Informe Técnico – Technical Report

DPTOIA-IT-2002-001

Febrero, 2002

Fundamentos para el desarrollo de aplicaciones distribuidas basadas en CORBA

Francisco José García Peñalvo

Jaime González González

Iván Álvarez Navia

María N. Moreno García

Belén Curto Diego

Vidal Moreno Rodilla

Departamento de Informática y Automática

Universidad de Salamanca

Revisado por:

Dr. Luis Antonio Miguel Quintales

Departamento de Informática y Automática

Universidad de Salamanca

lamq@usal.es

D. Diego Sevilla Ruiz

Departamento de Ingeniería y Tecnología de Computadores

Universidad de Murcia

dsevilla@ditec.um.es

Aprobado en el Consejo de Departamento de 26-2-2002.

Información de los autores:

D. Iván Álvarez Navia[ϕ], Dra. Belén Curto Diego[θ], Dr. Francisco José García Peñalvo[ξ],
D. Jaime González González[Ψ], Dra. María de Navelonga Moreno García[ϕ],
Dr. Vidal Moreno Rodilla[θ]

[ϕ]Área de Lenguajes y Sistemas Informáticos -Departamento de Informática y Automática
Facultad de Ciencias - Universidad de Salamanca
Plaza de la Merced S/N – 37008 - Salamanca

[θ] Área de Ingeniería de Sistemas - Departamento de Informática y Automática
Facultad de Ciencias - Universidad de Salamanca
Plaza de la Merced S/N – 37008 – Salamanca

[ξ] Área de Ciencias de la Computación e Inteligencia Artificial
Departamento de Informática y Automática
Facultad de Ciencias - Universidad de Salamanca
Plaza de la Merced S/N – 37008 - Salamanca

[Ψ] Becario de Investigación - Departamento de Informática y Automática
Facultad de Ciencias - Universidad de Salamanca
Plaza de la Merced S/N – 37008 – Salamanca

Este trabajo ha sido parcialmente financiado por la Junta de Castilla y León y la Unión Europea a través del Fondo Social Europeo mediante el proyecto de investigación SA02/00F.

Resumen

CORBA es actualmente una de las opciones tecnológicas más importantes a la hora del desarrollo de sistemas software distribuidos. Adentrarse en el mundo del desarrollo de sistemas distribuidos no es una tarea sencilla, y previamente se deben de conocer una serie de conceptos clave para posteriormente profundizar en los diferentes problemas que, tanto en el ámbito del desarrollo industrial como en el ámbito académico e investigador, aparecen en este tipo de sistemas (aplicaciones de tiempo real, tratamiento de eventos, asincronismo, calidad de servicio...).

En el presente documento se van a tratar dos temas fundamentalmente. El primero de ellos aborda los sistemas distribuidos en general, presentando sus características, el concepto de *middleware*, que en su acepción más general da soporte a la interacción de aplicaciones software distribuidas, esto es, hace referencia a la infraestructura que se encarga de conectar de alguna manera los procesos cliente y servidor que aparecen en este tipo de aplicaciones; y sobre todo el patrón *broker* como el elemento común que comparten las principales arquitecturas de *middleware* como son CORBA o COM+.

El segundo punto que se estudia en este trabajo es CORBA. Este *framework* es sumamente amplio, por lo que se tratarán de abordar los elementos más fundamentales y básicos para iniciarse en esta tecnología. Así se abordan aspectos como su modelo objeto, su arquitectura y sus servicios. También se abordan los primeros pasos para desarrollar una aplicación CORBA.

Abstract

Actually, CORBA is one of the most important technological options in distributed software systems development. Going inside of the distributed systems' world is not an easy activity, for this reason every one should know a set of key concepts before. With these concepts you can deepen in the different issues related to the distributed systems that appear in both industrial and academic/research areas, such a way real-time applications, events, asynchronous responses, quality of service and so on.

In this document two topics are fundamentally studied. The first one is related to general distributed systems, presenting their main characteristics, the middleware notion – general middleware systems support the interaction of arbitrary application programs, this means that the middleware layer seeks primarily to hide the underlying networked environment's complexity by insulating applications from explicit protocol handling, disjoint memories, data replication, network faults, and parallelism – and specially we introduce the broker pattern as the common element shared by the main middleware architectures like CORBA or COM+.

The second aspect studied in this work is CORBA. This framework is very wide, and then we are going to introduce the most basic notions needed to begin working with this technology. This way, aspects such its object model, its architecture and its services are introduced. Also, firsts necessary steps to develop a CORBA application are treated.

Tabla de Contenidos

1. Introducción

La evolución de las aplicaciones software de los últimos años ha venido de la mano de la integración de la informática con el mundo de las telecomunicaciones, debido en gran medida al desarrollo de las redes locales y especialmente al auge de Internet.

Los programas tienden a superar el modelo monolítico según el cual dada una entrada obtenían una salida, caminando hacia la federación de un conjunto de componentes que interactúan entre sí, cada vez más en un entorno distribuido. Cada uno de estos componentes ofrece unos servicios a otros componentes lo que constituye la aplicación software final.

Las primeras soluciones que incluían servicios de red vinieron con los denominados *sistemas centralizados*, en los que un solo computador con una o varias unidades centrales de proceso recibía las peticiones de los usuarios que se conectaban a él vía terminales *tontas* o *green screens*. Sin embargo, debido a razones tales como coste, confianza, falta de escalabilidad, falta de flexibilidad y la diversa naturaleza de las diferentes divisiones que conformaban una organización, hacían que este modelo no fuera especialmente atractivo.

El abaratamiento del hardware, la aparición en escena de los ordenadores personales (PC – *Personal Computer*), el aumento de la potencia de los PCs y de las estaciones de trabajo y el desarrollo de redes locales, provocaron una evolución en los modelos arquitectónicos de las aplicaciones software, pareciendo razonable repartir el proceso entre algunos sistemas más o menos distribuidos, en los que normalmente seguirían residiendo las bases de datos, y las máquinas desde las que se conectarían los usuarios, PCs o estaciones de trabajo donde se ejecutaría al menos la parte de interfaz de usuario.

Había surgido así el modelo Cliente/Servidor (C/S) [Bohnhoff et al., 1994], caracterizado por dividir la funcionalidad de la aplicación en dos papeles perfectamente definidos: cliente y servidor.

De modo abstracto, el servidor ofrece una serie de servicios que pueden ser utilizados por los clientes para completar la funcionalidad de la aplicación. Una interacción básica implica a un cliente que inicia una petición de algún servicio a un servidor. El servidor entonces realiza la función especificada por el cliente, devolviendo los posibles resultados que el servicio genera. En la práctica, los clientes y servidores se implementan como procesos que se están ejecutando en máquinas conectadas a una red. La infraestructura que se encarga de conectar a los procesos cliente y servidor se denomina *middleware*.

La arquitectura C/S se organiza en niveles o capas, existiendo arquitecturas de dos, tres o en general *n* capas. En las arquitecturas de dos capas, existen diversas configuraciones, que van desde los clientes livianos y servidores gordos (*thin clients & fat servers*), donde la mayor carga se la llevan los servidores, siendo los clientes sólo responsables de la lógica de presentación, a los clientes gordos (*fat clients*) donde los servidores se ven relegados a simples repositorios de datos.

La arquitectura C/S más popular es la de tres capas (Figura 1). El cliente se encarga de mantener la interfaz de usuario. En un nivel intermedio se encarga de implementar la lógica de la aplicación. Finalmente, en el último nivel se encuentra la lógica de datos. Los clientes se construyen sobre la base de unos servicios encapsulados en los procesos que implementan la lógica de la aplicación, siendo más robustos frente a cambios en esta lógica o en la lógica de datos. La funcionalidad que implementan los clientes es muy sencilla, pudiendo varios clientes reutilizar servicios estándares definidos en alguno de los niveles intermedios. La aplicación al estar dividida en partes más pequeñas, hace que el proceso de distribución de funcionalidad en los procesadores más adecuados sea mucho más flexible.

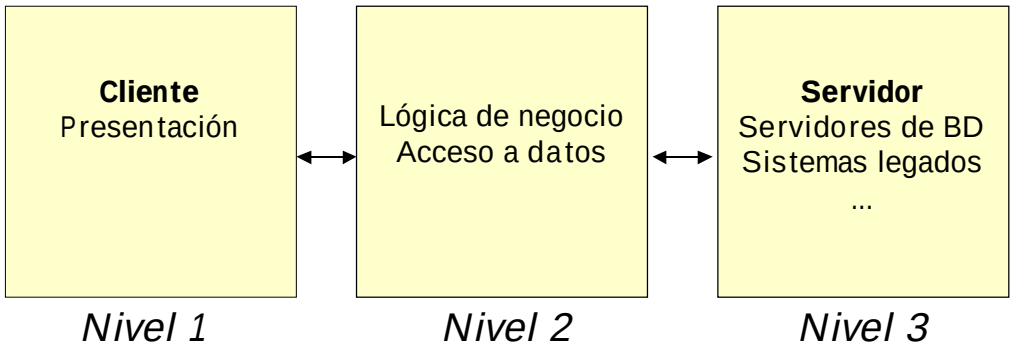

Figura 1. Esquema de una arquitectura de tres niveles

Un ejemplo de este tipo de sistemas C/S multicapa son las aplicaciones web. El primer nivel es la interfaz de usuario, mediante un navegador que interpreta el código HTML y puede ejecutar componentes ActiveX [Chappel, 1996] o *applets* de Java [SUN, 2001], que residen en el segundo nivel. El último nivel son los datos que se encuentran en el servidor web y en el servidor de base de datos.

Los sistemas distribuidos son el último paso en la computación C/S [Lewis, 1995]. En vez de diferenciar entre las distintas partes de la aplicación, los sistemas distribuidos ofrecen toda la funcionalidad en forma de "objetos", con un significado similar al término "objeto" de la Programación Orientada a Objetos (POO), pero no existen los papeles explícitos de cliente y servidor, esto significa que aunque existen los roles de cliente y servidor, éstos pueden cambiar en el tiempo.

Un sistema de programación distribuida basado en objetos, va a ofrecer, por tanto, las características de un sistema de programación orientado a objetos, además de las características de un sistema descentralizado o distribuido. Dichas características típicamente se resumen en los siguientes tópicos [Chin y Chanson, 1991]: *distribución, transparencia, integridad de datos, tolerancia a fallos, disponibilidad, capacidad de recuperación, autonomía de los objetos, concurrencia de programas, concurrencia de objetos* y *mejora en el rendimiento*.

Normalmente, al ejecutar un programa desarrollado con un lenguaje de orientado a objetos, el flujo de ejecución principal se encarga de crear y destruir los objetos dinámicamente. Sin embargo, al distribuir los objetos, esto se ejecutarán en procesos distintos (espacio de memoria disjuntos). El *middleware* tiene que encargarse de que las invocaciones a objetos remotos sean similares a las que hacen los objetos locales (transparencia de distribución), pasar los mensajes necesarios por la red, y devolver las respuestas de las operaciones al que hizo la llamada.

Los procesos que componen la aplicación, y que se están ejecutando en distintas máquinas de la red, son o pueden llegar a ser a la vez clientes y servidores, los cuales cooperan para conseguir la funcionalidad total de la aplicación. El mundo de los sistemas distribuidos es un mundo de "entidades pares", esto es, elementos de procesamiento con distintas disponibilidades de recursos, distinta capacidad de almacenamiento..., que cooperan ofreciendo servicios en forma de objetos y requiriendo servicios de otros objetos implementados en otros nodos de la red.

El hecho de ofrecer una serie de servicios en forma de objetos hace que el diseño y desarrollo se haga basándose en interfaces bien definidas que facilitan y apoyan la modularidad y la reutilización, a la vez que permiten un diseño mucho más flexible.

Aunque la complejidad inherente de los sistemas distribuidos es muy alta, haciendo que los componentes distribuidos sean más difíciles de diseñar, depurar y mantener que otras piezas de software, las ventajas que ofrecen son demasiado prometedoras como para ignorarlas, pudiendo destacar entre éstas [Ozsu y Valduriez, 1991], [Tanenbaum, 1992]:

- **Reducción potencial de costes:** Las redes de computadores incorporan tanto PCs como estaciones de trabajo ofrecen una mejor relación precio/rendimiento que los grandes ordenadores.
- **Capacidad para compartir recursos:** Es la propiedad por la que se permite a los elementos involucrados en el sistema utilizar los recursos de los otros.
- **Mejora en el rendimiento y la escalabilidad:** Cuando el número de clientes que accede a un recurso crece, el tiempo de respuesta comienza a caer. La naturaleza dispersa de los sistemas distribuidos ayuda a evitar esto porque los recursos se hallan físicamente en máquinas diferentes. Las peticiones por los recursos se envían a diferentes máquinas, haciendo que el proceso de la petición sea inherentemente distribuida. Además, el número de computadores en un sistema distribuido beneficia al sistema global en el sentido de que aporta mayor poder de procesamiento.
- **Autonomía local:** Un sistema distribuido es responsable de administrar sus recursos. Ofrece autonomía local a los nodos sobre sus propios recursos. Cada uno de ellos puede aplicar políticas, configuraciones o accesos de control locales sobre los recursos y los servicios.
- **Distribución intrínseca:** Algunas aplicaciones son inherentemente distribuidas.
- **Mejora de la confianza y la disponibilidad:** Una máquina de la red puede fallar sin afectar al resto del sistema.

Independientemente del balance que se pudiera hacer entre las ventajas de los sistemas distribuidos y sus problemas de complejidad de desarrollo, prueba y seguridad, se encuentra el contexto real de la demanda de aplicaciones software que hoy en día se tiene, donde la presencia de los servicios web es cotidiana, ya sea a través de Internet o intranets, la heterogeneidad resulta evidente como suma de diversos factores (diversidad de plataformas hardware, software y de comunicaciones, inexistencia de soluciones únicas, sistemas legados...) y se camina cada vez más hacia la idea de sistemas abiertos.

El objetivo principal debe ser conseguir la interoperabilidad, esto es, la capacidad de los objetos de interoperar, independientemente de su ubicación en la red, del lenguaje de implementación y de la plataforma donde se ejecuten.

Una de las características más notables, y a su vez un claro problema, de los sistemas distribuidos es su dependencia de la existencia de unos servicios de red subyacentes. La comunicación es proporcionada por la red, siendo posible construir sistemas distribuidos sólo con lo que la red por sí misma ofrece. De todas formas la distribución es más fácil de conseguir, mantener y extender si la plataforma de distribución proporciona una abstracción de la capa de red y permite un uso natural de los componentes remotos.

Normalmente, las redes de ordenadores son heterogéneas. Así, si desarrollar software para un sistema distribuido es difícil, desarrollar software para un sistema distribuido heterogéneo a veces es imposible, aumentando las, ya de por sí, grandes barreras de la programación distribuida.

De forma muy general, para desarrollar aplicaciones para entornos distribuidos heterogéneos habría que tener en cuenta los siguientes aspectos:

1. Buscar modelos y abstracciones independientes de plataforma que puedan ser útiles para resolver distintos tipos de problemas.
2. Ocultar todo lo posible la complejidad de bajo nivel sin sacrificar demasiado la ejecución.

La utilización de modelos y abstracciones correctos puede proporcionar una capa de aplicación homogénea sobre el sistema distribuido. Esta capa oculta los detalles de bajo nivel y permite a los desarrolladores resolver problemas sin tener que preocuparse de la capa de red para las distintas plataformas.

Una solución es la aproximación *middleware*, que consiste en extender un lenguaje de programación existente introduciendo una nueva capa entre la de aplicación y la de red que oculte la complejidad de la comunicación y de la transferencia de datos. Para el usuario, la invocación de un procedimiento remoto debería ser similar a la invocación de uno local, es decir, el *middleware* presenta objetos locales y realiza el envío transparente de los datos a los distintos recipientes.

En el presente documento se abordan los elementos fundamentales de una de estas arquitecturas *middleware*, CORBA (*Common Object Request Broker Architecture*) [OMG, 1995], [OMG, 1997], [OMG, 1998a], [OMG, 2001a], [OMG, 2001e] caracterizada por permitir la computación de objetos distribuidos.

De esta manera, en una primera aproximación, puede verse a CORBA como una norma o conjunto de normas que especifican la interoperabilidad entre objetos en un entorno distribuido heterogéneo de manera transparente.

Que sea una norma o conjunto de normas, implica que puede haber diferentes implementaciones que, cumpliendo la norma, difieran en cuanto a eficiencia, robustez, precio... La interoperabilidad en entornos heterogéneos se hace posible definiendo mecanismos que tengan en cuenta una triple vertiente: lenguaje, plataforma y sistema operativo. Por último, la transparencia establece que todos los aspectos relacionados con el acceso a red y a los objetos, locales o remotos, los lenguajes de programación... queden bajo responsabilidad de la implementación CORBA.

