuales, ya que prácticamente todas ellas cuentan con servicios TCP/IP.

De los protocolos derivados de GIOP el único que obligatoriamente ha de implementar un ORB es IIOP. De hecho, un ORB no será considerado compatible con CORBA 2.0 si no implementa dicho protocolo. Esto no significa, no obstante, que el fabricante no pueda añadir cualquier otro protocolo que desee, por ejemplo un protocolo nativo más eficaz que IIOP y que sólo usaría cuando los ORBs fuesen del mismo fabricante. Cuando dos ORB quieren comunicarse entre sí lo primero que hacen es ponerse de acuerdo en cuanto al protocolo que utilizarán, lo cual facilita la elección del más apropiado según los casos.

En la Figura 4 se muestra un esquema simplificado de la relación existente entre una aplicación cliente que quiere usar cierto servicio implementado por un servidor CORBA. Tanto cliente como servidor cuentan con un ORB, utilizando el protocolo IIOP para comunicarse entre ellos.

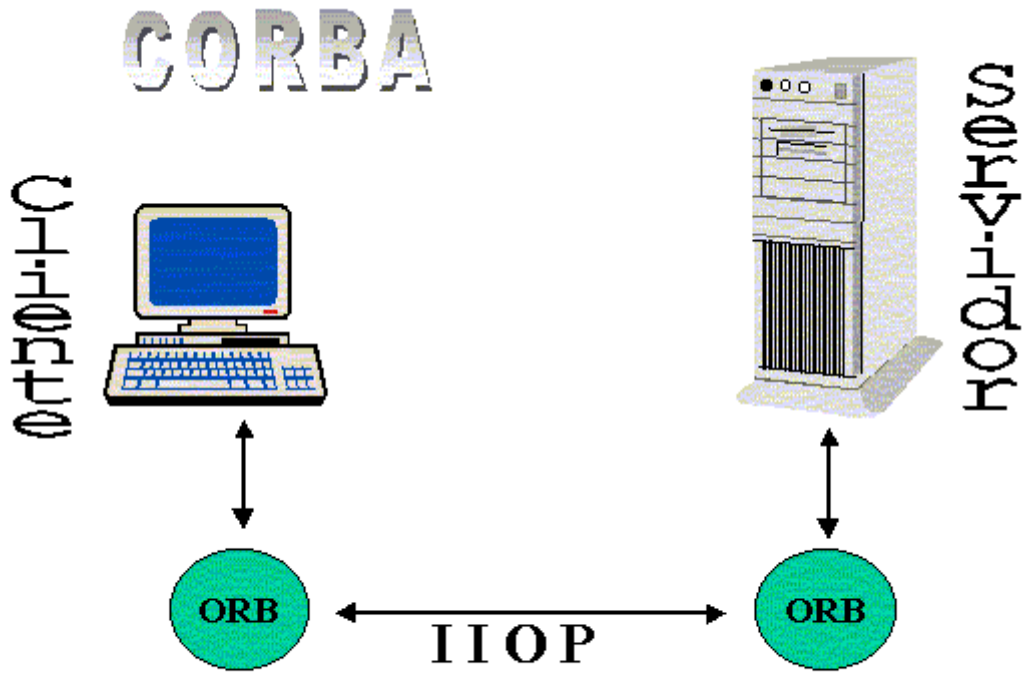


***Figura 4.*** *En este esquema simplificado se aprecia la relación de las aplicaciones, cliente y servidor, con los ORB y el protocolo IIOP que les permite comunicarse.*

## Servicios CORBA

Disponiendo de un compilador IDL y un ORB que cumpla con la especificación CORBA 2.0, el desarrollo de una aplicación distribuida no implica mucha más complejidad que el aprendizaje de algunos servicios nuevos, accesibles en forma de métodos principalmente de dos objetos: el ORB y el BOA (*Basic Object Adapter*). Para que dicha aplicación funcione, sin embargo, son necesarios algunos servicios adicionales.

