Desarrollo Orientado a Objetos en un Ambiente Distribuido

María Soledad Escobar

Trabajo de Grado

Facultad de Informática

Universidad Nacional de La Plata

Dirección del trabajo a cargo del Ingeniero Gustavo Antonelli

Diciembre de 1999

INDICE

Introducción

Computación distribuida

Es un hecho que en una red encontramos una cada vez más amplia diversidad en el hardware y software. En el “high end” nos encontramos con supercomputadoras trabajando en conjunto para complejos cálculos científicos y de ingeniería o que sirven bases de datos masivas como las de una línea aérea. En el “midrange”, servidores poderosos y computadoras de escritorio que proveen una amplia gama de servicios. Y en el “low-end”, sistemas embebidos, que nosotros no llamamos computadoras pero que interactúan con otros sistemas, como por ejemplo PDAs, televisores y VCRs, pagers, teléfonos celulares y muchos más. Explorando esta lista, somos conscientes de que utilizan diferentes sistemas operativos y lenguajes de programación. Pero la necesidad de interoperabilidad es creciente y trasciende a la máquina o al sistema operativo.

Veamos los problemas que surgen en estos ambientes complejos:

- Es difícil lograr que el hardware trabaje junto: la necesidad de varios tipos de computadoras, dispositivos de red y sistemas operativos dificultan considerablemente la interoperabilidad.
- Es aún más difícil lograr que el software trabaje junto: una compañía es una colección de diferentes partes que trabajan en cierta medida juntas para lograr determinados objetivos. Estos atributos deben verse reflejados en el software.
- Finalmente, el desarrollo de software toma mucho tiempo y cuesta demasiado: típicamente cada proyecto comienza de cero, y esto era aceptable cuando los mismos eran pequeños y simples, pero no lo es cuando se trabaja para aplicaciones complejas. Es necesario un modo de construir el software sobre bases comunes, para permitir reusabilidad y ahorrar en tiempo y costos.

El software basado en componentes está haciendo esto posible. Las aplicaciones están cambiando y los clientes son cada vez mas reacios a aceptar aplicaciones inmensas y monolíticas que hagan todo. Hoy están buscando componentes más pequeñas y flexibles que se puedan combinar y adaptar a las necesidades de sus negocios. Las componentes trabajarán juntas sólo si han sido diseñadas y construidas sobre interfaces estándares.

Ya que la interoperabilidad debe extenderse en la empresa y aún más allá de ella para integrarse con clientes y proveedores, estas interfaces deben ser independientes de la plataforma, sistema operativo, lenguaje de programación y hasta del protocolo de red. Cualquier otra cosa creará "islas de interoperabilidad", aisladas las unas de las otras por barreras tecnológicas arbitrarias.

Tecnología de objetos

Lo que necesitamos es un nuevo modo de encarar los problemas (desde su establecimiento y análisis, a través del diseño de su solución e implementación y hasta el uso, mantenimiento y extensión de la misma) que integre cada componente y nos lleve de una manera ordenada desde un paso al siguiente. La tecnología de objetos es el nuevo camino.

En el pasado la computación estaba basada en los *datos* y los *procedimientos* en vez de en los procesos de negocio. Esto se debe en parte a que las computadoras eran muy costosas y todos los esfuerzos conducían al ahorro de recursos (memoria, CPU, entrada/salida).

El hardware ha cambiado mucho desde sus primeros pasos, y en la actualidad los recursos dejaron de ser un problema crucial. El desafío actual es lograr explotarlos al máximo focalizándonos en la soluciones que se adapten a las necesidades reales. La tecnología de objetos apunta a modelar el mundo real. Un objeto en una computadora debe corresponder a un objeto en el mundo real. Los objetos trabajan juntos para modelar las interacciones del mundo real.

Un desafío es tratar con la complejidad, que es la palabra que mejor describe al mundo actual. A través del trabajo de generaciones de analistas se ha probado que el mejor modo de tratar con un problema complejo es dividirlo en partes más pequeñas y manejables, teniendo especial cuidado en realizar las divisiones correctamente para no sumar complejidad en vez de reducirla.

Separando los problemas en componentes que modelen el mundo real tratamos con cosas que nos son familiares. Los componentes son más fáciles de entender para todos (clientes, analistas, programadores, etc.) y las interacciones entre los objetos aparecen como lógicas y naturales.

Pero la computación orientada a objetos nos da más que esto, también nos ayuda a resolver los tres principales problemas de la programación que se describen a continuación:

- *Los programas son difíciles de cambiar y extender:* cuando los programas se implementan como objetos discretos, muchos cambios afectarán sólo a un objeto o a un grupo reducido de ellos.
- *Los programas son difíciles de mantener:* cuando se programa usando un paradigma procedural, los efectos del mantenimiento son muy difíciles de medir. Basando nuestros programas en objetos que modelan el mundo real, estos efectos son mucho más previsibles y mesurables.

- *Lleva mucho tiempo escribir programas:* los objetos bien concebidos invitan a su reutilización, haciendo que los programas puedan ser escritos en menor tiempo y con menores costos.

Object Management Group

El OMG es un consorcio internacional formado por más de 800 compañías involucradas con la informática, donde el gran ausente es Microsoft que compite con su propio producto llamado Distributed Component Object Model (DCOM). El OMG no produce software sino especificaciones. Nació en 1989 con ocho miembros: 3Com, American Airlines, Canon, Data General, Hewlett-Packard, Philips Telecomunications N.V., Sun Microsystems, y Unisys.

Los objetivos del OMG han sido y son promover soluciones orientadas a objetos para la ingeniería de software y desarrollar un "framework" de desarrollo común para escribir aplicaciones de objetos distribuidos basados en especificaciones de interfaces para los objetos de la aplicación. Como resultado el OMG creó la Common Object Request Broker Architecture (CORBA). La interoperabilidad y la transparencia de localización, son premisas fundamentales sobre las cuales nació CORBA.

Organización del trabajo

La primera parte de este trabajo tiene como objetivo describir la especificación de CORBA (Common Object Request Broker Arquitecture) en general.

En la segunda parte veremos cómo CORBA se Integra con Java.

La Tercera parte incluye la comparación de CORBA con otros sistemas que permiten construir aplicaciones Client/Server y se perfilan como sus competidores.

Como apéndice presentaré un ejemplo práctico del uso de CORBA con JAVA a través de una aplicación.

Parte I:

Descripción de la aquitectura de CORBA

Esta primera parte se compone de ocho capítulos que describen a CORBA. En el primero, se introducen conceptos fundamentales como la definición de un objeto CORBA, el *Interface Definition Language* (IDL) para la especificación de las interfaces de los objetos y la anatomía del *Object Request Broker* (ORB), con una breve descripción inicial de cada uno de sus componentes.

En el segundo capítulo se define el rol del IDL y se analizan sus características; describiendo su sintaxis, los tipos de datos que soporta y la definición de operaciones.

En el tercer capítulo vemos cuáles son las operaciones que se pueden realizar sobre el ORB a través de su interfaz.

En el capítulo cuarto nos concentramos en la *Dynamic Invocation Interface* (DII), especificando su función y cómo afecta a la semántica de las operaciones. También vemos los pasos necesarios para su utilización y las operaciones de las interfaces involucradas.

El capítulo quinto detalla las responsabilidades de un *Object Adapter* (OA). Se introducen dos tipos de OAs especificados por el OMG: el *Basic Object Adapter* (BOA) y el *Portable Object Adapter* (POA), viendo en cada caso sus características y políticas de activación de objetos que soportan.

En el capítulo sexto nos abocamos a la interoperabilidad de CORBA a través de protocolos para la comunicación entre ORBs como el *Internet Inter-ORB Protocol* (IIOP) y el *Enviroment-Specific Inter-ORB Protocol* (ESIOP).

En los capítulos séptimo y octavo se describen los servicios y facilidades de CORBA. Nos focalizamos principalmente en los servicios ya que se encuentran en un estado más avanzado y su utilidad está directamente relacionada con toda implementación de aplicaciones distribuidas.

Capítulo I : Descripción General de CORBA

Conceptos fundamentales

En esta sección veremos algunos conceptos fundamentales para introducirnos en el mundo de CORBA.

Qué es un objeto CORBA?

Un objeto CORBA cumple con tres características que permiten la reusabilidad de los módulos de software y el cambio de los mismos sin que afecten el resto del sistema. Estas tres características son: encapsulamiento, herencia y polimorfismo.

Encapsulamiento

El encapsulamiento nos da la posibilidad de software *plug-and-play*. El software encapsulado contiene dos partes: una interfaz, que el módulo presenta al mundo exterior, y una implementación que es privada. La interfaz representa las funcionalidades del objeto: si un cliente envía al objeto un mensaje especificado en su interfaz con los parámetros apropiados, el objeto responderá en el modo apropiado. El modo en el que el objeto produce o calcula la respuesta y ubica los resultados en los lugares correctos, concierne solamente al objeto. El cliente no puede examinar ese proceso ni obtener ninguna información adicional.

Si tenemos más de una implementación del objeto con la misma interfaz (la implementación puede haber sido escrita usando diferentes lenguajes), podríamos sustituir una por otra sin que el cliente se entere ya que las respuestas a sus mensajes no cambiarían.

El encapsulamiento también le da a CORBA la transparencia de localización. Los clientes envían una invocación a su ORB local, no al objeto en sí; el ORB rutea el mensaje a su destino. El cliente no sabe dónde realmente se encuentra implementado el objeto del cual ha solicitado un servicio. Incluso la implementación del objeto puede moverse sin que el cliente necesite saberlo, ya que esto es conocido por el ORB. Esto funciona a través de la referencia al objeto que se explicará más adelante en esta sección.

El encapsulamiento nos da un ambiente dividido en componentes, pero no es suficiente en un ambiente orientado a objetos. Podemos sustituir implementaciones y moverlas a través de una red a voluntad. Pero para beneficiarnos de la programación orientada a objetos necesitamos también herencia y polimorfismo.

Herencia

Se pueden crear nuevos *templates* de objetos (clases) más fácilmente adaptándolos a otros que creándolos de cero. Se pueden definir nuevas clases como subclases de otras, heredando de las mismas las definiciones de los tipos de datos y operaciones. Los objetos creados en base a subclases tendrán todas las propiedades de su superclase más las de ellos mismos.

El concepto de herencia de la programación orientada a objetos, ahorra mucho trabajo al programador.

Polimorfismo

Naturalmente, algunas operaciones pertenecen a más de una clase de objetos. Por ejemplo, podríamos invocar la operación *draw* a un círculo, un cuadrado, un rectángulo, etc. El polimorfismo es una propiedad que permite al cliente invocar a la misma operación con resultados diferentes, ya que cada objeto tiene sus propios métodos para implementarla. En consecuencia, el resultado obtenido no depende sólo de la operación invocada sino de sobre qué objeto la invocamos.

OMG IDL

En CORBA, la interfaz de un objeto se define en un lenguaje llamado IDL (Interfece Definition Language). La definición de la interfaz especifica las operaciones que el objeto puede realizar, los parámetros de entrada y salida requeridos y cualquier excepción que puede generar. Esta interfaz constituye un contrato con los clientes del objeto, que la usan para realizar invocaciones, como así la usa la implementación del objeto para recibir y responder a esas invocaciones. De este modo el cliente y la implementación del objeto se encuentran aislados por al menos tres componentes: el *IDL stub* del lado del cliente, el ORB (o varios de ellos), y los correspondientes *server skeletons* del lado de la implementación del objeto. Este aislamiento provee gran flexibilidad y muchos beneficios.

De este modo se puede ver como CORBA fuerza el encapsulamiento: los clientes sólo acceden a los servicios de un objeto a través de su interfaz IDL. No existe posibilidad en la arquitectura de CORBA de permitir que el cliente acceda a la implementación del objeto en modo directo.

En la arquitectura de CORBA el escenario es este: para el cliente, la interfaz IDL representa una promesa: cuando envía un mensaje bien formado según la interfaz del objeto destino, la respuesta debe ser la correcta. Para el que implementa el objeto, la interfaz constituye una

obligación: el programador debe implementar todas las operaciones que se especifican en la interfaz para poder satisfacer los requerimientos del cliente.

Object Reference

Cada objeto CORBA en un sistema tiene su propia referencia. Y ésta es asignada por el ORB cuando se crea el objeto y permanece válido hasta que el mismo es explícitamente borrado. Los clientes obtienen la referencia al objeto de varios modos. El OMG da la libertad al creador de cada ORB para implementar esta referencia en el modo que más le convenga; siempre que cumplan con una propiedad muy importante: un cliente puede almacenar la referencia a un objeto para usarlo en un futuro, el objeto puede cambiar de lugar en el ínterin, lo que no debe impedir que el cliente pueda usar esa referencia para localizarlo. Por lo tanto la referencia al objeto no puede ser una dirección de memoria o una de red.

Conclusión

De estos conceptos se desprenden dos propiedades fundamentales de CORBA:

* Tanto el cliente como la implementación del objeto están aislados del *Object Request Broker* (ORB) por una interfaz IDL. CORBA requiere que toda interfaz de un objeto sea expresada en OMG IDL. Los clientes ven sólo la interfaz del objeto y nunca los detalles de su implementación. Esto garantiza la sustituibilidad de la implementación detrás de la interfaz.

* La invocación no pasa directamente del cliente a la implementación del objeto. En vez, las invocaciones son siempre manejadas por el ORB. Cada invocación de un objeto CORBA es pasada al ORB; la forma de la invocación es siempre la misma, sea el objeto local o remoto. Los detalles de la distribución de las implementaciones permanecen en el ORB. El código de la aplicación está liberado de detalles administrativos y se concentra en su propio problema.

La anatomía del ORB

El ORB es responsable de todos los mecanismos necesarios para:

⇒ Encontrar la implementación del objeto al cual un cliente ha hecho un pedido.

⇒ Preparar la implementación del objeto para recibir el pedido.

⇒ Comunicar los datos que se incluyen en el pedido.

Como se muestra en la figura 1.1.1, el Cliente (Client) es la entidad que desea realizar una operación sobre el objeto, y la Implementación del Objeto (Object Implementation) es la suma del código más los datos que implementan el objeto.

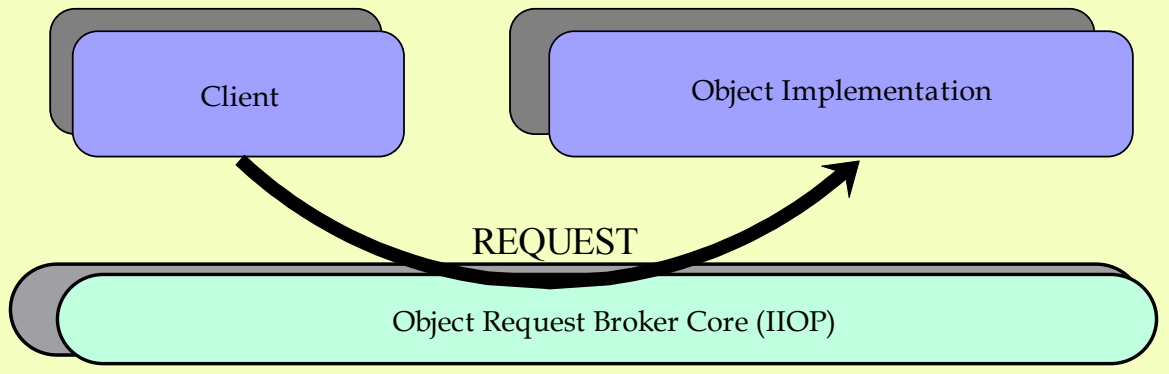

Figura 1.1.1 Object Request Broker, cliente y servidor

El cliente no sabe la ubicación del objeto al cual le realiza el pedido, desconoce los mecanismos por los cuales se comunica con él, o el modo en el que el objeto se activa o se almacena.

La siguiente es una pequeña lista de los beneficios que todo ORB provee:

- **Invocación dinámica y estática de métodos**. Un ORB permite definir estáticamente los métodos en tiempo de compilación (fuerte chequeo de tipos en tiempo de compilación), o permite descubrirlos dinámicamente en tiempo de ejecución.
- **Lenguaje de alto nivel**. Se pueden invocar métodos en los objetos del servidor usando un lenguaje de alto nivel a elección del programador. No interesa el lenguaje en el cual fueron implementados los objetos del servidor. CORBA separa la interfaz de la implementación y provee tipos de datos independientes del lenguaje, lo que hace posible llamar a objetos a través de distintos lenguajes y sistemas operativos.
- **Sistema que se auto-describe**. CORBA provee “run-time metadata” para describir la interfaz de un servidor conocido para el sistema. Cada ORB posee un Repositorio de Interfaces (Interface Repository) que contiene información en tiempo real que describe las funciones que provee un servidor y sus parámetros. Los clientes usan esta propiedad para descubrir cómo invocar servicios en tiempo de ejecución.
- **Transparecia local y remota**. Un ORB puede ejecutarse en modo “standalone”, o puede interconectarse con cualquier número de ORB’s usando los

servicios de CORBA 2.0 Internet Inter-ORB Protocol (IIOP). Un ORB puede derivar llamadas a objetos dentro de un solo proceso, a objetos en diferentes procesos en una misma máquina, o a múltiples procesos corriendo a través de redes y distintos sistemas operativos.

- **Mensajes polimórficos**. El ORB no invoca simplemente una función, sino que invoca una función en un objeto destino. Esto significa que la misma invocación puede tener diferentes efectos dependiendo del objeto que la recibe.

En la arquitectura, no se requiere que el ORB sea implementado como un componente simple, sino que debe ser definido por sus interfaces. Cualquier implementación de un ORB que provee las interfaces apropiadas, es aceptable. Las interfaces se pueden agrupar en tres categorías:

1. Operaciones que son las mismas para todas las implementaciones.
2. Operaciones que son específicas para tipos particulares de objetos.
3. Operaciones que son específicas a estilos particulares de implementaciones de objetos.

Distintos ORB's pueden tomar diferentes elecciones en cuanto a la implementación, y junto con los compiladores de IDL, repositorios, y varios "Object Adapters", proveer un conjunto de servicios para clientes e implementaciones de objetos que tienen diferentes propiedades y calidades. Pueden tener, por ejemplo, diversas representaciones para las referencias de los objetos y diferentes formas de realizar una invocación.

En la figura 1.1.2 vemos con mayor nivel de detalle la anatomía de un ORB, tanto la parte del cliente como la del servidor.

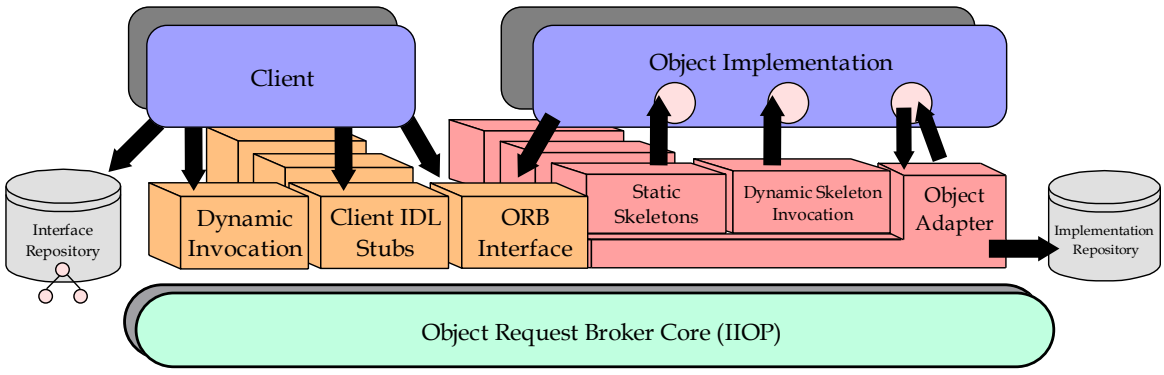

Figura 1.1.2 La estructura de CORBA 2.0

Object Request Broker Core

Es la parte del ORB que provee la representación básica de los objetos y la comunicación de llamadas.

Cliente

El cliente de un objeto tiene acceso a la referencia al mismo, e invoca las operaciones sobre ella. El cliente conoce sólo la estructura lógica del objeto de acuerdo con su interfaz, y sólo experimenta el comportamiento del objeto a través de sus invocaciones.

A continuación se describe lo que CORBA realiza del lado del cliente.

Client IDL Stubs

Proveen las interfaces estáticas de los servicios de los objetos. Estos stubs precompilados definen cómo los clientes invocan los servicios correspondientes en los servidores. Desde el punto de vista del cliente, el stub actúa como una llamada local, es un proxy local para el objeto en el servidor remoto. Los servicios se definen usando IDL, y se compilan usando el compilador de IDL.

El stub incluye código para codificar y decodificar la operación y sus parámetros en mensajes “chatos” que pueden ser enviados al servidor (marshaling).

Los stubs realizan llamadas al resto del ORB usando interfaces que son privadas a ellos.

La interfaz cliente-a-stub es definida por mapeo standard del OMG para el lenguaje elegido.

Dynamic Invocation Interface

Permite descubrir métodos en tiempo de ejecución para la invocación de los mismos. CORBA define API’s standard para que el cliente pueda conocer en tiempo de ejecución:

- ⇒ La referencia al objeto a ser invocado.
- ⇒ La operación a realizar.
- ⇒ El conjunto de parámetros que necesita para realizar la invocación del método.

El servidor desconoce si el cliente usó el modo estático para realizar la invocación o si usó DII.

En el capítulo IV se explicará con mayor detalle este mecanismo.

Object Adapters

Estamos ya del lado del servidor. Un object adapter es un componente que la implementación de un objeto utiliza para hacerse disponible a través de un ORB.

Los servicios prestados por el ORB a través de un object adapter incluyen:

⇒ Generación e interpretación de referencias de objetos.

⇒ Invocación de métodos.

⇒ Seguridad de interacciones.

⇒ Activación y desactivación de objetos e implementaciones.

⇒ Mapeo de referencias de objetos a implementaciones.

⇒ Registración de implementaciones.

En el Capítulo V veremos con profundidad este tema y los diferentes objects adapters que propone CORBA.

Server Skeletons

Los server skeletons son las interfaces estáticas para cada servicio exportado por el servidor.

Una vez que un pedido llega al servidor, debe existir un camino para invocar el método correcto en el objeto correcto. La decodificación del mensaje recibido por el servidor en una estructura válida para realizar la invocación, es una función del *skeleton* generado por el compilador IDL (*unmarshaling*).

Dynamic Skeleton Interface

Provee un mecanismo de *binding* en tiempo de ejecución para servidores que necesitan manejar llamadas entrantes a métodos para componentes que no tienen *skeletons* compilados por el compilador IDL.

El *dynamic skeleton* mira los valores de los parámetros del mensaje entrante para determinar cual es el objeto destinatario del mismo y el método correspondiente. Constituye el equivalente al DII en la parte del cliente.

Object Implementations

La implementación de un objeto provee la semántica del mismo, definiendo los datos para la instancia del objeto y el código de sus métodos.

Puede ser soportada una variedad de implementaciones de objetos, incluyendo servidores separados, librerías, un programa por método, una aplicación encapsulada, una base de datos orientada a objetos, etc.

Generalmente, la implementación de un objeto no depende del ORB o de como el cliente invoca operaciones sobre el objeto, pero se pueden seleccionar interfaces a los servicios dependientes del ORB mediante la elección de *Object Adapters*.

ORB Interface

Interactúa tanto con el cliente como con el servidor. Es la interfaz que va directamente al ORB y es igual para todos los ORB's. No depende ni de la interfaz de los objetos ni del *object adapter*.

Como la mayor parte de la funcionalidad del ORB se encuentra a través del object adapter, stubs, skeletons o dynamic invocation, hay solamente pocas operaciones que son comunes a través de todos los objetos y son útiles tanto para los clientes como para la implementación de los objetos. Las operaciones que provee serán explicadas en el Capítulo III.

Repositorio de Interfaces

El repositorio de interfaces es crucial para el funcionamiento de CORBA. CORBA requiere que cada ORB soporte e implemente la interfaz de IR (Interface Repository), permitiendo a las definiciones IDL ser almacenadas, modificadas y devueltas. Estas definiciones pueden ser usadas para diferentes propósitos:

⇒ Proveer interoperabilidad entre diferentes implementaciones de ORB's.

⇒ Proveer chequeo de tipos en las invocaciones de los clientes.

⇒ Chequear la corrección de los grafos de herencia.

Además la información puede ser útil para los objetos clientes y usuarios para:

⇒ Manejar la instalación y distribución de las interfaces a través de la red.

⇒ En el proceso de desarrollo, por ejemplo, para navegar a través de las definiciones de interfaces o modificarlas.

⇒ Los compiladores de los lenguajes podrían compilar los *stubs* y *skeletons* directamente desde el IR en vez que desde los archivos IDL.

Existen dos modos de acceder al repositorio de interfaces. Uno es usar utilidades proveídas por el vendedor del ORB. Otra es a través de la interfaz standard del IR definida por el OMG.

No debe asumirse que existe una correspondencia uno a uno entre ORB's e IR's. Las especificaciones del OMG son flexibles en este sentido y sólo requieren que cada ORB tenga acceso al menos a un IR; con lo cual, un ORB puede tener acceso a más de un IR, o un IR puede ser compartido por más de un ORB.

Los IR's son reconocidos por los ORB's por un identificador: el RepositoryID. Los ORB's asumen que dos repositorios con el mismo identificador, poseen exactamente el mismo contenido.

La implementación de un IR requiere alguna forma de almacenamiento permanente de objetos, pero el OMG da la libertad a los implementadores de ORB's de elegir el método que mejor se adapte a sus necesidades.

Capítulo II : Especificaciones IDL

El Interface Definition Language (IDL) es un lenguaje que se usa para describir las interfaces de los objetos que serán llamados por los clientes. La definición de una interfaz escrita en IDL, define y especifica completamente la interfaz del objeto con sus operaciones y parámetros. Esta especificación determina la visión que el cliente tiene del objeto y las operaciones que puede invocar sobre el mismo.

La especificación IDL debe ser procesada por un "compilador" que mapea esta interfaz a un lenguaje de programación determinado. El OMG especifica cómo debe llevarse a cabo dicho mapeo para cada lenguaje (C, C++, Smalltalk, Java, Ada, y Cobol). Como salida de esta "compilación" se generan los Client Stubs y Server Skeletons (como se ve en la figura 1.2.1).

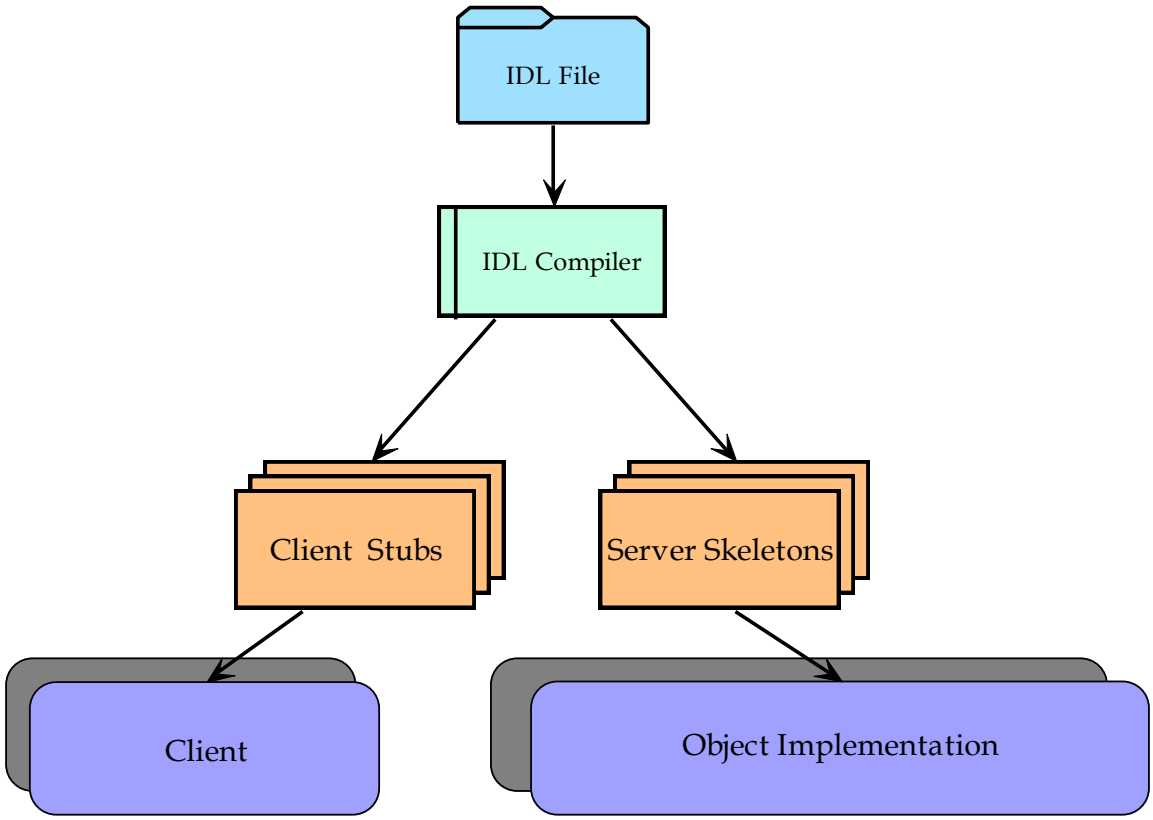

Figura 1.2.1 Función del IDL Compiler

En este capítulo se explicarán la sintaxis y semántica de este lenguaje.

Análisis lexical

Identificadores: deben comenzar con una letra y ser seguidos por cero o más letras, números o underscores. La única característica poco común es que los identificadores son "case sensitive" pero no pueden coexistir con otros identificadores que difieren solo en una mayúscula o minúscula.

Preprocesamiento: Se usa el preprocesamineto standard de C++. Se incluyen el #include, #define, #ifdef, #pragma.

Palabras Clave: las palabras clave son sólo en minúsculas y otros identificadores no se pueden usar con ese nombre aunque estén con mayúsculas.

Comentarios: se usan los dos estilos de C++ (/* ...comentario..*/, y // para una línea simple).

Puntuación: las llaves se usan para abrir y cerrar bloques y son siempre seguidas por un punto y coma, como así también cada declaración. Los parámetros son encerrados entre paréntesis y separados entre ellos por comas.

Módulos e Interfaces

El propósito del IDL es definir interfaces y sus operaciones. Para evitar el conflicto de nombres cuando se utilizan varias declaraciones IDL juntas, se usa el *module* que define un alcance de nombres. Los mismos incluyen declaraciones IDL bien formadas y se pueden anidar.

Las interfaces abren también un nuevo alcance de nombres y pueden contener constantes, declaraciones de tipos de datos, atributos y operaciones.

Cualquier nombre de interfaz en un alcance de nombres, puede ser usado como nombre de tipo, y se pueden referenciar nombres en otros alcances usando la sintaxis de C++ para dicho propósito ("::"). Por ejemplo:

```
module outer {
        module inner{ //módulo anidado
                Interface inside{};
}
interface outside { //puede referenciar a inner como un nombre local
        inner::inside get_inside();
}
```

La operación get_inside() retorna la referencia a un objeto del tipo ::outer::inner::inside.

Las interfaces pueden ser referenciadas mutuamente, esto significa que las declaraciones en cada interfaz pueden usar el nombre de otras interfaces como tipo. Para evitar errores de

compilación, una interfaz puede ser declarada antes y especificada después como veremos en el siguiente ejemplo:

```
interface A; //especificada luego
interface B {
        A get_an A();
};
interface A {
        B get_a_B();
};
```

Cuando una declaración en un módulo necesita algunas referencias mutuas a una declaración en orto módulo, esto se consigue cerrando el primer módulo y reabriéndolo después en otras declaraciones:

```
module X {
        interface A; // especificada luego
}; // cierro el módulo y A debe ser especificada luego

module Y {
        interface B { B puede usar X::A como nombre de tipo
                X::A get_an_A();
        };
};

module X { // el módulo se reabre
        interface C {
                A get_an_A(); // A puede ser usado como nombre de tipo si usar ::
                                //ya que   está en el mismo alcance
        };
        interface A { // A puede usar Y::B como nombre de tipo
                Y::B get_a_B();
        };
};
```

Herencia

El conjunto de operaciones ofrecidos por una interfaz puede ser extendido declarando una nueva interfaz que hereda de una ya existente. La interfaz existente es llamada *interfaz base* y la nueva interfaz se llama *interfaz derivada*. La herencia se declara usando dos puntos después del nombre de la interfaz, seguido del nombre de la interfaz base:

```
module ejemplo_de_herencia{
        interface A {
                typedef unsigned short ushort;
                ushort op1();
        };
        interface B : A {
                boolean op2(ushort num);
        };
};
```

En este ejemplo, la interfaz B extiende a la interfaz A. La declaración de tipos también es heredada, permitiendo el uso de ushort como el tipo de un parámetro en op2(). Todas las interfaces heredan de CORBA::Object.