El resto del documento se organiza como sigue: en la sección dos se introduce con más profundidad el concepto de *middleware*, haciendo especial hincapié en la noción de *middleware* orientado a objetos distribuidos. La sección tercera está dedicada a presentar el patrón *broker*, patrón que recoge los fundamentos conceptuales que se encuentran tras la mayoría de los *frameworks* o arquitecturas *middleware* para el desarrollo de sistemas distribuidos. En la cuarta sección se hace una primera aproximación a CORBA con el objeto de situar su génesis y presentar su evolución hasta nuestros días, así como para introducir una terminología básica que establezca el vocabulario que se va a utilizar en el resto de secciones. Las secciones cinco y seis abordan los modelos de referencia de CORBA, esto es, el modelo objeto presente en CORBA y su arquitectura de gestión de objetos, OMA (*Object Management Architecture*), donde se introducen las diferentes interfaces con las que se ha de trabajar en un entorno CORBA. La séptima sección aborda más en detalle la arquitectura CORBA, desgranando sus componentes principales, estudiando la forma en que interactúan éstos para posibilitar la comunicación entre los elementos distribuidos, y presentando sus características fundamentales; así se abordará el estudio del ORB, el lenguaje de definición de interfaces, los adaptadores de objetos... La octava sección volverá a tratar el tema de los servicios CORBA, ya introducido al presentar OMA, pero esta vez se van a presentar en detalle dos servicios fundamentales en los trabajos del Grupo de Investigación de Robótica del Departamento de Informática y Automática de la Universidad de Salamanca, como son el servicio de nombres y el servicio de eventos. En la sección nueve se presentan algunos de los ORBs de carácter gratuito que más se utilizan en el desarrollo de aplicaciones CORBA en general, y en el seno del grupo de investigación en particular. La sección diez explica los pasos que hay que dar para crear una aplicación CORBA. Y por último, la sección once cierra el trabajo con unas conclusiones generales del mismo.

2. Middleware

En la última mitad de la década de los años ochenta los diseñadores de software introdujeron las plataformas *middleware* para ofrecer un soporte a los sistemas software distribuidos. El término *middleware* hace referencia a la infraestructura que se encarga de conectar de alguna manera los procesos cliente y servidor. Engloba a todos los elementos que hacen posible esta comunicación, desde las líneas físicas hasta los protocolos de alto nivel. El soporte estándar de desarrollo de aplicaciones distribuidas es Internet, por lo que la infraestructura *middleware* está cubierta hasta el nivel de transporte de la arquitectura OSI (*Open Systems Interconnection*), e incluso en algunos casos hasta el de aplicación.

Las primeras muestras básicas de *middleware* se pueden encontrar en elementos como llamadas a procedimientos remotos, servicios de ficheros y directorios que estaban basados en los avances en la tecnología hardware y los servicios de red. En la actualidad el ámbito del *middleware* es mucho más amplio, y los sistemas distribuidos ocupan un lugar importante tanto en la industria como en el mundo académico e investigador.

Entre los primeros ejemplos de plataformas para sistemas distribuidos cabe destacar *Athena* en el MIT (*Masschusetts Institute of Technology*) [Balkovich et al., 1985], *ITC/Andrew* en la *Carnegie Mellon University* [Morris et al., 1986], *ANSAware* [Herbert, 1987] y DACNOS de IBM y la Universidad de Karlsruhe [Geihs y Hollberg, 1990]. El modelo de programación de DACNOS comprendía una combinación de un modelo de comunicaciones asíncronas y de un modelo objeto [Geihs y Hollberg, 1990]. Posteriormente, actividades de estandarización dirigidas por consorcios industriales dieron como resultado especificaciones de arquitecturas *middleware* estándares, tales como DCE (*Distributed Computing Environment*) [Rosenberry et al., 1992] o CORBA [OMG, 2001a]. Actualmente, OMG CORBA y Microsoft COM+ [Platt, 1999] son los *frameworks* más utilizados para el desarrollo de arquitecturas *middleware* para entornos distribuidos de propósito general, aunque últimamente las soluciones basadas en el lenguaje Java [SUN, 2001], tales como *Jini* o los *Enterprise JavaBeans* [SUN, 2000], están recibiendo una creciente atención.

Dar una definición del concepto de *middleware* es sumamente difícil, y la mayoría de las existentes en la bibliografía comparte un problema común: dependiendo del entorno de aplicación varían en la opinión de los componentes que aparecen en la capa de *middleware*. El *middleware* general da soporte a la interacción de aplicaciones software arbitrarias; así funciones específicas como por ejemplo acceso remoto a bases de datos, trabajo en grupo o sistemas de *workflow* requieren de la existencia de una solución basada en *middleware* [Geihs, 2001].

Una definición de *middleware* que se ajusta especialmente bien al contexto en el que se desarrolla el presente trabajo es la que aparece en [Bacon et al., 2000], donde se define este concepto como *la capa de software que se ejecuta encima de sistemas operativos y sistemas de comunicaciones heterogéneos, ofreciendo una interfaz uniforme a las aplicaciones distribuidas*.

En el presente documento la acepción que interesa destacar del término *middleware* es la de DOC (*Distributed Object Computing*) *middleware*, el cual reside entre las aplicaciones y los elementos subyacentes (sistemas operativos, pilas de protocolos y dispositivos físicos) para simplificar y coordinar la conexión de estos componentes y la forma en que ellos interoperan. Adoptando la filosofía de paso de mensajes que subyace a la tecnología de objetos, el DOC *middleware* hace que el patrón de interacción entre las partes de una aplicación distribuida sea mucho más flexible.

Según argumenta Douglas C. Schmidt en el prólogo de [Tari y Bukhres, 2001], el DOC *middleware* puede dividirse en las siguientes capas:

- ***Middleware* de infraestructura**: Potencia y encapsula los mecanismos de comunicación y concurrencia nativos de los sistemas operativos para crear componentes de red orientados a objetos. Estos componentes ayudan a eliminar muchos aspectos no portables, tediosos y tendentes a error, propios del desarrollo y mantenimiento de aplicaciones de red mediante APIs de bajo nivel propias de los sistemas operativos, como por ejemplo *sockets* o POSIX *pthreads*.
- ***Middleware* de distribución**: Extiende el anterior para crear un modelo de programación de alto nivel. Define APIs y componentes reutilizables que automatizan tareas de programación de aplicaciones de red tales como la gestión de conexión, *(un)marshaling*, demultiplexación... Este tipo de *middleware* permite que las aplicaciones distribuidas puedan ser desarrolladas como si se utilizasen las técnicas tradicionales que se emplean para el desarrollo de aplicaciones no distribuidas, permitiendo que los clientes invoquen las operaciones de los objetos destino (*target objects*) sin conocer ningún dato sobre la localización, lenguaje de programación, sistema operativo, protocolos de comunicaciones y hardware de estos objetos destino. El núcleo de este tipo de *middleware* son los ORBs (*Object Request Broker*), tales como el modelo de componentes de Microsoft COM+ (*Component Object Model*) [Microsoft, 1996] el RMI (*Remote Method Invocation*) de Sun [SUN, 2001] o CORBA de OMG [OMG, 2001a].
- ***Middleware* de servicios comunes**: Define servicios de alto nivel independientes del dominio, tales como notificaciones de eventos, autentificación, persistencia, seguridad, planificación de tiempo real, calidad de servicio... Mientras que el *middleware* de distribución se centra en la gestión de los recursos de los nodos finales para dar soporte a un modelo de programación orientada a objetos distribuida, este tipo de *middleware* se centra en la gestión de los recursos a través del sistema distribuido. Los desarrolladores pueden reutilizar estos servicios para alojar, planificar y coordinar los recursos globales y realizar tareas de distribución comunes que de otra forma se implementarían de una forma desorganizada.
- **Servicios específicos del dominio**: Construidos para satisfacer los requisitos de dominios concretos, tales como telecomunicaciones, comercio electrónico o salud por ejemplo. A diferencia de otras capas de *middleware*, que ofrecen mecanismos y servicios para una reutilización horizontal, éstos servicios están destinados a mercados verticales. Esta es, hoy en día, la capa menos madura, debido en parte a la falta de estándares en el *middleware* de distribución y de servicios.

3. El patrón *broker*

El patrón *broker* [Buschmann et al., 1996] es un patrón que ayuda a resolver los problemas derivados del diseño de sistemas de computación distribuida, presentando un conjunto de componentes desacoplados que interactúan a través de invocaciones a servicios remotos.

En el presente apartado se van a presentar los conceptos y la experiencia de diseño que están detrás de este patrón arquitectónico, aunque previamente se va a hacer una breve introducción a los patrones de diseño.

3.1. Patrones de diseño

El concepto de patrón, como elemento reutilizable de experiencia y conocimiento ha calado profundamente en el área del desarrollo de aplicaciones software, teniendo su caldo de cultivo más activo en la comunidad de orientación a objeto. De este hecho se deriva el término patrón

software y más concretamente el de patrón de diseño para hacer referencia al uso de patrones en el Diseño Orientado a Objeto (DOO).

Según el diccionario de la Real Academia Española [DRAE, 1995] *un patrón es un dechado que sirve de muestra para sacar otra cosa igual*, estando esta definición muy cercana a la idea que se persigue con los patrones software, donde los patrones, en lugar de servir como muestra física, sirven para almacenar y documentar soluciones recurrentes a problemas tipo.

El uso del término patrón, con el significado que actualmente se le da en la ingeniería del software, y más concretamente en el área de la tecnología de objetos, se deriva de los trabajos del arquitecto Christopher Alexander desde mediados de los años sesenta. Sin embargo, aún estando su trabajo centrado en la arquitectura, sus ideas son aplicables a muchas otras disciplinas, entre las que cabe destacar su aplicación en el desarrollo de software.

No existe una definición estándar para el término patrón, así se pueden encontrar varias en la bibliografía. De hecho, parafraseando a James Coplien: "*Alexander podría haber escrito una frase de definición de lo qué es un patrón, pero en su lugar él escribió un libro de 550 páginas para hacerlo porque el concepto es difícil*".

Entre las muchas definiciones de patrón software se ha elegido la que ofrece el propio James Coplien, por ser una de las más cercanas al concepto de patrón difundido por el Dr. Alexander. Así, un patrón *es una regla constituida por tres partes, la cual expresa una relación entre un cierto contexto, un cierto sistema de fuerzas que ocurren repetidamente en ese contexto, y una cierta configuración software que permite a estas fuerzas resolverse a sí mismas* [Coplien, 1998].

Esta definición sirve para introducir una serie de términos que configuran la terminología propia de los patrones. En primer lugar el término **sistema de fuerzas**, significa que un problema se refiere a un conjunto de fuerzas. La noción de **fuerza** generaliza el tipo de criterios que los ingenieros del software utilizan para justificar los diseños y las implementaciones. En el caso de los patrones, éstos casan con conjuntos de objetivos y restricciones que se encuentran en el desarrollo de cada elemento que se vaya a crear. Estas fuerzas son grandes, difíciles de medir y conflictivas. También es interesante resaltar el término **configuración software** en el sentido de un diseño en una forma canónica o un conjunto de reglas de diseño que alguien puede aplicar para solucionar las fuerzas del problema.

En resumen, un patrón involucra una descripción general de una solución recurrente para un problema recurrente repleto de diferentes objetivos y restricciones.

Existen varios tipos de patrones software, que alcanzan casi todas las parcelas de la ingeniería del software, aunque los más conocidos y utilizados son los denominados patrones de diseño. Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí, adaptada para resolver un problema de diseño general en un contexto particular [Gamma et al., 1995].

Dentro de los patrones de diseño existen diversas categorías, así por ejemplo en [Buschmann et al., 1996] se distinguen las siguientes categoría de patrones:

- **Patrones Arquitectónicos**: Expresan una organización estructural fundamental para un sistema software, que se refiere a un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y guías para organizar las relaciones entre ellos.
- **Patrones de Diseño**: Ofrecen esquemas para refinar subsistemas y componentes de un sistema software, o las relaciones entre ellos. Describen normalmente una estructura de comunicación recurrente entre componentes, que sirve para resolver un problema general de diseño dentro de un contexto particular.

- **Patrones de implementación (*Idioms*)**: Un *idiom* es un patrón de bajo nivel, específico de un determinado lenguaje de programación. Describen como implementar aspectos particulares de los componentes, o de sus interrelaciones, utilizando las características de un determinado lenguaje. Como ejemplo de un *idiom* se tiene la siguiente construcción en C++ para copiar cadenas de caracteres: `while (*destino++ = *src++);`.

El patrón *broker* es un patrón clasificado dentro de los patrones arquitectónicos, pues su misión es transmitir qué subsistemas componen un sistema software distribuido.

3.2. Descripción del patrón broker

El patrón *broker* [Buschmann et al., 1996] se utiliza para estructurar sistemas software distribuidos con componentes desacoplados que interaccionan por invocaciones de servicio remotas, donde el *broker* es el componente responsable de coordinar la comunicación. Diversas plataformas, como Microsoft OLE 2.x (*Object Linking and Embedding*) [Brockschmidt, 1994], IBM SOM/DSOM (*System Object Model / Distributed System Object Model*) [Campagnoni, 1994] o CORBA [OMG, 2001a], comparten este patrón arquitectónico.

El contexto en el que se utiliza este patrón es el de un sistema distribuido, y posiblemente heterogéneo, con componentes independientes que cooperan.

Construir sistemas software complejos mediante un conjunto de componentes desacoplados e interoperativos, en lugar de aplicaciones monolíticas, ofrece flexibilidad, mantenibilididad y capacidad de cambio. Así, al repartir la funcionalidad entre un conjunto de componentes independientes se logra que el sistema potencialmente distribuido y escalable.

Sin embargo, cuando los componentes distribuidos se comunican entre sí, se requiere algún tipo de comunicación entre procesos. Si los componentes manejan ellos mismos la comunicación el resultado será un sistema con demasiadas limitaciones y dependencias.

Además, se necesitan servicios para añadir, eliminar, cambiar, activar y localizar componentes. Las aplicaciones que utilizan estos servicios no deben ser dependientes de los detalles específicos de cada sistema, para garantizar así la portabilidad y la interoperabilidad, aunque se esté en un entorno heterogéneo.

Desde el punto de vista de los desarrolladores, éstos no deben encontrar diferencias esenciales a la hora de desarrollar una aplicación distribuida. Una aplicación que use un objeto sólo debe ver la interfaz ofrecida por el objeto, no debiendo conocer nada en absoluto sobre los detalles de implementación del objeto o de su localización física.

De esta manera, el patrón *broker* se utiliza para equilibrar las siguientes fuerzas que aparecen en el contexto descrito:

- Los componentes deben ser capaces de acceder a los servicios que ofrecen otros componentes mediante invocaciones de servicio remotas y transparentes a la localización de los servidores.
- Se necesita cambiar, añadir o eliminar componentes en tiempo de ejecución.
- La arquitectura debe ocultar los detalles específicos del sistema o de la implementación a los usuarios de los componentes y de los servicios.

La solución pasa por introducir un componente *broker* para manejar mejor los clientes y los servidores desacoplados. Los servidores se registran ellos mismos al *broker*, poniendo sus servicios a disposición de los clientes a través de las interfaces de sus métodos. Los clientes acceden a la funcionalidad de los servidores enviando las peticiones a través del *broker*. Las

tareas del *broker* incluyen la localización del servidor apropiado, enviar la petición al servidor y transmitir los resultados y excepciones de vuelta al cliente.

Utilizando el patrón *broker*, una aplicación puede acceder a los servicios distribuidos enviando un mensaje al objeto apropiado, en vez de centrarse en las comunicaciones de bajo nivel entre procesos. La arquitectura *broker* ofrece una gran flexibilidad, permitiendo cambios, incorporaciones, eliminaciones de objetos de forma dinámica.

Este patrón reduce la complejidad en torno al desarrollo de sistemas distribuidos porque hace que la distribución sea transparente al desarrollador. Esto lo consigue introduciendo un modelo objeto en el que los servicios distribuidos se encuentran encapsulados dentro de los objetos. Los sistemas *broker* ofrecen un camino para integrar la tecnología de los sistemas distribuidos con la tecnología orientada a objetos. Además, extienden los modelos objeto desde las aplicaciones centralizadas a las aplicaciones distribuidas formadas por componentes desacoplados que pueden ejecutarse en plataformas heterogéneas y que pueden estar implementados en diferentes lenguajes de programación.

La estructura del patrón *broker* incorpora seis tipos de componentes, tal y como se puede apreciar en el diagrama de clases UML (*Unfied Modeling Language*) [OMG, 2001d] de la Figura 2: clientes, servidores, *brokers*, puentes, *proxies* del lado del cliente y *proxies* del lado del servidor,

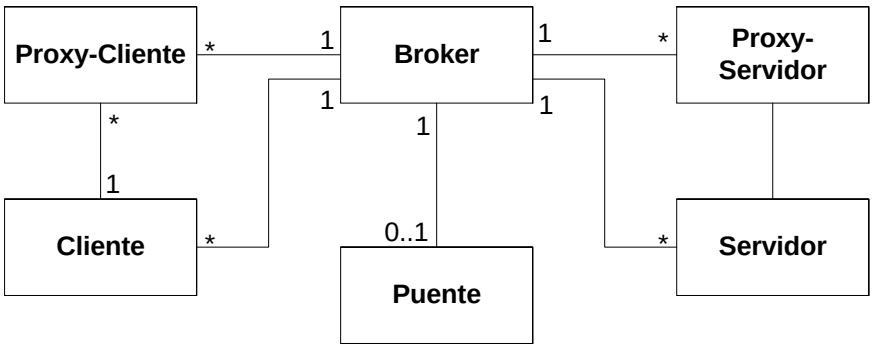

Figura 2. Estructura del patrón *broker* [Buschmann, 1996]

Un servidor implementa objetos que ofrecen su funcionalidad a través de interfaces que consisten en operaciones y atributos. Estas interfaces se realizan mediante lenguajes de definición de interfaces o mediante estándares binarios. Las interfaces normalmente agrupan funcionalidad semánticamente relacionada. Existen dos tipos de servidores:

- Los que ofrecen servicios comunes a varios dominios de aplicación.
- Los que implementan funcionalidad específica para una sola aplicación o tarea.