Los servicios CORBA, conocidos habitualmente como *CORBAservices*, no forman parte de CORBA sino de OMA, la arquitectura global que persigue el OMG. Esto significa que la adquisición de un producto CORBA, como suele ser un ORB y el compilador IDL, no implica la disponibilidad de servicios adicionales. Muchos de estos servicios, sobre todo los más complejos, de hecho se venden como productos separados.

Existen servicios que permiten a los componentes CORBA gestionar transacciones, añadir elementos de seguridad, hacer que los objetos sean persistentes, etc. De todos los servicios, no obstante, los dos más comunes y necesarios son el de nombres y eventos. El primero de ellos facilita la resolución de referencias a objetos, mientras que el segundo permite la gestión de eventos entre objetos COBA.

Un servicio fundamental es el de nombres o directorio distribuido. Asumiendo que una aplicación CORBA usa servicios implementados por objetos que están distribuidos por una red, que puede ser más o menos grande, la localización de esos objetos puede ser un problema. El ORB es capaz de realizar una llamada a los métodos de un objeto siempre que se disponga de una referencia a éste, referencia que es preciso obtener previamente de alguna forma.

El servicio de nombres se ejecuta en un determinado puesto de la red bien conocido por las aplicaciones. Los servidores registran sus implementaciones de interfaces facilitando la referencia a los objetos que hay disponibles. Cuando un cliente necesita una cierta interfaz, consulta al servicio de nombres y obtiene esa referencia. A partir de aquí ya es tarea de los ORBs la comunicación entre ambos extremos.

Todos los servicios CORBA se exponen en forma de objetos CORBA cuyas interfaces están claramente definidas en la correspondiente especificación, lo cual significa que dichos servicios deberían de poder utilizarse desde cualquier cliente CORBA, a pesar de que esté desarrollado con otro lenguaje y ejecutándose en una plataforma o sistema operativo diferente.

# Esquema de desarrollo

En los puntos anteriores se han ido describiendo los elementos fundamentales de la arquitectura CORBA, elementos que permiten la creación de aplicaciones en forma de componentes u objetos que se ejecutan de forma distribuida. Todas estas piezas, analizadas individualmente, han de unirse de forma adecuada para conseguir el objetivo final: un servidor que ofrezca uno o más servicios y un cliente que los utilice.

El primer paso que hay que dar, tras el lógico proceso de análisis, consiste en escribir los módulos IDL que contendrán las definiciones de interfaces. Éstas serán la especificación que tendrá que seguir el servidor a la hora de implementar sus servicios, así como la seguida por cualquier cliente que necesite utilizarlos. Tan sólo hay que definir en forma de interfaces IDL aquellos elementos que vayan a ejecutarse distribuidamente, no todo objeto que pueda forma parte de la aplicación.

Disponiendo del módulo IDL el siguiente paso será la compilación, obteniendo como producto dos módulos de código dependientes del lenguaje: un *stub* y un *skeleton*. En caso de que servidor y cliente vayan a desarrollarse con diferentes lenguajes será preciso compilar varias veces el módulo IDL, una vez con cada compilador IDL.

El proceso de compilación del código IDL depende del lenguaje y la herramienta de desarrollo que se use. Java IDL, por ejemplo, dispone de la herramienta `idltojava`, que se usa desde la línea de comandos. Entornos como C++ Builder, por el contrario, integran estas operaciones en el propio entorno, siendo preciso tan sólo seleccionar la opción adecuada.

También el nombre y número de los módulos generados por el compilador dependerá del lenguaje destino. Como mínimo existirá un módulo de *stub* y otro de *skeleton*, pero adicionalmente pueden existir otros. Al usar Java, compilando con `idltojava`, se obtienen también unos módulos con clases auxiliares denominadas genéricamente *helper*.

# Implementación del servidor

Partiendo del módulo IDL el compilador ha generado otro conocido como *skeleton*, que servirá para implementar el servidor propiamente dicho. En los lenguajes C++ y Java el módulo *skeleton* contiene una clase cuyos métodos realmente se encargarán de la comunicación con el ORB, utilizando la API que éste expone. Son estos métodos los que provocarán que las llamadas remotas a los métodos del servidor parezcan locales, ocupándose de todos los detalles de bajo nivel.