Herencia Múltiple

Una interfaz puede heredar de varias otras. La sintaxis es la misma que en el caso de herencia simple, y las interfaces base son separadas por comas.

Los nombres de las operaciones en cada una de las interfaces heredadas (incluyendo las operaciones que heredan de otras interfaces) deben ser únicas y no pueden ser redeclaradas en interfaces derivadas. La excepción a esta regla es cuando las operaciones son heredadas en dos o más clases de la misma interfaz base. Esto se conoce como *diamond inheritance*:

```
module ejemploDiamondInheritance{

        interface Base {
                string BaseOp();
        };
```

```
interface Izquierda : Base {
        short IzqOp(in string izqParam);
};

interface Derecha : Base {
        any DerOp(in long derParam);
};

interface Derivada : Izquierda, Derecha {
        octect DerivadaOp(in float DerInParam,
                                out unsigned long DerOutParam);
};
```

La figura 1.2.2 muestra la especificación IDL en modo gráfico.

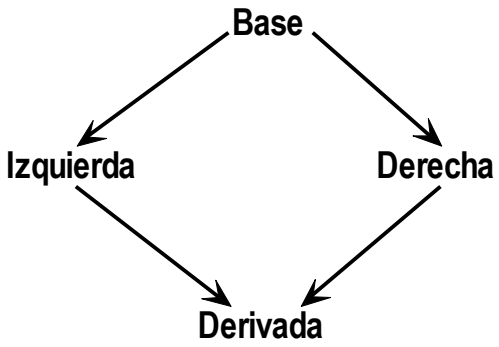

Figura 1.2.2 Herencia Múltiple

Tipos y Constantes

Los nombres de las interfaces en IDL se convierten en nombres de tipos de objetos que pueden ser usados como el tipo de un parámetro en cualquier operación o valor de retorno de una función, o como miembro en un tipo estructurado de datos.

Los tipos básicos son lo suficientemente ricos como para representar números, strings, caracteres y booleanos. Las definiciones son lo suficientemente precisas como para permitir *marshaling* no ambiguo.

Los tipos estructurados disponibles son: estructuras, uniones discriminadas, arreglos y secuencias. Las excepciones pueden ser consideradas como tipos especiales de estructuras que son sólo usadas en sentencias *raises*.

La siguiente es una tabla de los tipos básicos de datos que provee IDL:

Palabras clave	**Descripción**
[unsigned] short	Enteros de 16 bits en complemento a base dos, con o sin signo
[unsigned] long	Enteros de 32 bits en complemento a base dos, con o sin signo
float	Número en punto flotante de 16 bits.
Double	Número en punto flotante de 32 bits.
Char	Carácter ISO Latin-1
boolean	Tipo booleano que toma los valores TRUE y FALSE
string	Cadena de caracteres de largo variable
octet	Tipo sin interpretación de 8 bits
enum	Tipo enumerativo.
Any	Puede representar cualquier valor de los posibles en IDL, básicos o construidos, de objetos o no.

La palabra clave typedef permite crear alias para cualquier declaración legal de tipos.

A continuación se detallan las propiedades de los tipos estructurados y el tipo *any*.

Any

Este tipo tiene una API definida en pseudo-IDL que describe cómo los valores son insertados y extraídos de él, y cómo el tipo de sus valores contenidos pueden ser descubiertos.

Estructuras

Son declaradas con la palabra clave *struct*, que tiene que ser seguida de un nombre. Este nombre se puede usar luego como un nombre de tipo. Ejemplo:

```
interface Vidrios    {
        enum vid_color{transparente, gris, blanco, verde}
        struct vidrio_spec {
                vid_color color;
                float   ancho;
                float   largo;
        }
}
```

Uniones discriminadas

Son declaradas con la palabra clave *union* seguida por un nombre, que también puede ser usado como un nombre de tipo. La palabra clave *switch* va a continuación del nombre de tipo y es parametrizada por un tipo escalar (*integer*, *char*, *boolean*, *enum*), que actúa como discriminante. El cuerpo de la unión se encierra entre llaves y contiene un número de sentencias case. Ejemplo:

```
enum tipo_cerradura (ventana, puerta);
union cerradura switch (tipo_cerradura){
        case ventana: ventana_spec vent;
        case puerta: puerta_spec prta;
        default: float altura;
}
```

El caso *default* no es obligatorio pero no puede repetirse más de una vez. En cada mapeo a un lenguaje existe un modo de acceder al valor del discriminante para determinar qué campo de la unión contiene un valor. El valor de la unión consiste del valor del discriminante y el del elemento que nomina.

Secuencias

Las secuencias son tipos *templates*. Esto significa que sus declaraciones nombran otros tipos que serán contenidos por la secuencia. Una secuencia es una colección ordenada de ítems que puede crecer en tiempo de ejecución. Sus elementos son accedidos por un índice. Pueden o no tener un límite fijo. Todas las secuencias tienen dos características en tiempo de ejecución: una longitud máxima y una actual. La longitud máxima de las secuencias con número fijo de ítems se conoce en tiempo de compilación. La ventaja de las secuencias es que solamente el numero de ítem corriente es transmitido al objeto remoto cuando una secuencia es pasada como parámetro.

Las secuencias deben declararse con un alias usando typedef para poder ser usadas como tipos en los parámetros de una operación o como valor de retorno de las mismas. Ejemplo:

```
typedef sequence <cerradura> cerraduraSeqSinLimites;
typedef sequence <cerradura, 10> cerraduraSeqConLimite;
```

También se pueden declarar secuencias de secuencias con el siguiente formato:

```
typedef sequence <sequence <cerradura>, 10> cerraduraSeqSeq;
```

Arreglos

Son usualmente declarados con typedef, y deben tener un nombre antes de ser usados como parámetros o valores de retorno. Pueden ser incluidos como elementos de una unión, o como miembros de una estructura.

En tiempo de compilación tienen una longitud fija. Se declaran de la siguiente manera:

```
typedef ventana[10] ventanaArr;
typedef ventana[3][10] ventanaMatriz;
```

Excepciones

Son declaradas de la misma manera que las estructuras usando al palabra clave **exception** en vez de struct.

Es una buena práctica incluir los valores que provocaron la falla como argumentos de la excepción. De este modo el manejador de la excepción puede ser un manejador general que no sabe cuáles fueron los argumentos que provocaron la excepción. El manejador puede determinar el contexto de la operación que produjo la excepción por el valor de sus argumentos.

Constantes

Pueden ser declaradas con alcance global o dentro de módulos e interfaces. La declaración comienza con una palabra clave *const* y los tipos que puede tomar son numéricos, caracteres, strings o booleanos.

```
const short maximo = 100;
```

Operaciones y atributos

Las operaciones requieren tres partes en sus declaraciones, más tres partes opcionales. Las primeras tres partes requeridas son: el tipo de retorno de la operación, el nombre de la misma y la lista de parámetros.

El tipo de retorno debe ser void cuando no se debe retornar un tipo específico de datos. Los parámetros pueden ser *in*, *out* o *inout*.

Las operaciones pueden declararse como *oneway*, en cuyo caso el tipo de retorno debe ser *void*. Las operaciones no declaradas como *oneway*, pueden retornar valores ya sea como valores de retorno de la operación como en parámetros *out* o *inout*.

Se pueden incluir en las operaciones las excepciones que la misma puede producir usando la palabra clave *raises*.

Ejemplos:

```
boolean OrdenDeVenta(in short artCod, in string descripcion, out long precio)
        raises (ArtNoDefined);

oneway void EnviarMensaje (in string mensaje);
```

Los atributos son equivalentes a un par de funciones accesorias, una para acceder al valor y otra para modificarlo. Los atributos *readonly* solo tienen función de lectura del valor.

Es más simple declarar un atributo que las operaciones y para esto se utiliza la palabra clave *attribute* seguida por el tipo del mismo y una lista de nombres de atributos.

Ejemplos:

```
readonly attribute short artCod;
attribute long precio;
```

Estas declaraciones equivalen a especificar las siguientes operaciones:

```
short artCod();
long get_precio();
set_precio(in long precio);
```

Contextos

Un objeto *context* contiene una lista de pares de nombres llamados propiedades. Actualmente el OMG restringe los valores a strings. Son equivalentes a las variables de enviroment de UNIX o DOS.

Algunos ambientes pueden tener un contexto para el sistema, otro para cada usuario y otro para cada aplicación. Solamente el ORB puede crear contextos, por lo que existe una operación para ello.

En IDL se declara un contexto para una operación del siguiente modo:

```
context(context1, context2,...);
```

Capítulo III: ORB Interface

La ORB Interface está formada por aquellas funciones que no dependen del *object adapter* usado. Estas operaciones son las mismas para todos los ORB's y todas las implementaciones de objetos. Algunas de ellas parecen ser operaciones sobre el ORB, otras sobre las referencias de los objetos. Como estas funciones están implementadas sobre el ORB mismo, no son, de hecho, operaciones sobre objetos; aunque pueden ser descriptas en ese modo y el *binding* del lenguaje, por consistencia, las hará parecer de esa manera. La interfaz del ORB define también operaciones para crear listas y determinar el contexto usado en la *Dynamic Invocation Interface* (estas operaciones se describen en el capítulo IV).

Inicialización del ORB y referencias iniciales

Antes de que una aplicación pueda entrar en el ambiente CORBA, primero debe:

⇒ Ser inicializada en el ORB y posiblemente en ambientes del *object adapter* (OA*)*.

⇒ Obtener referencias al pseudo-objeto ORB y quizás de otros *object adapters*.

Existe un orden para estas operaciones, ya que no se puede llamar a la inicialización de un OA sin antes haber inicializado el ORB. La operación de inicializar una aplicación en el ORB y obtener una referencia al pseudo-objeto ORB no se realiza sobre un objeto. Esto es porque las aplicaciones no tienen inicialmente un objeto sobre el cual realizar operaciones.

La inicialización se define en pseudo-IDL (PIDL); esto es, una operación sobre el pseudo-objeto ORB.

```
// PIDL
module CORBA {
        typedef string ORBid;
        typedef sequence <string> arg_list;
        ORB ORB_init (inout arg_list argv, in ORBid orb_identifier);
};
```

Para obtener la referencia al ORB la aplicación llama a **ORB_init**. Los parámetros contienen un identificador del ORB y una lista que se usa para pasar argumentos específicos del ambiente.

El identificador del ORB es del tipo **CORBA::ORBid**. Es un string no vacío que se establece por el administrador del ORB y no lo maneja el OMG.

Si no se le indican argumentos a la operación, se devuelve el ORB por omisión.

Para conseguir referencias iniciales a objetos se opera sobre el pseudo-objeto ORB, que da facilidades para listar y resolver referencias iniciales a objetos. El PIDL para estas operaciones es:

```
// PIDL interface for getting initial object references
module CORBA {
        interface ORB {
                typedef string ObjectId;
                typedef sequence <ObjectId> ObjectIdList;
                exception InvalidName {};
                ObjectIdList list_initial_services ();
                Object resolve_initial_references (in ObjectId identifier)
                raises (InvalidName);
        };
};
```

ObjectIDs son strings que identifican un objeto cuya referencia es requerida. Son pocos los objetos que se pueden obtener mediante este método, por motivos de simplicidad de la interfaz. Actualmente, los ObjectIDs reservados para el CORBA Core son **RootPOA**, **POACurrent** e **InterfaceRepository**; para CORBA Services están **NameService**, **TradinService**, **SecurityCurrent**, **TransactionCurrent**.

Para que la aplicación pueda determinar cuáles son los servicios disponibles, se cuenta con la operación **list_initial_services**. Esta operación retorna una lista de ObjectIDs.

La operación es la responsable de hacer el "*narrowing*" de la referencia al objeto retornada por **resolve_initial_references** al tipo que fue requerido en el ObjectId. El *narrowing* consiste en convertir un objeto de un tipo genérico a un tipo más especifico derivado del original. Por ejemplo, para **InterfaceRepository**, el objeto devuelto debe ser convertido al tipo **Repository**.

Convertir referencias de objetos a strings

Como la referencia de los objetos puede cambiar de ORB a ORB, no es un dato apropiado para ser almacenado en forma persistente. Existen dos problemas a ser resueltos: permitir que la referencia al objeto sea convertida a un valor que el cliente pueda guardar, y asegurar que ese valor puede ser reconvertido en la apropiada referencia.

Una referencia a un objeto puede ser convertida a un string usando la operación **object_to_string.** El valor puede ser almacenado o comunicado en el mismo modo que cualquier string. La operación **string_to_object** acepta el string producido por **object_to_string** y devuelve la correspondiente referencia al objeto.

Existe otro caso en el cual es muy útil utilizar esta función del ORB, y este es cuando el servidor genera un objeto al cual el cliente quiere referenciar. El servidor pasa la referencia al objeto en formato de string y el cliente utiliza la operación string_to_object para obtener dicha referencia.

Operaciones sobre referencias a objetos

Existen operaciones que pueden hacerse sobre cualquier objeto. No son operaciones en el sentido normal, ya que son implementadas directamente sobre el ORB y nunca llegan a la implementación del objeto. La siguiente es una lista de estas operaciones con una breve descripción sobre su utilidad.

⇒ **Determinar la interfaz del objeto**

La operación **get_implementation** será despreciada en futuras especificaciones de CORBA, y sirve para trabajar en ambientes sin chequeo de tipos en tiempo de compilación.

⇒ **Duplicar y liberar copias de referencias a objetos**

Como las referencias a los objetos son opacas y dependientes del ORB, no es posible para los clientes alocar espacio para ellas. Por ello existen operaciones para copiar y liberar referencias:

```
Object duplicate();   //PIDL
void release();
```

Ninguna de estas operaciones afectan a la implementación del objeto, y la misma no puede saber si una referencia original o duplicada fue usada para su acceso.

⇒ **Referencias a objetos nulas**

Se puede determinar si una referencia a un objeto es nula invocando la operación **is_nil().**

⇒ **Operación de chequeo de equivalencia**

La operación

```
boolean is_a(in RepositoryID logical_type_id); //PIDL
```

se define para facilitar el manejo seguro de tipos. EL logical_type_id es un string que denota un identificador de tipo. La operación retorna true, si el objeto es realmente una instancia de ese tipo (incluyendo si el objeto es una instancia de un tipo derivado).

* **Probar la no existencia**

La operación **non_existent()** puede ser usada para testear si un objeto ha sido destruido. Esto se hace sin invocar una operación al nivel del objeto, por lo cual nunca lo afectará.

⇒ **Identidad de la referencia al objeto**

Para manejar eficientemente entornos que incluyen gran cantidad de referencias, los servicios necesitan soportar la noción de identidad de referencias a objetos. Son dos las operaciones que proveen esta funcionalidad.

```
unsigned long hash(in unsigned long maximum); // PIDL
```

Las referencias están asociadas a los identificadores internos del ORB, que pueden ser indirectamente accedidos por las aplicaciones usando la operación hash. El valor de este identificador no cambia a lo largo del tiempo de vida de la referencia. El valor de esta operación puede no ser único, es decir, otra referencia puede retornar el mismo valor de hash. De todos modos, sirve para determinar que dos referencias no son idénticas con certeza.

La operación **is_equivalent()** sirve para determinar si dos referencias son equivalentes. Si dos referencias son idénticas, son equivalentes. Si dos referencias diferentes refieren al mismo objeto, también lo son.

⇒ **Obtener la política y el manejador de dominio asociados al objeto**

Para esto se utilizan las operaciones **get_policy()** y **get_domain_managers()**

Objeto Current

Los servicios de CORBA pueden desear tener acceso a la información de contexto asociada a un thread de ejecución en el que están corriendo. Esta información se accede en forma estructurada a través de interfaces derivadas de **Current**, en el módulo **CORBA**.

Cada servicio que necesita su propio contexto, deriva una interfaz del módulo **Current**. Los usuarios de este servicio pueden obtener una instancia de la interfaz Current apropiada invocando **ORB::resolve_initial_references**.

Las operaciones sobre interfaces derivadas de **Current**, acceden al estado asociado con el thread en el que son invocadas, no con el estado asociado al thread del cual se obtuvo la interfaz **Current**. Esto previene que un thread manipule el estado de otro thread.

Objeto Policy

Un servicio de CORBA puede permitir el acceso a ciertas elecciones que afectan su funcionamiento. Esta información se accede de manera estructurada a través del uso de interfaces derivadas de **Policy** en el módulo **CORBA**. Un ejemplo de esto es *"Security Service"* que usa esta técnica para asociar políticas de seguridad a objetos del sistema.

```
module CORBA {
        typedef unsigned long PolicyType;
        // Basic IDL definition
        interface Policy{
                readonly attribute PolicyType policy_type;
                Policy copy();
                void destroy();
        };
        typedef sequence <Policy> PolicyList;
};
```

PolicyType define el tipo del objeto **Policy**. Los valores de PolicyTypes son establecidos por el OMG.

Manejo de Policy Domains

En esta sección se describen los conceptos de políticas y dominios.

* **Policy Domain**

Es un conjunto de objetos a los cuales se les aplica una política de dominio. Estos objetos son los miembros del dominio. Las políticas representan reglas y criterios que limitan las actividades de los objetos que pertenecen al dominio. Cuando se crea un objeto, el ORB implícitamente asocia al objeto con una o más políticas de dominio.

* **Policy Domain Manager**

Un dominio de políticas incluye a un único objeto (uno por dominio) llamado **domain manager,** que es asociado con los objetos policy de ese dominio. El manejador de dominio también controla la pertenencia de objetos al dominio, y provee métodos para agregar y remover miembros.

* **Policy Objects**

Un objeto policy encapsula una política de un tipo específico. La política encapsulada en uno de estos objetos es asociada con el dominio mediante la asociación del objeto **policy** con el **domain manager** del dominio.

Pueden haber muchas políticas asociadas a un dominio, con un objeto **policy** por cada una. Existe al máximo una política de cada tipo asociada a un dominio. Los objetos **policy** son entonces compartidos por los objetos en el dominio, en vez de existir uno asociado a cada objeto individual.

* **Pertenencia de objetos a los Policy Domains**

Un objeto puede pertenecer simultáneamente a más de un dominio de políticas. En ese caso el objeto es gobernado por todas las políticas de los dominios a los que pertenece. El modelo de referencia permite que un objeto sea miembro de múltiples dominios, que se pueden superponer para el mismo tipo de política. Esto requiere que los conflictos entre políticas que se superponen sean resueltos, pero la especificación de CORBA no incluye soporte específico para esto.

Los manejadores de políticas de dominios y los objetos **policy** tienen dos tipos de interfaces:

- Las interfaces operacionales usadas cuando se imponen las políticas. Estas interfaces son usadas por el ORB durante la invocación al objeto. El que invoca, pide una política de un determinado tipo y usa la política retornada para imponerla.

- Las interfaces administrativas usadas para establecer políticas (por ejemplo qué eventos auditar, o quién puede acceder objetos de un determinado tipo en un dominio).

* **Asociación de Dominios en la creación del objeto**

  Cuando un nuevo objeto es creado, el ORB implícitamente lo asocia con los siguientes elementos:

- Uno o más **Policy Domains**, definiendo todas las políticas de las cuales el objeto es sujeto.
- Los *"dominios tecnológicos"*, que caracterizan variantes particulares de los mecanismos disponibles en cada ORB.

La especificación de CORBA define todas las operaciones necesarias para el manejo de los conceptos vistos, por lo cual no se describirán en este trabajo.

Capítulo IV: Dynamic Invocation Interface

La interfaz de invocación dinámica (DII) le da al cliente la capacidad de invocar cualquier operación, en cualquier momento, a cualquier objeto al cual puede tener acceso en una red. Esto incluye objetos para los cuales no existen *stubs* (objetos recientemente agregados a la red o descubiertos por los servicios de *Naming* o *Trading*). Se puede programar exclusivamente en DII algo que no se pudo hacer en SII (Static Invocation Interface) sin restringir el acceso a objetos nuevos.

Semántica de las operaciones

Las operaciones vía SII son sincrónicas al menos que se defina la operación como **oneway**. Las operaciones sincrónicas bloquean al cliente, y no retornan hasta que el ORB pueda despachar al cliente la respuesta y los valores de retorno del objeto invocado o una excepción. DII provee mecanismos para el sincronismo diferido. Esto es, el cliente puede invocar una operación sobre un objeto y retornar en seguida para realizar algún otro procesamiento mientras la invocación es transmitida y ejecutada.

En los siguientes gráficos veremos la diferencia entre SII y DII.

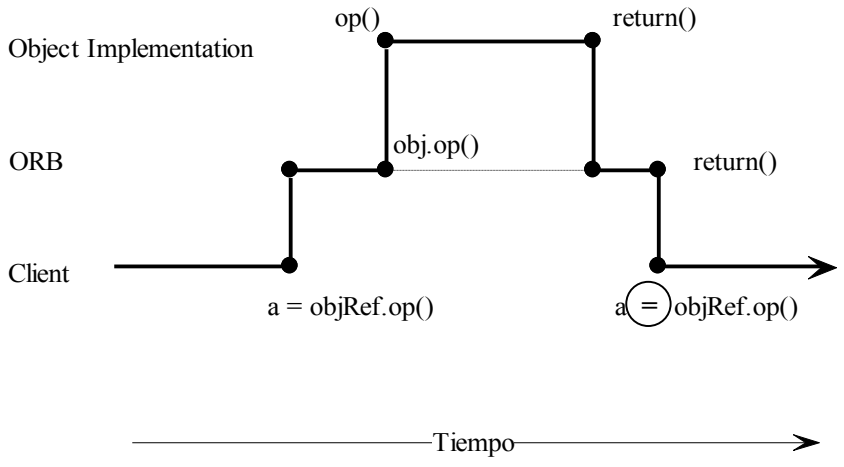

Figura 1.4.1 Semántica de las operaciones en SII

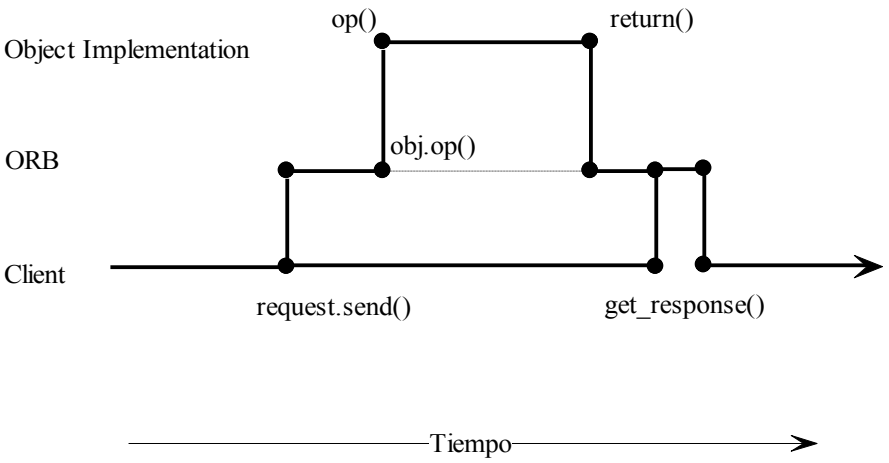

Figura 1.4.2 Semántica de las operaciones en DII

Se pueden invocar operaciones en modo sincrónico usando DII, con la operación **invoke**. En consecuencia, usando DII podemos usar tanto operaciones sincrónicas como asincrónicas, mientras que en SII sólo existen las primeras.

Cómo construir una invocación dinámica

Se deben seguir cuatro pasos para realizar una invocación dinámica:

1. **Identificar al objeto que se desea invocar**. En un ambiente de objetos distribuidos dinámicos, los usuarios probablemente consultarán un servicio de *Trading* para localizar las implementaciones de los objetos. El servicio lo veremos en el Capítulo VII.
2. **Obtener su interfaz**. Gracias al servicio de *Trading*, ya identificamos al objeto que deseamos, pero necesitamos conocer su interfaz para poder construir una invocación. Para esto, el ORB ofrece una función llamada **get_interface**, que no retorna la interfaz en sí misma sino que retorna una referencia a un objeto del tipo **InterfaceDef**, que es un objeto dentro del repositorio de interfaces (IR).

   El objeto **InterfaceDef** lo podemos usar como punto de entrada al IR para conocer su contenido. Podemos obtener toda clase de información detallada de la interfaz y los métodos que soporta. CORBA especifica diez operaciones para navegar por el IR y describir los objetos que contiene.

3. **Construir la lista de argumentos**. Para construir la invocación necesitamos construir una lista de argumentos. CORBA provee estructuras de datos para pasar estos parámetros, esta lista se llama *Names Value List* (*NVList*). La lista de argumentos la construimos usando un pseudo-objeto *NVList*. La lista se puede crear invocando **create_list** y tantos **add_item** como sean necesarios según la cantidad de argumentos requeridos por la operación a invocar. Alternativamente se puede dejar que el ORB cree esa lista por nosotros invocando **create_operation_list** sobre un objeto CORBA::ORB. A este llamado se le debe indicar la operación de la cual queremos obtener la lista de parámetros.
4. **Construir la invocación.** La invocación se construye usando un pseudo-objeto del tipo **Request**, que contiene el nombre del método, la lista de argumentos, y el valor de retorno. Para generar este pseudo-objeto llamamos a **create_request**, pasando como parámetros el nombre del método, la **NVList** y un puntero al valor de retorno. También se puede usar una forma abreviada que es llamando a **_request** y pasando como parámetro sólo el nombre del método. Esta última forma se utiliza cuando la operación a invocar no necesita argumentos.
5. **Invocar la operación.** Existen tres formas de hacerlo:
   - ⇒ Usando **invoke**, que envía el pedido y obtiene los resultados.
   - ⇒ Usando **send_deferred**, que es el modo asincrónico que ya vimos en la sección anterior. Esta llamada retorna el control a la aplicación que luego debe pedir los resultados usando **poll_response** (para saber si los resultados ya están disponibles) y **get_response**.
   - ⇒ En el caso de que no sea necesaria una respuesta se puede utilizar la operación **send_oneway**.

Como hemos visto, lleva bastante más trabajo realizar una invocación dinámica, pero es el precio que se debe pagar para contar con una mayor flexibilidad.

Interfaces de la invocación dinámica

Los servicios que se necesitan para construir una invocación dinámica forman parte del *CORBA Core.* Pero estos métodos se encuentran dispersos en cuatro interfaces en el módulo CORBA. Estas interfaces son las siguientes:

- **CORBA::Object** es la interfaz de un pseudo-objeto que define las operaciones que todo objeto CORBA debe soportar. Es la interfaz raíz de todos los objetos CORBA. Incluye tres métodos que se pueden usar para construir una invocación dinámica. Llamamos a **get_interface** para obtener la interfaz que soporta el objeto, y **create_request** o **_request** para generar un objeto del tipo **Request**.
- **CORBA::Request** es la interfaz de un pseudo-objeto que define las operaciones sobre un objeto remoto. Pertenecen a esta interfaz la definición de las operaciones: **add_arg**, **invoke**, **send_deferred**, **poll_response**, **get_response** y **send_oneway**. Llamamos a la operación **delete** para borrar el objeto **Request** de memoria.
- **CORBA::NVList** es la interfaz de un pseudo-objeto que nos ayuda a construir la lista de argumentos. Un objeto **NVList** mantiene una lista de datos que lo autodescriben llamados NamedValues, cuya interfaz es:

```
struct NamedValue {
        Identifier name; // nombre del argumento
        any argument; // argumento
        long len; //cantidad d elementos de la lista
        Flags arg_modes;// banderas que indican el modo en el que se pasa el argumento
};
```

Se pueden invocar: **add_item**, para agregar un elemento a la **NVList**; **add_value** para establecer su valor; **get_count** para saber el número de ítems que hay en la lista; **remove** para eliminar un elemento; **free_memory** para liberar memoria cuando se elimina un ítem, y **free** para liberar la memoria que la estructura entera no usa, llamando a **free_memory** por nosotros por cada ítem que contenga.

* **CORBA::ORB** es la interfaz del pseudo-objeto que define métodos del ORB de propósito general. Estos métodos pueden ser invocados tanto desde el cliente como desde la implementación del objeto en el servidor. Son seis los métodos de esta interfaz que se utilizan para generar invocaciones dinámicas. El método **create_list**, para generar una lista de elementos **NVList** vacía, o **create_operation_list** para que el ORB la genere por nosotros. El ORB provee cuatro operaciones para enviar la operación remota: **send_multiple_request_oneway** para enviar múltiples pedidos de los cuales no esperamos respuestas; **send_multiple_request_deferred** para enviar múltiples pedidos de los cuales sí esperamos respuestas; y utilizamos **poll_next_response** para saber si hay respuestas y **get_next_response** para obtenerlas.

Capítulo V: Object Adapters

Como ya vimos en la breve descripción del Capítulo I, un Object Adapter (OA) es responsable de:

- registrar implementaciones
- generar e interpretar referencias a objetos
- mapear referencias a objetos en sus respectivas implementaciones
- activar y desactivar implementaciones de objetos
- invocar operaciones a través del **skeleton** o vía DSI
- coordinar la seguridad en la interacción

El OMG no desea la proliferación de muchos tipos de **Object Adapters**. Para evitar esto, en la especificación de CORBA 2.0, el OMG describe el **Basic Object Adapter** (BOA), y en la versión 3.0 introduce el **Portable Object Adapter** (POA).

Basic Object Adapter

Es un adaptador bastante flexible y diseñado inicialmente para servidores que residen en sus propios procesos y separados del ORB.

BOA diferencia entre un servidor, que corresponde a una unidad de ejecución, y un objeto, que implementa un método o una interfaz.

CORBA::BOA Interface	Descripción de la operación
create	Crea un objeto y retorna su referencia.
change_implementation	Actualiza la información de la implementación de un objeto existente.
get_id	Retorna la referencia (en un formato específico).
Dispose	Destruye la referencia al objeto.
set_exception	Informa al ORB que algo no anduvo bien.
impl_is_ready	Activa la implementación del objeto.
obj_is_ready	Activa el objeto.
deactivate_impl	Desactiva la implementación.
deactivate_obj	Desactiva el objeto.

La interfaz BOA soporta cuatro políticas diversas de activación, cubriendo las diferentes configuraciones de servidores y objetos: *shared server policy, unshared server, server-per-method y persistant server.*

BOA Shared Server

En una política de activación **shared server**, múltiples objetos pueden residir en el mismo proceso. BOA activa el servidor la primera vez que una invocación es realizada sobre un objeto implementado por el mismo. Después de que el servidor ha sido inicializado, notifica al BOA de que está preparado para recibir pedidos invocando a *impl_is_ready()*. Todos los pedidos siguientes serán enviados al proceso del servidor, BOA no activará otro proceso servidor para esta implementación.

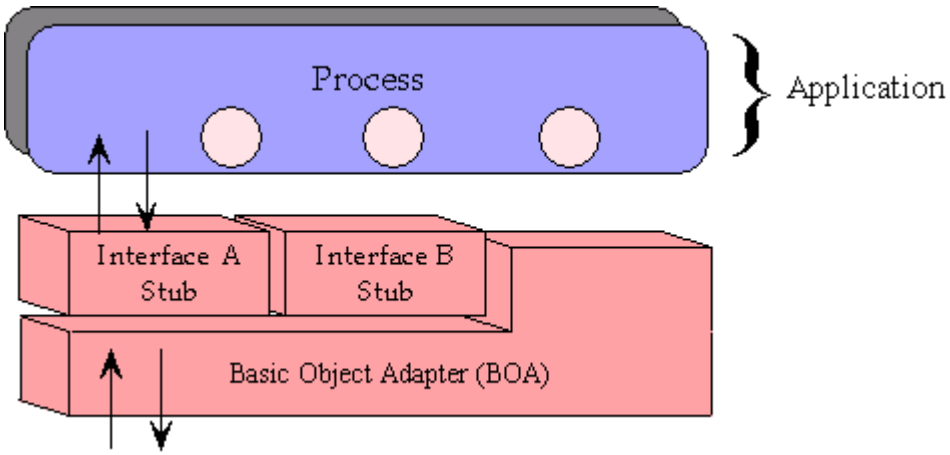

Figura 1.5.1 Shared Server Activation Policy

Cuando el proceso está listo para terminar, notifica a BOA llamando a *deactivate_impl()*. En este punto, todos los objetos que están corriendo en el proceso, son desactivados. También se puede desactivar un objeto singular llamando a *deactivate_obj()*. Se debe notar que CORBA no requiere que los objetos de un shared server invoquen obj_is_ready() cuando se activan por primera vez. Sin embargo, la mayoría de las implementaciones de los ORB's así lo requieren.