Los clientes son las aplicaciones que acceden a los servicios de al menos un servidor. Para llamar a servicios remotos, los clientes envían peticiones al *broker*, después de que la operación se ha llevado a cabo, reciben las respuestas y las excepciones de éste.

La interacción entre clientes y servidores se basa en un modelo dinámico, que significa que los servidores pueden actuar también como clientes, lo cual difiere de la concepción habitual del modelo C/S.

En la Figura 3 se resumen las responsabilidades y colaboraciones de las entidades servidor y cliente, utilizando para ello sendas tarjetas CRC (*Class, Responsability and Collaboration*) [Beck and Cunningham, 1989].

Clase *Cliente*	**Colaboraciones** • *Proxy* del lado del cliente • *Broker*
Resposabilidades • Implementar la funcionalidad de usuario • Enviar peticiones a los servidores a través de un *proxy* del lado del cliente	
Clase *Servidor*	**Colaboraciones** • *Proxy* del lado del servidor • *Broker*
Resposabilidades • Implementar los servicios • Registrarse en el *broker* • Enviar las respuestas y excepciones al cliente a través del *proxy* del lado del servidor	

Figura 3. Resposabilidades de los clientes y servidores en el patrón *broker*

Un *broker* es el responsable de la transmisión de peticiones de los clientes a los servidores, y de las respuestas y excepciones de éstos últimos a los clientes. Debe ser capaz de localizar al receptor de una petición basándose en un identificador único del sistema. Además, el *broker* ofrece APIs (*Application Programming Interface*) a los clientes y servidores que incluyen operaciones para registrar a los servidores y para invocar a los métodos del servidor.

Cuando llega una petición a un *broker* la pasa directamente al servidor. Si el servidor está inactivo, el *broker* lo activará. Todas las respuestas y excepciones procedentes de la ejecución de un servicio son enviadas por el *broker* al cliente que envió la petición inicial. Si el servidor especificado está localizado en otro *broker*, el *broker* local encontrará una ruta para llegar la petición al *broker* remoto, esto implica la necesidad de una forma para que los diferentes *brokers* puedan interactuar.

Dependiendo de los requisitos del sistema pueden integrarse servicios adicionales al broker: servicio de nombres, servicio de conversión...

En la Figura 4 se resumen las responsabilidades y colaboraciones de la entidad *broker*.

El *proxy* del lado del cliente representa una capa entre los clientes y el *broker*. Esta capa ofrece transparencia, de forma que un objeto remoto parece al cliente como si fuera un objeto local. Los *proxies* permiten ocultar los detalles de implementación a los clientes, tales como:

- Mecanismos de comunicación entre procesos utilizados para la transferencia de mensajes entre clientes y *brokers*.
- Creación y borrado de bloques de memoria.
- Traducción a formato independiente de la máquina de parámetros y resultados (*marshaling*).

El *proxy* del lado del servidor es análogo al de la parte cliente, la diferencia está en que ellos son los responsables de recibir las peticiones, desempaquetar los mensajes que llegan, traducir al formato dependiente de plataforma los parámetros (*unmarshaling*) y llamar al servicio apropiado. También deben traducir los resultados y excepciones al formato independiente de la plataforma antes de mandárselos al cliente.

En la Figura 5 se presentan responsabilidades y colaboraciones de los *proxies* presentes en la estructura del patrón *broker*.

Clase *Broker*	**Colaboraciones** • Cliente • Servidor • *Proxy* del lado del cliente • *Proxy* del lado del servidor • Puente
Resposabilidades • Registrar y dar de baja servidores • Ofrecer APIs • Transferir mensajes • Recuperar errores • Interoperar con otros *brokers* a través de puentes • Localizar servidores	

Figura 4. Responsabilidades de la entidad *broker*

Clase *Proxy del lado del cliente*	**Colaboraciones** • Cliente • *Broker*
Resposabilidades • Encapsular la funcionalidad específica del sistema • Mediar entre el cliente y el *broker*	
Clase *Proxy del lado del servidor*	**Colaboraciones** • Servidor • *Broker*
Resposabilidades • Llamar a los servicios del servidor • Encapsular la funcionalidad específica del sistema • Mediar entre el servidor y el *broker*	

Figura 5. Responsabilidades de los *proxies*

Por último, los puentes son componentes opcionales que se emplean para ocultar los detalles de implementación cuando dos *brokers* interactúan. Sus responsabilidades y colaboraciones se muestran en la.

Clase *Puente*	**Colaboraciones** • *Broker* • Puente
Resposabilidades • Encapsular la funcionalidad específica de la red • Mediar entre el *broker* local y el puente de otro *broker* remoto	

Figura 6. Responsabilidades de la entidad puente

La estructura presentada en el patrón *broker* puede presentar variantes, diferenciándose fundamentalmente dos tipos de *brokers*:

- Aquéllos que usan comunicación indirecta entre clientes y servidores, ideas que se han presentado en el patrón *broker* genérico [Buschmann et al., 1996].
- Aquéllos que usan comunicación directa entre clientes y servidores. Para buscar un mejor rendimiento, algunas implementaciones de *brokers* se limitan a establecer la conexión inicial entre cliente y servidor, y el resto de las comunicaciones se hace directamente entre los participantes. De esta forma, los mensajes, las excepciones y las respuestas se transfieren entre los *proxies* del lado cliente y del lado servidor sin hacer uso del *broker* como capa intermedia. Esta comunicación directa requiere que tanto clientes como servidores entiendan el mismo protocolo. Un caso de este tipo de *broker* se encuentra definido en el patrón *Client-Dispacher-Server* [Buschmann et al., 1996].

Un ejemplo práctico de la utilización del patrón *broker* en el diseño de sistemas software distribuidos se puede encontrar en [González et al., 2000].

4. Una primera aproximación a CORBA

En este apartado se van a presentar las nociones básicas de este *framework* o *middleware*, con el objetivo de conocer tanto sus orígenes como la terminología básica de CORBA.

4.1. Orígenes de CORBA

CORBA es un estándar publicado por el OMG (*Object Management Group*) para una red de objetos distribuida y heterogénea.

El OMG (http://www.omg.org) es un grupo de empresas formado en 1989 con el propósito de promover la orientación a objetos en la ingeniería del software y el establecimiento de una plataforma arquitectónica común para el desarrollo de programas basados en objetos distribuidos. Actualmente reúne más de 800 miembros entre los que se incluyen las mayores empresas del mundo de la informática y de las telecomunicaciones (entre ellas Sun, Hewlett-Packard, IBM, Telefónica I+D, Netscape, Nokia, Oracle, Rational SW, Xerox o Fujitsu), y también multitud de empresas de distintos sectores de la industria, las finanzas, la educación...

Su objetivo no es producir software sino especificaciones para lograr la interoperabilidad. No es una solución realista que todo el mundo acepte trabajar con un mismo software, un mismo sistema operativo o un mismo lenguaje de programación. Así que se propone buscar una solución para soportar la heterogeneidad de entornos que se pueden encontrar en un sistema distribuido.

La primera versión del estándar CORBA (CORBA 1.1) se publicó en 1991. Ésta incluía: un lenguaje de definición de interfaces o IDL (*Interface Definition Language*), unas interfaces de programación de aplicaciones y un bus de que gestiona las peticiones de los objetos distribuidos, el ORB (*Object Request Broker*). En estas primeras versiones de CORBA, el OMG se centra en la especificación de la parte central de CORBA, el ORB, además de unos cuantos servicios (servicio de nombres, *trading* y servicio de eventos).

En 1994 aparece la versión CORBA 2.0 [OMG, 1995], [OMG, 1997], [OMG, 1998a], [OMG, 2001a] cuya principal mejora fue la especificación de un protocolo para integrar ORBs de diferentes vendedores.

Recientemente, a lo largo del año 2000, el consorcio OMG ha comenzado la publicación de la versión CORBA 3 [Siegel, 2000], [Tari y Bukhres, 2001], que ofrece soluciones en tres áreas: integración en Internet, calidad de servicio y una arquitectura de componentes para CORBA.

4.2. Aportaciones de CORBA

CORBA aporta un conjunto de especificaciones que, al ser incorporadas a las aplicaciones distribuidas, ofrecen una forma de conseguir la interoperabilidad entre sistemas distribuidos de naturaleza heterogénea. Para ello CORBA se centra en principios fundamentales: *la separación entre interfaz e implementación, la independencia de localización* y *la independencia del fabricante y la integración de sistemas a través de la interoperabilidad*.

CORBA es una aproximación completamente basada en objetos, ya que en el OMG se ha tenido siempre el convencimiento de que la tecnología de objetos es la más adecuada para el desarrollo de aplicaciones distribuidas porque simplifica el problema.

La filosofía que siguen los sistemas distribuidos puede verse como análoga a la filosofía que siguen los objetos en el paradigma objetual, dado que los objetos se entienden como componentes software discretos que contienen datos y que pueden manipularlos; así los clientes (otros componentes software) envían mensajes a los objetos con peticiones, y los objetos devuelven su respuesta con otros mensajes.

En CORBA todos los componentes son objetos. Cada objeto se puede implementar con un lenguaje de programación distinto, y ejecutarse sobre cualquier plataforma hardware y sistema operativo. Para OMG, un objeto es una entidad que encapsula una funcionalidad a través de una interfaz. Un objeto ofrece servicios a través de sus operaciones y atributos que son visibles mediante su interfaz. Esta entidad permite a los clientes solicitar la realización de operaciones en un objeto con independencia de su localización. El elemento que permite la transparencia de localización y de acceso a los objetos es el ORB, que podría ser el equivalente en software al bus (como se define en el hardware) que interconecta componentes. Esta idea se resume en la Figura 7, donde se puede apreciar como los objetos presentan, mediante interfaces, a otros objetos los servicios que ofrecen, y las peticiones de estos servicios, así como las respuestas que se produzcan, serán canalizadas a través del ORB, que oculta la localización de los objetos que intercambian mensajes, de forma que éstos actúan como si los orígenes y los destinatarios de los mensajes fueran objetos locales. En relación con el apartado 3, el ORB se puede ver como el *broker* del patrón del mismo nombre, o incluso como el *broker* más el puente.

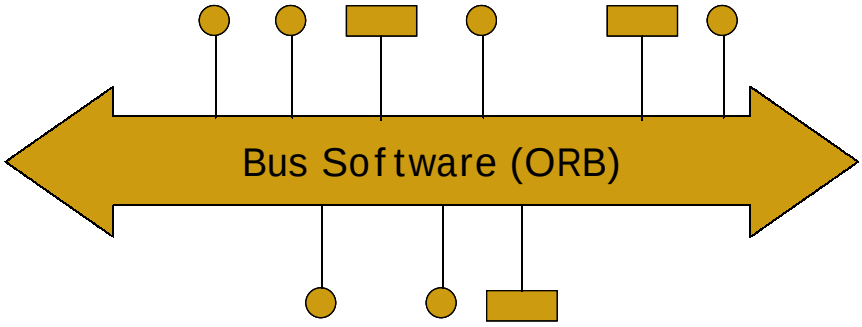

Figura 7. Interacción entre los objetos distribuidos a través de un ORB

Otro aspecto de suma importancia en la aproximación orientada a objetos que planta CORBA es la clara apuesta por la separación entre la interfaz y la implementación de los objetos.

Las interfaces de los objetos se especifican en un lenguaje especialmente definido para este fin, IDL (*Interface Definition Language*) [OMG, 2001a], que forma parte del estándar CORBA.

IDL es un elemento clave para lograr la interoperabilidad perseguida por CORBA. IDL es un lenguaje orientado a objetos para definir interfaces, pero no para programar los servicios de

éstos. Para la implementación de los servicios OMG define cómo se corresponden las construcciones de IDL con las de los lenguajes de programación[1].

IDL aísla la interfaz de un objeto de su implementación, permitiendo así ganar portabilidad e interoperabilidad. Es posible crear una nueva aplicación o utilizar una ya existente a la que se añade un envoltorio (*wrapper*) que le permita ser accedida mediante CORBA (simplemente adaptándole una interfaz IDL).

CORBA logra la independencia del fabricante y la integración de sistemas gracias a que a partir de la versión 2.0 [OMG, 1995] se define un estándar para que ORBs de distintos fabricantes puedan integrarse en organizaciones heterogéneas y escalables de objetos distribuidos. El protocolo GIOP (*General Inter-ORB Protocol*) y su versión para Internet, IIOP (*Internet Inter-ORB Protocol*) [Curtis et al., 1997], permiten a ORBs de distintos fabricantes comunicarse de una manera estándar. Esto, de cara al desarrollador de aplicaciones, ofrece dos claras ventajas: la independencia del vendedor y una invocación de métodos independiente de si ambos objetos (cliente y servidor) están en el mismo o en diferentes ORBs.

4.3. Terminología básica de CORBA

En este subapartado, a forma de glosario, se van a definir los términos que se manejan a la hora de explicar los componentes y funcionamiento de la arquitectura y modelos de referencia propuestos por CORBA.

- *Objeto CORBA*: Entidad virtual capaz de ser localizada por un ORB y de recibir peticiones por parte de un cliente.
- *Objeto destino (target)*: Objeto CORBA al que se dirige una petición por parte de un cliente.
- *Cliente*: Entidad que realiza peticiones sobre un objeto CORBA.
- *Servidor*: Aplicación en la que existen uno o más objetos CORBA.
- *Petición*: Invocación de una operación sobre un objeto CORBA por un cliente.
- *Referencia a objeto*: Etiqueta usada para identificar, localizar y direccionar un objeto CORBA de forma única. Son opacas e inmutables.
- *Sirviente (servant)*: Entidad de un lenguaje de programación que implementa uno o más objetos CORBA. Los sirvientes existen dentro del contexto de una aplicación servidor.

5. El modelo objeto de CORBA

Este apartado describe el modelo objeto de CORBA. Dicho modelo objeto es similar a cualquier modelo objeto de referencia, aunque con algunas particularidades especialmente centradas en el significado que se le da a algunos conceptos clave.

El modelo objeto de CORBA es abstracto y no se implementa en ninguna tecnología particular. Un sistema objeto en CORBA es una colección de objetos que aísla mediante una interfaz bien definida a los que solicitan los servicios (clientes) de los que los ofrecen (servidores). El modelo objeto de CORBA es definido como dos modelos [OMG, 1995]:

[1] Estas correspondencias se denominan *bindings* o *mappings*, existiendo para diversos lenguajes de programación (C, C++, Java, Ada, Smalltalk, COBOL...).

- El modelo cliente, que describe la semántica de los objetos, esto es, los conceptos que son significativos para los clientes, tales como identidad de objeto y creación de objetos, peticiones y operaciones, y tipos y signaturas.
- El modelo de implementación que describe los conceptos relacionados con la implementación de los objetos, incluyendo conceptos como métodos, motores de ejecución y activación de objetos.

El modelo objeto de CORBA es un caso particular de un modelo objeto convencional, donde el cliente envía un mensaje a un objeto [Tari y Bukhres, 2001]. Conceptualmente, el objeto interpreta el mensaje para decidir que servicio ejecutar. En el modelo convencional, un mensaje identifica a un objeto y cero o más parámetros actuales. Como en la mayoría de los modelos más convencionales, se requiere de un primer parámetro diferenciador, que es el que identifica la operación a ejecutar. La interpretación del mensaje por el objeto implica la selección de un método basado en la operación especificada. En el caso de CORBA, la selección del método puede ser llevada a cabo tanto por el objeto como por el ORB.

5.1. Semántica de los objetos

La semántica de los objetos define los conceptos que son relevantes para los clientes, y que se resumen e los siguientes puntos [OMG, 1995]:

- Como en los modelos objeto convencionales, los objetos son entidades de un sistema objeto. Un objeto es una entidad encapsulada e identificable que ofrece uno o más servicios que pueden ser solicitados por un cliente.
- Los clientes solicitan servicios. El término *petición* se utiliza para hacer referencia a una secuencia completa de eventos relacionados que ocurren entre el cliente que inicia la petición y el último evento asociado con dicha iniciación. La información asociada con la petición está compuesta de una operación, un objeto destino, cero o más parámetros actuales, y un contexto de petición que es opcional.
- Los parámetros en una petición son instanciados con valores. Un valor es una instancia de un tipo de dato de OMG. Hay valores que no son objetos (por ejemplo, las cadenas), y también referencias a objetos. Una referencia a objeto es un valor que inequívocamente denota a un objeto particular. Una referencia a objeto identificará al mismo objeto cada vez que la referencia se utilice en una petición, aunque está sujeta a ciertos límites prácticos de espacio y tiempo.
- Los objetos pueden ser creados y destruidos. Desde la perspectiva del cliente, no hay un mecanismo especial para estas operaciones, como pueden existir en los sistemas orientados a objetos (por ejemplo, los constructores en Java [SUN, 2001] o en C++ [Stroustrup, 1997]). Los objetos se crean y se destruyen como resultado de las peticiones. El resultado de la creación de un objeto es revelado al cliente en forma de una referencia a un objeto que denota a un nuevo objeto.
- Los tipos en CORBA son similares a los definidos en los modelos objeto convencionales. Los tipos se emplean en las signaturas para restringir a un parámetro o caracterizar un resultado. La extensión de un tipo es el conjunto de entidades que satisfacen al tipo en cualquier momento. El tipo de un objeto es un tipo cuyos miembros son referencias a objetos.
- Hay diferentes tipos, los definidos por CORBA y los definidos por los clientes. Un tipo en CORBA puede ser básico (lógico, carácter...) o un tipo estructurado. Un tipo estructurado puede ser un tipo tupla (consistente en un conjunto de parejas ordenadas <nombre, valor>), un tipo unión, un tipo secuencia (consistente en un vector de longitud variable de un solo tipo), un tipo matriz (consistente en una

matriz multidimensional de dimensión fija de un solo tipo), una interfaz (que especifica un conjunto de operaciones que una instancia de ese tipo debe soportar), o un tipo valor (que especifica estado además de un conjunto de operaciones que una instancia del tipo debe soportar).