La implementación del servidor puede realizarse generalmente siguiendo uno de dos métodos diferentes: herencia o delegación. El primero consiste en usar el *skeleton* como base derivando una clase nueva, que será en la que se implementen los métodos originalmente definidos en la interfaz IDL. El hecho de que la clase de implementación tenga que derivar del *skeleton* en ocasiones puede implicar ciertas limitaciones. Por ello existe el método de delegación, mediante el cual la clase en la que se implementan los métodos puede ser cualquiera ya que lo que se hace es llamar directamente desde el *skeleton* a sus métodos.

Aparte de implementar los servicios que el cliente espera encontrar, el servidor también tendrá que dar algunos pasos adicionales. Tras inicializar el ORB y el BOA, lo normal es que obtenga una referencia al servicio de nombres y la utilice para registrarse, de tal forma que pueda ser localizado por cualquier cliente que precise sus servicios.

Un servidor no puede saber de antemano qué peticiones va a recibir, de dónde van a venir ni en que momento. Por ello lo habitual es que tras dar los anteriores pasos de preparación quede a la espera indefinidamente, procesando las peticiones a medida que éstas llegan.

# Implementación del cliente

Si para desarrollar el servidor se hace uso del *skeleton*, la creación de cualquier cliente implicará la utilización del correspondiente *stub*. Éste hace que las llamadas a los métodos del servidor por parte del cliente parezcan llamadas locales, como las que es posible realizar a cualquier otro objeto con que cuente la aplicación. Esto es posible porque la clase que hay en el *stub* implementa la interfaz definida en el módulo IDL, si bien su código no se encarga de hacer el trabajo real del servidor.

La clase *stub* cuenta con los métodos necesarios para comunicarse con el ORB del cliente, indicándole todos los datos necesarios para poder invocar al método remoto, en el servidor. Como ya se indicó antes, será el ORB el que en último término prepare los parámetros y se comunique con el otro extremo.

Lo habitual es que el cliente comience por inicializar su ORB, tras lo cual hará uso del servicio de nombres para obtener una referencia al objeto que implementa la interfaz que él necesita utilizar. Con esa referencia, conocida como IOR (*Interoperable Object Reference*), se llamará directamente a los métodos y se obtendrá el resultado esperado, sin necesidad de ningún paso adicional.

Para que un cliente pueda usar los servicios de un servidor lógicamente éste deberá estar ejecutándose en algún punto de la red. Opcionalmente es posible usar un repositorio de interfaces y un servicio de activación a demanda. De esta forma el cliente buscaría en el repositorio la interfaz que necesita, obteniendo los datos precisos para ejecutar servidor.

Aunque previamente se ha dicho que cualquier cliente que necesite utilizar un cierto servicio tiene que disponer del correspondiente *stub*, esto es una verdad a medias. Usando una técnica conocida como DII (*Dynamic Invocation Interface*) y un repositorio de interfaces es posible resolver ciertas referencias durante la ejecución en lugar de hacerlo en la compilación. Es una técnica similar a la que permite el uso polimórfico de objetos en lenguajes como C++. El uso de DII, no obstante, implica una mayor carga y tiempo de proceso, por lo que su uso no es muy extendido y se limita a casos concretos.

En la Figura 5 se muestra esquemáticamente el proceso de desarrollo comentado en estos últimos puntos. Partiendo de un módulo IDL se obtiene, mediante el compilador IDL, un *stub* y un *skeleton*. Éstos sirven a continuación para implementar el cliente y el servidor, que serían instalados en las correspondientes máquinas de red. A partir de aquí se pondrían en funcionamiento según el esquema de la Figura 4.