La mayoría de los servidores CORBA son de este tipo. La pregunta lógica a este punto es: cómo se corren múltiples objetos concurrentemente en un mismo proceso? La respuesta es: usando threads. Es recomendable utilizar un thread por cada objeto activo.

BOA Unshared Server

En este caso, cada objeto reside en diferentes procesos servidores. Un nuevo servidor es activado la primera vez que un pedido llega al objeto. Cuando el objeto se ha inicializado, notifica a BOA de que está preparado para recibir nuevos pedidos invocando a *obj_is_ready()*.

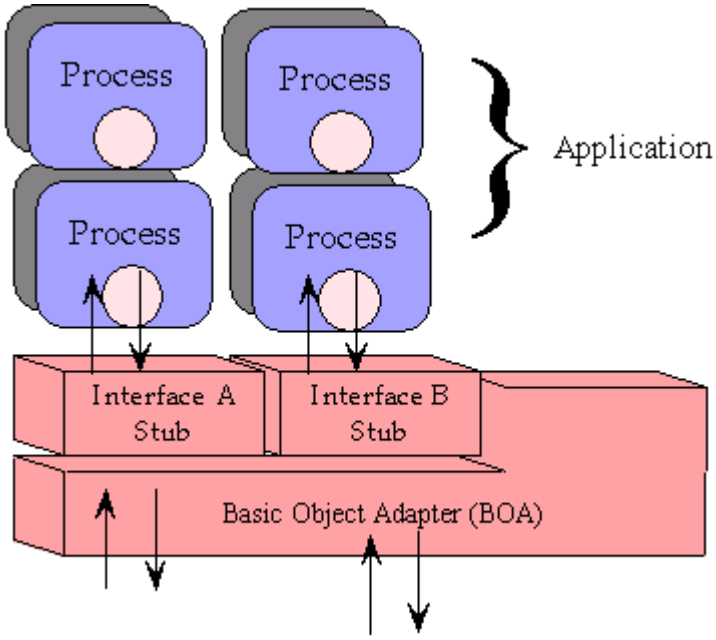

Figura 1.5.2 Unshared Server Activation Policy

Cualquier vez que una invocación se realiza sobre un objeto que no ha sido activado, se arranca un nuevo proceso servidor, no importa si existe ya otro objeto activo para la misma implementación. El objeto del servidor permanecerá activo recibiendo pedidos hasta que se invoque sobre él *deactivate_obj()*.

Este método se utiliza cada vez que se necesitan objetos dedicados, por ejemplo, para representar una impresora. En vez de manejar múltiples objetos, la aplicación maneja uno solo.

BOA Server-per-Method

En esta política, un servidor se arranca cada vez que un pedido llega a un objeto. El servidor se ejecuta solamente el tiempo que el método invocado es ejecutado, y luego termina.

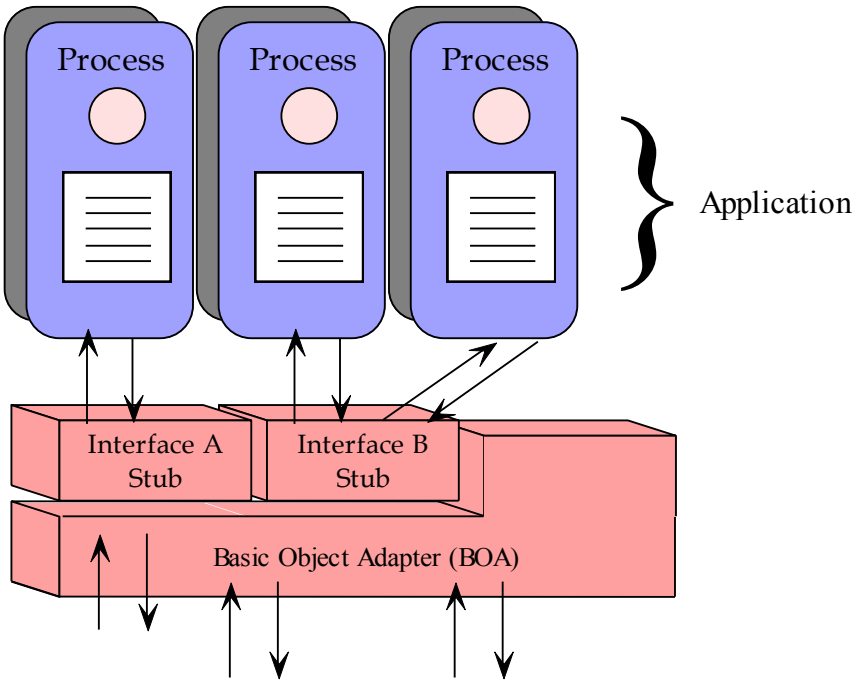

Figura 1.5.3 Server-per-Method Activation Policy

Pueden estar activos concurrentemente muchos procesos servidores para el mismo objeto, o incluso para el mismo método.

Como un nuevo servidor se arranca para cada pedido, no es necesario notificar a BOA cuando el objeto está listo o es desactivado.

Este es el método menos frecuente. Su utilidad se encuentra para ejecutar scripts o programas que se ejecutan sólo una vez y terminan.

BOA Persistent Server

En esta política, los servidores son activados fuera de BOA. Típicamente se arranca un servidor que luego notifica a BOA de que está listo para manejar pedidos usando *impl_is_ready()*. BOA trata todos los pedidos como en la política **shared server** y envía pedidos de activación para objetos y métodos específicos a un solo proceso. Si la implementación no está lista cuando arriba un pedido, se produce un error.

El **persistent server** es en realidad un caso especial de **sahred server**. La diferencia está en que la activación del servidor no es manejada por el ORB.

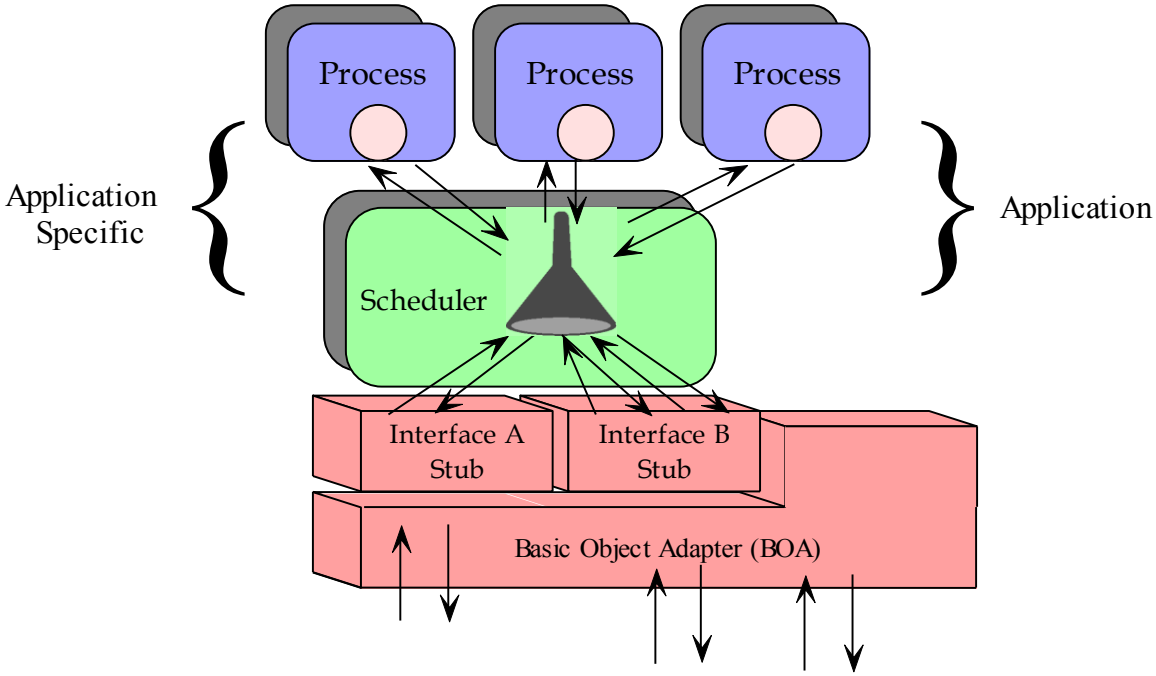

Figura 1.5.4 Persisten Server Activation Policy

Ejemplo de Activación de objetos

Veremos ahora un ejemplo de cómo se activan los objetos en un escenario de **shared server**. Se mostrará la activación de un nuevo objeto y de uno ya existente.

1. **El servidor crea dos instancias de objetos.** El servidor invoca, por ejemplo, un constructor de Java.
2. **El nuevo objeto se registra en el BOA.** El nuevo objeto servidor debe invocar *BOA::create*. El objeto debe pasar a BOA su nombre de interfaz, su nombre de implementación y algún dato de referencia único o ID. La llamada a *create* retorna una referencia al nuevo objeto que luego se pasa a *obj_is_ready()* para que el ORB sepa que está listo para recibir pedidos.
3. **El objeto existente se registra con el BOA.** El otro objeto que ya existe y del cual ya tenemos la referencia, se registra llamando a *obj_is_ready()*.

4. **El servidor informa que está listo para funcionar.** El servidor ha inicializado todos sus objetos, entonces invoca a impl_is_ready() para informar al ORB que está listo para aceptar las invocaciones de los clientes.

5. **Los objetos se desactivan**. Cada objeto se desactiva a sí mismo invocando *deactivate_obj()*.

6. **El servidor termina su ejecución**. El servidor invoca *deactivate_impl()*.

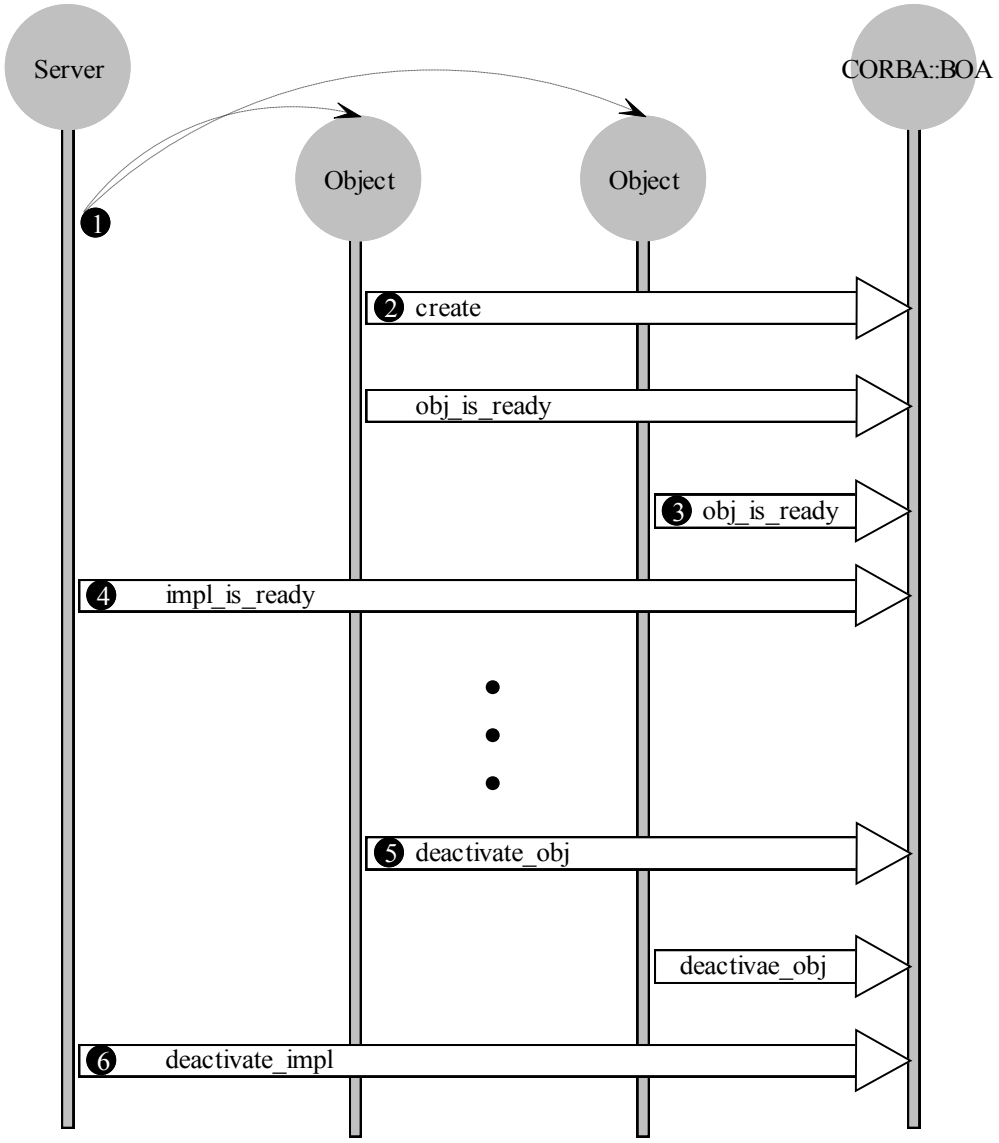

Figura 1.5.5 Ejemplo de activación de objetos

Portable Object Adapter

El POA es como el BOA con varias mejoras. Como BOA, POA puede iniciar:

1. Un servidor por método.
2. Un servidor por objeto.
3. Un servidor compartido por todas las instancias de todos los tipos de objetos.

Los objetivos del diseño de POA son:

- Permitir a los programadores construir implementaciones de objetos que son portables entre diferentes ORBs.
- Proveer soporte para objetos con identidades persistentes. Este es uno de los aspectos principales en cuanto a lo que POA aporta.
- Proveer soporte para la activación trasparente de objetos.
- Permitir a un solo servant soportar múltiples objetos simultáneamente.
- Permitir que múltiples instancias de POA existan en un mismo servidor.
- Proveer soporte para activación implícita de servants.
- Permitir que la implementación de los objetos tengan la máxima responsabilidad sobre el comportamiento de los mismos.
- Evitar pedir al ORB que mantenga información persistente que describa a objetos individuales.

POA soporta objetos transitorios y permanentes. Los transitorios viven sólo mientras el proceso que los creó vive.

POA también introduce algunos otros nuevos conceptos. Opcionalmente se puede elegir un **servant manager** para cada implementación de la interfaz de un objeto. Un servant es un objeto o entidad que implementa pedidos sobre uno o más objetos. De este modo, POA invoca operaciones sobre el servant manager para crear, activar y desactivar servants. Se puede pensar en los servants managers como manejadores de instancias que asisten a POA en el manejo de los objetos del lado del servidor.

Los servants managers deben ser registrados en el POA para que puedan generar instancias de los objetos. El POA mantiene un mapa de los servant managers activos. También tiene un mapa de todos los identificadores de los objetos activos llamado **Active Object Map**, que mapea **ObjectIDs** en servants en ejecución. Un objeto activo se identifica por su referencia, que a su vez encapsula su ObjectID. Un servant de POA puede soportar simultáneamente múltiples ObjectIDs. El cliente sólo conoce la referencia al objeto, la que utiliza para invocar operaciones sobre el mismo. Del lado del servidor, el ORB, POA, y el servant manager cooperan para hacer llegar el pedido al correspondiente servant.

Como se muestra en la figura 1.5.6, un servidor puede soportar múltiples POAs que son derivados de un objeto distinguido POA llamado *root POA*. Cada instancia de POA puede ser configurada para implementar diferentes políticas del lado del servidor.

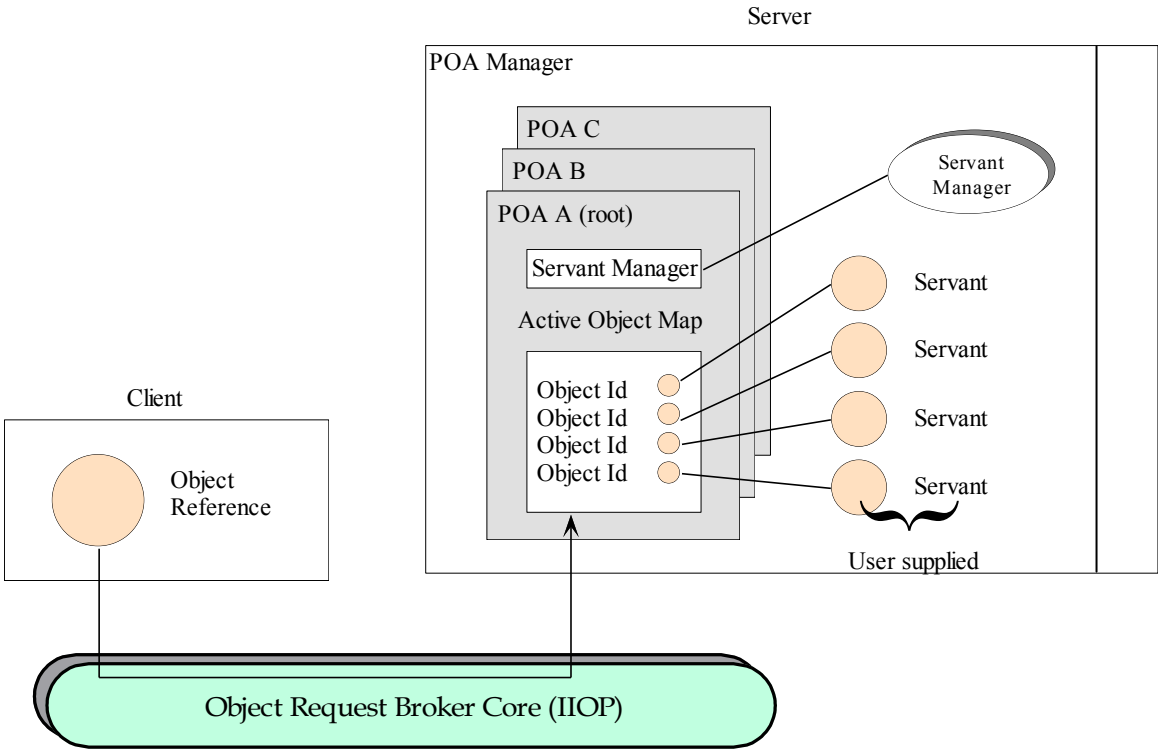

Figura 1.5.6 Portable Object Adapter

Para que el servidor pueda descubrir la instancia root, debe usar las siguientes operaciones:

```
// obtener el root POA
org.omg.CORBA.Object object = orb.resolve_initial_references("RootPOA");
// convertilo al tipo correcto
org.omg.PortableServer.POA rootpoa = POAHelper.narrow(object);
```

Se usa luego la referencia *rootpoa* para crear o destruir nuevos POAs invocando a *create_POA()* y *destroy()*.

Referencias a objetos persistentes

Lo más interesante que introduce POA son las referencias a objetos persistentes. La implementación puede controlar el funcionamiento de objetos persistentes de las siguientes maneras:

1. Especificando la información que define la identidad del objeto.
2. Determinando la relación entre la identidad del objeto y su estado.
3. Manejando el almacenamiento y la devolución del estado de los objetos.
4. Proveyendo código que será ejecutado en respuesta a los pedidos.
5. Determinando si el objeto existe o no en cualquier momento.

El ObjectID es un valor que la implementación y POA usan para identificar un objeto abstracto particular de CORBA. El valor del ObjectID puede ser asignado tanto por POA como por la aplicación, pero igualmente se mantiene siempre encapsulado y oculto al cliente. Los ObjectIds no tienen una forma standard, sino que son manejados por el POA como secuencias de octetos sin interpretación. Los ObjectIDs deben ser únicos dentro de un espacio de nombres de POA. Una referencia a un objeto CORBA encapsula al POA y al ObjectID, lo que la hace única.

.

Servant Managers

Se pueden controlar prácticamente todos los aspectos del comportamiento de un servidor especificando explícitamente la política del POA. El programador también puede especificar su propio servant manager.

Un servant manager que se asocia con un POA, debe ser local al proceso que contiene al POA. Un servant manager permite al POA activar objetos bajo demanda, cuando el POA recibe un pedido para un objeto inactivo. Un servant manager es registrado con el POA como un objeto callback, lo que significa que es invocado por el POA cuando lo necesita. Una aplicación servidor que activa todos sus objetos cuando se inicializa, no necesita un servant manager.

Si están presentes, los servant managers son responsables de mantener la asociación de un objeto con un servant particular, y de determinar si un objeto existe. Un servant manager puede implementar una de las dos siguientes interfaces: **ServantActivator** o **ServantLocator**. La interfaz que se elija depende de la política de POA que se desee implementar. La interfaz ServantActivator, típicamente maneja objetos persistentes, que el POA también mantiene en su *Active Object Map*. La interfaz ServantLocator trabaja con objetos transitorios que no son seguidos por el POA. Cada tipo de servant manager contiene dos operaciones: la primera es usada para encontrar y devolver un servant, y la segunda para desactivarlo. La interfaz ServantActivator corresponde a la política de POA llamada RETAIN, y la ServantLocator a la NON_RETAIN. Se explican estas políticas en la próxima sección.

Políticas de POA

El POA y su servant manager refuerzan juntos las políticas que permiten controlar íntimamente el ambiente de ejecución de un servant. Estas políticas permiten especificar los siguientes comportamientos para cada POA: el modelo del manejo de threads, la extensión del tiempo de vida de los objetos, la unicidad del ObjectID, técnicas de asignación de Ids, conservación de servants, y el modelo de activación.

Para especificar las políticas de un nuevo POA, se deben primero crear objetos **Policy**, que deben ser pasados como parámetros a la operación *create_POA*.

En la siguiente tabla veremos cuáles son las políticas de POA.

Tipo de Política	Descripción de la pólitica y elecciones
Request Processing	Esta política especifica cómo los pedidos son procesados por el POA. Se pueden especificar los siguientes valores: **USE_ACTIVE_OBJECT_MAP_ONLY:** Si el POA no encuentra el Object Id en su mapa, retorna una excepción al cliente. El POA debe estar usando también la política RETAIN. **USE_DEFAULT_SERVANT:** Si el POA no encuentra el Object Id en su mapa, o si la política NON_RETAIN está presente, despachará el pedido al servant por defecto. Este debe haber sido registrado previamente en el POA usando *set_servant* . **USE_SERVANT_MANAGER:** Si el POA no encuentra el Object Id en su mapa, o si la política NON_RETAIN está presente, despachará el pedido al servant manager. Este debe haber sido registrado previamente en el POA usando *set_servant_manager* . El root POA usa por defecto: USE_ACTIVE_OBJECT_MAP_ONLY

Tipo de Política	Descripción de la pólitica y elecciones
Thread	Esta política especifica el modelo de threads que se usan en este POA. Se pueden especificar los siguientes valores: **ORB_CTRL_MODEL:** El POA es responsable de asignar los pedidos a los threads. **SINGLE_THREAD_MODEL:** El POA procesará todos los pedidos secuencialmente. El root POA usa por defecto: ORB_CRTL_MODEL.
Lifespan	Esta política especifica el modelo de persistencia para el POA. Se pueden especificar los siguientes valores: **TRANSIENT:** Los objetos no pueden sobrevivir a los procesos que los crearon. **PERSISTENT:** Los objetos pueden sobrevivir a los procesos que los crearon. El root POA usa por defecto: TRANSIENT
Object Id Uniqueness	Especifica si los servants activados por el POA tienen Object Ids únicos. Se pueden especificar los siguientes valores: **UNIQUE_ID:** Los servant activados por este POA soportan solamente un Object Id. **MULTIPLE_ID:** Un servant activado por este POA puede soportar múltiples Object Ids. El root POA usa por defecto: UNIQUE_ID
Id Assignment	Esta política especifica si los Object Ids son asignados por la aplicación o por el POA. Se pueden especificar los siguientes valores: **USER_ID:** Se asigna un Object id a los objetos desde la aplicación. **SYSTEM_ID:** Solamente el POA asigna los Object Ids. Si el POA también tiene la política PERSISTENT, los Object Ids asignados deben ser únicos a través de todas las instancias del mismo POA. El root POA usa por defecto: SYSTEM_ID
Servant Retention	Esta política especifica si el POA conservará a los servants en el Active Object Map. Se pueden especificar los siguientes valores: **RETAIN:** El POA conservará a los servants activos en el Active Object Map. **NON_RETAIN:** Los sevants no son conservados por el POA. En este caso, se deben habilitar las políticas USE_DEFAULT_SERVANT o USE_SERVANT_MANAGER El root POA usa por defecto: RETAIN
Activation	Especifica si el POA soporta la activación implícita de objetos. Se pueden especificar los siguientes valores: **IMPLICIT_ACTIVATION:** Este POA soportará activación implícita. **NO_IMPLICIT_ACTIVATION:** El POA no soportará activación implícita. El root POA usa por defecto: IMPLICIT_ACTIVATION

Conclusiones

En este capítulo hemos visto cómo el Portable Object Adapter agrega mucha más flexibilidad y control, desde el lado de un servidor, sobre el ambiente de ejecución, gracias a la variedad de las políticas propuestas.

POA introduce el manejo de objetos persistente, algo para que los programadores debían diseñar sus propias implementaciones si usaban BOA.

La flexibilidad del modelo de POA está dada entre otras cosas por la posibilidad de utilizar muchos Object Adapters para un mismo servidor, permitiendo distintas políticas acordes a las necesidades de los servants.

Capítulo VI: CORBA Interoperability

La especificación de CORBA 1.1 se ocupaba solamente de la creación de aplicaciones orientadas a objetos portables. El resultado fue cierto nivel de portabilidad pero no interoperabilidad. CORBA 2.0 agrega la interoperabilidad especificando un protocolo llamado **Internet Inter-ORB Protocol** (IIOP). IIOP es básicamente TCP/IP con algunos mensajes de intercambio definidos por CORBA que sirven como el *backbone* común. Cada ORB llamado **CORBA-compliant** debe implementar este protocolo.

Comunicación ORB-to-ORB

La interoperabilidad se basa en la comunicación de ORB a ORB. La figura 1.6.1 muestra cómo debe funcionar básicamente la interoperabilidad.

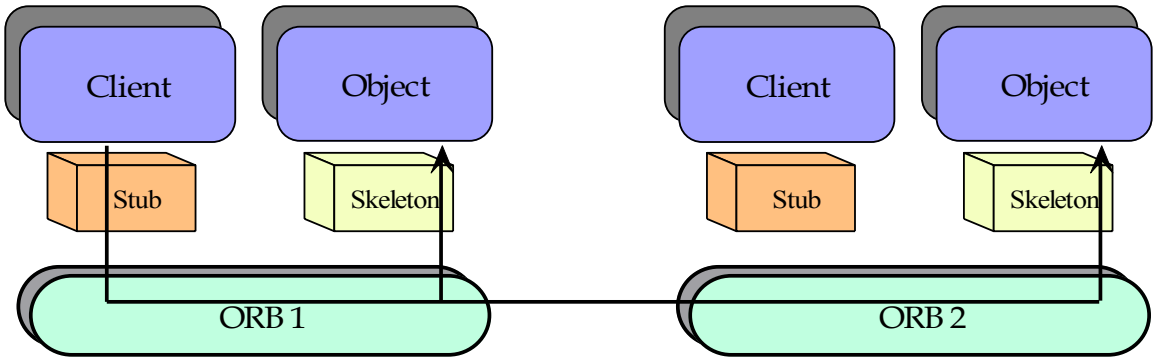

Figura 1.6.1 CORBA Interoperability

La invocación desde el cliente 1 pasa a través de su stub IDL al ORB Core. El ORB examina la referencia del objeto destino y busca la ubicación de su implementación en su repositorio local. Si la implementación es local, el ORB pasa la invocación a través del skeleton al objeto correspondiente. Si la implementación es remota, el ORB pasa la invocación a través del camino de comunicación al ORB 2 que la rutea al objeto. La implementación del objeto no tiene modo de saber si la invocación viene desde un cliente local o remoto.

Este escenario funciona independientemente de las diferencias de plataforma, protocolo o implementación que puedan existir entre el ORB1 y el 2.

Debe notarse que ni el cliente ni la implementación del objeto se encuentran involucrados en la comunicación entre ORBs. En CORBA la comunicación es siempre entre ORBs.

Existen dos modos para hacer que dos ORBs se comuniquen entre ellos. Uno es hacerlos usar el mismo protocolo, y el otro, si los protocolos son diferentes, es instalar **bridges** para traducir de un protocolo a otro. CORBA 2.0 provee ambas soluciones. Obviamente la primera es más simple, directa y eficiente. Todo ORB usa IIOP, pero se puede optar por utilizar DCE CIOP (que se ve más adelante en este capítulo) o un protocolo propietario.

Existen diferentes modos de construir bridges. Las dos aproximaciones principales son: immediate bridging y mediated bridging. En la figura 1.6.2 se puede apreciar como funciona cada una.

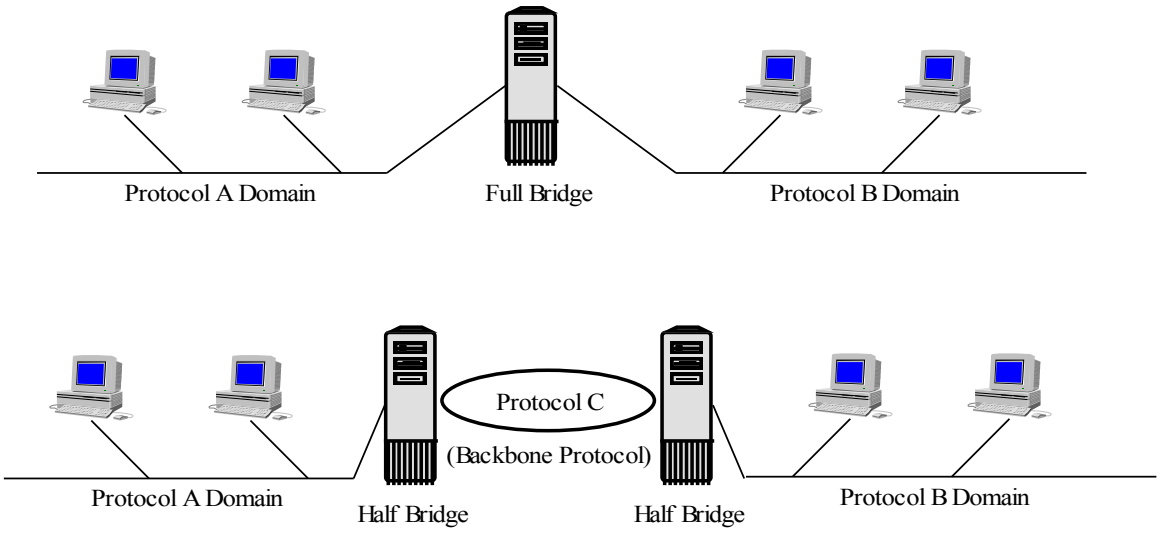

Figura 1.6.2 Immediate y mediate bridging

En immediate bridging, dos dominios hablan directamente entre sí a través de un solo bridge que traduce los mensajes o partes de ellos que así lo requieren. Se mantiene un concepto de dos dominios. Es rápido y eficiente, ya que cada bridge se diseña para el par de dominios a los que sirve. Pero es inflexible ya que pueden requerirse muchos bridges si la cantidad de dominios se extiende.

En mediate bridging todos los dominios “puentean” a un mismo protocolo común. Cuando un mensaje pasa a través del primer bridge desde su dominio de origen, es traducido al protocolo común, y cuando pasa de la región común a su dominio de destino, es traducido al protocolo de este último. La gran ventaja de esta solución es que la cantidad de bridges crece solo a medida que aumenta el número de dominios. Las desventajas radican en que los mensajes deben ser traducidos dos veces y no es eficiente si los dominios son pocos.