- Las interfaces son el concepto clave en el modelo objeto de CORBA. Describen un conjunto de operaciones posibles que un cliente puede solicitar de un objeto, a través de sus interfaces. Comparándolo con los modelos objeto tradicionales, las interfaces de CORBA constituyen la parte de operación de un tipo. La implementación del tipo está separada de la especificación del tipo. Una interfaz puede tener varias implementaciones, y la particularidad del modelo objeto de CORBA es que estas implementaciones pueden estar realizadas en diferentes lenguajes de programación como C, C++ o Java por ejemplo. No obstante, las interfaces deben estar definidas en el IDL definido por OMG.
- Una interfaz ofrece una descripción sintáctica de cómo se ofrecen los servicios por el objeto que soporta la interfaz. Un objeto satisface la interfaz si ofrece sus servicios a través de las operaciones de la interfaz de acuerdo a la especificación de las operaciones.
- El tipo interfaz para una interfaz dada es un tipo de objeto de forma que una referencia a objeto satisfará el tipo si y sólo si la referencia a objeto también satisface la interfaz.
- Una interfaz puede tener atributos. Un atributo es equivalente a declarar una pareja de funciones de acceso, una para recuperar el valor del atributo y otra para establecer el valor del mismo. Un atributo puede ser de sólo lectura, y en este caso sólo se provee del método selector, omitiéndose el método modificador. Desde la perspectiva de CORBA la mayor diferencia entre un atributo y una operación es que no se pueden asociar excepciones a los atributos.
- La herencia de interfaces ofrece un mecanismo de composición que permite al objeto soportar múltiples interfaces. La interfaz principal se compone de todas las operaciones presentes en el cierre transitivo del grafo de herencia de interfaces.
- El concepto de herencia tal cual se encuentra en los modelos objeto tradicionales debe ser por tanto redefinido. El concepto de herencia tradicional no es adecuado en los entornos distribuidos heterogéneos al estilo de CORBA. Esto es así porque la herencia tradicional implica tanto herencia de especificaciones (operaciones) y la herencia de implementaciones (datos y código de los métodos). Dado que un tipo en CORBA se define únicamente como parte relacionada con los clientes, en las operaciones, entonces la herencia de operaciones parece la única que tiene sentido en los entornos CORBA. La herencia de implementación no se considera porque la parte de implementación de los tipos se implementa en diferentes lenguajes de programación.
- Una operación es una entidad identificable que denota una primitiva de servicio indivisible que puede ser solicitada. Una operación se define de igual forma que en los modelos objeto convencionales, esto es, consta de un identificador (su nombre) y tiene una signatura (que describe los valores correctos que pueden tomar sus parámetros y valor de retorno). La forma general que presenta una signatura de una operación es:

```
[oneway] <tipo> <identificador> (param1, ..., paramL)
         [raises(excep1, ..., excepM)]
         [context(nombre1, ..., nombreN)]
```

Donde, `oneway` es una palabra clave opcional que indica que la operación se llama sólo una vez de forma informativa y no se espera la ejecución (normalmente se utiliza en método de finalización o de *ping*); `<tipo>` especifica el tipo del valor de retorno (`void` sino retorna nada); `<identificador>` es el nombre de la operación. Los parámetros vienen especificados mediante una lista separada por comas de valores `{in | out | inout} <tipo> <identificador>`, de manera que `in` especifica un parámetro de entrada, `out` uno de salida e `inout` uno de entrada y salida. La cláusula `<raises>` utiliza la palabra reservada `raises` y una lista entre paréntesis de excepciones que la operación puede lanzar; y por último la cláusula `<context>` se compone de la palabra reservada `context` seguida de una lista de identificadores encerrados entre paréntesis y separados por comas, utilizándose para establecer el contexto de la operación, es decir, para pasarle información adicional.

5.2. Objetos implementación

Describe los conceptos relevantes para implementar el comportamiento de los objetos en un sistema computacional. La implementación de un sistema de objetos conlleva las actividades necesarias para atender el comportamiento de las operaciones requeridas por los clientes. Estas actividades pueden incluir resultados a las peticiones, así como la actualización del estado del sistema.

El modelo de implementación se compone de dos partes [OMG, 1995]: el modelo de ejecución y el modelo de construcción.

Un servicio solicitado se ejecuta en un sistema computacional mediante la ejecución del código asociado a la operación solicitada y actuando sobre algunos datos. Los datos representan una forma de estado del sistema computacional. El código realiza el servicio solicitado, y éste puede cambiar el estado del sistema. El código que se ejecuta para realizar el servicio es denominado en *método*. Un método es una descripción inmutable de una rutina que puede ser interpretada por un motor de ejecución. Un método tiene un atributo que no cambia y que recibe el nombre de formato del método, que define el conjunto de motores de ejecución que son capaces de interpretar el método. Un motor de ejecución es una máquina abstracta (no un programa) que puede interpretar métodos en ciertos formatos, provocando las acciones computacionales a realizar. Un motor de ejecución define un contexto dinámico para la ejecución del método. La ejecución de un método se denomina activación del método. Cuando un cliente solicita una petición, se llama a un método en el objeto destino. Los parámetros de entrada son pasados del cliente al método, y los parámetros de salida y el valor de retorno (o las excepciones y sus parámetros) son devueltos al cliente.

El modelo de construcción ofrece los mecanismos adecuados para implementar el comportamiento de las peticiones. Estos mecanismos incluyen la definición del estado del objeto, la definición de los métodos y la definición de cómo la infraestructura del objeto permite seleccionar los métodos a ejecutar así como las porciones relevantes del estado del objeto para que sean accesibles para los métodos. Estos mecanismos deben ser capaces de describir las acciones asociadas con la creación de objetos, tales como la asociación de un nuevo objeto con los métodos apropiados. La implementación de un objeto es una definición que ofrece la información necesaria para la creación de un objeto y permitir que el objeto participe en los contratos con los clientes ofreciendo el conjunto apropiado de servicios. Una implementación típicamente incluye, entre otras cosas, las definiciones de los métodos que operan junto con el estado de un objeto.

6. OMA (Object Management Architecture)

Todas las especificaciones relacionadas con los objetos distribuidos y propuestas por el OMG comparten una infraestructura conceptual denominada OMA (*Object Management Architecture*), que define una arquitectura común que posibilita la existencia de un sistema unificado de cómputo basado en objetos distribuidos interoperando entre sí tanto en contextos homogéneos como heterogéneos.

Como se muestra en la Figura 8, la arquitectura de referencia OMA consta de los siguientes componentes:

- El ORB (*Object Request Broker*)
- Un conjunto de interfaces funcionales que se organizan en cuatro categorías: interfaces de aplicación, utilidades comunes, interfaces de dominio y servicios de objetos.

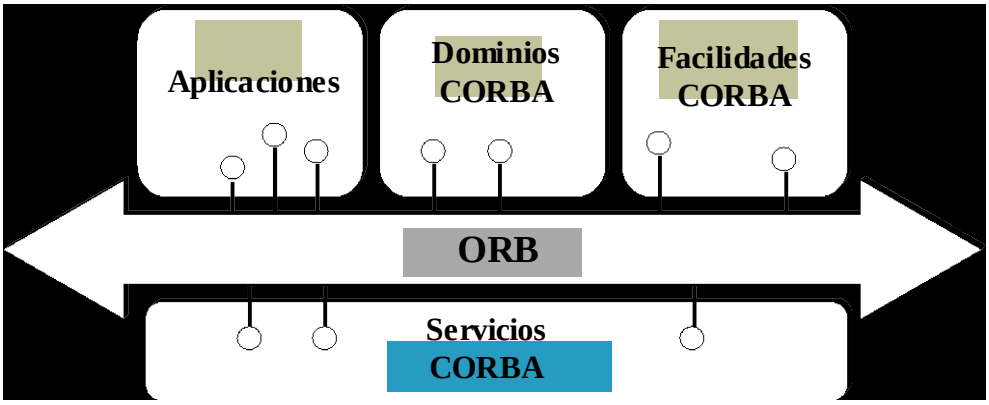

Figura 8. Estructura de OMA

El ORB es el máximo responsable de facilitar la comunicación entre los clientes y los objetos. Proporciona mecanismos estándares para que los objetos interactúen de forma transparente.

Los servicios CORBA son un conjunto de servicios de nivel de sistema con interfaces IDL. Se pueden entender como un complemento a la funcionalidad del ORB. Los servicios se requieren para construir aplicaciones distribuidas, siendo independientes éstos de los dominios de aplicación específicos. Un ejemplo de servicio CORBA es el servicio de ciclo de vida, que define la forma de crear, borrar, copiar y mover objetos, sin embargo, no se ocupa de los detalles de implementación de los objetos dentro de las aplicaciones. Más adelante se volverá a hablar de los servicios CORBA y, más concretamente, se detallaran los servicios de nombrado y de eventos.

Las utilidades o facilidades comunes de CORBA son interfaces orientadas a las aplicaciones finales, esto es, servicios que pueden ser compartidos por varias aplicaciones (servicios horizontales), pero que no son fundamentales como los servicios CORBA. Un ejemplo puede ser la interfaz DDCF (*Distributed Document Component Facility*), una utilidad para la composición de documentos basada en *OpenDoc*, que permite la presentación e intercambio de objetos basados en un modelo de documento.

Por su parte, los dominios CORBA son interfaces orientadas a dominios de aplicación específicos, es decir, a dominios verticales de mercado, como pueden ser las telecomunicaciones, la medicina o las finanzas por ejemplo.

Las interfaces de aplicación son las interfaces desarrolladas específicamente para una determinada aplicación. Al ser específicas de las aplicaciones finales y OMG no desarrolla aplicaciones, sino sólo especificaciones, estas interfaces no están estandarizadas.

En la Figura 9 se presenta una descripción detallada de cada uno de los componentes de OMA.

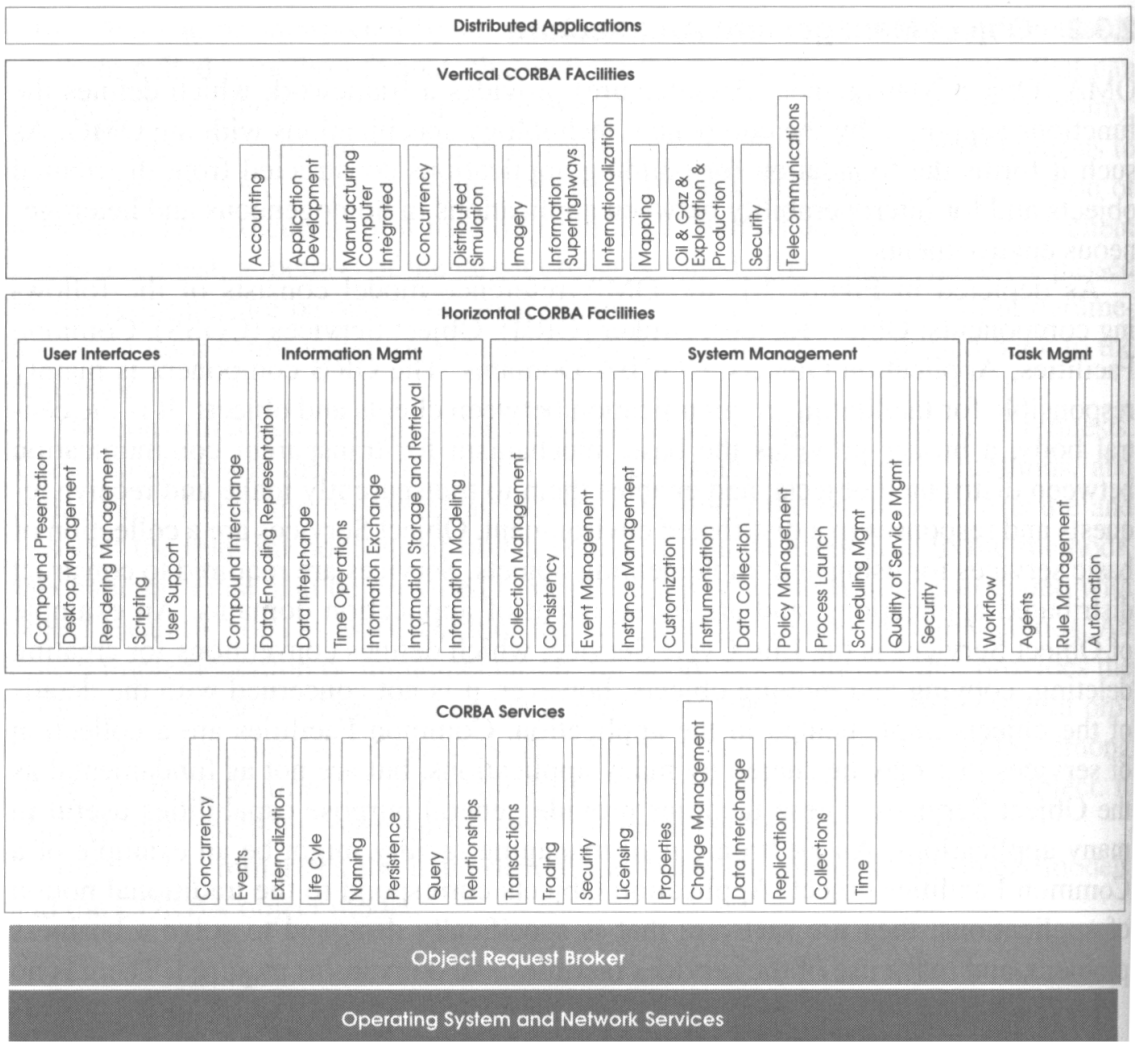

Figura 9. Vista detallada de OMA [Tari y Bukhres, 2001]

7. La arquitectura de CORBA

Los componentes de la arquitectura CORBA (ilustrados en la Figura 10) permiten que una aplicación cliente pueda solicitar la realización de operaciones en un objeto CORBA (objeto implementación en la terminología de CORBA) que reside en un servidor. Las implementaciones de los objetos que están en el lado servidor definen los métodos que implementan las interfaces IDL. Estos objetos pueden estar implementados en diversos lenguajes de programación.

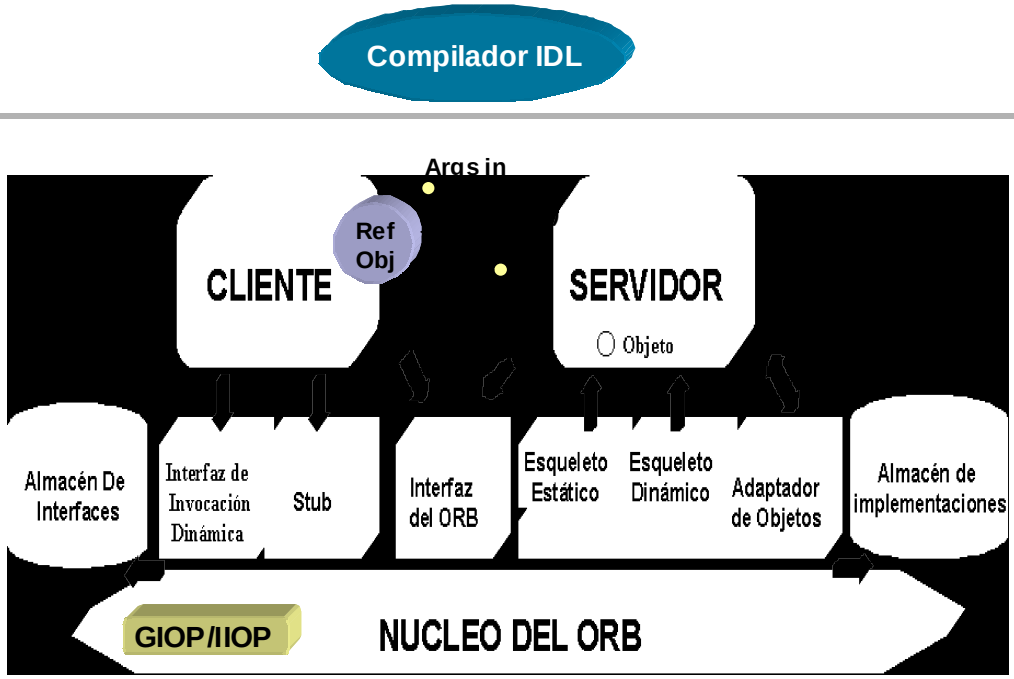

Figura 10. Arquitectura CORBA

El principal responsable de localizar el objeto CORBA al que va destinada una operación, y hacérsela llegar, es el *ORB Core* o núcleo del ORB. El ORB ofrece además una interfaz tanto al cliente como al servidor con varios servicios muy básicos, como por ejemplo para iniciar el uso del ORB.