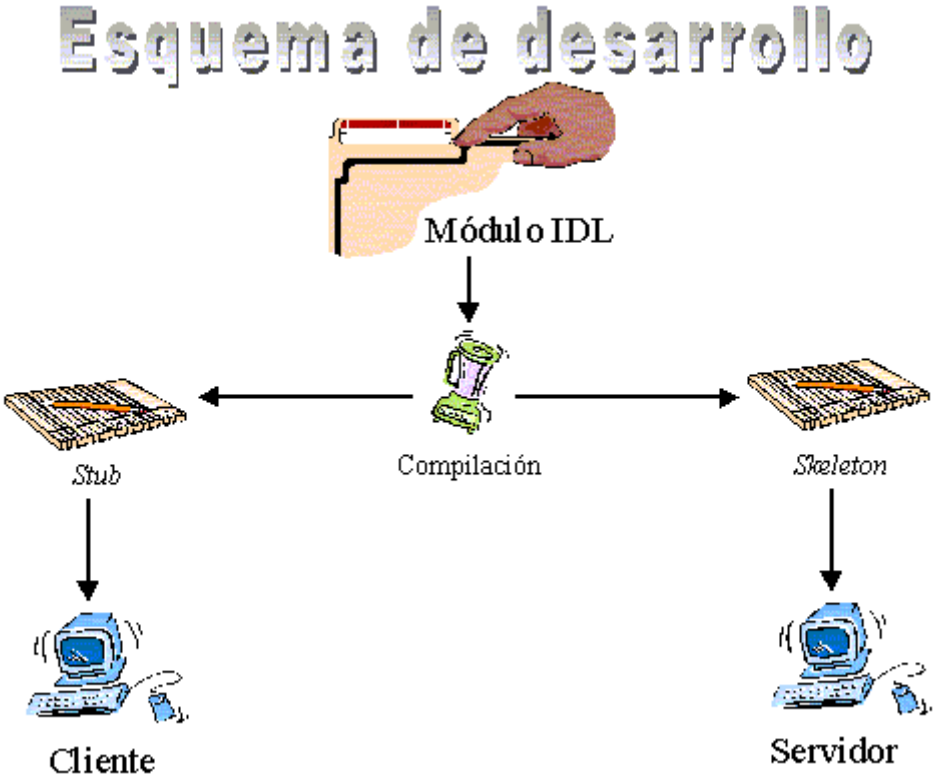


***Figura 5.*** *El proceso de desarrollo de una aplicación CORBA se inicia con la definición de interfaces en el módulo IDL, que es posteriormente compilado para obtener un stub y un skeleton. Son estos módulos los que permiten después implementar el cliente y el servidor.*

# CORBA en la práctica

Todos los conocimientos teóricos acerca de CORBA, parte de los cuales se introducen en este artículo, son necesarios para comprender el funcionamiento de una aplicación distribuida y poder desarrollarla correctamente. Los casos prácticos, sin embargo, siempre son un recurso inestimable como ejemplos a seguir.

En próximas entregas de esta seria se demostrará cómo es posible desarrollar servidores y clientes RMI y CORBA usando la plataforma Java 2, profundizando en las particularidades del compilador IDL de Java, las clases *stub* y *skeleton* y el uso del servicio de nombres. También se ofrecerá un ejemplo de uso de la tecnología CORBA, concretamente se mostrará como desarrollar servidores y clientes CORBA, usando Borland C++ Builder 4. Esta herramienta de Borland, concretamente su versión superior, incorpora todos los elementos necesarios para los desarrollos CORBA: un compilador IDL con un editor específico, un ORB (Visibroker), etc.

# Resumen

La arquitectura CORBA es una tecnología sobradamente probada cuya finalidad es facilitar la creación de aplicaciones distribuidas, con una interoperabilidad entre componentes *software* independientemente del lenguaje en que están desarrollados, la plataforma *hardware* y el sistema operativo sobre el que se ejecutan. Ciertamente CORBA no es la única tecnología que facilita el desarrollo de aplicaciones distribuidas, pero no es menos cierto que actualmente las otras opciones están aún por detrás en muchos aspectos

En este artículo se ha realizado una breve introducción al mundo de las aplicaciones distribuidas, la arquitectura CORBA y sus pilares principales: el lenguaje de descripción de interfaces IDL, los ORB y el protocolo IIOP. También se ha hecho referencia a algunos otros elementos de OMA, como los servicios de nombres y eventos.