Interoperable Object References (IORs)

CORBA especifica una forma standard de las referencias de los objetos llamada Interoperable Object Reference (IOR). Estas referencias no son nunca usadas internamente por un ORB solo ni es jamás conocida por un cliente o implementación de objeto. Es sólo utilizada para las invocaciones entre distintos ORBs.

La IOR requiere contar con la siguiente información:

- *El tipo de objeto*: los ORBs deben conocer el tipo de objeto para preservar la integridad del sistema.
- *Qué protocolos puede usar el ORB que realiza la invocación*: cuando la IOR deja el ORB, lista el o los protocolos que ese ORB acepta. Pero tan pronto como cruza el bridge, la IOR es alcanzable sólo a través de los protocolos que el bridge conoce. Esto requiere que los bridges reconozcan las IORs y actualicen esta información a medida que pasan a través de ellos. Es necesario tener en cuenta que las IORs no tienen por qué ser solo referencias de objetos destinos de una invocación, sino que también pueden ser parámetros.
- *Qué servicios se encuentran disponibles en el ORB*: la invocación puede involucrar servicios extendidos del ORB. Agregando esta información a la IOR se evitan negociaciones entre ORBs referidas al contexto.
- *Si la referencia al objeto es nula*: reconociendo la nulidad de una referencia, un bridge puede evitar trabajo innecesario.

La IOR especificada por el OMG consiste de un ID y uno o más **tagged profiles**. Cada protocolo de comunicaciones que CORBA puede utilizar para conectar dos ORBs, contiene un **tag** y un **profile**. El tag consta de cuatro bytes registrados por el OMG. El profile contiene toda la información que un ORB remoto necesita para realizar una invocación usando el protocolo.

Estructura de la especificación de la Interoperabilidad

Existen dos protocolos standard especificados por el OMG: IIOP y DCE-ESIOP. En la figura 1.6.3 vemos la estructura de la especificación de CORBA para la interoperabilidad.

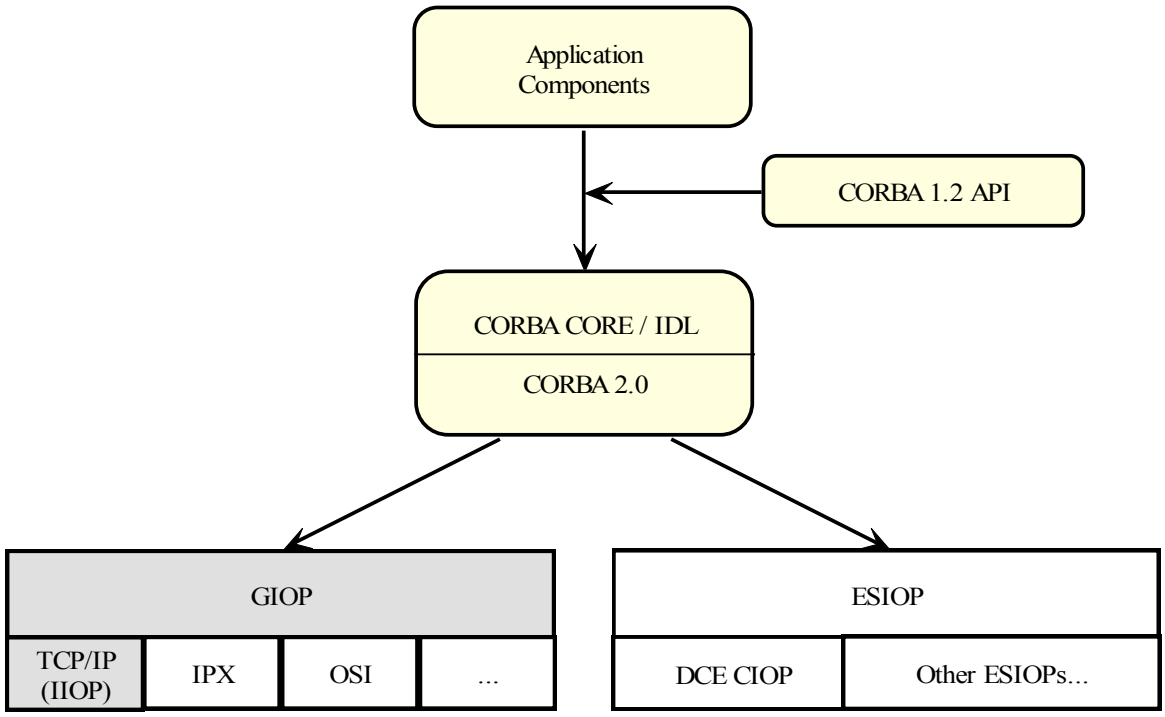

Figura 1.6.3 Estructura de la Interoperabilidad de CORBA

La caja llamada CORBA 2.0 contiene la arquitectura básica de bridging, la IOR con sus protocolos, profiles y componentes, y las interfaces de interoperabilidad incluyendo DSI.

GIOP, o **General Inter-ORB Protocol**, contiene las especificaciones para el protocolo de mensajería general inter-ORB, que ha sido diseñado para ser implementado sobre cualquier transporte confiable. Se incluye también el Common Data Representation y los siete mensages de GIOP con sus formatos. Debajo de esta caja se encuentran otras que representan a transportes confiables.

ESIOP significa Enviroment-Specific Inter-ORB Protocol. Todo ESIOP llega a definir su protocolo independientemente en tanto cumple con los requerimientos para la comunicación ORB-to-ORB.

El DCE ESIOP especifica la comunicación entre ORBs basada en invocaciones y respuestas DCE RPC. Adopta el mismo Common Data Representation que el GIOP, pero reemplaza los siete mensajes de GIOP por dos operaciones: *invoke* y *locate*. Para este ESIOP, los contenidos de las invocaciones no sólo mapean a IIOP, sino que son idénticas. Pero la semántica de los mensajes y los mecanismos de localización son diferentes. DCE ESIOP es el único ESIOP standard del OMG.

General Inter-ORB Protocol (GIOP) e Internet Inter-ORB Protocol (IIOP)

El protocolo GIOP/IIOP fue designado para cumplir con las siguientes expectativas:

- *Amplia disponibilidad*: el IIOP se basa en TCP/IP, el protocolo más ampliamente utilizado y flexible, y define sólo la mínima cantidad de capas adicionales para transferir invocaciones de CORBA entre ORBs.
- *Simplicidad*: GIOP fue diseñado tan simple como era posible.
- *Escalabilidad*: está diseñado para escalar a la medida de Internet.
- *Bajo costo*: los costos de reingeniería e implementación de los ORBs fue minimizada.
- Generalidad: el GIOP fue diseñado para ser implementado sobre cualquier protocolo confiable orientado a la conexión, no solamente TCP/IP.
- Neutralidad en su arquitectura: GIOP asume muy pocas cosas sobre la arquitectura e implementación de los ORBs y de los bridges que lo soportan.

El GIOP consiste de tres especificaciones:

- La definición del Common Data Representation (CDR).
- Formato de los mensajes GIOP.
- Asunciones sobre el transporte.

El IIOP no es una especificación separada, sino que es el mapeo de GIOP sobre TCP/IP.

Common Data Representation (CDR)

El CDR define representaciones para todos los tipos de datos de OMG IDL. La especificación toma en cuenta el ordenamiento y la alineación de los bytes. La semántica *receiver-makes-right* evita traducciones innecesarias de mensajes entre máquinas con el mismo ordenamiento de bytes.

Formato de los mensajes GIOP

GIOP define siete mensajes distintos para las comunicaciones entre ORBs. Estos cargan pedidos, localizan implementaciones de objetos y manejan los canales de comunicación. Soportan todos las funciones y comportamientos requeridos por CORBA, incluyendo el reporte de excepciones y el paso de la información de contexto. Algunas características de la transferencia de mensajes son:

- *La conexión es asimétrica*: los roles del cliente y el servidor son distintos y asignados en la conexión. El cliente origina la conexión y puede enviar pedidos y no respuestas. El servidor acepta la conexión y puede enviar sólo respuestas.
- *Los pedidos pueden ser multiplexados*: múltiples clientes de un mismo ORB pueden compartir una conexión a un ORB remoto.
- Los pedidos se pueden superponer: cualquier orden de pedidos y respuestas en un modelo asincrónico, es soportado a través de identificadores de pedidos y respuestas.

Todos los mensajes GIOP comienzan con un encabezado. En IDL es como sigue:

```
struct MessageHeader{
        char            magic[4];
        Version         GIOP_version;
        boolean   byte_order;
        octect          message_type;
        unsigned long   message_size;
}
```

La siguiente es una breve descripción de las operaciones:

- **Request message**: el mensaje de request consiste del encabezado GIOP, un encabezado de pedido y un cuerpo de pedido. El encabezado contiene el contexto del servicio, un ID de pedido, el identificador del objeto y la operación. El cuerpo del pedido contiene los parámetros **in** y **out**.
- **Reply message**: el mensaje de reply consiste del encabezado GIOP, un encabezado de respuesta y un cuerpo de respuesta. El encabezado de respuesta contiene el contexto del servicio, el mismo ID del pedido y un código de estado. Si éste último es **no_exception** entonces el cuerpo de la respuesta contiene el valor de retorno (si existe), todos los valores de los parámetros **inout** y **out**. Si el código de estado es **USER_EXCEPTION** o **SYSTEM_EXCEPTION**, el cuerpo contiene información codificada sobre la excepción. Y finalmente, si el código de estado es **LOCATION_FORWARD**, el cuerpo contiene una IOR a la cual el ORB debe redireccionar el pedido original. Este redireccionamiento se realiza sin el conocimiento del cliente.
- **CancelRequest**: estos mensajes contienen el usual encabezado GIOP, seguido de un encabezado de CancelRequest con el ID del pedido. Este mensaje notifica al servidor de que el cliente no espera más el especificado Request o LocateRequest pendiente.

- **LocateRequest**: estos mensajes son enviados del cliente al servidor para determinar si una referencia a un objeto particular es válida, si un servidor es capaz o no de servir pedidos; y , si no, una nueva dirección para los pedidos sobre esa referencia.
- **LocateReply**: Esta es la respuesta del servidor a LocateRequest. Contiene el encabezado GIOP, el encabezado de LocateReply con el ID del request, y el valor de estado, y el cuerpo del LocateReply que contiene, si es aplicable, la IOR del nuevo destino.
- **CloseConnection**: este mensaje es enviado por el servidor para notificar a los clientes (más precisamente a sus ORBs), que el servidor pretende cerrar la conexión y no proveerá futuras respuestas. Al servidor le está permitido enviar este mensaje cuando no tiene pedidos pendientes. También informa a los ORBs de que ellos no aceptarán más pedidos sobre esa conexión, y de que las mismas pueden se reasumidas en modo seguro sobre otra conexión. El ORB enviará a los clientes una excepción de COMM_FAILURE a los clientes. El mensaje CloseConnection consiste de el encabezado GIOP cuyo valor de tipo de mensaje especifica todo lo necesario.
- **MessageError**: Estos mensajes son enviados en respuesta a cualquier mensaje IIOP que por alguna razón no puede ser interpretado.

Requerimientos de transporte de los mensajes GIOP

Los requerimientos fueros elegidos específicamente para mapear a TCP/IP. Básicamente requiere:

- Un protocolo orientado a la conexión.
- Entrega confiable.
- Los participantes de la conexión deben ser notificados si la misma se pierde.
- El modelo para la iniciación de una conexión debe cumplir ciertos requisitos.

Internet Inter-ORB Protocol (IIOP)

El mapeo de GIOP a TCP/IP es directo, por lo que esta especificación es muy breve. Consiste principalmente de la especificación de la IOR, que contiene el nombre del host y el puerto que escucha a la conexión iniciada por el cliente.

Enviroment-Specific Inter-ORB Protocols (ESIOP) y DCE ESIOP

ESIOP está basado en CORBA 2.0 y su arquitectura básica incluye dominios y bridging, la IOR y las interfaces de interoperabilidad incluyendo DSI. No todo protocolo Inter-ORB es un ESIOP. Es posible construir protocolos propietarios para cumplir con demandas específicas.

El protocolo basado en DCE (Distributed Computing Enviroment) que es adoptado por el OMG como una parte de CORBA 2.0 es un ESIOP. Cumple con todos los requisitos de CORBA 2.0 y utiliza el mismo CDR que el GIOP, una característica que facilita el bridging entre él y los dominios GIOP. En la especificación el nombre del protocolo es DCE Common Inter-ORB Protocol, abreviado DCE CIOP, como será mencionado de ahora en adelante.

DCE CIOP reemplaza los siete mensajes de GIOP con dos llamadas DCE RPC: *locate* e *invloke*. También reemplaza las asunciones sobre el transporte de GIOP por la confianza en el servicio DCE RPC para manejar estas dos llamadas.

El DCE CIOP es un protocolo para comunicaciones entre ORBs, en cuanto cumple el mismo rol que IIOP. El cliente y el servidor interactúan con su ORB local, y el hecho de que sus invocaciones pueden viajar a través de una red siguen permaneciendo ocultas a ellos. De hecho no existe en CORBA ningún mecanismo que le permita a un cliente determinar qué protocolo se utiliza.

Los RPCs usados por DCE CIOP fueron seleccionados específicamente para comunicaciones ORB-to-ORB, y no están relacionadas con operaciones específicas de un cliente en particular. La operación invocada por el cliente es un parámetro el la operación invoke.

Estructura de la especificación DCE CIOP

El DCE CIOP corre sobre RPC que es interoperable con los protocolos DCE especificados en muchos lugares. El DCE RPC:

- Define protocolos sin conexión y orientados a la conexión.
- Soporta múltiples protocolos subyacentes, incluyendo TCP/IP.
- Soporta múltiples pedidos a múltiples objetos CORBA sobre la misma conexión.
- Soporta fragmentación de mensajes, ventaja para la transferencia de gran cantidad de pedidos y respuestas.

Cada interacción entre ORBs ocurre como una llamada a procedimiento remoto (RPC) a una de las dos interfaces de DCE RPC definidas en la especificación: *locate* e *invoke*. Cada una de éstas es una DCE RPC sincrónica, que consiste de un pedido y una respuesta. Las invocaciones asincrónicas de CORBA son implementadas internamente al ORB usando threads DCE.

DCE CIOP utiliza el mismo CDR que GIOP, y el encabezado de los mensajes DCE ESIOP también se proveen en IDL.

La IOR usa el profile **TAG_MULTIPLE_COMPONENTS**, que contiene componentes que:

- Identifican el proceso servidor.
- Identifican el objeto destino cuando el mensaje se envía a un servidor.
- Indican si el cliente debe enviar un mensaje *locate* o *invoke*.

Se definen dos interfaces DCE CIOP en DCE IDL que son: **dce_ciop_pipe** y **dce_ciop_array**. Ambas implementan las operaciones ya referidas (*locate* e *invoke*). Los contenidos de los mensajes son similares a los de GIOP.

Conclusiones

La interoperabilidad entre ORBs hace a CORBA ampliamente aplicable y extensible. Es una de sus grandes virtudes, ya que permite relacionar ambientes diversos, independientemente de la plataforma, implementaciones de ORBs y protocolos de redes.

Las especificaciones son bastante sencillas, por lo que su implementación para los fabricantes de ORBs no implica mayores gastos ni esfuerzos.

Capítulo VII: CORBA Services

Introducción sobre CORBA services y falicities

CORBA conecta sólo objetos, no aplicaciones. El OMG propone la **Object Management Arquitecture** (OMA), que está basada en CORBA. La OMA da cuerpo a la visión del OMG en cuanto al ambiente de las componentes de software. Esta arquitectura muestra como la estandarización de las interfaces de componentes sube hasta objetos de aplicación para crear un ambiente de alta reusabilidad de software.

Los objetos de aplicación, accederán a estos servicios y facilidades a través de interfaces standard. La OMA especifica un conjunto de interfaces y funciones standard para cada componente y se divide en dos componentes principales: **CORBA services**, de un nivel más bajo; y **CORBA facilities**, de un nivel intermedio.

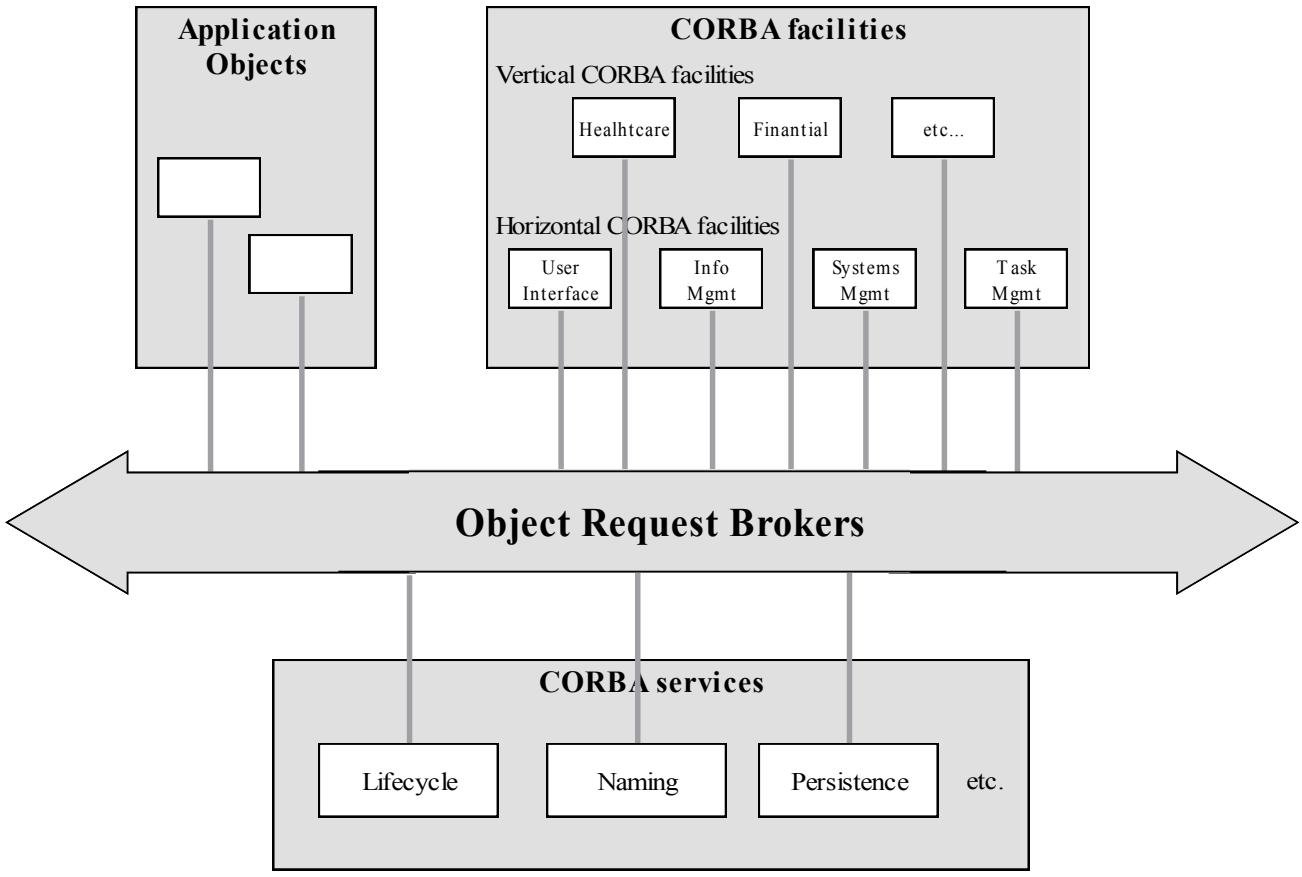

Figura 1.7.1 Object Management Arquitecture

Los CORBA services proveen funcionalidad básica que casi todo objeto puede necesitar: servicios de ciclo de vida, de nombres, de persistencia, y otros. Básico no necesariamente significa simple.

Mientras que CORBA services provee servicios para objetos, CORBA facilities provee servicios para aplicaciones. Por ejemplo la facility “Compound Document Management” le da a la aplicación un modo standard de acceder a las componentes de un documento compuesto.

Las CORBA facilities se dividen en horizontales y verticales. Las horizontales pueden ser usadas eventualmente en cualquier negocio; las verticales estandarizan el manejo de información especializada de grupos industriales particulares.

Object Lifecycle Service

Este servicio define convenciones para crear, borrar y mover objetos. Las mismas permiten a los clientes realizar operaciones relacionadas con el ciclo de vida de los objetos en diferentes lugares, y especificar ubicaciones de destino cuando es necesario, usando interfaces standard y sin violar la transparencia de localización de CORBA. Esto requiere la introducción de nuevos conceptos.

Factory Objects

Para crear un objeto usando una invocación de CORBA, el cliente debe hacer referencia a un objeto al cual le pueda enviar el pedido, éste objeto es el **factory object**. Existen varias cosas que un objeto de este tipo debe realizar:

- Determinar la ubicación del nuevo objeto.
- Reunir los recursos que el objeto necesita (memoria, almacenamiento permanente, recursos dependientes del sistema, etc.).
- Registrar el objeto en el OA (Object Adapter) para obtener una referencia la mismo.
- Crear el objeto usando los recursos reunidos.
- Avisar al OA que el objeto está listo para la activación.
- Devolver la referencia al objeto al cliente.

Objetos y Categorías de Interfaces

Los objetos que soportan interfaces de CORBA services, se dividen en dos categorías, y los tipos de interfaces se dividen en tres categorías. Para los objetos, las categorías son:

- *Specific Objects*: estos son los que soportan a la interfaz como su propósito primario.
- *Generic Objects*: soportan a la interfaz de CORBA services pero no lo hacen como su propósito primario, sino por necesidad.

Los tres tipos de interfaces son:

- *Functional Interfaces*: proveen el servicio.
- *Participant Interfaces*: nacen y generalmente son heredadas por los generic objects, y participan del servicio de algún modo pero no como las proveedoras principales.
- *Administrative Interfaces*: usadas para administrar el servicio.

Para el servicio de ciclo de vida, las interfaces de **move**, **copy** y **delete** son concebidas por objetos genéricos. El OMG reconoce que estas operaciones son muy dependientes de la implementación, y no pretenden más que proveer las interfaces de las mismas.

La transparencia de localización significa que los ambientes se dividen en la capa de aplicación y el nivel de infraestructura; y la localización queda a cargo de éste último.

Para expresar el concepto de localización en un modo independiente de la interfaz, el servicio lifecycle define el objeto **factory finder**.

Persistent Object Service

La OMA especifica reglas respecto de ciertos comportamientos de los objetos. De este modo, el cliente sabe qué esperar de un objeto remoto en la red. Quizás, una de las partes más importantes de la interacción involucra el estado de persistencia de los objetos.

La OMA dice que cada estado de un objeto (para aquellos que mantienen un estado interno), debe permanecer intacto entre una invocación y la siguiente. Ya que CORBA permite que una referencia sea almacenada durante una sesión y recuperada en otra, el estado debe ser conservado incluso por períodos de tiempo arbitrarios. Si un cliente crea un objeto, se espera que el estado del mismo sea reflejado siempre según se lo haya establecido al momento de la creación, al menos que el objeto sea compartido con otros clientes que puedan modificar su estado, o un cliente que tenga una copia de la referencia, haya eliminado al objeto.

Desde el punto de vista del cliente, la persistencia es un concepto simple, ya que para él, la instancia del objeto está siempre activa y preserva su estado. Para el cliente esto representa un servicio.

Desde el punto de vista de la implementación del objeto, la persistencia deja de ser un servicio para transformarse en una obligación. La implementación debe proveer los mecanismos para preservar el estado del objeto.

La OMA provee el **Persistent Object Service** (POS) para almacenar componentes en una variedad de medios de almacenamiento permanente.

Arquitectura del POS

Para proveer flexibilidad el POS ha sido diseñado como la unión de una cantidad de componentes que se describen en la figura 1.6.2.

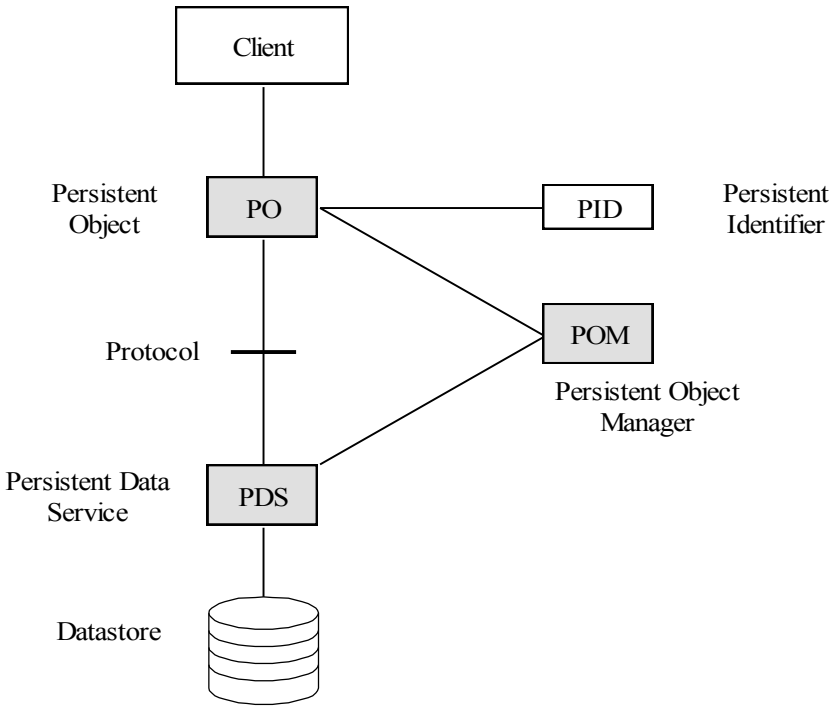

Figura 1.7.2 Persistent Object Service Arquitecture

A continuación se describen las componentes del POS:

- **Persistent Object**: representa al objeto del cual se espera que el cliente pueda acceder a su estado y que el mismo sea invariable a través del tiempo, excepto bajo las condiciones anteriormente mencionadas.
- **PID**: este es un identificador del objeto (distinto de los demás ya descriptos en capítulos precedentes). Este identificador representa al objeto en un medio de almacenamiento permanente.
- **Persistent Object Manager**: es la componente más compleja, ya que es la encargada de manejar el almacenamiento permanente en múltiples medios y con múltiples protocolos.
- **Persistent Data Service and Protocol**: el PDS es el componente que implementa la interfaz del protocolo y coordina las operaciones de persistencia.
- **Datastore**: es el medio donde se almacenan los objetos. Puede ser desde un simple archivo a una compleja base de datos.

Control de la presistencia

El POS provee dos alternativas para controlar la persistencia de los objetos:

- **Connection**: establece una relación entre el estado dinámico del objeto y su información persistente. Cuando se establece la conexión, la información del objeto debe ser vista como la misma. Cuando la conexión se termina, la información persistente debe mantener los valores que se tenían inmediatamente antes de concluirla. Las operaciones son **connect** y **disconnect**.
- **Store/Restore**: permite al cliente u objeto, controlar el movimiento de información entre el almacenamiento permanente y el uso dinámico. Las operaciones son **store** y **restore**.

Naming Service

El servicio de nombres (NS) permite asociar un nombre o jerarquía de nombres con una referencia a un objeto, para que los objetos sean localizados por nombre. A la acción de obtener una referencia de un objeto a través de un nombre se la llama *resolver el nombre*. La acción de asociar un nombre a una referencia se conoce como *ligar*, y el resultado de esta acción es una *ligadura*.

Los nombres pueden ser estructurados jerárquicamente usando *contextos*. Los contextos son similares a los directorios en un sistema de archivos y pueden contener tanto ligaduras como subcontextos.

El uso de referencias a objetos presenta dos dificultades, una es que la referencia es opaca (invisible al cliente), la segunda es que está compuesta por una larga secuencia de números. El servicio de nombres soluciona estos inconvenientes proveyendo una capa extra de abstracción para la identificación de objetos.

El uso típico de este servicio implica ligar a las implementaciones de los objetos al servicio cuando estas se inicializan y desligarlas antes de terminar.

Especificación de la interfaz

La interfaz central se llama **NamingContext** y contiene las operaciones para ligar nombres a referencias y crear subcontextos. Los **Names** son secuencias de **NamesComponents**. Los NamingContexts pueden resolver nombres con más de un componente resolviendo primero el primer componente a un subcontexto y pasando el resto al mismo para su resolución. Es un mecanismo recusrisvo.

El tipo Name

El módulo CosNaming provee las definiciones de los tipos que permiten resolver nombres:

```
module CosNaming{
        typedef string Istring;
        struct NameComponent {
                Istring id;
                Istring kind;
};
typedef sequence <NameComponent> Name;
```

Ligaduras (bindings)

El tipo Binding provee información sobre las ligaduras en un contexto:

```
// module CosNaming{

enum BindingType {nobject, ncontext};
struct Binding {
        Name binding_name;
        BindingType binding_type;
};
typedef sequence <Binding> BindingList;
```

El tipo **CosNaming::Binding** provee un nombre y un *flag* de tipo **BindingType**. El valor **ncontext** indica que un objeto ligado a un nombre es un **NamigContext**, en el cual se realizarán las futuras resoluciones. El valor **noobject** indica que esa ligadura no podrá ser usada para resolución de nombres.

Agregar Nombres a un Contexto

Existen dos operaciones para ligar un objeto a un nombre en un contexto y otras dos para ligar un contexto a un nombre.

```
// module CosNaming{

interface NamingContext{
        //se pasan por alto las declaraciones de las excepciones
        void bind(in Name n, in Object obj)
                raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void rebind(in Name n, in Object obj)
                raises(NotFound, CannotProceed, InvalidName);

        void bind_context(in Name n, in NamingContext nc)
                raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
        void rebind_context(in Name n, in NamingContext nc)
                raises(NotFound, CannotProceed, InvalidName);
```

Las operaciones **bind()** y **bind_context()** asocian un nuevo nombre con un objeto. En la segunda, el objeto debe ser del tipo **NamingContext**.

Las operaciones **rebind()** y **rebind_context()** trabajan del mismo modo que las anteriores, pero en vez de lanzar una excepción si el nombre ya existe, simplemente reemplazan la referencia al objeto.

Remover nombres de un contexto

La operación **unbind()** removerá el nombre asociado a la referencia al objeto de un contexto o uno de sus subcontextos:

```
void unbind(in Name n)
        raises(NotFound, CannotProceed, InvalidName);
```

Resolución de nombres

La operación **resolve()** retorna la referencia al objeto ligada a un nombre pasándole el mismo como parámetro:

```
Object resolve (in Name n)
        raises(NotFound, CannotProceed, InvalidName);
```

Esta operación se comporta del siguiente modo:

- Resuelve el primer componente del nombre (n), a una referencia a un objeto.
- Si no quedan más componentes por resolver, retorna la referencia al cliente.
- Si quedan más componentes para resolver, convierte la referencia al tipo **NamingContext** y llama a **resolve()** pasando al resto de las componentes como parámetros.

Excepciones

Estas son las excepciones que fueron omitidas anteriormente:

```
//interface NamingContext
enum NotFoundReason { missing_node, not_context, not_object};
exception NotFound {
```

```
        NotFoundReason why;
        Name rest_of_name;
};
exception CannotProceed {
        NamingContext cxt;
        Name rest_of_name;
};
exception InvalidName{};
exception AlreadyBound {};
exception NotEmpty{};
```

La excepción **NotFound** indica que el nombre no identifica a una ligadura.

La excepción **CannnotProceed** retorna una referencia a un objeto del tipo **NamingContext** y una parte del nombre original. Indica que la operación **resolve()** ha abandonado, por ejemplo por problemas de seguridad.

La excepción **InvalidName** indica que el nombre es sintácticamente incorrecto.

La excepción **AlreadyBound** puede ser disparada por operaciones **bind**. Informa al que invoca la operación que el nombre ya ha sido usado.

Por último, la excepción **NotEmpty** es disparada por la operación **destroy()** que veremos luego. Los contextos que contienen ligaduras no pueden ser destruidos.

Creación de contextos

Existen operaciones definidas para crear contextos en la interfaz NamingContext.

```
//interface NamingContext

NamingContext new_context();
NamingContext bind_new_context(in Name n)
        raises(NotFound, AlreadyBound, CannotProceed, InvalidName);
```

Los nuevos contextos pueden ser creados y utilizados solos o pueden ligarse a otros contextos usando **bind_context()**. También pueden ser creados y ligados a otros contextos en el momento de su creación usando **bind_new_context()**.