El cliente es la entidad (programa) que invoca una operación de un objeto residente en el lado servidor. Una aplicación cliente debe estar desarrollada utilizando algún lenguaje de programación soportado por el ORB. Cada ORB soporta equivalencias para un solo lenguaje de programación en el se generan los *proxies* del lado cliente (*stub* - cabo) y del lado servidor (*skeleton* – esqueleto). El programa cliente puede referenciar a los tipos de objetos definidos en un fichero IDL. Cuando el fichero IDL es compilado, el ORB genera los *stubs* y los *skeletons* que servirán de *pegamento* entre las aplicaciones cliente y servidor y el propio ORB. Estos *proxies* son fundamentalmente los responsables de realizar las operaciones de *marshalling* y *un-marshalling* derivadas de las peticiones del cliente. Cuando un cliente invoca una operación presente en una interfaz IDL, esta operación será traducida por el *stub* al formato de adecuado para su transmisión (*marshalling*) y enviada al ORB. Este tipo de invocación se denomina estática porque el *stub* conoce la operación invocada en tiempo de compilación.

Existe otra alternativa donde el cliente puede llamar a operaciones que son desconocidas al *stub*. Esta situación puede darse cuando la implementación del objeto ha sido cambiada en el servidor (añadiendo una implementación de una operación por ejemplo) y el IDL no se ha recompilado para generar un nuevo *stub* (que conozca la nueva operación). También puede darse este caso cuando un cliente quiera delegar la ejecución de una operación a un objeto implementación sin que el *stub* conozca la interfaz IDL de ésta (por ejemplo, el cliente recibe una referencia a objeto remota a través del servicio de *trading* y desea invocar una operación en dicho objeto). Este tipo de invocaciones recibe el nombre de invocaciones dinámicas, y para satisfacerlas dentro del lado cliente de la arquitectura CORBA se cuenta con el DII (*Dynamic Invocation Interface*), que permite a los clientes acceder directamente a los mecanismos subyacentes ofrecidos por el ORB. Las aplicaciones cliente utilizan el DII para solicitar peticiones a la parte del servidor de forma dinámica, sin verse obligadas a contar con *stubs*. Al contrario que sucede con los *stubs*, el acceso a través de DII permite a los clientes hacer

llamadas asíncronas (separando las operaciones de envío y recepción) y llamadas de dirección única (*send-only or oneway calls*).

Por lo que respecta al lado del servidor, como se puede apreciar en la Figura 10, existen diversos componentes que son utilizados por el ORB para negociar la ejecución de una operación solicitada por el cliente, ya sea estática o dinámicamente. Si la operación ha sido invocada de forma estática mediante el mecanismo SSI (*Static Skeleton Invocation*), entonces el ORB enviará la petición al *adaptador de objetos* gracias a la información contenida en la referencia al objeto que ha utilizado el cliente para realizar la invocación.

Las referencias a objetos contienen detalles que ayudan a la ejecución de las operaciones en los servidores. Incluyen información relativa a la localización del servidor, la identidad del adaptador y del OID (*Object Identificator*) del objeto en el servidor.

El adaptador de objetos asiste al ORB en la tarea de hacer llegar las peticiones de los clientes a los objetos implementación adecuados, activando los objetos necesarios dentro del lado del servidor. Para hacer esto, el adaptador necesita tener la información necesaria para asociar los OIDs con las clases de los objetos implementación. Así, cuando el ORB delega la ejecución de la operación en el adaptador de objetos, lo primero que hace éste es identificar la clase del objeto implementación y solicitar al *skeleton* que descodifique la operación (*un-marshalling*) para invocarla correctamente.

El adaptador de objetos tiene otras funciones además de activar o desactivar los objetos en el servidor. Así, también es el responsable de llevar a cabo las políticas de seguridad especificadas en las implementaciones de los objetos.

Existe un mecanismo análogo al DII pero en el lado del servidor, éste se denomina DSI (*Dynamic Skeleton Interface*). El DSI permite que un ORB envíe peticiones a la implementación de un objeto que no tiene conocimiento en tiempo de compilación del tipo de objeto que implementa. El cliente que hace la petición no conoce si la implementación utiliza una interfaz especificada en un esqueleto o utiliza esqueletos dinámicos. Gracias a esto CORBA puede procesar operaciones que en principio no conocía cuando se implementó su código (esto sirve, por ejemplo, para implementar pasarelas entre plataformas, como la que OMG ha definido entre CORBA y COM/DCOM).

En la Tabla 1 se resumen los componentes de la arquitectura CORBA, tanto en el lado del cliente como en el lado del servidor.

PARTE CLIENTE	PARTE SERVIDOR
***Stub* del cliente**: Capa intermedia entre el cliente y el núcleo del ORB. Define cómo los clientes invocan los servicios que proporcionan los objetos servidores; se encargan de codificar la operación y sus parámetros, y de enviarla de forma remota.	**Esqueleto (*skeleton*) del servidor**: Proporciona los elementos necesarios para que los clientes invoquen los servicios exportados por el objeto. Tratan de hacer transparente el proceso de comunicación.
Interfaz de invocación dinámica o DII (*Dynamic Invocation Interface*): Permite descubrir en tiempo de ejecución métodos para ser invocados.	**Esqueleto de interfaces dinámicas o DSI (*Dynamic Skeleton Interface*)**: Proporciona un mecanismo de enlazado en tiempo de ejecución para servidores que necesitan manejar peticiones realizadas dinámicamente.
Almacén de interfaces: Permite obtener y modificar la descripción de todos los objetos que están registrados en él, métodos que soporta y los parámetros que requiere. No es más que una base de datos distribuida.	**Adaptador de objetos**: Elemento que se encarga de gestionar las peticiones que llegan al servidor y que asigna referencias e instancia objetos.
Interfaz del ORB: Bibliotecas de servicios locales para realizar labores auxiliares en la aplicación.	**Almacén de implementaciones**: Almacén de información acerca de las clases que soporta el servidor, los objetos instanciados y sus OIDs.

Tabla 1. Componentes de la arquitectura CORBA

Después de esta visión global se van a explicar de forma más detallada los principales componentes de esta arquitectura.

7.1. Núcleo del ORB

Como ya se ha comentado con anterioridad, el ORB ofrece un mecanismo para comunicar de forma transparente las peticiones realizadas por los clientes con los objetos implementación destino. El ORB simplifica el desarrollo de aplicaciones distribuidas desacoplando del cliente de los detalles de invocación de los métodos. Esto hace que las llamadas que realiza el cliente tengan la apariencia de invocaciones a rutinas locales. Cuando un cliente invoca una operación, el ORB es el responsable de encontrar el objeto implementación, activarlo de forma transparente si fuera necesario, enviarle la petición, y mandar la respuesta al cliente que invocó la operación.

El ORB toma el papel del elemento negociador dentro del *framework* que establece CORBA. Un ORB conoce las interfaces de ciertos objetos. Los IDLs de los objetos que él conoce están localizadas en una lista dinámica denominada *repositorio de interfaces*. Un ORB tiene la capacidad de conocer objetos que residen en otros sistemas (interoperabilidad). Cuando es consultado, el ORB intenta encontrar una correspondencia entre los datos inmersos en la petición y los que tiene en su repositorio de interfaces. Entonces el ORB, a través del *repositorio de implementaciones*, intenta enviar un mensaje al objeto. Si el objeto o su servidor no están en ejecución, el ORB encontrará en el repositorio de implementaciones la referencia de cómo y dónde iniciar al objeto. Una vez que el ORB tiene la referencia correcta, tratará de iniciar el objeto y mandarle el mensaje original. Si el proceso falla o el ORB no conoce al objeto solicitado retornará un mensaje de error al cliente.

El repositorio de interfaces se guarda en el *host* y es mantenido por el ORB. Está compuesto de una lista de todas las interfaces IDL de los objetos que el ORB tiene que conocer. Las interfaces IDL pueden residir en diferentes *hosts* al del ORB, pero deben estar registradas en el repositorio de interfaces para su uso. Aunque las aplicaciones clientes no necesitan ser compiladas o enlazadas con otros objetos CORBA, éstas pueden hacer llamadas, y usar la información contenida en el repositorio de interfaces, a través del DII.

El repositorio de implementaciones está también mantenido por el ORB. Este repositorio permite al ORB localizar y activar los objetos implementación.

En general, y tras lo expresado, se puede argumentar que el ORB simplifica el desarrollo de aplicaciones distribuidas ocultando a los desarrolladores:

- La localización del objeto: el cliente no necesita saber la localización del objeto al que realiza la petición.
- La implementación del objeto: el cliente no necesita saber nada sobre el lenguaje de implementación del objeto, ni el SO, ni la plataforma hardware.
- El estado de ejecución del objeto: el cliente no necesita sabe si el objeto está activado en el momento de realizar la petición. Si no fuera así, el ORB se encarga de activarlo.
- Los mecanismos de comunicación: el cliente no necesita conocer el mecanismo de comunicación (tipo de red, protocolo, memoria compartida).

7.2. Lenguaje de Definición de Interfaces (IDL)

El IDL [OMG, 2001a] es un elemento clave para la interoperabilidad. Separa la interfaz de la implementación permitiendo definir las interfaces de los objetos CORBA.

El IDL ofrece una forma de especificar las interfaces de los objetos implementación de una forma independiente de los lenguajes de programación elegidos para implementar los métodos

de los objetos implementación. Las interfaces se especifican en el lado cliente como ficheros IDL. En el lado del servidor, los *servants* se tienen como especificaciones abstractas de los objetos implementación. Los *servants* pueden ser clases propias del lenguaje soportado por el ORB, como por ejemplo C++ o Java.

Esto difiere de los sistemas centralizados, donde tanto la especificación como la implementación se definen en un único sistema, utilizando un único lenguaje para la especificación y la implementación. Dado que CORBA es acorde con la filosofía de los objetos distribuidos, lleva a cabo una división en dos componentes: uno que implica la interfaz IDL y otro que describe la implementación de dichas interfaces IDL. Puede haber varias implementaciones para la misma interfaz, pudiendo estar en diferentes servidores. Además, las implementaciones pueden estar realizadas en diferentes lenguajes. El ORB, conjuntamente con el adaptador de objetos, es el responsable de seleccionar la implementación apropiada cuando un cliente hace una invocación.

Antes de que un cliente pueda realizar peticiones sobre un objeto debe conocer los tipos de operaciones que soporta. La interfaz de un objeto especifica las operaciones y tipos que soporta un objeto y, por tanto, define las peticiones que pueden hacerse a ese objeto. IDL es un lenguaje declarativo, forzando de esta manera la definición de las interfaces de forma separada a la de sus implementaciones. Esto permite que aunque los objetos sean construidos utilizando distintos lenguajes de programación, puedan comunicarse. También se oculta la implementación del objeto a los posibles clientes.

Al aislar la interfaz de un objeto de su implementación, se permite ganar portabilidad e interoperabilidad. Es posible crear una nueva aplicación o utilizar una ya existente a la que se añade un envoltorio (*wrapper*) que le permite ser accedida mediante CORBA simplemente adaptándole una interfaz IDL.

Las definiciones en IDL son compiladas en un lenguaje de implementación particular por un compilador IDL. El compilador traslada las definiciones independientes del lenguaje en definiciones específicas de un lenguaje de programación. Esas definiciones específicas son utilizadas por el desarrollador para proporcionar funcionalidad a la aplicación e interactuar con el ORB. Los algoritmos de traslación para varios lenguajes de implementación están especificados por CORBA y se denominan *mappings* del lenguaje.

Las definiciones en IDL alcanzan a las interfaces de los objetos, las operaciones contenidas por esas interfaces y las excepciones que pueden ser lanzadas por esas operaciones. Una gran parte de IDL está relacionado con la definición de tipos de datos (siendo IDL un lenguaje fuertemente tipado), debido a que el cliente y el servidor sólo pueden intercambiar datos cuyos tipos hayan sido definidos previamente en IDL.

En la descripción IDL, el programador deberá plasmar todos los elementos del objeto que serán accesibles a los clientes (como un fichero de cabecera en C++). La sintaxis del lenguaje IDL es similar a la de C y proporciona un conjunto de tipos similares a los de otros lenguajes de programación.

Tipos de datos simples

Su significado es similar al que tienen en lenguajes de programación C/C++ o Java. Llama la atención que no existe tipo `int` como ocurre en otros lenguajes de programación. Una variable de tipo `any` permite almacenar valores de cualquier tipo CORBA IDL, incluyendo tipos definidos por el usuario... Realmente hay pocas ocasiones donde el tipo `any` sea necesario, aunque podría ser interesante cuando no se conoce en tiempo de compilación qué tipo de CORBA-IDL se va a necesitar.

La Tabla 2 recoge los diferentes tipos de datos simples soportados en CORBA.

void	boolean	char	wchar
unsigned long	short	unsigned short	long
double	float	enum	any
string	octet		

Tabla 2. Tipos de datos simples en CORBA

Tipos de datos estructurados

CORBA presenta dos tipos de datos estructurados: `struct` y `union`.

El tipo `struct` es similar al definido en C++. Pueden contener uno o varios miembros de tipo arbitrario, incluyendo tipos complejos declarados por el usuario.

```
struct Dia{
    short hora;
    short minuto;
    short segundo;
};
```

Las uniones IDL tienen una pequeña diferencia con las que existen en C, en IDL son discriminadas, es decir, tienen un campo que en cada momento indica qué elemento de los que contiene está en uso. También soportan opcionalmente un caso por defecto (`default:`).

```
union U swicht(boolean) {
  case FALSE: long dato;
  case TRUE : string palabra;
};
```

Sólo un miembro de la unión está activo al mismo tiempo. De todos modos, IDL añade un discriminador que permite saber que miembro está activo en un momento dado. En este ejemplo, `dato` es activo cuando el valor del discriminador es `FALSE` y `palabra` estará activo cuando su valor sea `TRUE`.

Tipos de datos contenedores

CORBA tiene en las matrices y en las secuencias los tipos contenedores elementales, estando representados por las palabras clave `array` y `sequence` respectivamente.

Los `arrays` son similares a los de C/C++, pero deben ser declarados como tipos definidos por el usuario (`typedef string estacion[4];`).

Las secuencias son vectores de longitud variable. Pueden contener elementos tanto definidos por el usuario como predefinidos, y pueden tener longitud limitada o ilimitada.

```
typedef sequense<estacion> estaciones;     //no acotada
typedef secuence<long,100> numeros;        //acotada
```

Excepciones

Cuando una operación genera una excepción durante su ejecución, el ORB será el responsable de notificar al cliente de la situación que acaba de producirse. En CORBA IDL existen dos tipos de excepciones: de usuario y las que la norma CORBA proporciona de forma predefinida.

Las excepciones de usuario son aquéllas que el programador declara al definir la interfaz de un objeto. La declaración se realiza a través de la palabra `exception`. Para indicar las diferentes excepciones que puede levantar un método se utiliza la palabra reservada `raises` y entre paréntesis el nombre de las excepciones separadas por comas.

```
//IDL
```

```
exception   localizaciondesconocida{ string   localización};
string   dar ( string libro )
     raises (localizaciondesconocida);
```

Métodos, atributos y módulos

Para generar un fichero IDL se deben que utilizar métodos, parámetros, interfaces y módulos.

Los métodos no son más que las funciones que pertenecen a la clase que se desea generar, teniendo en cuenta que sólo se declaran aquéllos que son públicos, es decir, los que pueden ser invocados por el cliente. La sintaxis es:

```
<tipo_retorno> <nombre> (<parámetros>);
```

IDL no permite la sobrecarga de funciones. Todos los parámetros han de tener en su declaración un calificador que indique el sentido que tendrá la información que va a pasar a través de ellos. Las palabras reservadas son `in` (entrada), `out` (salida) e `inout` (entrada-salida). Es obligatorio poner el nombre de los parámetros en la declaración, cosa que no es necesaria en C++.

```
void enviar(in string palabra);
long dar(out string mensaje);
```

Un atributo es una variable de un objeto para la cual se desean crear operaciones de acceso, que pueden ser de lectura y escritura o sólo de lectura (funciones `set` y `get`). La declaración de un atributo se hace con la palabra `attribute`. Una vez compilada la interfaz se generan en el lenguaje de implementación dos métodos, uno de lectura y otro de escritura o un solo método de lectura si el acceso es parcial.

```
attribute float radio;
```

La declaración anterior generaría dos métodos, por ejemplo, en C++:

```
float get_radio() {return radio;}
void  set_radio(float valor){radio=valor;}
```

Una interfaz es un conjunto de métodos que un cliente puede invocar sobre un conjunto de objetos del mismo tipo. Es la definición de una clase a la que le falta la sección de implementación.

Los módulos añaden un nivel de jerarquía en el espacio de nombrado del IDL, incluyen una o más interfaces.

Posteriormente se verá un ejemplo completo que mostrará el uso de este lenguaje.

En IDL está definida la herencia de interfaces. Las interfaces derivadas heredan operaciones y tipos definidos en las interfaces base. Exhibe las siguientes características:

- Todas las interfaces base son `public virtual`.
- Todas las operaciones son virtuales.
- Las operaciones no pueden ser redeclaradas en las interfaces derivadas.
- No hay herencia de implementación.

OMG IDL tiene un caso especial de herencia de interfaces, todas las interfaces derivan implícitamente de la interfaz `Object` definida en el `module` CORBA.