Destrucción de contextos

Cuando un contexto no será utilizado nuevamente, y todas las ligaduras que contenía fueron eliminadas, puede ser destruido.

```
//interface NamingContext
void destroy( )
        raises(NotEmpty);
```

Navegar contextos

Un NamingContext soporta la navegación de sus contenidos gracias a la operación list().

```
//interface BindingIterator se declara más adelante
        //interface NamingContext
        void list (in unsigned long how_many, out BindingList bl, out BindingIterator bi);
}; // fin de la interfaz NamingContext
```

Los parámetros de la operación **list()** permiten al cliente especificar cuántas ligaduras debe retornar en la secuencia **BindingList**. El resto será devuelto como un objeto iterador, que se explica a continuación.

Iteradores de ligaduras

Un objeto del tipo **BindingIterator**, será retornado al cliente si el número de ligaduras en un contexto excede al valor del argumento **how_many** en la operación **list()** invocada sobre el contexto.

```
//module CosNaming
```

```
interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long how_many,out BindingList bl);
        void destroy();
};
}; //fin del módulo CosNaming
```

Si existen ligaduras pendientes, la operación **next_one()** retorna TRUE y ubica un **Binding** en su parámetro de salida (out).

La operación **next_n()** retorna una secuencia de a lo sumo **how_many** ligaduras en el parámetro de salida **bl**.

La operación **destroy()** permite al iterador desalojar sus recursos y convertir su referencia en inválida.

Trading Service

Este servicio constituye las “Páginas Amarillas” para objetos, los cuales pueden “publicar” sus servicios.

Los **traders** son repositorios de referencias a objetos que se describen por un tipo de interfaz y un conjunto de valores de propiedades. Esta descripción de una interfaz se conoce como una *oferta de servicio*. Cada oferta de servicio tiene un *tipo de servicio*, que es una combinación del tipo de interfaz del objeto que está siendo publicado y una lista de propiedades que para las cuales el servicio debe proveer los valores.

Un **exporter** es un servicio que actúa como un agente que ubica la oferta de un servicio en un trader.

Un cliente que le pide a un trader que descubra un servicio es llamado **importer**. El importer provee al trader con la especificación de tipo de servicio y una expresión sobre las propiedades que ofrece ese servicio que describe los requerimientos del importer.

Los tipos de servicios son *templates* a partir de los cuales se crean las ofertas de servicios. Esto permite agrupar las ofertas de servicios que ofrecen la misma interfaz y realizar búsquedas eficientes en los traders. Y lo más importante es que permite a los exporters e importers usar la misma terminología para describir un conjunto de características, de modo que las expresiones de las propiedades sean siempre correctamente evaluadas.

La especificación de CORBA de este servicio define las siguientes interfaces:

- Service Type Repository.
- Trader Components.
- Lookup.
- Iterators.
- Register.
- Link.
- Admin.
- Proxy.
- Dynamic properties.

Al no ser el objetivo del trabajo describir en profundidad cada uno de los servicios de CORBA, se describirán brevemente los restantes de ellos. Las especificaciones de todos los servicios pueden ser encontradas en el sitio :http://www.omg.org.

Otros Servicios

Relationship Service

Este servicio provee un mecanismo para crear asociaciones dinámicas (o ligaduras) entre componentes que no saben nada el uno del otro. También provee mecanismos para navegar por dichas ligaduras que agrupan componentes.

Las relaciones se pueden caracterizar por:

- *Tipo*: tanto como los objetos, las relaciones también tienen tipo.
- *Roles*: las relaciones definen los roles que objetos relacionados deben asumir.
- *Grado*: las relaciones se caracterizan por el número de roles requeridos.
- *Cardinalidad*: es el máximo número de relaciones que puede involucrar un rol particular.
- *Semánticas*: definen los atributos y operaciones específicos de la relación.

Event Service

Permite a las componentes registrar dinámicamente su interés en eventos específicos. Este servicio define un objeto llamando **event channel** que reúne y distribuye eventos a través de las componentes.

Transaction Service

Extiende la semántica transaccional a aplicaciones distribuidas orientadas a objetos. Provee una coordinación de compromisos en dos fases entre componentes recuperables usando transacciones “chatas” o anidadas.

Concurrency Control Service

Provee un manejador de bloqueos, que puede realizarlos en función de transacciones o threads. Este servicio está especialmente diseñado para servir al Transaction Service.

Externalization Service

Provee un mecanismo standard para ingresar y extraer información de una componente usando un mecanismo de **streams**.

Query Service

Provee operaciones de SQL sobre objetos. Se basa en la especificación SQL3 y en el **Object Query Languaje** (OQL) del *Object Database Management Group* (ODMG).

Licensing Service

Provee operaciones para medir el uso de componentes para asegurar una justa compensación por su uso. El servicio soporta cualquier modelo de control de uso en cualquier momento del ciclo de vida de una componente. Soporta el cargo por sesión, por nodo, por creación de instancia y por sitio.

Properties Service

Provee operaciones que permiten asociar propiedades con cualquier componente. Usando este servicio se pueden asociar, por ejemplo, propiedades dependientes del estado de un componente, como ser un título o una fecha.

Time Service

Provee interfaces para la sincronización del tiempo en un ambiente de objetos distribuidos. También provee operaciones para el manejo de eventos disparados en función del tiempo.

Security Service

Provee un *framework* para la seguridad de objetos distribuidos. Soporta la autenticación, listas de control de acceso, confidencialidad, y delegación de credenciales entre objetos.

Collection Service

Provee interfaces para crear y manejar genéricamente las colecciones más comunes.

Capítulo VIII: CORBA Facilities

CORBA facilities son un conjunto de *frameworks* definidos en IDL que proveen servicios para el uso directo de objetos de aplicación. Como ya vimos en la introducción del capítulo anterior, se dividen en horizontales y verticales. Las horizontales son utilizables potencialmente por cualquier aplicación. Las verticales son compartidas por un número de aplicaciones en un mercado especializado (telecomunicaciones, servicios financieros, etc.).

CORBA Facilities Horizontales

Se dividen en cuatro áreas básicas:

- **User Interface**: de acuerdo al documento de la arquitectura, abarca las principales categorías de las interfaces de usuario: manejadores de ventanas, emuladores de terminales, y librerías de clases de objetos para la interfaz de usuario. Un **work management system** se encarga de controlar las sesiones y la visualización del escritorio; y el **task process automation**, permite la automatización a través de *scripts* y el grabado interactivo de macros. Esta categoría se divide en: **rendering management**, **compuond presentation management**, **user support**, **desktop management** y **scripting**.
- **Information Management**: facilita el modelado, la definición, el almacenamiento, la recuperación, el manejo y el intercambio de información.
- **Systems Management**: facilita el manejo de sistemas de información complejos. El objetivo es lograr estandarizar el desarrollo de aplicaciones de administración de sistemas abiertos. Define un *framework* para permitir reducir el esfuerzo en el desarrollo de aplicaciones para administrar sistemas distribuidos.
- **Task Management**: Facilita la automatización de los procesos de usuario y del sistema. El documento de la arquitectura divide esta categoría en: **workflow**, **agents**, **rule management** y **automation facility**.

CORBA Facilities Verticales

El objetivo del OMG es cubrir las necesidades de mercados verticales o sectores de la industria. No se encuentran aún en un estado avanzado de desarrollo.

Para mayor información se debe consultar la especificación en el sitio: http://www.omg.org/corba/cf2.html.

Parte II

CORBA y Java

Esta segunda parte está dedicada a la integración de CORBA con Java, y en el primer capítulo vemos los beneficios que aporta esta integración al desarrollo de aplicaciones distribuidas en la Web.

En el segundo capítulo se detalla la definición del mapeo de IDL a Java que es ejecutado a través del compilador correspondiente.

Capítulo I: CORBA se encuentra con Java

CORBA es mucho más que el *Object Request Broker* (ORB), es una plataforma ampliamente abarcativa de objetos distribuidos. CORBA extiende el alcance de las aplicaciones Java a través de las redes, los lenguajes de programación y los sistemas operativos. CORBA también le aporta a Java un amplio conjunto de servicios distribuidos: descubrimiento dinámico, transacciones, relaciones, seguridad y nombres.

Java es más que otro lenguaje de programación, es un *sistema de código móvil*, es un sistema operativo portable para correr objetos. Provee un modo de desarrollo más simple y nuevo, para generar aplicaciones cliente/servidor. Eventualmente, Java permitirá a los objetos CORBA, ejecutarse en cualquier ambiente, desde un *mainframe* y redes de computadoras hasta teléfonos celulares. Java simplifica la distribución de grandes sistemas CORBA gracias a sus *bytecodes*.

Mientras CORBA se ocupa de la transparencia de localización, Java se ocupa de la transparencia en la implementación. Por esto se complementan bien, ya que Java comienza donde CORBA termina.

La Web fue diseñada sobre standards abiertos y *cross-platform*. La figura 2.1.1 muestra el progreso de las tecnologías de la Web.

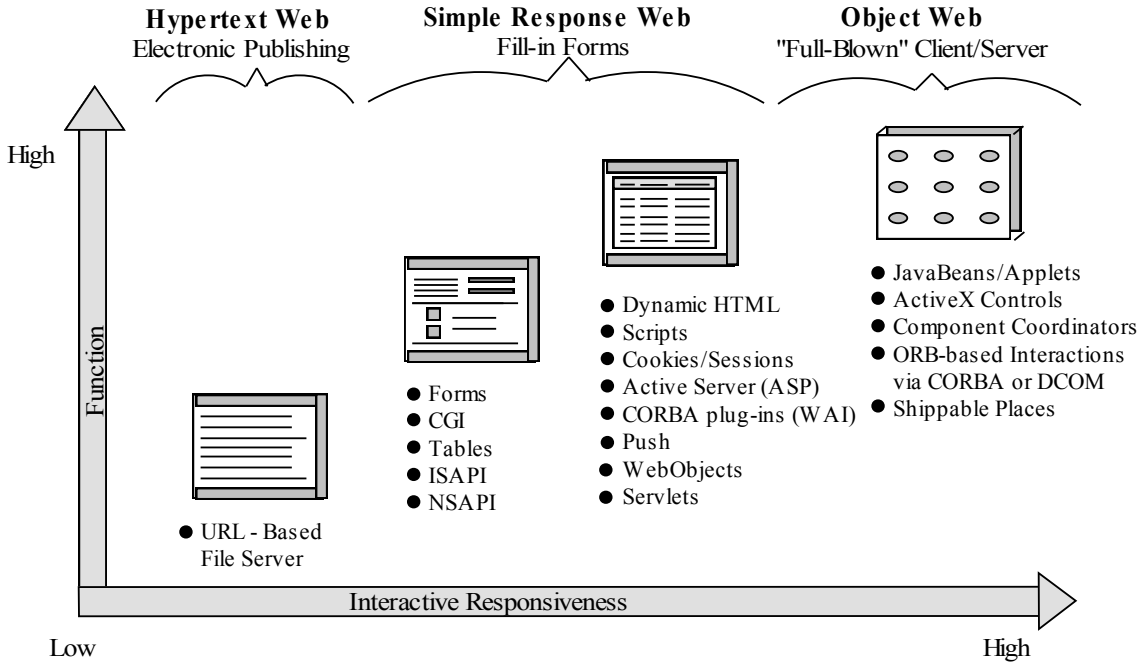

Figura 2.1.1 La evolución de la WEB

CORBA/Java y la Web

Para obtener los máximos beneficios de la interacción entre objetos, la Web debe ser agrandada con una infraestructura de objetos distribuidos. Los ORBs se especializan en la comunicación entre componentes sin limitaciones en cuanto a que la comunicación puede ser cliente-cliente, servidor-cliente y servidor-servidor. Las applets fueron el primer paso en cuanto a la comunicación servidor-cliente. Java es necesario pero no suficiente para crear la Object Web; Java necesita ser complementado con una infraestructura de objetos distribuidos, y es precisamente ahí donde se incorpora CORBA.

La Object Web nació oficialmente en junio de 1997, cuando Netscape lanza *Communicator* con un ORB CORBA/Java. Del lado del servidor, Netscape lanza ORBs CORBA/C++ y CORBA/Java con cada copia del *Enterprise Server 3.0*. La intersección entre la tecnología de objetos CORBA y Java es el primer paso en la evolución de la Object Web.

En la figura 2.1.2 vemos cómo un cliente típico Web interactúa con su servidor en la Object Web:

1. **El Web browser baja la página HTML:** en este caso, la página incluye referencias a applets Java embebidas.
2. **El Web browser recibe la applet del servidor HTTP:** el servidor HTTP envía la applet y el cliente la baja en el browser en forma de bytecodes.

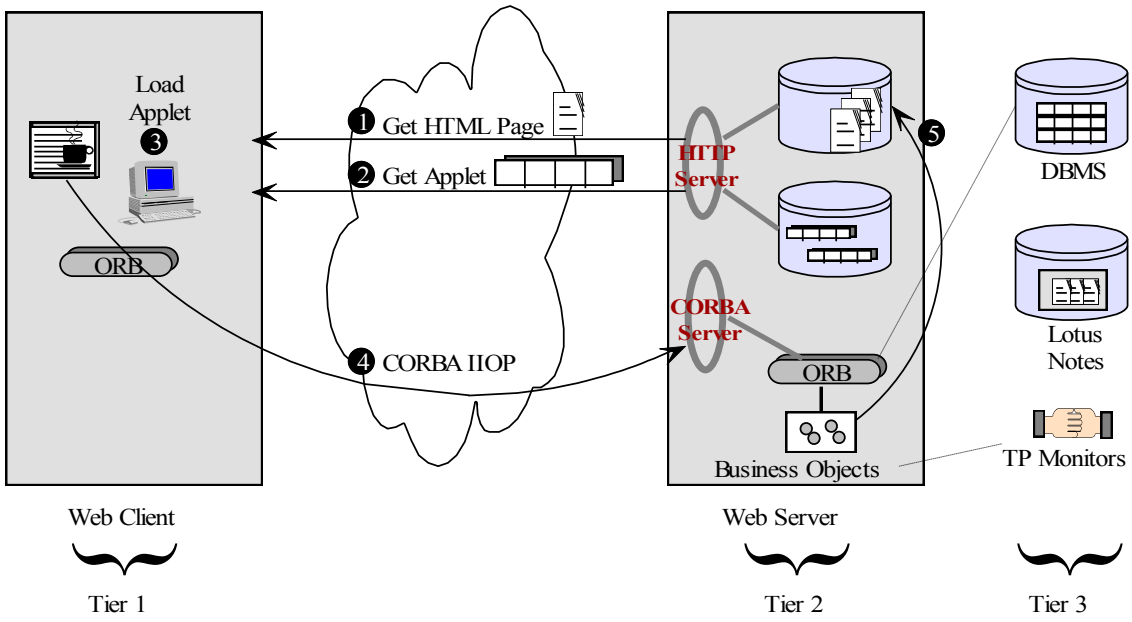

Figura 2.1.2 Cómo HTTP, CORBA y Java funcionan juntos

3. **El Web browser carga la applet:** la applet se carga en memoria luego de haber pasado por el control de seguridad.

4. **La applet invoca a objetos CORBA en el servidor:** la applet Java puede incluir stubs IDL, que permiten invocar a objetos en el servidor. La sesión entre la applet Java y el servidor CORBA persistirá hasta que cualquiera de las partes decida desconectarse.

5. **Generar dinámicamente la página siguiente:** los objetos del servidor opcionalmente pueden generar la próxima página HTML para el cliente. Esta generación dinámica de HTML es menos frecuente con la Object Web. En este nuevo ambiente, la aplicación del cliente es típicamente empaquetada como una sola página HTML con componentes embebidos. De este modo CORBA permite interactuar con el servidor cliqueando sobre una componente embebida sin tener que cambiar el contexto de la página para cada respuesta.

Aumentar la infraestructura de la Web con CORBA/Java provee dos ventajas inmediatas: se elimina el cuello de botella de CGI (que veremos en más detalle en la Parte III) y provee una infraestructura escalable y robusta para la comunicación servidor-servidor en la Web.

Sin CORBA, Java en la Web es simplemente un competidor de Dynamic HTML. De cualquier modo, CORBA no es el único modelo de objetos distribuidos para la Web. Veremos en la Parte III un serio competidor de CORBA: Microsoft DCOM.

Lo que Java le da a CORBA

Entre otras cosas, Java aporta a CORBA dos beneficios fundamentales:

- **Java simplifica la distribución de código en grandes sistemas CORBA**: el código Java puede ser desplegado y manejado centralmente desde un servidor. El código se actualiza una sola vez en el servidor y los clientes lo reciben como y cuando lo necesitan.
- **Java es un buen lenguaje para implementar objetos CORBA**: Java es un buen lenguaje para implementar tanto los objetos del cliente como los del servidor. Java es multithreading nativo, con *garbage collection* y manejo de errores, lo que simplifica el hecho de escribir objetos robustos. El modelo de objetos de Java complementa al de CORBA; y juntos utilizan el concepto de interfaces para separar la definición de un objeto de su implementación.

En definitiva, Java permite correr aplicaciones portables que “algún día” podrán correr sobre cualquier máquina existente. Tanto Java como CORBA nacieron sobre la base de la portabilidad, lo que hace que se complementen de la mejor manera.

Capítulo II: Mapeo de IDL a Java

CORBA IDL es la lengua que permite describir servicios en un mundo heterogéneo. A partir de la misma, con un compilador IDL-to-Java, se genera el código necesario en Java, evitando al programador realizar este trabajo manualmente.

Constructores Generales

En esta sección veremos los mapeos IDL-to-Java para los constructores generales de IDL, incluyendo módulos, excepciones, typecasts, atributos y el paso de parámetros.

CORBA Modules

Un *module* CORBA-IDL, mapea a un *package* Java con el mismo nombre que el módulo IDL. Ejemplo:

IDL:

```
module MisCosas{
}
```

Código Java:

```
package MisCosas;
...
```

CORBA Exceptions

La figura 2.2.1 muestra la jerarquía de excepciones de CORBA, donde se definen dos tipos de excepciones: 1) *system exceptions*, que son excepciones standard definidas por CORBA; 2) *user-defined exceptions*, que son definidas en IDL por el programador. Una excepción definida por el usuario es una estructura que contiene campos de datos.

En el mapeo CORBA/Java, las excepciones standard derivan indirectamente de **java.lang.RuntimeException**. Ellas mapean a clases finales Java que extienden **org.omg.CORBA.SystemException**. Es posible acceder tanto al código de la excepción como a strings que especifican el motivo de las mismas.

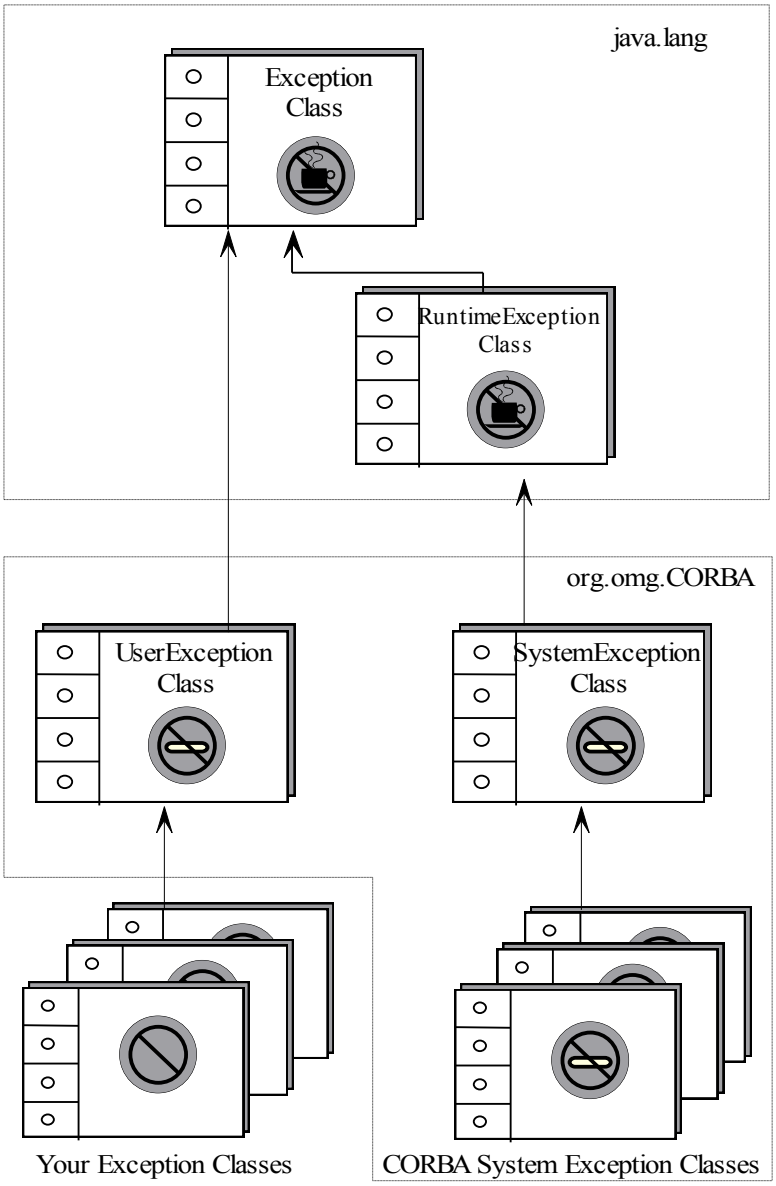

Figura 2.2.1 Jerarquía de Excepciones en CORBA

Las excepciones *user-defined* mapean a clases finales Java que extienden **org.omg.CORBA.UserException**. El compilador IDL genera clases Java por cada excepción que el usuario define; y estas clases proveen tanto variables de instancia por cada campo en la excepción como constructores. Ejemplo:

IDL:

```
module MiExcepcion{
        string explicacion;
};
```

Código Java:

```
final public class MiExcepcion extends org.omg.CORBA.UserException
{
        public java.lang.string explicacion;
        public MiExcepcion() {}
        public MiExcepcion(java.lang.string explicacion)
        {
                this.explicacion = explicacion;
        }
        public java.lang.String toString()
        {
                org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
                MiExcepcionHelper.insert(any,this);
                return any.toString();
        }
}
```

Parámetros CORBA y Clases Holder

CORBA IDL define tres modos para el pasaje de parámetros: *in*, *out* e *inout*. Java sólo soporta el modo *in*, que representa la semántica de pasaje de parámetros por valor. En consecuencia, el mapeo de los parámetros *in* es a parámetros normales de Java.

Los parámetros de IDL *out* e *inout* no tienen un mapeo directo en parámetros Java, por lo que es necesaria la implementación de mecanismos adicionales para proveer su funcionalidad. El mapeo define clases *Holder*, que sirven como contenedores para todos los tipos básico de IDL y los definidos por el usuario. Estos *Holders* implementan los modos adicionales de pasaje de parámetros que Java no provee. El cliente debe generar una instancia de la clase *Holder* apropiada que es pasada por valor por cada parámetro *in* e *inout*. Los contenidos de esta instancia son modificados por la invocación al servidor. El cliente luego utiliza los contenidos de esta instancia (posiblemente modificados) luego que retorna de la invocación.

El ORB provee clases *Holder* para todos los tipos de datos básicos como una parte del *package* **org.omg.CORBA**. El nombre de la clase *Holder* es el mismo que el tipo de dato más la palabra “Holder”. Por ejemplo, para el tipo **short**, la clase se llama **ShortHolder**. Cada clase *Holder* posee un constructor desde una instancia, un constructor por defecto y un miembro público llamado

value. El constructor por defecto asigna al campo **value** el valor por defecto que Java asigna al tipo determinado que se esté usando (por ejemplo, para boolean, false, para numéricos, cero, etc.).

El siguiente es el código para la clase ShortHolder:

```
final public class ShortHolder
{
        public short value;
        public ShortHolder() {}
        public ShortHolder(short initial)
        {
                value = initial;
        }
}
```

El siguiente es un ejemplo de como se utilizan las clases *Holder*:

```
/* Definición IDL */
interface Perro
{
                void ladrido (out short cant_veces);
        ....
}
```

Interfaz Java correspondiente:

```
public interface Perro extends org.omg.CORBA.Object
{
        public void ladrido(org.omg.CORBA.ShortHolder cant-veces);
        ...
}
```

Como se ve en el ejemplo, el parámetro CORBA *out* es sustituido por la clase *Holder* correspondiente. El cliente debe instanciar un objeto de esta clase antes de invocar el método.

Además, el compilador de IDL, genera clases Holder para los tipos definidos por el usuario, excepto para aquellos que se definen usando *typedef*. Para soportar stubs y skeletons portables, las clases Holder para los tipos definidos por el usuario también implementan la interfaz **org.omg.CORBA.portable.Streamable**.

Ejemplo de la clase *Holder* que un compilador generaría para la clase Perro:

```
final public class PerroHolder implements org.omg.CORBA.portable.Streamable
{
        public Perro value;
        public PerroHolder() {}

        public PerroHolder(Perro value)
        {
                this.value = value;
        }

        public void _read(org.omg.CORBA.portable.InputStream input)
        {
                value = PerroHelper.read(input);
        }

        public void _write(org.omg.CORBA.portable.OutputStream output)
        {
                PeroHelper.write(output, value);
        }

        public org.omg.CORBA.TypeCode _type()
        {
                return PerroHelper.type();
        }
}
```

Clases CORBA Helper

Las clases *Helper* contienen métodos que permiten manipular los tipos IDL en varios sentidos. El compilador de IDL genera una clase *Helper* por cada tipo IDL o interfaz que se define. El nombre de la clase es el nombre del tipo con la palabra “Helper” como sufijo. Las clases *Helper* proveen métodos estáticos que los clientes pueden usar para manipular los tipos de datos.

Estos métodos incluyen obtener el ID del repositorio, obtener el *typecode*, y leer y escribir el tipo desde y hacia un *stream*. Además provee un método estático *narrow* para *typecasts*.

El siguiente es un ejemplo de lo que un compilador IDL generaría para la clase Perro:

```
abstract public class PerroHelper
{
        public static Perro narrow(org.omg.CORBA.Object object)
        {
                ...
        }
        private static Perro narrow(org.omg.CORBA.Object object, boolean is_a)
        {
                ...
        }
        public static Perro bind(org.omg.CORBA.ORB orb)
        {
                ...
        }
        public static Perro bind(org.omg.CORBA.ORB orb, java.lang.Srting name)
        {
                ...
        }
        public static Perro bind(org.omg.CORBA.ORB orb,
                                 java.lang.String name,
                                 java.lang.string host,
                                 org.omg.CORBA.BindOptions options)
        {
                ...
        }


        private static org.omg.CORBA.ORB _orb()
        {
                ...
        }
        public static Perro read(org.omg.CORBA.portable.InputStream _input)
        {
```

```
        ...
    }
    public static void write(org.omg.CORBA.portable.OutputStream _output,
                             Perro value)
    {
        ...
    }
    public static void insert(org.omg.CORBA.Any any, Perro value)
    {
        ...
    }
    public static Perro excract(org.omg.CORBA.Any any)
    {
        ...
    }
    private static org.omg.CORBA.TypeCode _type;
    public static org.omg.CORBA.TypeCode type()
    {
        ...
    }
    public static java.lang.String id()
    {
        return "IDL:PERRO:".0";
    }
}
```

CORBA Attributes

Las interfaces de CORBA IDL tienen atributos. Cada uno de ellos es mapeado a un par de métodos Java con el mismo nombre del atributo, para asignar u obtener su valor. Si el atributo se declara como readonly en la interfaz IDL, solamente se mapea la operación que permite leer el valor.

El siguiente es un ejemplo:

```
// IDL
attribute short age;
readonly attribute long salary;
```

Código correspondiente en Java:

```
public void age(short age);
public short age();
public int salary();
```

Tipos básicos

Esta sección cubre los mapeos IDL-to-Java para los tipos básicos de CORBA, incluyendo: *constants*, *boolean*, *char*, *wchar*, *octet*, *string*, *wstring*, *short*, *long*, *longlong*, *float* y *double*.

CORBA Constants

Los valores *const* de CORBA son mapeados a campos *public static final* en la interfaz de Java correspondiente. Las constantes no declaradas dentro de un interfaz IDL son mapeadas a una interfaz pública con el mismo nombre de la constante. Ejemplo:

```
// IDL
interface ConstantTypes
{
        const long MyLong = -12345;
        const boolean MyTrue = TRUE;
        const char MyChar = 'A';
}
```

Código Java correspondiente:

```
public interface ConstantTypes extends org.omg.CORBA.Object
{
        final public static int MyLong = (int) -12345;
        final public static boolean MyTrue = (boolean) true;
        final public static char MyChar = (char) 'A';
}
```

Tipos Primitivos de CORBA

El resto de los tipos primitivos mapean en modo directo a los tipos primitivos de Java:

- **Booleans**: el tipo *boolean* de COBRA IDL mapea directamente en el tipo *boolean* de Java.
- **Caracteres**: el tipo *char* de CORBA IDL representa caracteres del conjunto ISO 8859.1. El tipo *wchar* es usado para representaciones de 16 bits como Unicode. En Java los caracteres son representados en Unicode, por lo tanto, tanto *char* como *wchar* mapean al tipo *char* de Java.
- **Octetos**: el tipo *octect* de CORBA IDL es un valor de 8 bits que mapea en el tipo *byte* de Java.
- **Strings**: Tanto el tipo *string* como *wstring* de CORBA IDL mapean en el tipo **java.lang.String** de Java.
- **Enteros**: CORBA IDL incluye 6 tipos de datos enteros; *shrot*, *long* y *longlong* en la versión con y sin signo. Java tiene 3 tipos de enteros: *short*, *int* y *long*. Los tipos de datos enteros de IDL mapean en los tipos correspondientes de Java de acuerdo a la longitud.
- **Punto Flotante**: los tipos de IDL *float* y *double* mapean a los tipos correspondientes de Java con los mismos nombres.

Los tipos básicos son usados como tipos de atributos, parámetros, constantes y campos en los tipos compuestos.

Tipos compuestos de datos

Los tipos compuestos de datos que se cubren en esta sección incluyen tanto a los tipos de datos definidos por el usuario (*interface*, *union*, *struct*, *enum*), como a los tipos parametrizados (*array* y *sequence*). También se incluye al tipo *Any*, que puede contener tanto a tipos simples como compuestos.

CORBA Interface

El tipo IDL *interface* mapea en una interfaz de Java con el mismo nombre. La interfaz Java contiene el mapeo de las operaciones definidas en la interfaz IDL. Los métodos son invocados sobre la referencia del objeto a esta interfaz. Ejemplo:

```
// IDL
interface Perro
{
        attribute short edad;
        void ladra (out short veces);
}
```

Código Java correspondiente:

```
public interface Perro extends org.omg.CORBA.Object
{
        public void edad(short edad);
        public short edad();
        public void ladra(org.omg.CORBA.ShortHolder veces);
}
```

CORBA Sequence

Una secuencia es un arreglo de elementos unidimensional y de longitud variable, donde los elementos pueden ser cualquier tipo IDL definido. Puede ser ligada opcionalmente a una longitud máxima. Se debe dar un nombre a la secuencia usando *typedef* antes de ser usada.

Mapea a un *array* de Java con el mismo nombre. Ejemplo:

```
interface ejemploSeq
{
        //unbounded sequence
        typedef sequence<Perro> myUnboudedSeq;
        // bounded sequence con un máximo de 60
        typedef sequence<Perro, 60> myBoundedSeq;
        // ejemplo del uso
        void seqtest(in myBoundedSeq val1, in myUnboundedseq val2);
}
```

Código Java correspondiente:

```
public interface ejemploSeq extends org.omg.CORBA.Object
{
        public void seqtest(Perro[] val1, Perro[] val2);
}
```

El límite de la secuencia myBoundedSeq es chequeado cuando el argumento es convertido a la representación interna de Java.

CORBA Array

Como en una secuencia, un arreglo puede contener elementos de cualquier tipo de datos válido de IDL. La diferencia con la secuencia, es que el arreglo puede ser multidimensional y es siempre de longitud fija. Por este último motivo, el arreglo es menos flexible que la secuencia en algunas circunstancias.

El arreglo mapea de la misma manera que las secuencias con longitud fija. La longitud se chequea cuando el arreglo es convertido al formato interno de Java como un argumento en una operación. Ejemplo:

```
interface ejemploArray
{
        const long limite = 60,.
```

```
        typedef Perro miArreglo[limite];
        // ejemplo del uso
        void arraytest(in miArreglo val2);
}
```

Código Java correspondiente:

```
public interface ejemploArreglo extends org.omg.CORBA.Object
{
        public void arraytest(Perro[] val1);
}
```

CORBA Structs

Una estructura permite empaquetar campos de distintos tipos. El tipo IDL *struct* mapea en una clase Java con el mismo nombre que contiene una variable de instancia por cada campo en la estructura. Existen dos constructores por cada estructura. El primero es el constructor por defecto que setea todos los campos en *null*. El segundo toma los valores de los campos como argumentos e inicializa cada uno de los campos con dichos valores. Ejemplo:

```
interface MyStruct
{
        short edad;
        string nombre
}
```

Código Java correspondiente:

```
final public class MyStruct
{
        public short edad;
        public java.lang.String nombre;

        //Constructores
```

```
        public MyStruct() {}
        public MyStruct(short edad, java.lang.String nombre)
        {
                this.edad = edad;
                this.nombre = nombre;
        }

        public java.lang.String toString()
        {
                org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
                MyStructHelper.insert(any, this);
                return any.toString();
        }
}
```

CORBA Enums

El tipo enumerativo permite asignar identificadores a los miembros de un conjunto de valores. El tipo *enum* de IDL mapea en una clase Java con el mismo nombre. La clase Java incluye:

1. Dos miembros de datos estáticos por label.
2. Un constructor privado.
3. Un método que retorna el valor entero.
4. Un método *from_int* que retorna el *enum* con el valor especificado.
5. Un método *to_string*.

Ejemplo:

```
enum MyEnum(amarillo, rojo, azul);
```

Código Java correspondiente:

```
final public class MyEnum
{
        // Un par de miembros estáticos por label;
        final public static int _amarillo = 0;
        final public static MyEnum none = new MyEnum(_amarillo);

        final public static int _rojo = 10;
        final public static MyEnum first = new MyEnum(_rojo);

        final public static int _azul = 3;
        final public static MyEnum second = new MyEnum(_azul);

        //Constructor privado
        private int _value;
        private MyEnumt(int value)
        {
                this._value = value;
        }
        //Método que retorna el valor entero
        public int value()
        {
                return _value(),
        }
        //Método from_int que retorna el enum con el valor especificado
        public static MyEnum from_int(int $value){
                switch($value){
                        case _amarillo:     return amarillo;
                        case _rojo:         return rojo;
                        case _azul: return azul;
                                default: throw new
        org.omg.CORBA.BAD_PARAM("Enum out of range:[0.."+ (5 - 1) + "]:" + $value);
                }
        }
        // tostring
        public java.lang.String toString()
```

```
        {
                org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
                MyEnumtHelper.insert(any, this);
                return any toString();
        }
}
```

CORBA Union

Una union discriminada permite definir una estructura que referencia sólo a uno de varios datos alternativos en un momento determinado (sólo un miembro está en memoria en ese momento). La union utiliza un discriminante para indicar qué miembro de la union debe seleccionarse.

La union mapea en una clase Java con el mismo nombre. La clase Java incluye:

1. Un constructor por defecto.
2. Un método para acceder al discriminante.
3. Un método para acceder a cada una de las bifurcaciones.
4. Un método para modificar cada una de las bifurcaciones.
5. Un método *to_string*.

Ejemplo:

```
union AnimalUnion switch (short)
{
        case 1: Perro perro;
        case2: Gato gato;
        default: Animal otro,
}
```

Código Java correspondiente:

```
final public class AnimalUnion
{
        private java.lang.Object _object;
        private short _disc;

        //Constructor
        public AnimalUnion() {}

        //Acceso al discriminante
        public AnimalUnion discriminator()
        {
                return _disc;
        }

        // Acceso a las bifurcaciones
        public Perro perro()
        { //No se muestra el codigo
        ....
        }
        public Gato gato()
        { //No se muestra el codigo
        ....
        }
        public Animal otro()
        { //No se muestra el codigo
        ....
        }

        // Modificadores
        public void perro(Perro value)
        { //No se muestra el codigo
        ....
        }
```

```
        public void gato(Gato value)
        { //No se muestra el codigo
        ....
        }
        public void otro(Animal value)
        { //No se muestra el codigo
        ....
        }

        // tostring
        public java.lang.String toString()
        {
                org.omg.CORBA.Any any = org.omg.CORBA.ORB.init().create_any();
                AnimalUnionHelper.insert(any, this);
                return any toString();
        }
}
```

CORBA Typedef

Permite definir nuevos nombres para tipos de datos (o alias), para un tipo IDL. Java no soporta construcciones de este tipo. En consecuencia, las declaraciones CORBA *typedef* mapean al original cada vez que aparece el tipo *typedef*. El ejemplo es trivial, por lo que se omite.

CORBA Any

El tipo IDL *Any*, mapea en la clase **org.omg.CORBA.Any**. Es una estructura que permite extraer e insertar valores, de tipos IDL predefinidos, en tiempo de ejecución. Se invocan los métodos *insert_xxx* para inicializar o actualizar el objeto *Any*. Las funciones *extract_xxx* retornan un valor contenido en el objeto *Any*.

El ORB es la fábrica para los objetos *Any*. El siguiente ejemplo muestra como se crea un objeto *Any* que luego se inicializa con un *short*:

```
org.omg.CORBA.Any x = org.create_any();
```

```
x.insert_short((short)4);
```

El valor se puede recuperar del siguiente modo:

```
short y = x.extract_short();
```

Sumario de los mapeos de IDL a Java

CORBA IDL	Java
Construcciones Generales	
module	package
user exceptions	Clase Java que extiende org.omg.CORBA.UserException que a su vez extiende java.lang.Exception
system exceptions	Clase Java org.omg.CORBA:SystemException que extiende java.lang.RuntimException
Parámetros in	Parámetros normales de Java
Parámetros out e intout	Clases Java Holder (generadas por IDL e instanciadas por el cliente
typecast (o narrow)	Clases Java Helper
atributos	Par de métodos con el mismo nombre del atributo para accederlos y modificarlos
Tipos primitivos	
const	Campo public static final
boolean, TRUE, FALSE	boolean, true, false
char, wchar	char
octet	byte
string, wstring	java.lang.String
short, unsigned short	short
long, unsigned long	int
long long, unsigned long long	long
float	float
double	double
Tipos construidos	
interface	interface
sequence	array
array	array
struct	Clase de Java con el mismo nombre que la estructura
enum	Clase Java con el mismo nombre que el tipo enumerativo
union	Clase Java con el mismo nombre que el tipo union, con operaciones get/set para cada uno de los campos
typedef	Java no tiene un constructor typedef
any	Clase Java **org.omg.CORBA.Any**

Mapeos del lado del servidor

Como ya hemos visto anteriormente, la especificación IDL, cuando es compilada, realiza mapeos tanto del lado del servidor como del cliente, que reflejan la visión que cada uno de ellos tendrá del objeto definido por la interfaz.

CORBA soporta dos tipos de mapeos del lado del servidor para implementar una interfaz IDL: *ImplBase inheritance* y *Tie delegation*. El primero es el más utilizado, y el programador implementa la interfaz especificada en IDL usando una clase que hereda de la clase **xxxImplBase**, donde xxx es el nombre de la interfaz IDL. Como la clase xxxImplBase generada por el compilador IDL es el equivalente en Java de la interfaz IDL, generando una clase que hereda de ésta, se implementan las operaciones definidas para la interfaz IDL.

En el mecanismo de delegación, la interfaz se implementa usando dos clases: 1) una clase generada por el compilador IDL que hereda de **xxxImplBase**, pero que delega todas las llamadas a una clase de implementación; 2) una clase que implementa la clase generada por el compilador IDL llamada **xxxOperatios**, en la que se definen las funciones.

El mecanismo *Tie* agrega niveles de indirección, por lo que es aconsejable evitar su uso, ya que degrada la performance del servidor.

Parte III:

CORBA/Java ORBs y sus Competidores

El objetivo de esta tercera parte es comparar a CORBA con otras tecnologías, describiéndolas en cada uno de los casos.

En el primer capítulo la comparación es entre los CORBA/Java ORBs y el mecanismo de sockets. En el segundo capítulo la comparación es contra HTTP/CGI, en el tercero contra RMI y por último en el quinto veremos como CORBA compite contra su más fuerte oponente que es DCOM de Microsoft.

Cada capítulo finaliza con una conclusión donde podemos apreciar los resultados en cuanto a las funcionalidades que provee cada tecnología.

Capítulo I: Sockets vs. CORBA/Java ORBs

Antes de que CORBA y RMI aparecieran en escena, el único modo por el cual un programa Java podía comunicarse con el resto del mundo era usando el *package java.net*. Este package provee dos útiles servicios de comunicación: 1) un conjunto de clases que permiten bajar y manipular objetos URL; 2) una implementación nativa de Java de los *Berkeley sockets*. En este capítulo veremos conceptualmente el funcionamiento de estos sockets, tanto los basados en datagramas como los basados en *streams*.

Berkeley Sockets

Son el primer protocolo de comunicación *peer-to-peer* para conectarse a través de TCP/IP. Nacieron en 1981 como una interfaz genérica del Unix BSD 4.2 que proveería comunicación interprocesos Unix-to-Unix (IPC). En 1985 Sun introdujo NFS y RPC sobre sockets. En la actualidad están soportados virtualmente sobre todos los sistemas operativos.

Un socket es un *endpoint* de la comunicación peer-to-peer, ya que tiene un nombre y una dirección de red. Desde la perspectiva del programador, el socket esconde los detalles de la red. Típicamente existen tres tipos:

- ***Datagram sockets****:* proveen una interfaz a UDP (User Datagram Protocol). UDP maneja las transmisiones de red como paquetes independientes y no provee garantías de que los paquetes lleguen a destino, ya que no existen acuses de recibo en el protocolo. UDP es un protocolo no orientado a la conexión.
- ***Stream sockets***: proveen una interfaz al protocolo de transporte confiable TCP. TCP sí es un protocolo orientado a la conexión que provee un servicio basado en sesiones. El protocolo garantiza que los paquetes lleguen y se encarga de la retransmisión en caso de error. No existen límites impuestos en los datos, ya que son considerados como un *stream* de bytes.
- ***Raw sockets***: proveen una interfaz a una capa más baja de protocolos, como la capa IP. En general se utilizan para testear nuevos protocolos o características más avanzadas de protocolos existentes.

Una dirección de un socket sobre una red TCP/IP consiste de dos partes: una dirección de IP y una de puerto. La dirección de puerto es el punto de entrada a una aplicación que reside en el host.

La clase **InetAddress** del package java.net se utiliza para encapsular las direcciones IP que son vistas como objetos que retornan información a través de las operaciones especificadas en esta clase.

El mejor modo de ver como funcionan los sockets es a través de un ejemplo que muestra una interacción cliente/servidor. La figura 3.1.1 muestra una transacción usando stream sockets. El cliente pide el balance de una cuenta y el servidor lo retorna. Los pasos son los siguientes:

1. Crear los endpoints de los sockets: cada proceso que utiliza sockets debe crearlos e inicializarlos utilizando la llamada *socket()*.

2. Establecer el puerto del servicio: el proceso servidor debe ligar sus sockets a un único puerto para poder hacerse conocido en la red.

3. Escuchar pedidos de conexiones: los procesos del servidor que utilizan stream sockets deben llamar a *listen()* para indicar que están listos para aceptar pedidos de los clientes.

4. Conectarse al servidor: el cliente emite una llamada *connect()* sobre el socket para iniciar una conexión sobre el puerto del servicio. El cliente puede bloquearse opcionalmente hasta que la conexión es aceptada por el servidor.

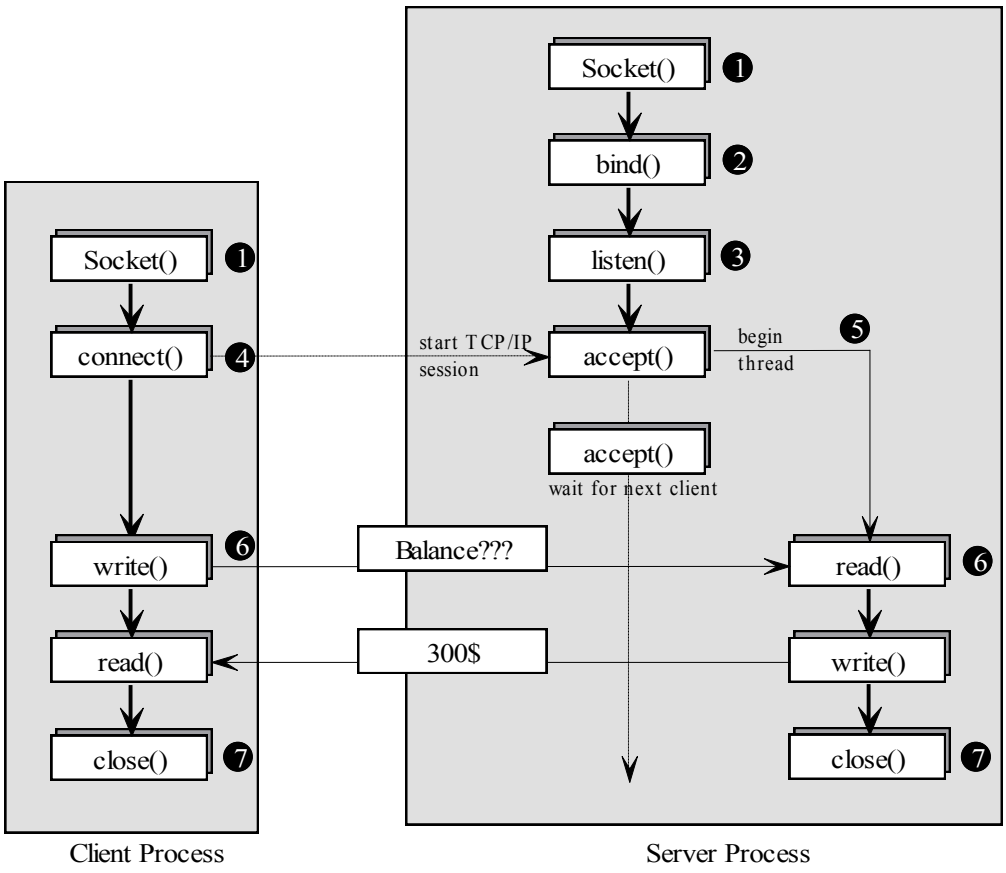

Figura 3.1.1 Stream Sockets

5. Aceptar la conexión: el servidor acepta la conexión sobre el socket con la llamada *accept()*. En un ambiente de threads múltiples, el servidor iniciará un nuevo thread para atender los pedidos del cliente sobre un socket separado. El thread original emite nuevamente la llamada *accept()* para esperar pedidos de otros clientes.

6. **Conducir la interacción cliente/servidor:** una vez realizada la conexión, los clientes y los servidores realizan llamadas para intercambiar información. Los stream sockets utilizan las llamadas *write()* y *read()* para enviar información sobre el socket, y los datagram sockets utilizan *send()* y *receive()*.

7. **Cerrar el socket:** los sockets deben cerrarse cuando la interacción entre el cliente y el servidor concluye.

Conclusiones

Los sockets son un mecanismo primitivo comparándolos con CORBA. La siguiente es una tabla comparativa:

Característica	CORBA IIOP	Java Sockets	
		Datagram	Data Stream
Comunicaciones confiables	SI	NO	SI
Marshaling de parámetros	SI	NO	NO
Descripción de interfaces	SI (vía IDL)	NO	NO
Descubrimiento dinámico	SI (vía IR y DSI)	NO	NO
Data typing de parámetros	SI (independiente de lenguaje y OS)	NO	SI (sólo Java-to-Java)
Seguridad	SI (vía CORBA y SSL)	SI (vía SSL)	SI (vía SSL)
Transacciones	SI	NO	NO

Los sockets han sido un gran aporte para la comunicación sobre TCP/IP, y actualmente se utilizan ampliamente. Pero hoy existen herramientas que proveen muchos más servicios y posibilidades, tanto para los diseñadores como para los programadores, a la hora de implementar sus aplicaciones distribuidas.

Capítulo II: HTTP/CGI vs. CORBA/Java ORBs

HTTP/CGI es actualmente el modelo predominante para crear soluciones cliente/servidor de 3 capas sobre Internet. El *Hypertext Transfer Protocol* (HTTP) provee semánticas simples similares a RPC sobre sockets. Se pueden usar muy efectivamente para acceder a recursos que viven en el espacio URL. Todos sabemos que los URLs apuntan a páginas HTML. De todos modos, los URLs también proveen un esquema de nombres que pueden utilizarse para identificar recursos en la Web, incluyendo páginas de la Web, documentos, imágenes, clips de sonido, applets y programas. De este modo se pueden usar los URLs dentro de un mensaje HTTP para localizar un programa servidor en Internet.

El Web server típco solo sabe como manejar documentos HTML. Cuando recibe el pedido para un programa, simplemente invoca el recurso nombrado en el URL y le dice que se haga cargo del pedido. El servidor pasa el pedido y sus parámetros al programa usando un protocolo llamado *Common Gateway Interface* (CGI). Se pueden escribir programas servidores CGI en cualquier lenguaje que pueda leer standard input y pueda escribir a standard output (incluyendo Java). El servidor HTTP lanzará la aplicación Java y la máquina virtual cada vez que el cliente invoque un pedido HTTP para esa aplicación. La aplicación se ejecuta y retorna sus resultados al servidor en formato HTML/HTTP, y el servidor devuelve los resultados al cliente.

Los clientes y los servidores HTTP usan representaciones de datos MIME para describir (y negociar) los contenidos de los mensajes. HTTP define protocolos para invocar comandos comunes y pasar sus parámetros. En consecuencia, HTTP/CGI provee todos los mecanismos necesarios para crear aplicaciones cliente/servidor basadas en la Web y usando Java. Por supuesto, todas las interacciones cliente/sevidor, deben pasar primero a través de un servidor HTTP para alcanzar al programa CGI.

Hypertext Transfer Protocol

Como ya se ha mencionado, HTTP es el RPC sobre TCP/IP. Se usa para acceder a recursos con nombres URL. HTTP es un RPC sin estado . El cliente establece una conexión con el servidor remoto. El servidor primero procesa el pedido, retorna un resultado , y luego cierra la conexión. Un cliente(típicamente un Web browser) pide una página de hipertexto y luego emite una secuencia de pedidos separados para obtener cualquier imagen referenciada en el documento. Una vez que el cliente ha obtenido las imágenes, el usuario puede activar un link de hipertexto y moverse a otro documento. HTTP típicamente crea una nueva conexión para cada pedido. El cliente debe esperar una respuesta antes de enviar un nuevo pedido.

Los clientes y los servidores deben negociar sus representaciones de datos cada vez que se realice una conexión. Los servidores de HTTP son sin estado, lo que significa que no tienen memoria de las conexiones anteriores. El hecho de que los servidores HTTP funcionen de esta manera los hace muy eficientes, ya que aceptan conexiones, cumplen con el pedido y rápidamente las cierran. De todos modos, el estado es necesario hasta para las más simples interacciones cliente/ servidor. Esto puede ser un problema en el actual protocolo HTTP.

Una interacción cliente/servidor usando HTTP consiste en un simple intercambio pedido/respuesta . Un pedido se compone de tres partes:

- ***Línea de pedido:*** está formada por tres campos de texto separados por espacios en blanco. El primer campo especifica el método (o comando) a ser aplicado a un recurso del servidor. El método más común es GET, que pide al servidor que envíe al cliente una copia del recurso. El segundo, especifica el nombre del recurso. El tercero identifica la versión del protocolo usado por el cliente.
- ***Campos de encabezado del pedido:*** pasan información adicional al servidor sobre el pedido y sobre el cliente mismo. Estos campos actúan como parámetros de RPC.
- ***Cuerpo:*** es utilizado a veces por los clientes para pasar información adicional al servidor.

La respuesta que el servidor envía al cliente está formada por:

- ***Línea de encabezado de la respuesta:*** retorna la versión de HTTP, el estado de la respuesta y la explicación de dicho estado.
- ***Campos de encabezado de la respuesta:*** retorna información que describe los atributos del servidor y del documento HTML enviado al cliente.
- ***Cuerpo:*** típicamente contiene el documento HTML.

CGI

Los formularios en la WEB son algo muy común. Cada vez que encontramos un formulario, debemos apretar un botón “Submit” que hace que el browser colecte la información ingresada y la envíe al servidor en un mensaje HTTP.

Un típico Web server no sabe que hacer con un formulario, ya que no es un documento HTML ordinario. Por lo tanto, el servidor invoca al programa o recurso nombrado en el URL y le dice que se haga cargo del pedido. El servidor pasa el método pedido y sus parámetros al programa en el “back-end” usando un protocolo *llamado Common Gateway Interface* (CGI). El programa ejecuta el método y retorna los resultados al Web server en formato HTML usando CGI. El Web server trata los resultados como un documento normal que envía al cliente.

CGI hace posible, para un cliente de Internet, actualizar una base de datos en un servidor en el “back-end”.

En la figura 3.2.1 vemos los elementos de una arquitectura cliente/servidor de tres capas.

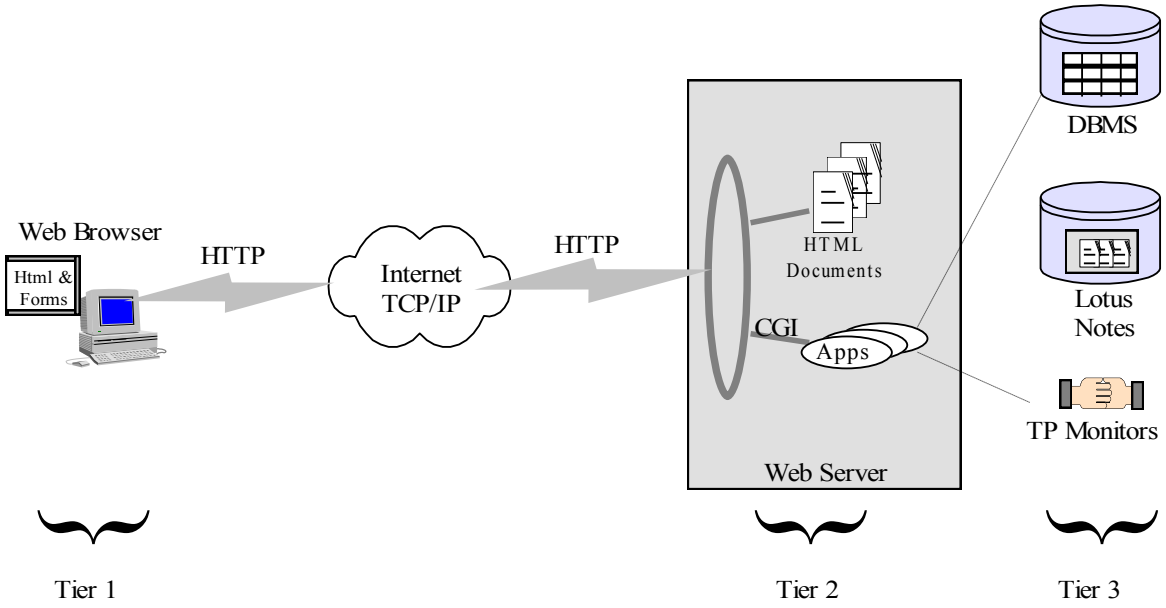

Figura 3.2.1 Arquitectura de tres capas cliente/servidor con HTTP/CGI

Conclusiones

En la siguiente tabla veremos una comparativa de algunas de las características de CORBA IIOP y HTTP/CGI.

Característica	CORBA IIOP	HTTP/CGI
Comunicaciones confiables	SI	SI
Estado a través de invocaciones	SI	NO
Marshaling de parámetros	SI	NO
Descripción de interfaces	SI (vía IDL)	NO
Descubrimiento dinámico	SI (vía IR y DSI)	NO
Data typing de parámetros	SI (independiente de lenguaje y OS)	NO
Seguridad	SI (vía CORBA y SSL)	SI (vía SSL o S-HTTP)
Transacciones	SI	NO

La conclusión es que HTTP/CGI es un protocolo muy limitado, que no es capaz de mantener estados, y además es lento. Muchos preveen que CORBA IIOP reemplazará a HTTP/CGI muy pronto, ya que es una elección mucho más conveniente no sólo por performance sino también porque es una alternativa que aporta muchos más recursos a la hora de diseñar aplicaciones sobre Internet.

Capítulo III: RMI vs. CORBA/Java ORBs

El *Remote Method Invocation* (RMI), parte del JDK 1.1, fue diseñado para soportar invocaciones remotas a métodos a través de *Java virtual machines*. RMI integra directamente un modelo de objetos distribuidos en el lenguaje Java.

Remote Method Invocation

Los objetos RMI son objetos remotos Java cuyos métodos pueden ser invocados desde otra JVM, incluso a través de una red. Los métodos sobre objetos remotos pueden invocarse del mismo modo que sobre los locales. En este aspecto, es mucho más parecido a CORBA, y también se parece en que permite pasar una referencia a un objeto remoto como un argumento, o retornarla como un resultado.

Los clientes RMI interactúan con objetos remotos a través de sus interfaces públicas. No interactúan nunca con las clases que implementan esas interfaces.

Contrariamente a lo que sucede en una invocación local, RMI pasa objetos locales como parámetros por valor y no por referencia, ya que una referencia a un objeto local es de utilidad sólo dentro de una única máquina virtual. RMI usa el servicio de serialización de objetos para aplanar el estado de un objeto Java en un *stream* que puede ser pasado como parámetro dentro de un mensaje.

Cuando un objeto remoto es pasado como parámetro o devuelto como resultado en una invocación remota, lo que se pasa es el stub del objeto remoto.

RMI provee nuevas interfaces y clases que permiten encontrar objetos remotos, cargarlos y ejecutarlos en modo seguro. Actualmente RMI provee un servicio de nombres que permite la localización de objetos remotos. También provee un cargador de clases que permite a los clientes bajar stubs del servidor.

Para el manejo de excepciones remotas, RMI extiende las clases de excepciones de Java.

En un sistema distribuido, un recolector de basura debe poder eliminar automáticamente aquellos objetos que no son referenciados por ningún cliente. RMI utiliza un esquema de contador de referencias que mantiene la cantidad de referencias externas vivas a los objetos de un servidor en una máquina virtual. En este caso, una referencia viva no es más que una conexión cliente/servidor sobre TCP/IP.

Stubs y Skeletons

RMI usa un mecanismo standard para comunicar objetos remotamente: *stubs* y *skeletons*. Un stub para un objeto remoto, actúa como un representante local del objeto remoto en el cliente. Cuando el cliente desea realizar una invocación en un objeto remoto, lo que hace es invocar un método en el stub local. El stub se encarga de hacer llegar el pedido al objeto remoto. Cuando un método en el stub es invocado, éste hace lo siguiente:

- Inicia una conexión con la VM (*virtual machine*) que contiene al objeto remoto.
- *Marshals* (escribe y transmite) los parámetros a la VM remota.
- Espera el resultado de la invocación.
- *Unmarshals* (lee) el valor o la excepción devueltos.
- Retorna el resultado al que realizó la invocación.

El stub oculta la serialización de parámetros y la comunicación a nivel de red.

En la VM remota, cada objeto debe tener su skeleton correspondiente. El skeleton es responsable de despachar la invocación a la implementación del objeto. Cuando un skeleton recibe una invocación, hace lo siguiente:

- *Unmarshals* (lee) los parámetros.
- Invoca el método en la implementación del objeto remoto.
- *Marshals* (escribe y transmite) el resultado (valor o excepción) al cliente.

En el JDK 1.2 fue introducido un protocolo de stubs adicional que elimina la necesidad de usar skeletons. En su lugar se utiliza un código genérico que realiza las tareas de los mismos.

El proceso de desarrollo usando RMI

La figura 3.3.1 muestra los pasos que se deben seguir para construir una aplicación cliente servidor y ejecutarla. Veremos los pasos necesarios uno por uno.

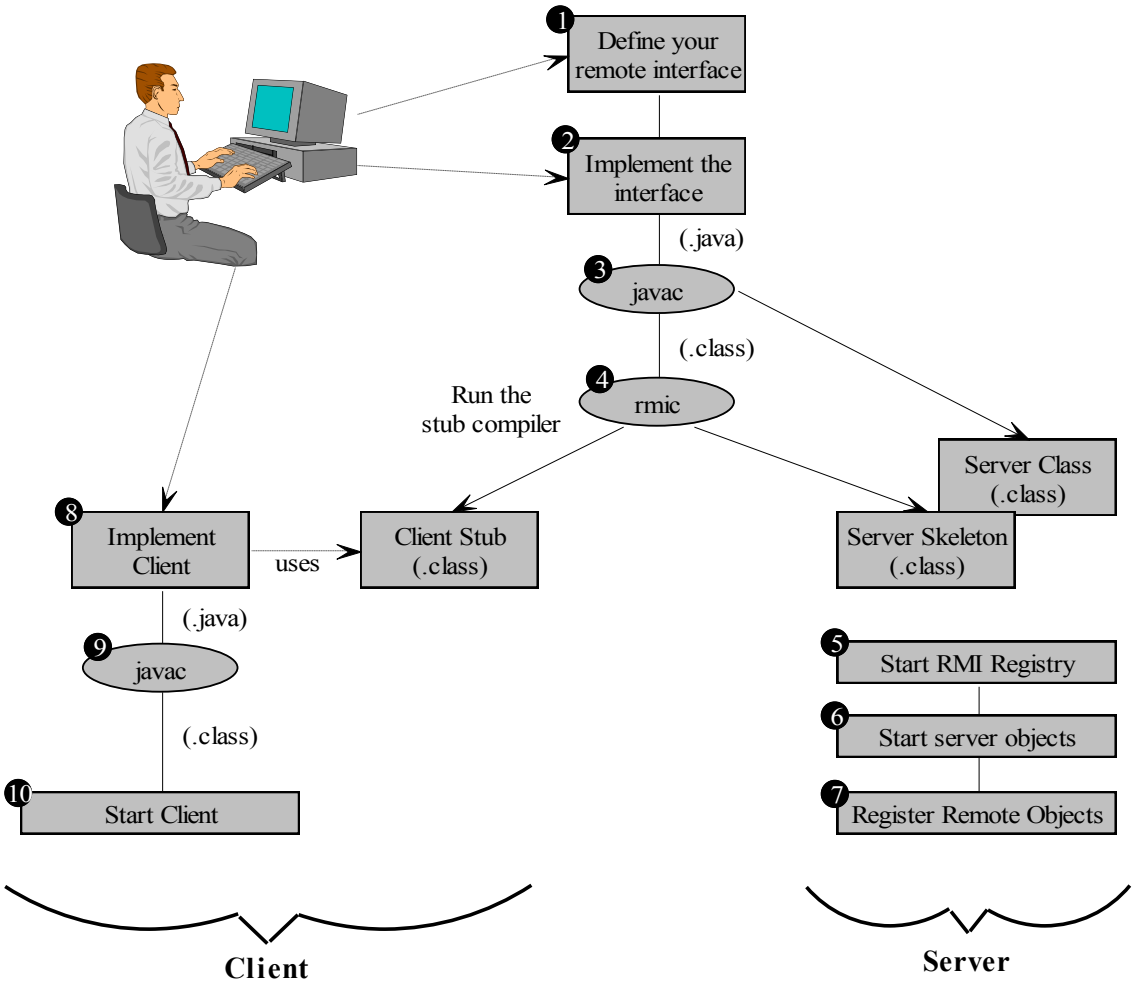

Figura 3.3.1 Creación de una aplicación cliente/servidor usando RMI

1. **Definir la interfaz remota:** el objeto servidor debe declarar sus servicios vía una interfaz remota. Esto lo hace extendiendo la interfaz **java.rmi.Remote**. Cada método en una interfaz remota debe "disparar" la excepción **java.rmi.RemoteException**.

2. **Implementar la interfaz remota:** se debe proveer una clase server que implemente la interfaz definida. Esta clase se debe derivar de **java.rmi.UnicastRemoteObject**.