Es fundamental mantener el lenguaje IDL tan simple como sea posible. Esto significa que sólo contiene tipos que puedan trasladarse a los lenguajes de programación de uso más común. Debido a la heterogeneidad de los sistemas de objetos distribuidos, la simplicidad de IDL es crítica para el éxito de CORBA como una tecnología de integración. Los compiladores de IDL

trasladan las construcciones en IDL al lenguaje que se utilice para implementar las operaciones definidas en la interfaz IDL.

En la Figura 11 se resume el proceso de creación y utilización de los ficheros IDL en una aplicación CORBA.

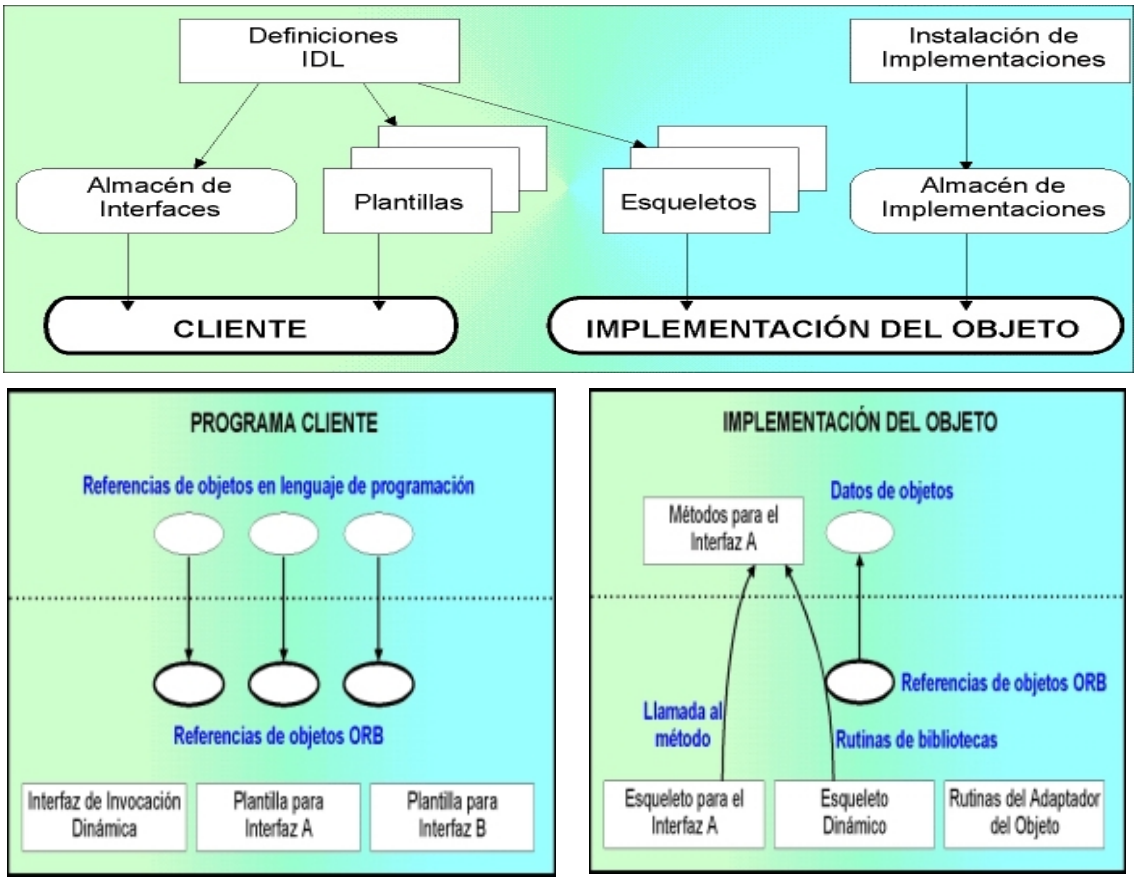

Figura 11. Parte cliente y parte servidor de una aplicación

7.2.1 Lenguajes de *mapping*

Se ha visto que IDL es un lenguaje declarativo, no de programación. Por tanto, se necesita otro lenguaje para implementar las aplicaciones distribuidas. En este momento OMG ha estandarizado el *mapping* para los lenguajes C, C++, Smalltalk, Cobol, Ada 95 y Java. Se espera que próximamente se amplíe a algún lenguaje de *script*. En el presente apartado se va a considerar el *mapping* para C++.

El *mapping* de IDL a C++ debe cumplir unos requisitos:

- Debe ser intuitivo y fácil de usar.
- Debe preservar usos habituales de C++ y ser tan parecido a él como sea posible.
- Debe tener tipos seguros.
- Debe utilizar de forma eficiente la memoria y los ciclos de la CPU.
- Debe trabajar en arquitecturas con memoria segmentada.
- Debe funcionar en entornos multihilo.

- El *mapping* debe preservar la transparencia en la localización del código de clientes y servidores.

El *mapping* a C++ es amplio y complejo, pero es consistente. Esto quiere decir que, por ejemplo, una vez entendido el manejo de memoria de memoria de los tipos `string`, ya se conocen las reglas para manejar otros tipos de longitud variable. Además, el *mapping* es seguro para los tipos, por lo que no es necesario realizar *casting* y hay muchos errores que se resuelven en tiempo de compilación.

El *mapping* para los tipos básicos se recoge en la Tabla 3. Su significado es similar al que tienen en C++, salvo en algunos casos que se comentan a continuación.

OMG IDL Type	C++ Mapping Type
`long, short`	`long, short`
`float, double`	`float, double`
`enum`	`enum`
`char`	`char`
`boolean`	`bool`
`octect`	`unsigned char`
`any`	`Any`
`struct`	`struct`
`union`	`class`
`string`	`char*`
`wstring`	`wchar_t*`
`sequence`	`class`
`object reference`	`pointer ó object`
`interface`	`class`

Tabla 3. *Mapping* a C++ de los tipos básicos de OMG IDL

Mapping de strings

Las cadenas (`strings`) en OMG IDL pueden ser limitados o ilimitados, ambos se corresponden con el tipo de C++ `char*`. El uso de los operadores `new` y `delete` no es portable, hay que utilizar funciones de la clase definidas por CORBA.

```
static char * string_alloc(Ulong len);  //localizar memoria
static char * string_dup(const char *); //hacer una copia
static void string_free(char *);        //liberar memoria
```

Tipos de longitud variable y tipos _var

Los tipos de longitud variable tienen requisitos de equivalencia especiales. Esto es debido a que no se conocen sus tamaños en tiempo de compilación y, por tanto, la reserva de memoria debe hacerse en tiempo de ejecución. Esto da a lugar a problemas relacionados con el manejo de memoria. El programador puede ser el responsable del manejo de la memoria, pero también puede hacer uso de un conjunto de clases conocidas como tipos `_var`. Si utiliza esos tipos, el programador no deberá preocuparse del manejo de la memoria dinámica.

Para cada tipo de longitud variable en IDL el compilador genera un par de tipos en C++. Un tipo está relacionado con el *mapping* de bajo nivel, con el que el programador es responsable de la memoria dinámica, y el otro tipo estará relacionado con una clase `_var`, como puede apreciarse en la Tabla 4.

`string`	`char *`	`CORBA::String_var`
`any`	`CORBA::Any`	`CORBA::Any_var`
`interface Hola`	`Hola_ptr`	`class Hola_var`
`struct Hola`	`struct Hola`	`class Hola_var`
`union Hola`	`class Hola`	`class Hola_var`
`typedef sequence<X> Hola`	`class Hola`	`class Hola_var`
`typedef X Hola[10];`	`typedef X Hola[10]`	`class Hola_var`

Tabla 4. Equivalencia en C++ con los tipos de longitud variable

Mapping de referencias a objetos

Además de las clases del *stub* y del *skeleton*, el *mapping* a C++ de una interfaz genera dos tipos de referencias a objetos, la referencia `nombre_ptr` y la referencia `nombre_var`. Ambas permiten trabajar con una instancia de un objeto, pero se diferencian en la forma de gestionar la memoria. Mientras que `nombre_ptr` deja toda la gestión en manos del programador, `nombre_var` gestiona la memoria automáticamente controlando la destrucción y el correspondiente liberado de memoria.

Mapping de estructuras

Los tipos `struct` de IDL son traducidos a estructuras (`struct`) de C++, y también a objetos de clase `_var`.

El *mapping* de tipos de longitud variable, como parámetros de salida o de retorno, se realiza con un puntero asociado a la clase o al `array`. Para hacer el *mapping* de este puntero la especificación IDL/C++ define una nueva clase que automáticamente borra el puntero cuando una instancia es destruida o reasignada, esta es la clase `_var`.

Los tipos de longitud fija son traducidos a sus correspondientes tipos en C++ con la excepción de las cadenas y las referencias a objetos que se traducen a su correspondiente tipo de clase `_var`.

Para permitir el acceso a un campo e iniciar las estructuras en C++ éstas no podrán tener constructores, destructores u operadores de asignación definidos por el usuario, se usarán los métodos definidos por defecto.

La asignación de cadenas y miembros que hacen referencia a objetos pertenecientes a clases `_var` se realiza copiando el dato, debido a que la asignación de estos miembros a punteros no da como resultado la copia del dato ni tampoco un puntero a la dirección que contiene ese dato.

El siguiente ejemplo muestra se accede a los miembros y el manejo de memoria cuando se utilizan estructuras de longitud variable; obsérvese como el miembro de tipo `string` y el miembro de tipo `interface` son traducidos a un objeto de clase `_var`.

```
//IDL
interface Interface_ej;
struct Struct_ej{
  string name;
  Interface_ej Interface_member;
};
//C++
struct Struct_ej{
  CORBA::string_var name;
  Interface_ej_var Interface_member;
};
```

```
...
class Struct_ej_var{
...
};
```

El siguiente ejemplo usa las estructuras definidas anteriormente para mostrar las diferentes formas de gestionar la memoria que existen. Obsérvese la diversidad de formas de acceder a los miembros de las estructuras dependiendo de si éstos pertenecen a un `struct` o a un `struct_var`. Se muestra, asimismo, como la asignación de un `const char*` da como resultado la liberación de la memoria antigua y la copia del dato; de manera similar la asignación entre una cadena o un miembro que hace referencia a un objeto y su correspondiente clase `_var` da como resultado la liberación del dato de la clase `_var` y la copia del nuevo dato. Por último, se muestra que cuando la asignación se realiza desde un puntero a un miembro de un `struct`, la memoria será liberada, pero el dato no será copiado a no ser que el puntero sea declarado como `const`.

```
//C++
Struct_ej struct1;
Struc_ej_var struct2=new Struct_ej;
Char *no_const;
String_var string_var;
const char *const_a;
const char *const_b=”string 1”;
const char *const_c=”string 2”;

struct1.name=const_b;
struct2->name=const_c;
```

Debido a que tanto `const_b` como `const_c` son de tipo `const`, el almacenamiento en el campo `name` de la estructura es liberado y el nuevo valor es copiado.

Cuando se trata de punteros, los punteros son asignados y el área de memoria no es liberada ni copiada.

```
no_const =struct1.name;
const_a=struct2->name;
```

En la primera línea del ejemplo siguiente, el área de memoria de `struct1.name` es liberada, pero el dato no es copiado sino que solamente es asignado un puntero ya que el valor asignado es de tipo `no_const char*`. En el segundo caso, el área de memoria utilizada es también liberada pero el valor es copiado ya que es de tipo `const char*`.

```
struct1.name=no_const;
struct1.name=const_b;
```

En los casos siguientes el área de memoria es liberada y copiada, en el primer caso un miembro es asociado a otro miembro, en los restantes se hacen asignaciones a partir de un `string_var` y de un `string_var`.

```
struct2->name=struct1.name;
struct1.name=string_var;
string_var=struct2->name;
```

Mapping de uniones

Las uniones de IDL se traducen como clases de C++. Las funciones-acceso son definidas para ajustar u obtener el valor de los datos miembro.

Cuando una función-acceso se utiliza para iniciar el dato se hace una copia, aunque la memoria anteriormente asociada al dato no es liberada en ningún caso.

```
//IDL
interface ej_objeto;
struct struct_ej{
   long miembro_long;
};
typedef string string_array[10];
union union_ej switch(long){
   case 1:
         long x;
   case 2:
         string y;
   case 3:
         struct_ej z;
   case 4:
         string_array lista;
   default:
         ej_objeto obj;
};
```

***Mapping* de arrays**

IDL traslada sus `arrays` a `arrays` C++ del tipo correspondiente. Si los elementos son cadenas se trasladan a `string_mgr`. Este *mapping* proporciona un tipo denominado `array_slice`, que es un `array` al que se le ha quitado la primera dimensión. También genera funciones de gestión dinámica de la memoria: `alloc, dup, copy, free`.

Así, por ejemplo, la declaración IDL: `typedef float cont[4];` equivaldría en C++ a:

```
typedef CORBA::Float cont[4];
typedef CORBA::Float cont_slice;
cont_slice * cont_alloc();
cont_slice * cont_dup(const con_slice *);
void cont_free(cont_slice *);
void cont_copy(cont_slice * dest,const cont_slice *ori);
```

***Mapping* de interfaces**

El *mapping* de interfaces OMG IDL a C++ genera varias clases con distintas funciones. El número de clases varía según el compilador IDL que se utilice, aunque al menos tiene que crear dos: una que represente al *skeleton* y otra al *stub*, que utilizarán el servidor y el cliente respectivamente. Por ejemplo, el compilador IDL del ORB MICO [MICO, 2001] genera cuatro clases.

```
// IDL
interface Libro
{...........};

// C++ generado automáticamente
class Libro : virtual public CORBA::Object
{...........};
class Libro_stub: virtual public Libro
{.......};
class Libro_stub_clp : virtual public Libro_stub,
virtual public PortableServer::StubBase
{........};
class POA_Libro :
virtual public PortableServer::StaticImplementation
{........};
```

Se ha generado una clase base virtual pura en la que están todas las definiciones de las operaciones que se especificaron en la interfaz IDL; dos clases para el *stub*, que son las que de forma transparente utilizan los clientes para realizar sus invocaciones; y una clase *skeleton* que será utilizada por el ORB como unión con la implementación del objeto.

***Mapping* de constantes**

Las constantes de IDL son traducidas directamente a una declaración de constantes de C++. El siguiente ejemplo ilustra una simple declaración IDL y el código C++ generado.

```
//IDL
const long Long_ej=1966;
interface Interface_ej
{
  const string String_ej=”Nombre Aquí”;
  const boolean Boolean_Ej=TRUE;
};

//C++
const CORBA::long Long_ej=1966;
class interface_ej : public virtual CORBA::object {
...
public:
static const char *string_ej;           //Nombre Aquí
static const CORBA::boolean Boolean_ej; //1
...
};
```

Si una constante es declarada y después es utilizada en la declaración de un `array` o de un tipo estructurado, el compilador reemplaza el uso del nombre de la constante por el valor real de la constante. El ejemplo siguiente muestra cómo se realiza esta operación:

```
//IDL
interface Interface_ej {
const long Array_index =10;
typedef long long_Array[Array_index];
};
//C++
class Interface_ej {
public:
  static const CORBA::long Array_index;
  typedef CORBA::long long_Array[10];
};
```

***Mapping* de enumeraciones**

Las enumeraciones definidas en IDL son traducidas directamente a enumeraciones de C++. Por ejemplo:

```
//IDL
enum enum_ej {fuego,tierra,mar,aire};
//C++
enum enum_ej {fuego,tierra,mar,aire};
```

***Mapping* del tipo any**

El tipo `any` es un tipo autodefinido que puede contener valores de cualquier tipo IDL (incluido el propio tipo `any`).

El mapeo de IDL a C++ del tipo `any` debe cumplir dos requisitos:

- Manejar los tipos de C++ de una manera segura.
- Manejar valores cuyo tipo es desconocido en tiempo de compilación.

En otras palabras, debe manejar las conversiones requeridas para insertar y extraer de un tipo `any` y debe acomodarse a las peticiones y respuestas que contengan un `any` que mantenga los datos de un tipo que es desconocido para el cliente que hace la invocación en tiempo de compilación.

El manejo de tipos de C++ de una manera segura requiere que el *mapping* a C++ proporcione funciones sobrecargadas para cada tipo de IDL. Para aquellos tipos de IDL que no producen distintos tipos de C++ (`boolean`, `octet`, `char` y `wchar`) deben proporcionarse funciones aparte para distinguir uno de otro.

Inserción en un any

La inserción en un tipo `any` se consigue con el operador sobrecargado `<<=`. Para tipos pequeños incluyendo `bounded string`, enumeraciones y referencias a objeto, el operador copia el dato.

```
//C++
void operator<<=(Any&, Data_type&); //copia el dato
void operator<<=(Any&, Data_type*); //no copia el dato
```

Obsérvese que con la forma del operador que no copia el dato cuando éste es insertado, el dato es consumido y no puede volver a ser accedido una vez que se ha insertado en el tipo `any`. Para la inserción de tipos IDL sin una equivalencia distinta en C++ (como `octet`, `char`, `wchar` o `bounded string`) se proporcionan algunos tipos de ayuda.