3. **Compilar la clase server:** se compila usando el compilador de Java *javac*.

4. **Ejecutar el compilador de stub:** RMI provee un compilador de stub llamado *rmic*. Se debe ejecutar a partir de la clase generada por el compilador Java para generar los stubs del cliente y los skeletons del servidor. Como en CORBA, los stubs de RMI proveen *proxies* en los clientes para los objetos remotos. Ellos preparan los llamados remotos y los envían al servidor. El skeleton en el lado del servidor, recibe el llamado, ordena los parámetros y luego invoca el método en la clase correspondiente.

5. **Iniciar el registry RMI en el servidor:** RMI define interfaces para un servicio de nombres no persistente llamado *Registry*. Esto permite registrar y recuperar objetos usando simplemente sus nombres. Cada proceso servidor soporta su propio registry, o se puede usar uno único que soporte todas las máquinas virtuales en el nodo servidor.

6. **Iniciar los objetos del servidor:** se deben cargar las clases del servidor y luego crear instancias de los objetos remotos.

7. **Registrar los objetos remotos en el registry:** se deben registrar todas las instancias de los objetos remotos para que los clientes puedan alcanzalros. Se utiliza la clase **java.rmi.Naming** para ligar un nombre al objeto servidor. Esta clase utiliza el registry de RMI subyacente para almacenar nombres. Los objetos del servidor están listos ahora para recibir invocaciones de los clientes.

8. **Escribir el código del cliente:** el cliente consiste de código Java ordinario. Se debe utilizar la clase **java.rmi.Naming** para localizar un objeto remoto. Luego se invocan sus métodos vía un stub que sirve como un proxy para el objeto remoto.

9. **Compilar el código del cliente:** se utiliza nuevamente el compilador de Java *javac*.

10. **Iniciar el cliente:** se cargan las clases del cliente y sus stubs.

Como se puede ver, RMI sigue un proceso de desarrollo muy similar al de CORBA.

Intefaces y clases de RMI

Las interfaces y las clases que son responsables de especificar el comportamiento remoto de un sistema RMI se definen en los siguientes cuatro packages: **java.rmi**, **java.rmi.server**, **java.rmi.dgc**, y **java.rmi.registry**. Juntos, estos packages definen más de 25 interfaces y clases. Esto hace de RMI un subsistema de Java relativamente grande, pero solo es necesario entender la mitad de las mismas para escribir aplicaciones cliente/ servidor usando RMI.

Dividiremos las interfaces de RMI en cuatro categorías funcionales:

1. **RMI core:** define las interfaces absolutamente necesarias para realizar invocaciones remotas a métodos. Consiste de seis clases e interfaces que definen la infraestructura de stubs y skeletons para invocaciones remotas. También definen extensiones para el manejo de errores.

2. **RMI naming service:** define las interfaces y las clases que implementan el registry. Antes de que un cliente pueda invocar un método en un objeto remoto, debe obtener primero una referencia a ese objeto. Típicamente el cliente obtiene esta referencia como un valor de retorno en la llamada a un método. El servicio de nombres permite obtener las referencias a los objetos en el servidor a partir de un nombre. El registry es un objeto en el servidor al que se puede acceder a través de la red, usando RMI. Este servicio no provee persistencia, es decir, cuando el servidor se reinicia, pierde todos los contenidos del registry.

3. **RMI security:** define un manejador de seguridad e interfaces para cargar clases vía URL.

4. **RMI marshaling:** define interfaces de bajo nivel para serializar objetos remotos y stubs. Mediante la serialización se pueden convertir los objetos Java en streams, para ser enviados como argumentos o devueltos como resultados.

RMI sobre IIOP

Sun e IBM trabajan para la convergencia de los modelos de objetos de Corba y RMI. Esta convergencia sucede en las dos áreas siguientes:

- **RMI-over-IIOP:** JavaSoft ofrecerá una versión de RMI que trabaja sobre el transporte IIOP. IIOP provee los siguientes beneficios a RMI: 1) soporte para la propagación de transacciones, 2) interoperabilidad con objetos escritos en otros lenguajes, 3) un standard abierto de objetos distribuidos.

- **RMI/IDL:** el objetivo es permitir a los programadores Java especificar interfaces CORBA usando la semántica de RMI en vez la de CORBA IDL. El compilador utiliza estas semánticas para generar automáticamente los stubs y skeletons de CORBA. RMI/IDL permite a los programadores que usan RMI, que sus objetos sean invocados por clientes CORBA (desarrollados en cualquier lenguaje) usando IIOP. También permite a los programas que usan RMI, invocar objetos escritos en otros lenguajes. Este standard está muy relacionado con otro standard de CORBA llamado *Objects-by-Value*, que permitiría la posibilidad de pasar parámetros por valor usando CORBA, cosa que en la actualidad no es posible ya que los parámetros son solo pasados por referencia.

Conclusiones

Existen dos posibles comparativas: una es CORBA vs. RMI, y la otra es RMI-over-IIOP vs. RMI-over-RMP (*Remote Method Protocol*). En la tabla siguiente veremos la primera que es una comparativa general entre CORBA IIOP y RMI.

Característica	CORBA IIOP	RMI
Tecnología	De integración	De programación
Lenguaje	Independiente del lenguaje	Java-to-Java
Definición de interfaces	Usando IDL	Usando interfaces Java
Servicios	Amplia cantidad	Limitados
Interoperbilidad	Vía IIOP	No provee

Lo más importante que se deduce de esta tabla es que CORBA es una tecnología que apunta a la integración, y a partir de ahí se explica el por qué de las demás características y de las diferencias con RMI que es una tecnología de programación.

La siguiente tabla es más detallada, ya que en ella podremos ver lo que CORBA le aporta a RMI.

Característica	RMI-over-IIOP	RMI-over-RMP
Marshaling de parámetros	SI	SI
Pasaje de parámetros	in, out, in/out	in
Download dinámico de stubs	SI (vía codebase)	SI
Pasaje de objetos por valor	SI (vía Pass-by-Value)	SI
Recolección de basura	SI (vía POA y ORB)	SI (vía el lenguaje)
Descripción de interfaces	SI (vía interfaces Java)	SI (vía interfaces Java)
Descubrimiento dinámico	SI (vía IR)	NO
Invocaciones dinámicas	SI (vía DII)	NO
Seguridad basada en SSL	SI	SI (con JDK 1.2)
Persistencia de nombres	SI	NO
Invocaciones independientes del lenguaje	SI (vía RMI/IDL)	NO
Escalabilidad	SI (vía IIOP)	NO

El diseño de RMI ha tenido una gran influencia sobre CORBA. RMI es realmente una nueva clase de ORB construido sobre el modelo de objetos de Java, con la ventaja de que el ORB es transparente. RMI provee otras innovaciones: permite mover código y no solo datos, permite pasar objetos por valor y utiliza Java tanto para la definición de las interfaces como para la implementación.

Pero hay algo con lo que RMI no puede competir con CORBA, y esto es la interoperabilidad y la escalabilidad que CORBA provee a través de IIOP.

La convergencia de las dos tecnologías es un avance importante, ya que integra ventajas de ambas en una única solución.

Capítulo IV: DCOM vs. CORBA/Java ORBs

El *Distributed Component Object Model* (DCOM) constituye el mayor competidor de CORBA, ya que es el "standard" de objetos distribuidos de Microsoft.

DCOM ha evolucionado a partir de diversas tecnologías de Microsoft, comenzando en 1990 con *Object Linking and Embedding* (OLE). OLE nació como un simple mecanismo *cut-and-paste* usando una tecnología llamada *Dynamic Data Exchange* (DDE). Luego fue extendido a OLE2 con la introducción del *Microsoft Component Object Model* (COM), para proveer comunicación entre aplicaciones (siempre de Microsoft). OLE2 fue también extendido para soportar "*drag and drop*" y capacidad de crear *scripts* para permitir que una aplicación realice tareas simples en otra (*OLE Automation*).

En la misma época, Microsoft lanza *Visual Basic* y gana credibilidad como un modelo de propósito general para ensamblar componentes (*Visual Basic custom controls* - VBX). De todos modos, la arquitectura VBX no era abierta y esto motivó a Microsoft a proveer una infraestructura más generalizada para la comunicación y control entre aplicaciones.

Este framework fue definido como la combinación entre COM y controles OLE (OCX). COM fue el standard a nivel de sistema y OLE fue el servicio de aplicación construido sobre COM.

Distributed COM emerge para soportar componentes remotos, y ActiveX se convierte en el sucesor de OCX.

Esta es una breve reseña de como Microsoft llega a lanzar DCOM. A lo largo de este capítulo veremos el funcionamiento de DCOM y al final se realizará una comparación con CORBA.

DCOM: Conceptos generales

Como CORBA, DCOM separa la interfaz del objeto de su implementación, y requiere que todas las interfaces sean declaradas usando un *Interface Definition Language* (IDL) basado en DCE y que no es compatible con CORBA.

Un objeto DCOM no es un objeto en el sentido *Object Oriented*, ya que las interfaces DCOM no tienen estado y no pueden ser instanciadas para crear un objeto único (un objeto con una única *object reference*). Una interfaz DCOM es simplemente un conjunto de funciones relacionadas. Los clientes deben obtener un puntero para acceder a las funciones de una interfaz, y este puntero no está relacionado con información de estado. Un cliente DCOM no puede conectarse exactamente a la misma instancia de un objeto con el mismo estado en un tiempo futuro. Sólo puede reconectarse a un puntero a la interfaz de la misma clase. Dicho de otro modo, un objeto DCOM no mantiene estado entre conexiones, lo que puede ser un problema en ambientes donde las conexiones fallan (como Internet).

DCOM provee interfaces estáticas y dinámicas para la invocación de métodos. La *Type Library* (de aquí en más "librería de tipos") es la versión de DCOM del *Interface Repository* (IR) de CORBA. El precompilador de DCOM pueden divulgar la librería de tipos con descripciones de objetos definidos en IDL, incluyendo sus interfaces y parámetros. Los clientes pueden consultar esta librería para descubrir qué interfaces son soportadas por un objeto y qué parámetros se necesitan para invocar un método particular.

Veremos más detalladamente estos y otros conceptos a lo largo del capítulo.

Interfaces DCOM

Una interfaz DCOM es un conjunto de funciones o métodos que se definen independientemente de la implementación. Los clientes interactúan entre ellos o con el sistema llamando a estas funciones. Las interfaces son independientes del lenguaje de programación y pueden realizarse llamados a funciones a través de distintos espacios de direcciones y redes. DCOM define simplemente una especificación de interoperabilidad binaria, que establece de que manera se accede a las interfaces a través de punteros y proxies remotos.

Una interfaz es definida como una API binaria de bajo nivel basada en una tabla de punteros. Para acceder a una interfaz, un cliente DCOM utiliza punteros a un arreglo de punteros a funciones, conocido como *virtual table* (o *vtable*). Las funciones referenciadas por estos punteros de la vtable son las implementaciones de los métodos de los objetos del servidor. Cada objeto tiene una o más vtables. Este concepto se ve más claramente en la figura 3.4.1.

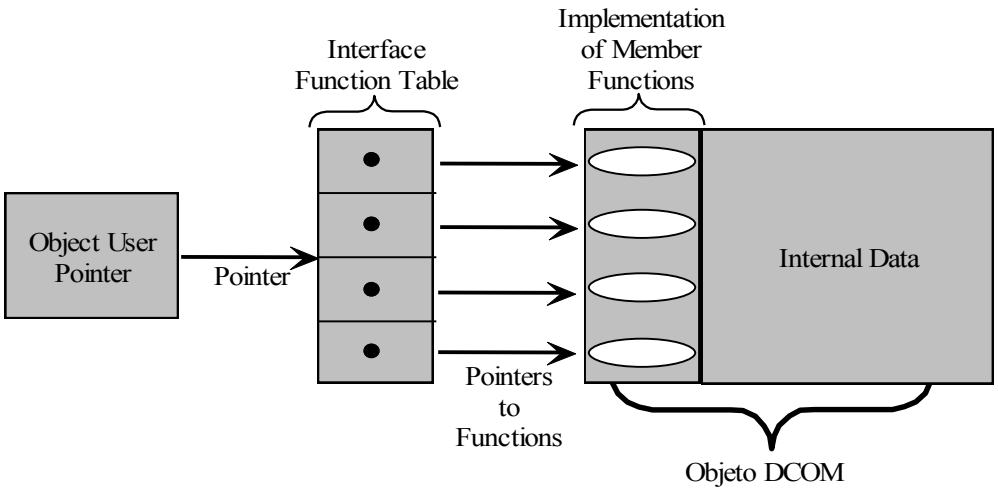

Figura 5.4.1 Interfaz DCOM: Representación del puntero a la vtable.

Por convención, el nombre de una interfaz comienza con "I" y tiene sólo un sentido simbólico, ya que en tiempo de ejecución cada interfaz es conocida por su único *Interface Identifier* (IID). Un IID es un *globaly-unique identifier* (GUID) generado por DCOM para las interfaces. Un GUID es un identificador de 128 bits que se crea llamando a la función *CoCreateGuid* de la API de DCOM. Esta función ejecuta un algoritmo que calcula un número globalmente único basado en la hora, la fecha, la dirección de la placa de red y un contador. Prácticamente no existe posibilidad de que este número se repita. El IID permite al cliente preguntar a un objeto, sin ambigüedades, si soporta una interfaz. Para que un cliente pueda dilucidar si una interfaz es soportada por un objeto, debe llamar a la función *QueryInterface* de la interfaz **IUnknown** que veremos más adelante.

Objetos DCOM

Un objeto DCOM, también conocido como *ActiveX*, es un componente que soporta una o más interfaces, según se define en la clase del objeto. Una interfaz DCOM refiere a un grupo predefinido de funciones relacionadas. Una clase DCOM implementa una o más interfaces y se identifica por un único identificador de 128 bits llamado *Class ID* (CLSID). Un objeto DCOM es, en tiempo de ejecución, la instanciación de una clase. Un objeto particular provee las implementaciones de las funciones que se especifican en las interfaces que soporta (ya sea una o más).

Los clientes tratan con los objetos a través de punteros a sus interfaces, nunca acceden directamente a ellos. Debe notarse que los objetos DCOM no soportan el concepto de object ID.

Los clientes obtienen un puntero a la interfaz y no al objeto, por lo tanto, lo único que conocen del objeto es si soporta o no una determinada interfaz. En consecuencia, las instancias de los objetos juegan un papel mucho menos significativo en DCOM que en los sistemas tradicionales orientados a objetos.

Todos los objetos DCOM deben implementar la interfaz **IUnknown**, a través de la cual los clientes pueden controlar el tiempo de vida de un objeto y a través de la cual pueden averiguar si el objeto soporta una determinada interfaz.

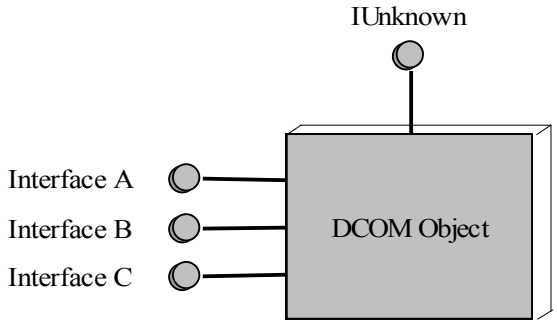

Figura 3.4.2 Objeto DCOM y las interfaces que soporta

Servidor DCOM

Un servidor DCOM es una pieza de código (DLL, EXE, o una clase Java) que contiene una o más clases de objetos con sus respectivos CLSIDs. Cuando un cliente pide un objeto de una determinada clase, DCOM carga el código del servidor y le pide que cree un objeto de esa clase. El servidor debe proveer una *class factory* para crear un nuevo objeto. Una vez que el objeto es creado, se le devuelve al cliente el puntero a su interfaz primaria.

De este modo, el servidor DCOM provee la estructura necesaria alrededor de los objetos para hacerlos disponibles a los clientes. Un servidor DCOM debe:

- **Implementar una interfaz *class factory*:** el servidor debe implementar un objeto que soporte la interfaz **IClassFactory** para crear instancias de objetos.
- **Registrar las clases que soporta:** cada clase soportada por el servidor debe ser registrada en la *NT Registry*. Por cada CLSID, debe crear una o más entradas que provean el *pathname* al servidor.
- **Inicializar la librería DCOM:** el servidor llama la función *CoInitialize* (de la API DCOM) para iniciar DCOM. La librería provee tanto servicios en tiempo de ejecución como APIs.

- **Verificar que la librería sea de una versión compatible:** realiza esto llamando a funciones de la API *CoBuildVersion*.
- **Proveer un mecanismo de descarga:** el servidor debe terminar su ejecución cuando no existen más clientes activos para sus objetos.
- **"Uninitialize" la librería DCOM:** llama a la API CoUninitialize cuando no se usa más.

La implementación del servidor (incluyendo la registración, la *class factory* y el mecanismo de descarga) varía dependiendo si éste es empaquetado en un archivo DLL, EXE o en una clase Java.

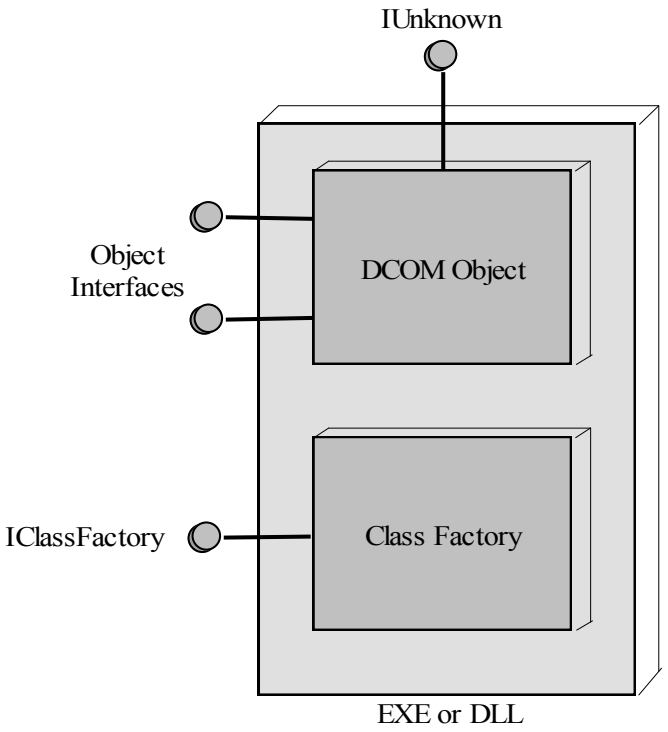

Figura 3.4.3 Estructura de un servidor DCOM

Transparencia local y remota en DCOM

Los servidores DCOM se pueden implementar de diferentes maneras, dependiendo de la estructura del módulo de código y su relación con los procesos del cliente que lo utilizarán. Los tipos de servidores son:

- ***In-process servers*:** se ejecutan en el espacio del proceso del cliente. En Microsoft Windows y Windows NT, son implementados como *Dynamic Link Libraries* (DLLs), que son cargadas directamente en los procesos del cliente.
- ***Local servers:*** se ejecutan en procesos separados de los del cliente pero en la misma máquina (y mismo sistema operativo). Los clientes usan un mecanismo de DCOM llamado *Lightweight RPC* (LRPC) para comunicarse con el servidor local.
- ***Remote servers:*** el cliente y el servidor se ejecutan en procesos separados y en diferentes máquinas y posiblemente con diferentes sistemas operativos. Los clientes utilizan un mecanismo como el de RPC para comunicarse con el servidor.

Teóricamente, DCOM permite a los clientes comunicarse en forma transparente con los objetos del servidor, sin importar en dónde están corriendo los mismos. Desde el punto de vista del cliente, todos los objetos del servidor se acceden a través de punteros a interfaces. El puntero debe estar en el proceso del cliente, ya que cada llamada a una función de una interfaz siempre utiliza algún pedazo de código en el proceso cliente. Si el objeto está en el proceso cliente, entonces es alcanzado directamente, sin intervención del código de infraestructura de DCOM. Si el objeto no está en el proceso del cliente, la llamada primero alcanza al objeto *proxy* provisto por DCOM. Este objeto genera la llamada a procedimiento remoto para el otro proceso, en la mismo o en otra máquina. El elemento de DCOM que localiza a los servidores y que se involucra con las invocaciones RPC, se llama *Service Control Manager* (SCM).

Desde el punto de vista del servidor, todas las llamadas a funciones de una interfaz de objetos, son realizadas a través de un puntero a esa interfaz. Si el cliente y el servidor se encuentran en el mismo proceso, el objeto del servidor recibe la llamada desde el mismo cliente. De lo contrario, el objeto servidor recibe la llamada a través de un *stub object* provisto por DCOM que toma la llamada a procedimiento remoto que efectúa el *proxy* del cliente y la convierte en una llamada a la interfaz del objeto servidor.

En la figura 3.4.4 vemos gráficamente estos mecanismos.

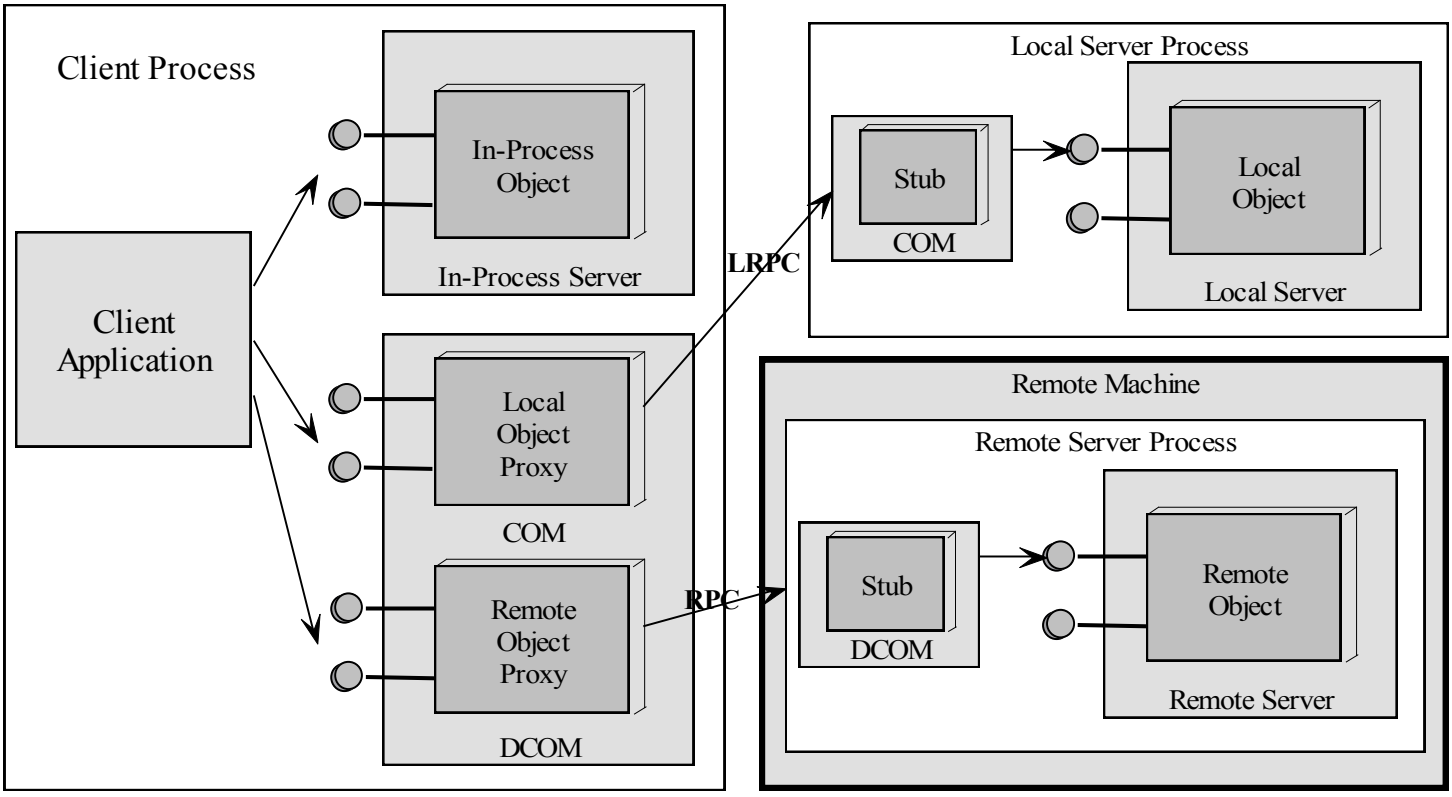

Figura 3.4.4 Límites entre el cliente y el servidor en DCOM

Interfaz IUnknown

La interfaz **IUnknown** es el corazón de DCOM. Se utiliza para las negociaciones de interfaces en tiempo de ejecución, manejo del ciclo de vida y agregación. Cada interfaz DCOM debe implementar tres funciones de la interfaz **IUnknown**: *QueryInterface*, *AddRef*, y *Release*.

IUnknown soporta negociaciones de interfaces a través de *QueryInterface*; y el ciclo de vida de una instancia de interfaz, a través de *AddRef*, y *Release*. Como todas las interfaces DCOM derivan de **IUnknown**, las funciones mencionadas pueden ser invocadas a través de cualquier puntero a una interfaz. Dicho de otro modo, **IUnknown** es la interfaz base de la cual todas las otras interfaces heredan estas funciones y las implementan de forma polimórfica.

Negociación de interfaces usando *QueryInterface*

Cuando inicialmente un cliente gana el acceso a un objeto DCOM, obtiene un puntero a una interfaz. El resto de los punteros a las otras interfaces que vaya a utilizar los recibe llamando a *QueryInterface*. Esta función permite al cliente descubrir, en tiempo de ejecución, si una interfaz (mediante su IID) es soportada por un objeto. Si la interfaz es soportada por el objeto, la función devuelve al cliente el puntero a la interfaz. Luego de que el cliente recibe el puntero a la misma, DCOM deja de participar y permite que el cliente interactúe directamente con el servidor invocando a las funciones de su interfaz.

De este modo, la unidad de negociación entre cliente y servidor es una interfaz y no una función individual. La función *QueryInterface* retorna un código de error si la interfaz no es soportada o si el servidor se niega a servir al cliente.

QueryInterface tiene la capacidad de retornar punteros a otras interfaces que el objeto DCOM soporta, lo que es la clave para el trabajo de agregación. DCOM requiere que una llamada a *QueryInterface,* para una interfaz específica, retorne siempre el mismo valor de puntero, de modo tal que el cliente pueda realizar tests de identidad, para determinar si dos punteros apuntan a la misma interfaz.

Manejo del ciclo de vida con contadores de referencias

En sistemas de objetos no distribuidos, el ciclo de vida de los mismos es manejado implícitamente por el lenguaje de programación o explícitamente por el programador; existe siempre un modo de saber cuándo un objeto debe ser creado o eliminado. En los sistemas de objetos distribuidos esto no aplica para la eliminación de los objetos, ya que deja de ser cierto que alguien o algo sepa cuando un objeto deja de ser necesario. De este modo surge la incógnita de quién es el responsable de la recolección de basura en un sistema de objetos distribuidos.

Existen tres aproximaciones para resolver la recolección de basura en sistemas de objetos distribuidos. La primera es ignorar el tema y dejar que el sistema operativo se encargue de eliminar los objetos cuando realiza un *shutdown*, cosa que no es aceptable para un servidor. La segunda es dejar que el ORB mantenga información de conexiones activas. La tercera es darle a los objetos la responsabilidad de controlar su utilización mediante contadores de referencias. En este caso, los clientes deben cooperar diciéndole a los objetos del servidor cuando los están utilizando y cuando dejan de hacerlo. Los objetos deben autoeliminarse cuando su contador de referencias indica que no hay clientes que estén requiriendo sus servicios.

La versión Java de DCOM se basa en la primera aproximación, dejando la tarea al recolector de basura de Java; y la versión de C++ de DCOM usa la tercera aproximación. Los ORBs de CORBA se basan en la segunda solución.

La versión de C++ de DCOM usa las funciones *AddRef* y *Release*, de la interfaz **IUnknown**, para que los clientes informen a los objetos servidores si deben incrementar (llamando a *AddRef*) o decrementar (llamando *Release*) su contador de referencias.

Estilo de herencia DCOM: Agregación y Contención

Contrariamente a lo que sucede con CORBA y Java, DCOM no soporta herencia múltiple de interfaces. De todos modos, los componentes DCOM pueden soportar múltiples interfaces. Mediante las llamadas a la función QueryInterface, los clientes DCOM pueden determinar que tipos de interfaces son soportadas por un componente. Estos dos mecanismos permiten que los desarrolladores puedan crear componentes externos que encapsulen los servicios de componentes internos y representarlos de ese modo a los clientes. DCOM provee dos métodos para encapsular servicios: *contención/delegación* y *agregación*. En ambos casos el ciclo de vida de los objetos internos es manejado por el objeto externo, y su interfaz **IUnknown** representa a las interfaces de los objetos internos.

En *contención/delegación*, el objeto externo debe recrear la invocación de los métodos que recibe en nombre de sus objetos internos. En la figura 3.4.5(a), el objeto externo A contiene a los objetos internos B y C. La interfaz **IUnknown** de A conoce a las interfaces A, B y C. Cuando un cliente quiere hablar con B o C, el objeto externo llama a los métodos provistos por los objetos internos. Esto se conoce como *delegación*.

Cuando se usa *agregación*, en vez de recrear cada llamada, la interfaz **IUnknown** del componente externo expone a los clientes los punteros a las interfaces de los componentes internos. Esto significa que el objeto externo expone a las interfaces de los objetos internos como si éstas le pertenecieran. De este modo los objetos internos se comunican directamente con los clientes, pero delegan su interfaz **IUnknown** al objeto externo. En la figura 3.4.5(b), los objetos internos B y C forman parte de la interfaz **IUnknown** de A, pero igualmente los clientes hablan directamente con B y C sin la intervención de A.

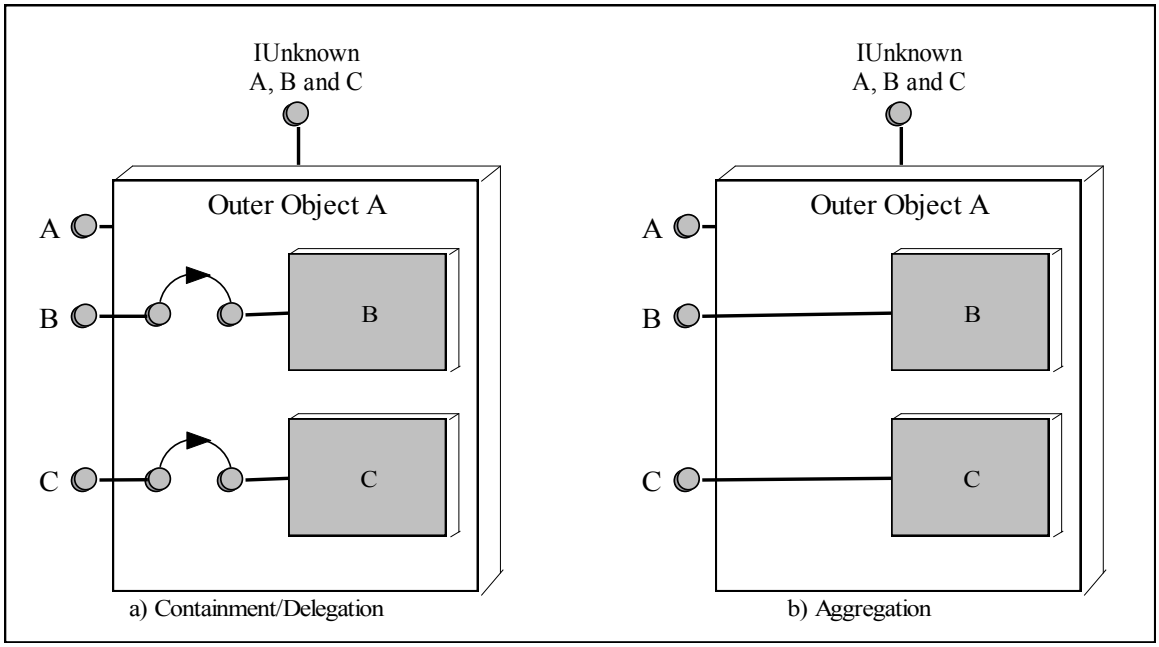

Figura 3.4.5 Delegación vs Agregación

Todo esto significa que el ambiente DCOM es inherentemente chato. La “herencia” se consigue a través de cadenas de punteros que ligan o agregan diferentes interfaces.