Inserción de arrays en el tipo any

La inserción de `arrays` en el tipo `any` se consigue con el tipo `array_forany` que se define para cada `array` definido en IDL. Una inserción que copia es la que se utiliza por defecto. Dependiendo de la implementación suministrada por el proveedor ORB, el usuario será capaz de configurar un *flag* para evitar la copia mencionada en el constructor del `array_forany`, si dicho *flag* contiene el valor `TRUE`, el valor insertado será consumido por el tipo `any`.

```
//IDL
typedef long long_array[10][20];
//C++
typedef CORBA::long long_array[10][20];
class long_array_forany{
...
};
...
operator<<=(CORBA::Any& a, const long_array_forany&  _val);
```

Los tipos generados arriba podrían usarse en una aplicación como sigue:

```
//C++ app
long_array array1;
//...iniciar array
Any any_data;
Any_data <<= long_array_forany (array1);
```

Extracción del tipo any

Para extraer un valor del tipo `any` se utiliza el operador sobrecargado `>>=` propio de cada tipo IDL. La función devuelve un `boolean` indicando si el tipo extraído es del mismo tipo al cual

va a ser asignado o no, en caso de éxito el valor será copiado o el puntero será asignado (dependiendo del tipo).

```
//C++
Any valor_any;
//Se le asigna un valor cualquiera
...
if(valor_any>>=long_value)
{
//usar el valor
...
}
else if(valor_any>>=struct_ptr){
      //usar el valor
      ...
      }
      else if(valor_any>>=array_forany_ref)
                  {
                  //usar el valor
                  ...
                  }
```

La clase Any

En el constructor por defecto de la clase Any su Typecode se asigna a tk_null, es decir, el tipo contenido por la clase Any es indefinido. El constructor de copia, copia el tipo any que se le pasa como parámetro. El constructor final duplica una referencia al seudo-objeto y asume la propiedad del almacenamiento del parámetro si el *flag* de liberación está a TRUE, si el *flag* está a FALSE el que invoca mantiene la propiedad de su almacenamiento.

```
//C++
CORBA_Any();
CORBA_Any(const CORBA_Any&);
CORBA_Any(CORBA_Typecode_ptr tc, void *value,
      CORBA::Boolean release=0);
```

También se define una clase Any_var, que es muy útil para la correcta gestión de memoria por el ORB.

La clase Any es una construcción muy útil para el manejo de tipos genéricos que pueden ser desconocidos en tiempo de compilación. Crear un objeto Any es tan simple como declarar un tipo de dato estándar, aunque puede construirse también duplicando otro Any como se muestra en los constructores de Any. El ejemplo siguiente es una muestra de la sencillez de creación de un tipo Any.

```
//C++
struct_ej struct_1={10,20.0};
struct_ej struct_2;
//Se crea un any y se le introduce una estructura
any_struct =new CORBA::Any();
*any_struct<<=struct_1;
//Ahora se extrae la estructura
If(*any_struct >>=struct_2) {
      //usamos el valor
...
}
```

Mapping de parámetros

Según el sentido con el que se hayan definido los parámetros en la interfaz, CORBA define *mappings* diferentes. En la Tabla 5 se muestran los *mappings* más significativos, teniendo en cuenta que los que no estén presentes tendrán una conversión similar.

OMG-IDL	in	out	inout	return
long	CORBA::Long	CORBA::Long&	CORBA::Long&	CORBA::Long
char	CORBA::Char	CORBA::Char&	CORBA::Char&	CORBA::Char
Referencia a objeto	objref_ptr	objref_ptr&	objref_ptr&	objref_ptr
struct (tamaño fijo)	const struct&	struct&	struct&	struct
struct (tamaño variable)	const struct&	struct&	struct*&	struct
any	Const CORBA::Any	CORBA::Any&	CORBA::Any*&	CORBA::Any*
string	const char*	char*&	char*&	char*
array (tamaño fijo)	const array	array	Array	array_slice *
array (tamaño variable)	const array	array	Array	array_slice *

Tabla 5. *Mapping* de parámetros

Con los lenguajes de *mapping* se implementan las abstracciones y conceptos definidos en CORBA, así que su importancia no debe ser despreciada, siendo su especificación revisada periódicamente para añadir nuevas facilidades y extensiones.

En la Tabla 6 se recogen las equivalencias básicas entre OMG IDL y Java. Para una mayor información del *mapping* OMG IDL – Java se recomienda la consulta de [Sevilla, 1998].

Tipo IDL	Tipo Java	Excepciones
bolean	bolean	
char	char	CORBA::DATA_CONVERSION
wchar	char	
octet	byte	
string	java.lang.String	CORBA::MARSHALL CORBA::DATA_CONVERSION
wstring	java.lang.String	CORBA::MARSHALL
short	short	
unsigned short	short	
long	int	
unsigned long	int	
long long	long	
unsigned long long	long	
float	float	
double	double	

Tabla 6. Equivalencia de tipos entre OMG IDL y Java

7.3. Direccionamiento e invocación

Como ya se ha indicado para acceder a un objeto remoto un cliente necesita conocer la dirección del servidor. Esta información está contenida en las referencias a objeto. En las primeras revisiones la implementación de estas referencias quedaba en manos del programador, actualmente están estandarizadas para facilitar la comunicación entre diferentes ORBs. Las referencias a objeto se denominan IOR (*Interoperable Object Reference*) en la terminología CORBA, y son opacas al usuario, ninguna información se puede extraer de ellas.

La vida de una referencia a objeto es independiente de la vida de su servidor. Un cliente no sabe si el servidor está activado o no en el momento de realizar la petición. Aunque pueda parecer que este desconocimiento del estado del servidor es perjudicial, permite el restablecimiento transparente de un servidor sin que el cliente se dé cuenta. Así, un servidor puede estar fuera de servicio temporalmente y más tarde ser restablecido.

Formalmente una IOR es una tupla I = (t, {pi}), con un tipo IDL declarado t (opcional) y un conjunto de perfiles (*profiles*) pi. Estos perfiles pueden tener diferentes niveles, por ejemplo, el más utilizado es el IIOP (*Internet Inter-ORB Protocol*) que puede ser descompuesto en cinco componentes: un número de la revisión IIOP, el nombre del *host* (o la dirección IP en la notación punto decimal), el número de puerto TCP en el que esté escuchando el servidor, una clave de objeto específica del servidor y etiquetas opcionales.

Debido a la opacidad de las referencias a objetos, el cliente tiene un problema al iniciarse porque debe adquirir una referencia a un objeto, pero no puede crearla. Una solución es el intercambio de una referencia a objeto convertida en una cadena de caracteres. El sirviente (instancia de una clase implementada) genera una referencia a objeto a sí mismo y la escribe como cadena de caracteres en un fichero. El cliente lee el fichero y utiliza una transformación inversa para convertir la cadena a una referencia a objeto. Esta forma de iniciar el cliente es poco efectiva si existen muchas referencias a objetos que deben ser intercambiadas. El servicio de nombres (*naming service*), que es un servicio de objetos, ha sido definido para reducir este problema. Los servidores pueden registrar sus objetos con el servicio de nombres, donde pueden ser encontrados más fácilmente por los clientes. Aún así el problema no está solucionado porque los clientes y servidores siguen necesitando una referencia a objeto al servicio de nombres (considerado como objeto).

Una vez que se conocen la interfaz y la dirección de un objeto, los métodos pueden ser invocados y se ejecutarán remotamente. Como se ha visto esto se hace a través del *mapping* del lenguaje. En C++, el compilador IDL genera *stubs* para cada interfaz, que no son más que clases de C++ que exportan los métodos declarados en el fichero IDL. Un objeto *stub* encapsula una referencia a objeto específica. Cuando un cliente adquiere una referencia a objeto, el ORB crea un objeto *stub*. El código generado en el *stub* se encarga de convertir el método invocado en una petición y se la pasa al ORB, que descifra la referencia a objeto y contacta con el servidor. Esto es transparente para el usuario, una invocación remota es similar a una invocación local. Esta operación se plasma gráficamente en la Figura 12.

En la parte del servidor el compilador IDL genera un *skeleton*. El programador deriva sus clases de ese esqueleto e implementa los métodos. El esqueleto generado recibe las peticiones del ORB y realiza la llamada sobre el objeto invocado (ver Figura 13). El ORB está implementado tanto en el cliente como en el servidor, pero estos detalles no son importantes para el usuario.

Los *stubs* y los *skeletons* usan información estática, los nombres de los métodos y de los parámetros se deciden en tiempo de compilación. Como alternativa se tiene una invocación dinámica: Interfaz de Invocación Dinámica (DII) en la parte de cliente y la Interfaz de Esqueleto Dinámica (DSI) en la parte del servidor. Mediante el uso de DII un cliente puede realizar una petición utilizando información obtenida en tiempo de ejecución proporcionada por un Almacén de Interfaces (*Interface Repository*). En la parte del servidor, un sirviente puede utilizar el DSI para responder las peticiones. Utilizar el DII en la parte del cliente no implica utilizar el DSI en la parte del servidor ni viceversa. El DII permite trabajar asíncronamente. Las peticiones realizadas a través de la interfaz estática siempre son síncronas, el cliente se bloquea hasta que la petición es procesada y respondida por el servidor. El código generado estáticamente es más rápido construyendo peticiones que el código que utiliza interfaces dinámicas, ya que este último debe utilizar interfaces genéricas para encapsular cada parámetro en un valor de tipo Any.

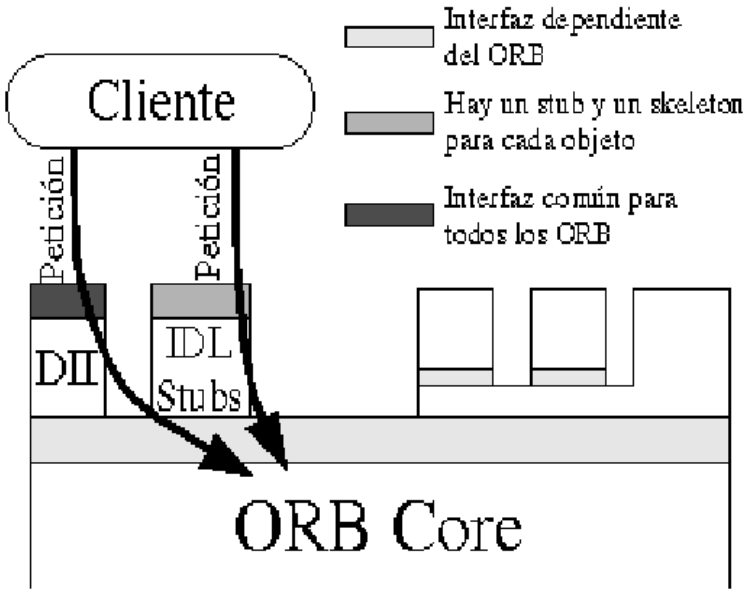

Figura 12. Invocación remota desde el cliente

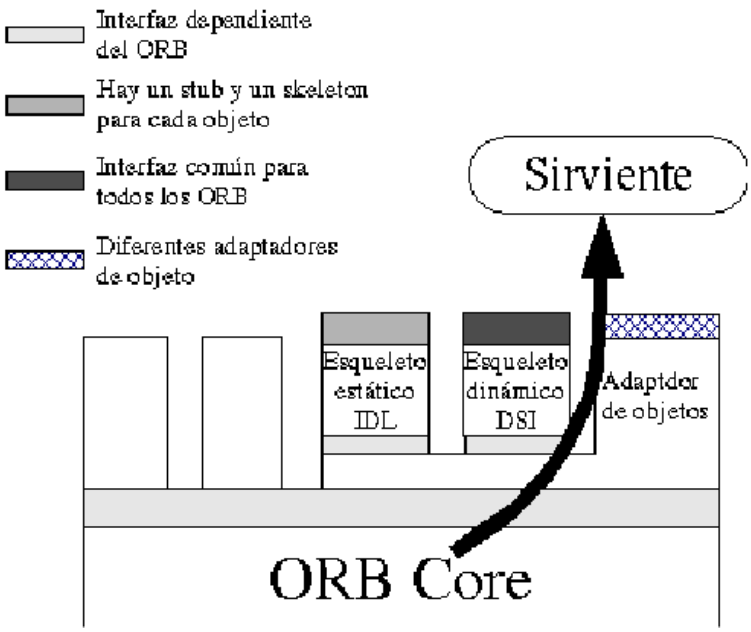

Figura 13. Recepción de la petición en el servidor

7.4. Adaptadores de objetos

Se ha visto que utilizar servicios CORBA es simple: después de recibir y trasladar el fichero IDL al lenguaje de implementación elegido, lo único que se necesita es una referencia a objeto. A partir de aquí los métodos del objeto remoto serán llamados a través de un objeto *stub* como si fueran locales. En el lado del servidor se hace necesario algo más, así OMG llegó a la conclusión de que los requisitos para la implementación dependían demasiado del sistema operativo, del lenguaje de programación y del entorno. Para permitir diferentes posibilidades se desacopló a los sirvientes del procesamiento normal del ORB introduciendo los adaptadores de objetos. Un adaptador de objeto se sitúa como mediador entre el ORB y las implementaciones de los objetos CORBA, consiguiendo que las peticiones que llegan a través del ORB sean procesadas por la implementación del objeto.

Cuando se realiza una invocación el ORB de la parte del cliente es responsable de interpretar los métodos de los objetos, de localizar el servidor donde se encuentra el objeto y de enviar una petición a ese servidor. En la parte del servidor, la petición es recibida por el ORB donde se realizan tres pasos:

1. El ORB debe encontrar el adaptador de objetos en el que está implementado el objeto y pasar la petición a ese adaptador.
2. El adaptador de objetos debe encontrar el sirviente que implementa el objeto.
3. Si el sirviente utiliza un esqueleto estático, la petición es interpretada por el código generado por IDL y el método deseado es invocado.

Pero antes de que todo esto se realice el adaptador de objetos tiene que conocer al sirviente. Entre las responsabilidades de un adaptador de objetos están:

- *Registro de los objetos*. Los adaptadores de objetos deben proporcionar funciones para registrar implementaciones para los objetos CORBA.
- *Generación de referencias a objetos*. Deben generar referencias para los objetos que tengan registrados.
- *Activación de procesos servidores*. Deben activar los objetos registrados.
- *Multiplexación de peticiones a los objetos registrados*. Deben asegurar que todas las peticiones sean recibidas por los objetos, aunque tengan múltiples conexiones, sin que ninguna se bloquee indefinidamente.
- *Gestión de las invocaciones*. Deben despachar las peticiones de objetos registrados.

Los adaptadores de objetos representan otro de los esfuerzos por mantener el ORB tan simple como sea posible. Como se ha dicho sin adaptadores de objetos la capacidad de CORBA para soportar diversos estilos de implementaciones de objetos estaría comprometida debido a que las implementaciones de los objetos se conectarían directamente al ORB para recibir peticiones. Así, si el estándar cubriera un pequeño número de interfaces para objetos, significaría que se soportarían muy pocas implementaciones de objetos. En el otro extremo, estandarizar muchas interfaces de objetos aumentaría innecesariamente el tamaño y la complejidad del ORB. CORBA permite múltiples adaptadores de objetos, son diferentes para cada lenguaje de implementación, aunque sólo proporciona dos: inicialmente OMG propuso BOA (*Basic Object Adapter*), pero debido a sus limitaciones, en CORBA 2.2 se introdujo POA (*Portable Object Adapter*).

7.4.1 BOA (*Basic Object Adapter*)

El adaptador de objetos más sencillo sería una tabla que relacionara claves de objetos (la parte del servidor de una referencia a objeto) a los sirvientes. La activación y desactivación causarían inserciones y eliminaciones de la tabla. El adaptador BOA, que fue el primer adaptador definido por el OMG, es poco más que eso. De hecho ya está fuera de uso, aunque en el presente apartado se resumen sus propiedades.

BOA cubría de forma mínima las funciones del adaptador de objetos, tales como el registro de objetos, la generación de referencias, la activación de implementaciones y la gestión de las peticiones. CORBA distingue entre objetos, unidades que son susceptibles de exportar la funcionalidad, y servidores, que son los procesos que alojan a los objetos.

BOA define tres estados en los que puede estar un objeto durante su vida: inexistente, activo e inactivo. Los procesos de activación y desactivación pueden suceder más de una vez y son transparentes al cliente.

Otro concepto introducido es el de almacén de implementaciones (*Implemetation Repository*), que se corresponde con su equivalente en el lado cliente, el de almacén de interfaces, y que es una base de datos de código para las implementaciones de objetos. Una IOR contiene una dirección a una implementación.

BOA emplea esa idea, es decir, la creación de un objeto se hace asociando una interfaz con una implementación usando el método `create` (proporcionado por la interfaz BOA). En el almacén de implementaciones se guarda información sobre la política de activación porque determina cuando un nuevo servidor debe ser activado para responder a una petición. Considerando las relaciones entre los objetos y los servidores se tienen cuatro políticas de activación para BOA:

1. *Servidor compartido*. Múltiples objetos pueden residir en el mismo servidor. BOA activa el servidor la primera vez que recibe una petición.
2. *Servidor no compartido*. Cada servidor implementa exactamente un objeto. La activación de un objeto causa la iniciación de un nuevo servidor.
3. *Servidor por método*. Un nuevo servidor se inicia cada vez que llega una petición sobre un objeto. El servidor permanece activo sólo durante la ejecución de un método.
4. *Servidor persistente*. La única diferencia con el servidor compartido es que no es BOA quien activa el servidor.

BOA puede ser dividido en dos partes. Una parte debe ser incluida en cada servidor para recibir peticiones y realizar llamadas sobre las implementaciones de objetos. La otra parte tiene que mantener todas las implementaciones en el almacén de implementaciones e iniciar nuevos servidores si es necesario. Esta segunda parte puede ser realizada como un *demonio* o ser integrada en cada cliente.