IDL de DCOM

Para especificar una interfaz de DCOM, se debe crear primero un archivo que describe los métodos de la misma y sus argumentos utilizando IDL (en la versión de DCOM). Luego se debe compilar este archivo mediante el compilador Microsoft IDL (MILD) para crear los *proxies* de los clientes, los *stubs* de los servidores y el código que ordena los argumentos para su transferencia entre ambos.

El MILD es en realidad un precompilador que transforma descripciones textuales de las interfaces en código para los *proxies* y los *stubs*. Para concluir el trabajo, se debe implementar, compilar, ligar y registrar el código del servidor.

Facilidades de DCOM para la invocación dinámica

Como CORBA, DCOM provee facilidades para la invocación dinámica. Como se ha mencionado anteriormente, la librería de tipos de DCOM, como el *Interface Repository* de CORBA, permite que los clientes puedan descubrir en tiempo de ejecución, los métodos y propiedades de un servidor DCOM. El compilador MILD genera la librería de tipos a partir de una especificación IDL si la palabra clave *library* se encuentra encabezando algún bloque de la misma. La librería de tipos puede ser navegada por los clientes a través de un puntero a la interfaz **ITypeLib**.

Los servidores que soportan la invocación dinámica implementan una interfaz específica llamada **IDispatch**. Esta interfaz combina todos los métodos de una interfaz regular en un único método llamado *Invoke*, que utiliza un tipo de registro variante para combinar todos los posibles parámetros en uno. Cada campo de este registro es un par compuesto por un tipo y un valor. El método al cual se desea invocar es especificado por un *dispatch ID* (dispID), que es simplemente un número de método dentro de una "*dispinterface*" (interfaz que implementa los métodos de **IDispatch**).

IDispatch agrega cuatro métodos que deben ser implementados por un servidor además de los de **IUnknown**:

- ***Invoke:*** mapea un dispID en una llamada a un método o un acceso a una propiedad.
- ***GetIDsOfNames:*** convierte nombres de propiedades o métodos (incluyendo parámetros) en su correspondiente conjunto de dispIDs.
- ***GetTypeInfoCount:*** sirve para que un cliente pueda saber si existe o no información de tipos para la *dispinterface*.
- **GetTypeInfo:** devuelve la información de tipo de la *dispinterface*. La información de tipo es la descripción de los métodos y parámetros que soporta la interfaz.

Además de crear entradas en la *Registry* para un servidor DCOM, un programa de instalación debe crear las correspondientes entradas para las librerías de tipos llamando a la función *RegisterTypeLib* de la API de DCOM.

Veremos gráficamente como se unen estas piezas en la figura 3.4.6.

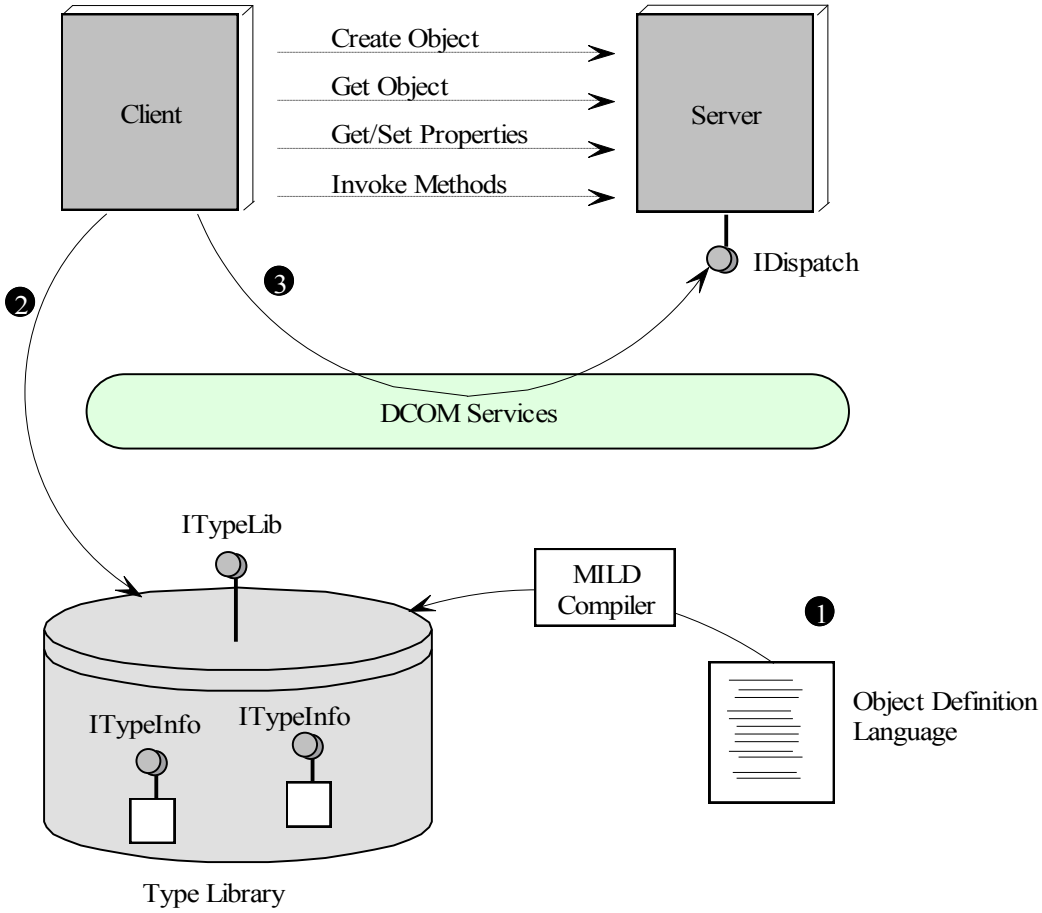

Figura 3.4.6 Facilidad de invocación dinámica de DCOM

Proceso de desarrollo con DCOM y Java

La figura 3.4.7 muestra los pasos que deben seguirse para implementar aplicaciones cliente/servidor usando Java y DCOM, y que son los siguientes:

1. **Crear el archivo IDL para el objeto :** se deben definir todas las interfaces para la clase.

2. **Generar los *GUIDs* para las interfaces IDL:** deben generarse los GUIDs y UUIDs para cada clase e interfaz que se define en IDL. Esto puede lograrse usando una herramienta que provee Microsoft con el Developer Studio (VJ++ o VC ++).

3. **Crear el archivo de la librería de tipos:** es necesario ejecutar el compilador MILD para generar los archivos .TBL que corresponden a la librería de tipos.

4. **Generar las clases Java basadas en la librería de tipos:** para esto se utiliza una herramienta llamada *JavaTBL* que genera clases Java para cada interfaz DCOM. También genera un archivo Summary.txt que es usado luego como template para implementar las clases DCOM.

5. **Implementar las clases DCOM en Java:** deben crearse las clases de Java que implementan las interfaces que se definieron en IDL.

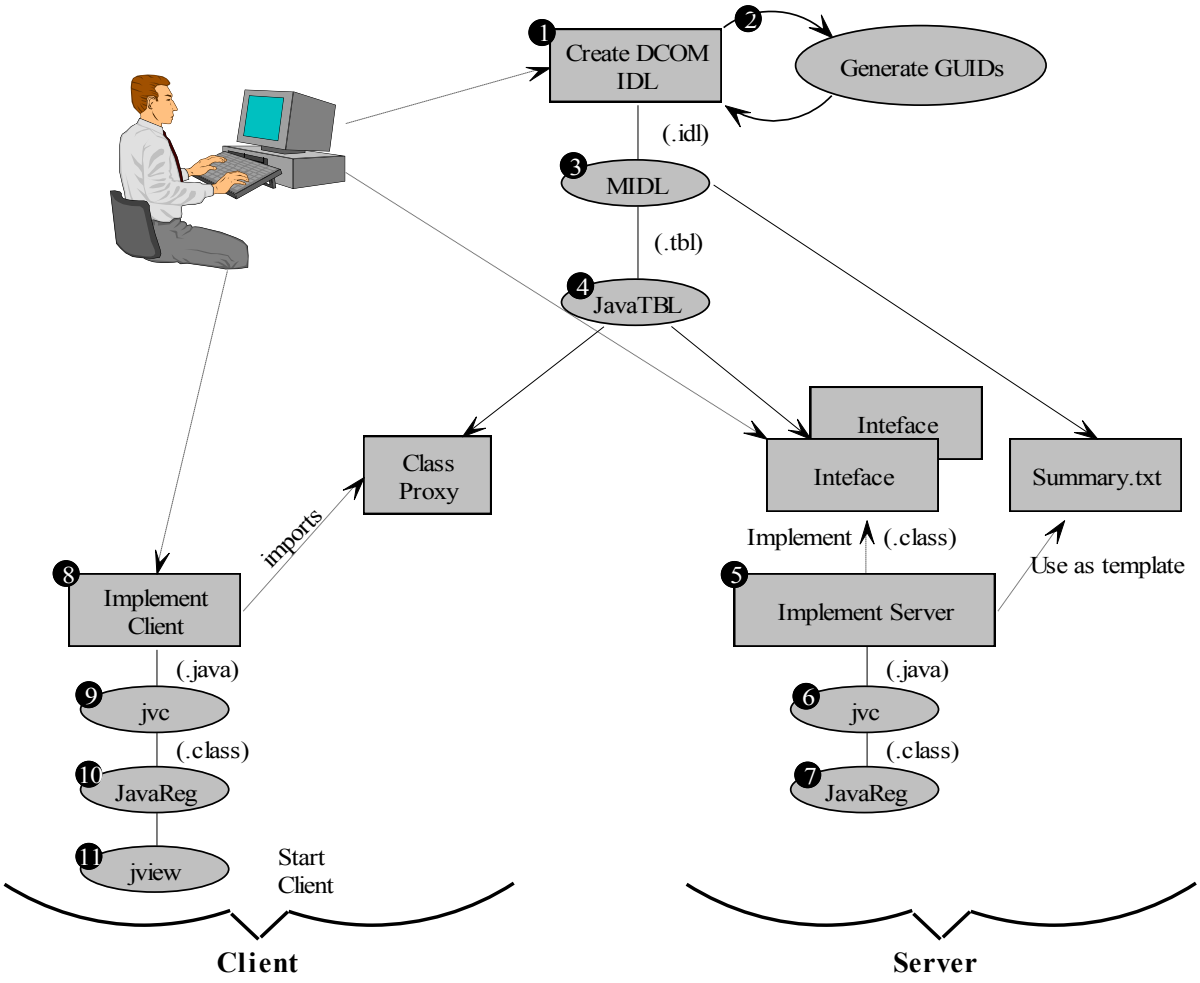

Figura 3.4.7 Como escribir aplicaciones cliente/servidor usando DCOM

6. **Compilar la implementación:** deben compilarse las clases Java (archivos .java) usando el compilador *jvc* que viene con VJ++.

7. **Registrar la clase Java:** debe usarse la utilidad *JavaReg* desde la línea de comandos para registrar la clase Java como una clase DCOM.

8. **Escribir el código del cliente:** el cliente consiste en su mayor parte de código Java común. La única restricción es que no puede utilizarse una instancia de una clase DCOM directamente, se la accede siempre a través de su interfaz.
9. **Compilar el código del cliente:** usando el compilador *jvc*.
10. **Registrar al cliente:** usando *JavaReg*.
11. **Ejecutar el cliente:** éste se arranca usando la JVM de Microsoft llamada *jview*.

Comparación paso a paso

Para comparar DCOM vs. CORBA, se utilizará un criterio diferente, en el cual se toman como referencia los elementos de la siguiente tabla:

Interoperabilidad
Cross-Language Support
Cross-Platform Support
Comunicaciones de red
Servicios comunes
Confiabilidad
Transacciones
Seguridad
Tolerancia a fallas
Viabilidad
Madurez del producto

Interoperabilidad

Cross-Language Support

CORBA

CORBA fue diseñado desde el comienzo para ser independiente de la plataforma y del lenguaje de programación, a través del uso del *Interface Definition Language* (IDL). IDL se usa para definir la "interfaz" de un componente, y no su funcionamiento interno. Las interfaces IDL se traducen a un lenguaje a través de "mapeos". Actualmente existen mapeos para C, C++, Smalltalk, Ada, OLE (Visual Basic, PowerBuilder, Delphi, etc.), Java y pronto para Eiffel y Objective C.

Los beneficios de la interoperabilidad tienen un costo: IDL limita los tipos de datos de los lenguajes a un subconjunto común que es soportado por todos ellos. IDL provee un tipo "any" para superar este escollo.

DCOM

La portabilidad de DCOM se basa en un "standard binario". La compatibilidad binaria se logra en los niveles cero y uno, área que anteriormente fue del dominio de los compiladores e intérpretes. Para garantizar la compatibilidad a estos niveles, el modo en que cada lenguaje se traduce a código binario debe ser controlado. Esto puede presentar algunos obstáculos, pero también tiene sus beneficios. Primero, existen diferencias fundamentales en cómo se traducen los diferentes lenguajes: algunos son compilados y otros interpretados, por lo que se deben proveer soluciones para ambos casos. Segundo, existen varios compiladores/interpretes para un lenguaje dado, donde cada uno aborda distintos aspectos de la traducción. Por último, especificar compatibilidad a este nivel tan bajo, crea vulnerabilidad debido al avance del hardware.

COM es actualmente soportado por los más populares productos de Microsoft, como Java, PowerBuilder, Delphi y Microsoft Focus COBOL. De todos modos, para DCOM se requiere soporte adicional de Microsoft o de terceras partes.

Conclusiones

Tanto CORBA como DCOM proveen soporte para diferentes lenguajes, pero utilizando diferentes técnicas.

Cross-Platform Support

CORBA

Este tema ha sido siempre un foco central de CORBA. Existen ORBs para más de 30 plataformas y soporta hasta más sistemas operativos de Microsoft que DCOM mismo. Orbix, uno de los ORBs líderes soporta más de 20 plataformas.

DCOM

DCOM abordó el tema con retraso. En 1993, Microsoft se acercó a una compañía alemana, llamada Software AG, para portar a COM a otras plataformas. Software AG portó DCOM a diferentes variantes de Unix, pero perdiendo en el camino varias de las componentes de DCOM.

Conclusiones

Queda claro que CORBA es muy superior a DCOM cuando se habla de soporte para diferentes plataformas desde el momento que CORBA nace con este tema como un objetivo central y Microsoft lo encara tardíamente. Aunque DCOM sea portado a otras plataformas, estas versiones estarán siempre un paso atrás de la versión para NT.

Comunicaciones de Red

CORBA

El modelo de red predominante de CORBA para la comunicación inter-ORB se basa en el protocolo IIOP sobre TCP. IIOP fue diseñado para asegurar que todos los ORBs pudieran utilizar un protocolo de comunicaciones común. De todos modos, un ORB particular puede usar internamente otros protocolos.

DCOM

Inicialmente DCOM utilizó UDP/DCOM, un protocolo sin conexión basado en la especificación DCE RPC de OSF con algunos cambios. DCOM ofrece en la actualidad una configuración usando TCP pero sacrificando características de eficiencia.

Conclusiones

CORBA tomó la delantera especificando un protocolo de comunicación común. DCOM provee opciones pero no las soporta de la misma manera.

Servicios Comunes

CORBA

El OMG se concentró en el desarrollo e integración de servicios claves en la arquitectura. La especificación de CORBA define 15 servicios, aunque no todos los ORBs comerciales soportan el conjunto completo.

DCOM

Los servicios de DCOM no son definidos desde la misma arquitectura, aunque existen algunos equivalentes a los provistos por CORBA. DCOM ofrece actualmente un servicio de nombres, transacciones y seguridad limitados e integrados a NT.

Conclusiones

No se encuentra aún disponible un conjunto de servicios completo ni en CORBA ni en DCOM. Pero en la actualidad CORBA es superior en cantidad, madurez y alcance de los servicios ofrecidos.

Confiabilidad

Transacciones

CORBA

La especificación del *Object Transaction Service* (OTS) de CORBA ofrece un rango de servicios para el soporte de transacciones distribuidas. OTS permite que tanto aplicaciones que usan ORBs como aquellas que no, participen en la misma transacción, de modo tal que las transacciones procedurales y las basadas en objetos puedan interactuar. También soporta interacciones entre distintos ORBs. Una interfaz IDL puede soportar implementaciones transaccionales y no-transaccionales. Para crear un objeto transaccional, los desarrolladores deben usar una interfaz que hereda de la clase abstracta OTS. Si se toman juntos el OTS y el servicio de concurrencia y control *(Concurrency and Control service*), se obtienen amplias capacidades para el soporte de transacciones. Varios implementadores de ORBs han ofrecido links a herramientas de *TP monitors* tradicionales.

Tuxedo, el TP monitor más escalable para ambientes ampliamente distribuidos, se ha integrado exitosamente con dos ORBs. Además, Hitachi y Visigenic han integrado TPBroker; y IONA ha integrado Transarc en OrbixOTS.

DCOM

También Microsoft ha atacado agresivamente el soporte para transacciones a través de su Microsoft Transaction Server (MTS). Con MTS, las transacciones se soportan implícitamente, liberando al implementador de la complejidad de tratar con el servicio de transacciones directamente.

Conclusiones

En este aspecto tanto CORBA como DCOM ofrecen soluciones válidas.

Seguridad

CORBA

El servicio de seguridad de CORBA es una de las especificaciones más exhaustivas disponibles para la computación distribuida. Cubre prácticamente todos los aspectos de seguridad, incluyendo integridad, manejo de cuentas, disponibilidad y confidencialidad.

Los ORBs disponibles difieren en cuanto a su soporte de seguridad. Por ejemplo, el ORB de IDL llamado DAIS fue el primero en soportar seguridad basada en standards de Kerberos y GSS; Orbix de IONA provee tanto SSL-IIOP como una implementación del servicio de seguridad de CORBA del nivel 1; y por último, Visigenic provee una implementación del servicio de seguridad de CORBA en nivel 2.

DCOM

DCOM utiliza los mecanismos de seguridad basados en NT, cuya versión 3.5 fue evaluada como de nivel 2 por el National Computer Security Center. De todos modos esto hace que la seguridad de DCOM se vea limitada sólo a un ambiente NT.

Conclusiones

Tanto CORBA como DCOM están construyendo completos mecanismos de seguridad. CORBA aventaja a DCOM desde el punto de vista de la variedad de soluciones que puede ofrecer, ya que DCOM se mantiene atado sólo a lo que NT ofrezca en este tema.

Tolerancia a Fallas

CORBA

La especificación de CORBA no soporta directamente servicios de tolerancia a fallas, aunque varios fabricantes de ORBs proveen soporte para esto. La mayoría de los ORBs proveen un mecanismo simple de “time out” para detectar la caída o la desconexión de clientes.

DCOM

El soporte básico de tolerancia a fallas se provee a nivel de protocolo. Utilizando el mecanismo ya descripto de contador de referencias, se envían mensajes cada dos minutos a los objetos clientes esperando sus respuestas, si esta no llega luego de tres intentos, el contador de referencias se decrementa. Esta solución consume recursos de red y corre el peligro de no poder escalar bien a medida que crece el número de conexiones.

Conclusiones

Ninguno de los dos soporta directamente tolerancia a fallas.

Viabilidad

Madurez del producto

CORBA

Muchos ORBs ya se encuentran en su tercera generación de desarrollo. Aunque la especificación del OMG se encarga de diseñar los servicios para asegurar su integración, ningún fabricante los provee en un modo estrictamente compatible con la especificación de CORBA. Igualmente, los ORBs están siendo actualmente utilizados para el desarrollo de los sistemas en industrias exigentes, incluyendo telecomunicaciones, aéreas y financieras.

DCOM

DCOM abordó con retraso el tema de la interoperabilidad, por lo que queda pasos atrás de CORBA en este sentido. DCOM evolucionó a partir de DDE, OLE y ActiveX, por lo que no fue diseñado desde un inicio con la idea de computación distribuida, sino que fue un agregado a tecnologías ya existentes, con las complicaciones que eso acarrea.

Conclusiones

Según lo visto en la sección anterior, CORBA supera claramente a DCOM en cuanto a sus posibilidades y características tecnológicas, pero este último no deja de ser un fuerte competidor (evidentemente por el hecho de ser creado e impulsado por Microsoft). DCOM es completamente dependiente tanto del sistema operativo de Microsoft como de su JVM. Esto

contrasta considerablemente con CORBA, en cuanto a que éste es un standard abierto con lo beneficios que eso implica.

Apéndice: Una Aplicación

En este apéndice veremos la descripción de una aplicación creada para complementar este trabajo.

La aplicación desarrollada tiene como objetivo mostrar la manera de realizar una implementación utilizando CORBA y Java. La misma se divide en tres partes: cliente, servidor e interfaz de monitoreo. El cliente debe conectarse al servidor para poder intercambiar mensajes con otros usuarios, enviar e-mails, entrar en "rooms" públicos, o establecer conexiones privadas con otro usuario para el "chateo" con la posibilidad de compartir documentos. En las siguientes secciones se verán las definiciones IDL y luego cómo el cliente se conecta al ORB, cómo se activa el servidor, cómo y por qué se realizan invocaciones dinámicas y la utilización de objetos "callback".

Definiciones IDL

La primera definición especifica las interfaces necesarias para la comunicación del cliente con el servidor y las interfaces de los objetos "callback" para que los clientes reciban invocaciones del servidor u otros clientes.

```
interface CallBack;
interface CallBackCon;
interface CallBackRoom;
interface UserInRoomActions;

interface UserActions{// Define las operaciones que el cliente puede invocar sobre el
                              //servidor, por lo que esta interfaz es implementada por este último
        oneway void RegisterClient(in CallBack obj,in string Name);
        oneway void UnRegisterClient(in CallBack obj,in string Name);
        oneway void getRoomsList(in CallBack obj);
        oneway void getNumOfRoomUsers (in CallBack obj, in string room);
        oneway void EnterRoom(in CallBackRoom cbr, in string room);
        oneway void WeAreTalking(in CallBackCon u1, in CallBackCon u2);
                //Operación que informa al servidor que dos usuarios han comenzado una
                //conexión privada
        oneway void QuitTalking(in CallBackCon u1, in CallBackCon u2);
                //Operación que informa al servidor que dos usuarios han concluido una
                //conexión privada
};
interface UserInRoomActions{
```

```
        // Define la interfaz de un room. El servidor implementa esta interfaz y se crea un
        //objeto en base a ella por cada room existente

        readonly attribute string RoomName;
        attribute long NumUsers;

        oneway void EnterRoom(in CallBackRoom cbr);
        oneway void SendStringRoom(in string name, in string message);
        oneway void leaveRoom(in CallBackRoom cbr, in string name);
};

interface CallBack{
        // Interfaz que implementa cada cliente. Se utiliza cuando el usuario no ha
        //establecido ninguna conexión privada ni ha entrado en ningún room

        attribute string Name;
        attribute string Email;

        oneway void NewUser(in CallBack cb, in string name);
        oneway void RemoveUser(in string Name);
        boolean Call(in string name, in CallBack objCall);
        oneway void SendSingleMessage(in string message, in string sender);
        CallBackCon SendMeNewCBC(in CallBackCon cbc);
        oneway void newRoom(in string room);
        oneway void setNumOfRoomUsers(in long num);
        oneway void setCanReceiveCalls(in boolean b);
};

interface CallBackCon{
        // Interfaz que implementa el cliente. Un objeto callback que se basa en esta
        //interfaz, se crea cada vez que el cliente realiza una conexión privada
```

```
        attribute string Name;

        oneway void ReceiveOtherCallBackCon(in CallBackCon otherCBC);
        oneway void StringChat(in string s);
        oneway void QuitConnection();
        oneway void OpenDocument(in string Name);
        oneway void DocumentText(in string text);
        oneway void DocumentSent();
};

interface CallBackRoom{
        // Interfaz que implementa cada cliente cuando entra en un room
        oneway void addMessageToRoom(in string message);
};
```

Existe otra especificación IDL para la parte del monitoreo y como es similar a anterior no se la transcribeaquí. La misma se adjunta en formato electrónico.

El cliente se conecta al ORB

Una vez que el cliente ingresa sus datos (nombre y dirección de e-mail) debe crear su objeto "callback", a través del cual recibirá llamadas o mensajes del servidor o de otros clientes, y luego debe conectarse al ORB.

La invocación a **ORB.init()** retorna una referencia al objeto ORB sobre la cual se podrán realizar invocaciones (las especificadas en la Parte I Capítulo III).

La operación **orb.connect(m_callbackObj)** conecta al ORB la implementación del objeto “callback” creado, para que se encuentre disponible a recibir llamadas remotas.

Por último, el cliente debe realizar un "bind" al servidor a través de la utilización de la interfaz **UserActionsHelper**. La operación **m_uref = UserActionsHelper.bind(bStr, m_host)** devuelve una referencia a la implementación de la interfaz IDL **UserActions** que será utilizada para realizar invocaciones al servidor como por ejemplo registrarse como un nuevo usuario.

```
boolean initOrbixWebObjects(){
```

```
        orb = ORB.init();
        bStr = "ConnServ:IntServ";
        //Create the callback object
        try{
              m_callbackObj = new local_implementation(this, m_name,
                                                          m_email,my_interface);
        }
        catch (SystemException ex){
                System.out.println("No anduvo la creacion del callback");
                ex.toString());
                System.exit(1);
        }
        try{
                orb.connect(m_callbackObj);
        }
        catch (SystemException se){
                System.out.println ("Unexpected exception: " + se.toString());
        }
        //Bind to server
        try{
                m_uref = UserActionsHelper.bind(bStr, m_host);
        }
        catch (SystemException se){
                System.out.println("Exception during binding " + se.toString());
                displayMsg("Exception during binding " + se.toString());
                System.exit(1);
        }
        return(true);
}
```

Activación del servidor

Normalmente, antes de que el servidor sea activado, se crea un conjunto inicial de objetos que se conectan al ORB. Luego se invoca a **impl_is_ready()** para indicar al ORB que el servidor ya ha sido inicializado y está disponible para recibir pedidos.

El objeto **my_imlp** es del tipo **UserRegistrationImplementation**, clase que hereda de **_UserActionsImplBase** (clase Java que equivale a la interfaz IDL **UserActions**). Por lo que el objeto **my_impl** es una referencia a la implementación de la interfaz **UserActions**.

```
public static void main(String args[])
        {
                ORB orb;
                String roomName = null;
                UserInRoomActions ri = null;
                UserRegistrationImplementation my_impl = null;

                orb = ORB.init();
                //Creo los rooms a partir de un archivo de texto con sus nombres.
                //Las implementaciones de estos se registran del mismo
                //modo que los callbacks de cliente por lo que se omite el código
                try{
                        my_impl = new UserRegistrationImplementation
                ("ConnServ",   new UsersConnectedManager(), roomsList);
                }
                catch (SystemException se){
                        System.out.println("Exception : " +
                                                        se.toString());
                }
                orb.connect(my_impl);
                try{
                        _CORBA.Orbix.impl_is_ready("IntServ",
                                        _CORBA.IT_INFINITE_TIMEOUT);
                }
                catch (SystemException se){....//Se omite el resto
```

Invocaciones dinámicas

Como hemos visto en el Capítulo IV de la primera parte, las invocaciones dinámicas nos permiten realizar llamadas asincrónicas. En el caso de la aplicación desarrollada, esto es de gran utilidad cuando un cliente llama a otro pidiéndole la posibilidad de establecer una conexión privada. No sería deseable que cuando un cliente llama a otro quede bloqueado esperando respuesta. Para que esto no ocurra se utiliza una invocación dinámica y el cliente pregunta periódicamente si existe respuesta a su pedido, teniendo la posibilidad de abandonar el intento.

El código que realiza la invocación dinámica es como sigue:

```
public void clickedCallButton(String name){
        TimerThread tt;
        boolean resp = false;
        boolean answered = false;
        CallBack cb =   SearchCallBack(name);
        if(cb != null){
                m_callbackObj.setCanReceiveCalls(false);
                others_name = name;
                ORB orb = ORB.init();
                NVList argsList = orb.create_list(2);
                NamedValue mn = argsList.add(ARG_IN.value);
                mn.value().insert_string(m_name);
                NamedValue cbObject = argsList.add(ARG_IN.value);
                cbObject.value().insert_Object(m_callbackObj);
                Context ctx = orb.get_default_context();
                Any a = orb.create_any();
                NamedValue respVal = orb.create_named_value("",a,0);
                Request request = cb._create_request(ctx, "Call",
                                                        argsList, respVal);
                try{
                        request.send_deferred();
                }
                catch (SystemException ex){
                        MessageBox.show("Exception: " + ex.toString(),
                                                "Exception", MessageBox.OK);
                }

                try{
                        boolean goOn = false;
                        int yesorno;
                        while(! goOn){
                                answered = request.poll_response();
                                if (! answered){
                                        tt = new TimerThread();
                                        answered = request.poll_response();
                //Se omite parte del código que pregunta al usuario si
```

```
//desea seguir esperando
if(answered){
        request.get_response();
        respVal = request.result();
        resp = respVal.value().extract_boolean();
```

Lo que apreciamos en esta porción de código es el armado del **Request** a partir de la **NVList** (lista de argumentos constituida por objetos del tipo **NamedValue**). Un objeto del tipo **NamedValue** consiste de un nombre (que es indiferente en este caso), un valor del tipo **Any** y una bandera que indica si el parámetro es **in**, **out** o **in/out**.. El objeto **respVal** del tipo **NamedValue** será el que nos permita extraer el valor de la respuesta.

A través de la llamada a request.poll_response() podemos controlar si ya existe respuesta disponible. Una vez que esto ocurre, utilizamos las operaciones **request.get_response()** y **request.result()** para obtenerla y extraer el resultado respectivamente.

Objetos callback

No deseo concluir este trabajo sin dejar de describir la importancia de los objetos *callback*. Es necesario primero definir qué es un objeto callback, que ya ha sido mencionado anteriormente asumiendo que, por el contexto, se entendía su significado o función. Una definición del término "objeto *callback*" es: objeto de un cliente que recibe, o está en grado de recibir, invocaciones de un servidor. Por lo tanto, nos da la posibilidad de que un cliente pueda recibir información de un servidor sin haberlo forzado a pedirla.

Otra definición indica que una "llamada *callback*" (que se realiza a un objeto *callback*) permite que un servidor realice una invocación al cliente; y que, en consecuencia, los roles de cliente y servidor se invierten (el cliente pasa a ser servidor y viceversa). Esta definición, según mi criterio, es más adecuada que la primera pero es limitada en cuanto a su alcance.

Los objetos callback nos dan la posibilidad, no sólo de que un servidor pueda invocar una función en un cliente, sino también la de que un cliente pueda invocar una función en otro cliente, y esto se puede apreciar el la aplicación desarrollada cuando un cliente envía un mensaje a otro directamente, sin que el pedido pase por ningún servidor.

Contar con una arquitectura como CORBA para la creación de aplicaciones orientadas a objetos en ambientes distribuidos, es un gran avance tecnológico que genera muchas posibilidades y campos para explorar.

Indice de Ilustraciones

Bibliografía consultada

- CORBA 2.0 Specification (Object Management Group)
- CORBA Services (Object Management Group)
- Java to IDL Languaje Mapping (Object Management Group)
- COM vs. COBRBA, A Decision Framework (Owen Tallman y J. Bradford Kain, whitepaper)
- The Component Object Model Specification (Microsoft Corporation, 1995)
- A Discussion of the Object Management Architecture (Object Management Group)
- Distributed Computing Overview (Ted Burghart)
- IIOP: OMG´s Internet Inter-ORB Protocol (David Curtis, Object Management Group)
- CORBA Fundamentals and Programming (editado y publicado por Jon Siegel, 1996)
- CORBA for Real Programers (Reaz Hoque, 1999)
- CORBA Specification and Architecture (especificación de la OMG).
- OrbixWeb Programming Guide (Iona)
- OrbixWeb Reference Guide (Iona)
- Client/Server Programming with Java and CORBA (Robert Orfali y Dan Harkey, 1998)
- Core Java (Cay S. Horstman y Gray Corneell, 1999)
- Java Networking and Communicactions (Todd Courtois, 1997)
- TCP/IP (Commer).