Sin embargo, BOA tiene varios problemas, siendo los más destacados:

- Los contenidos del almacén de implementaciones no están especificados.
- El nombre de las clases esqueleto no está especificado.
- El registro de los sirvientes no está especificado.
- El procesamiento de peticiones no está especificado.
- No existen interfaces para salvar o restaurar el estado de un objeto.
- La distinción entre estados del objeto (activo e inactivo) es muy simple.

Estos problemas deben ser resueltos por los ORB comerciales, pero al no estar fijados por especificaciones, introducen incompatibilidades, con lo que la portabilidad queda seriamente dañada. Para solucionar este problema OMG estableció las especificaciones para un nuevo adaptador de objetos, POA [OMG, 1998b].

7.4.2 POA (*Portable Object Adaptor*)

Los dos propósitos más importantes en la especificación de este adaptador son:

- *Portabilidad*. Debería ser posible usar código fuente con distintos ORBs comerciales sin cambiar código.
- *Flexibilidad*. Proporcionar herramientas para controlar el ciclo de vida de los sirvientes y la recepción de peticiones.

POA trata con tres entidades clave:

- *Referencias a objeto*. POA es responsable de crearlas.
- *Identificadores de objeto*. Dentro del alcance de un adaptador POA cada objeto es identificado por una secuencia de `octet` de forma única. Cuando POA crea un nuevo objeto introduce su identificador en la clave del objeto dentro de la referencia a objeto. Así, cuando un cliente invoca una petición usando una referencia a objeto, el ORB del cliente utiliza dicha referencia para determinar los puntos finales de la comunicación donde el objeto deseado puede ser encontrado, y envía allí la petición. El ORB del cliente envía la clave del objeto con la petición para identificar al objeto en el servidor. EL ORB servidor utiliza esta clave del objeto, que previamente había creado como parte de la referencia a objeto del objeto requerido, para determinar que POA del servidor controla dicho objeto. Una vez hecho esto, redirecciona la petición a ese adaptador POA. Finalmente, POA extrae el identificador del objeto de la clave del objeto, busca el sirviente que encarna el objeto buscado, y le pasa la petición a él.
- *Sirvientes*. Una aplicación puede crear y registrar sirvientes directamente con un adaptador POA para encarnar objetos. Además, puede proporcionar manejadores de sirvientes para POA que puedan crear sirvientes cuando sea necesario satisfacer una petición de un cliente.

Se pueden tener varias instancias de POA organizadas jerárquicamente en un servidor. El ORB crea una instancia de POA raíz y los siguientes POA pueden ser creados como hijos suyos. Cada POA mantiene su propio mapa de objetos activos, una tabla que relaciona los objetos activos con los sirvientes. Los objetos son activados dentro de una instancia de POA y se asocian con él identificados por un único identificador de objeto. La sincronización entre distintos POAs es realizada por los gestores de POA, como se puede ver en la Figura 14.

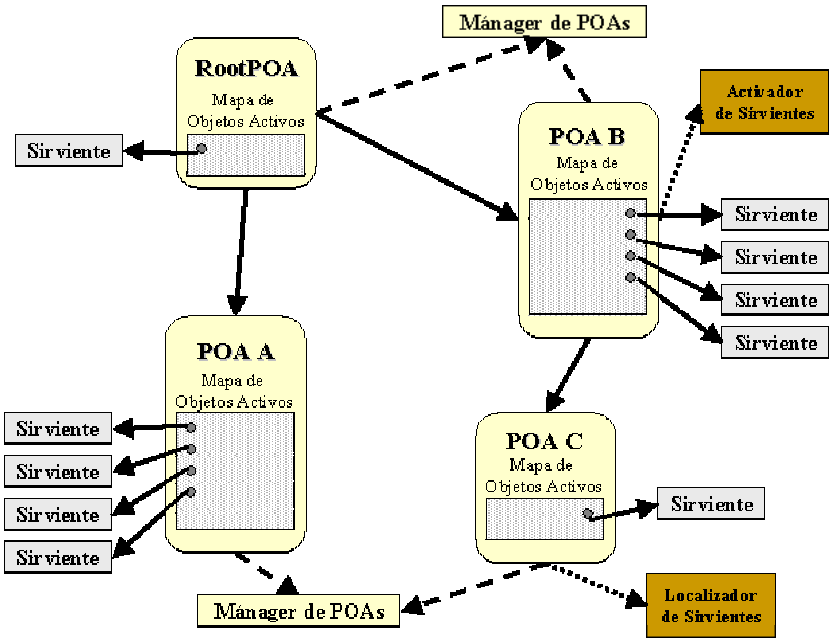

Figura 14. Estructura jerárquica del adaptador de objetos POA

La especificación CORBA ha definido un conjunto de políticas que indican la gestión del adaptador POA sobre sus objetos. El usuario puede definir los valores para estas políticas cuando cree un nuevo POA.

- **Política de hilos**. Es normal que las aplicaciones servidoras usen múltiples hilos para servir múltiples peticiones concurrentemente. Una aplicación puede proporcionar un hilo nuevo para cada nueva petición o bien puede manejar todas

las peticiones para un objeto en un hilo separado. También puede emplear un número fijo de hilos para manejar peticiones, creando colas si todos los hilos están ocupados. El valor `ORB_CTRL_MODEL` permite elegir un modelo multihilo, mientras que el valor `SINGLE_THREAD_MODEL` garantiza que todas las peticiones para todos los objetos en un adaptador POA concreto serán atendidas secuencialmente. El primer modelo permite que múltiples peticiones concurrentes sean procesadas por múltiples hilos. El valor por defecto de esta política es `ORB_CTRL_MODEL`.

- **Política de vida**. CORBA se diferencia de otras plataformas distribuidas en que proporciona una activación de objetos automática y transparente. Si un cliente realiza una petición a un objeto que en ese momento no está activo, el ORB activa un proceso servidor para ese objeto y luego activa al objeto mismo. Dicha activación es transparente para el usuario. Los objetos pueden ser persistentes o transitorios. Los persistentes pueden vivir fuera de su POA y de su servidor. Si se desactiva un servidor con objetos persistentes cuando se active de nuevo el adaptador POA correspondiente le generará idénticas referencias, por lo que para todos los clientes el objeto será el mismo que utilizaron anteriormente. El programador es el encargado de restaurar el estado de los objetos. La vida de los objetos transitorios está acotada por la vida del proceso en el que viven o del adaptador POA que los haya creado. Un objeto transitorio no podrá ser reactivado. La política de vida por defecto cuando se crea un adaptador POA es la de transitorio.

- **Política de identificador único**. Un adaptador POA puede permitir a un sirviente encarnar múltiples objetos CORBA o bien restringe a los sirvientes a encarnar un único objeto. Cuando llega una petición al servidor, POA extrae el identificador de objeto (que se encuentra incluido en la referencia a objeto) y lo usa para buscar dentro de su mapa de objetos activos el sirviente para ese objeto. Cada entrada del mapa de objetos activos consiste en una asociación entre un identificador a objeto y un puntero a un sirviente. Con el valor `UNIQUE_ID` para esta política, POA obliga a que cada identificador de objeto tenga un sirviente diferente. Por el contrario el valor `MULTIPLE_ID` permite que varios identificadores a objeto puedan referirse al mismo sirviente. Por defecto, el valor de esta política es `UNIQUE_ID`.

- **Política de asignación de identificación**. Se ha dicho que POA identifica cada objeto a través de su identificador de objeto. La principal diferencia entre referencias a objeto e identificadores de objeto es que un identificador de objeto no tiene significado fuera del alcance del adaptador POA que maneja ese objeto. Para invocar operaciones los clientes utilizan referencias a objeto y no identificadores de objeto. Como las referencias a objeto son opacas, los clientes no pueden extraer el identificador de objeto de una referencia a objeto, ni pueden crear una referencia a objeto conociendo el identificador de objeto. Dentro del domino de un adaptador POA los identificadores de objeto deben ser únicos. Se puede utilizar esta política para definir si los OID son generados por el adaptador POA o seleccionados por el usuario. El valor por defecto de esta política es `SYSTEM_ID`.

- **Política de procesamiento de peticiones**. Un aspecto importante para tener en cuenta la escalabilidad de una aplicación servidor es controlar las asociaciones entre objetos y sirvientes. Dependiendo del número de objetos que contenga la aplicación, se podría usar un sirviente para cada uno, un único sirviente para todos ellos, proporcionar dinámicamente un sirviente para cada petición, o usar una combinación de esas técnicas. Cuando un objeto es activado la asociación entre el

objeto y el sirviente se almacena en el mapa de objetos activos. Una alternativa es utilizar un administrador de sirvientes, que es un objeto llamado por el adaptador POA cuando recibe una invocación de un objeto que no tiene un sirviente asociado. El administrador puede crear un nuevo sirviente para atender la petición o reutilizar uno creado. Otra alternativa es proporcionar un sirviente por defecto, que encarnará todos los objetos CORBA para un adaptador POA, evitando la necesidad de crear un sirviente para cada objeto. La política de retención por defecto es `USE_ACTIVE_OBJECT_MAP_ONLY`.

- **Política de activación**. Cuando se crea un adaptador POA, en la aplicación se puede especificar que el nuevo POA permita que los objetos POA sean creados y activados implícitamente o especificar que simplemente permite crear objetos CORBA de forma explícita y registrar los sirvientes. La principal razón para controlar si se permite la activación implícita es prevenir la creación accidental de objetos CORBA. La política de activación por defecto al crear un adaptador POA es `IMPLICIT_ACTIVATION`.
- **Política de retención de sirvientes**. Cada vez que un adaptador POA atiende una petición almacena asociaciones entre objetos y sirvientes en su mapa de objetos activos. Si un adaptador POA retiene sus asociaciones entre objetos y sirvientes, cuando llegue una petición, puede utilizar el identificador del objeto como un índice a su mapa de objetos activos para buscar el sirviente que debe procesar dicha petición. Si el adaptador POA no retiene dichas asociaciones debe fiarse de que lo haga la aplicación en su lugar. Esto lo hace utilizando un administrador de sirvientes o un sirviente por defecto. Es importante controlar dicha política cuando se tiene en cuenta el uso de memoria de una aplicación servidor. Si una aplicación tiene muchos objetos, debe evitarse mantener en memoria todas las asociaciones entre identificadores de objeto y sirvientes, y utilizar un administrador de sirvientes. El valor por defecto de esta política es el de `RETAIN`.

7.5. Interoperabilidad en CORBA

El estándar de interoperabilidad de CORBA fue desarrollado para permitir que diferentes ORBs pudieran comunicarse. La interoperabilidad de CORBA ofrece una pasarela de infraestructura que hace que diferentes implementaciones de ORBs sean compatibles. Esta parte del estándar se asienta en la capa de transporte del modelo OSI.

La arquitectura de interoperabilidad entre ORBs está basada en el protocolo GIOP (*General Inter - ORB Protocol*) que especifica la sintaxis de transferencia de un conjunto de mensajes estándar para la comunicación entre ORBs basada en un protocolo de transporte orientado a conexión. La especificación consiste en los siguientes elementos:

- **CDR (*Common Data Representation*)**. Define cómo deben codificarse los datos para su transferencia entre clientes y servidores.
- **Formato de los mensajes GIOP**. Define ocho tipos de mensajes que pueden intercambiar los ORB de la parte del cliente y del servidor. Sólo dos de esos tipos de mensaje son necesarios para realizar la comunicación (`Request` y `Reply`), el resto son mensajes de control.
- **Condiciones de la capa de transporte.** Para poder transmitir mensajes GIOP.
  - Debe ser orientado a conexión.
  - Las conexiones son *full-duplex*.
  - Las conexiones son simétricas.

- Comunica si se produce una pérdida de conexión.

GIOP especifica la mayoría de los detalles de protocolo que son necesarios para que se puedan comunicar los servidores y los clientes. GIOP es un protocolo abstracto y, por tanto, es independiente de un tipo de transporte particular. El IIOP (*Internet Inter.-ORB Protocol*) [OMG, 2001a] es una implementación concreta del protocolo GIOP sobre el protocolo de transporte TCP/IP (Figura 15). Por ello, IIOP necesita especificar cómo las IOR codifican la información del direccionamiento de TCP/IP para que el cliente pueda establecer comunicación con el servidor y enviarle una petición. IIOP es el principal protocolo de interoperabilidad utilizado por CORBA.

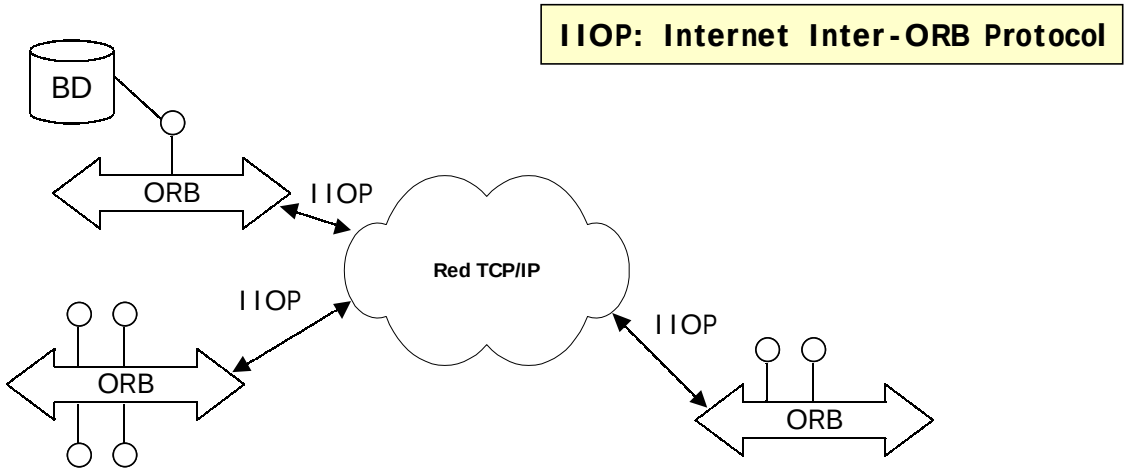

Figura 15. Interoperabilidad de ORBs a través de IIOP

8. Servicios de CORBA

Los servicios de CORBA representan un conjunto de servicios que complementan el funcionamiento básico de los objetos que dan lugar a una aplicación. Son un complemento a la funcionalidad del ORB. El número de servicios se amplía continuamente para añadir nuevas facilidades a los sistemas desarrollados con CORBA. Esto no quiere decir que se encuentren implementados en los ORBs comerciales.

Algunos de los principales servicios son:

- *Life cycle*: define operaciones para crear, copiar, mover y eliminar objetos.
- *Events*: permite registrarse para recibir eventos que pueden ser producidos por otros objetos.
- *Naming*: permite localizar objetos por un nombre.
- *Trader*: permite localizar objetos por sus propiedades.
- *Persistence*: ofrece una interfaz para almacenar objetos.
- *Transactions*: proporciona coordinación transacional.
- *Concurrency*: proporciona un gestor de bloqueos.
- *Externalization*: permite obtener y producir datos como *streams* (flujos).
- *Security*: da soporte a la autenticación, listas de control de acceso, confidencialidad...
- *Time*: permite definir y gestionar eventos temporizados.

- *Properties*: permite asociar propiedades a un objeto.
- *Query*: ofrece una interfaz SQL para realizar operaciones en objetos.
- *Licensing*: ayuda a medir el uso de los objetos.
- *Relationship*: permite crear asociaciones dinámicas entre objetos.
- *Collection*: proporciona las interfaces para crear y manipular colecciones de objetos (listas, conjuntos...)

A continuación se tratarán con más detalle los servicios de nombres [OMG, 2001c] y eventos [OMG, 2001b] por su gran utilidad y estar implementados en la mayoría de los ORBs comerciales.

8.1. Servicio de nombres

Este servicio [OMG, 2001c] es uno de los más básicos ofrecidos por CORBA. Proporciona una equivalencia entre nombres y referencias a objeto, es decir, dado un nombre el servicio devuelve una referencia a objeto asociada a ese nombre. Es similar al *servicio de dominio de nombres* de Internet (DNS – *Domain Name Service*) que traslada los dominios Internet a direcciones IP.

El servicio de Nombres ofrece una serie de ventajas. Una de ellas es que los clientes pueden utilizar nombres significativos para referirse a objetos en vez de tener que tratar con referencias a objeto en forma de cadenas. Además, resuelve el problema que tienen los clientes para acceder a los componentes al iniciar una aplicación.

La misma referencia a objeto puede ser almacenada varias veces con diferentes nombres, pero cada nombre identifica únicamente a una referencia. Un contexto de nombres (*naming context*) es un objeto que almacena relaciones entre referencias y nombres. Es decir, dicho objeto implementa una tabla que traslada nombres a referencias a objeto. Este servicio se puede estructurar como un sistema de ficheros jerárquico en el que los contextos de nombres se comportarían como directorios.

Las definiciones IDL para este servicio se proporcionan en un fichero llamado `CosNaming.idl` que debe existir en los ORBs que implementen dicho servicio. Este fichero contiene varias definiciones de tipo y dos interfaces: `NamingContext` y `BindingIterator`. Dichas interfaces proporcionan todos los servicios necesarios para utilizar este servicio.

A continuación se muestra cómo se puede obtener una referencia a un contexto de nombres. Hay que utilizar el método `resolve_initial_references()` que sirve para iniciar varios servicios de CORBA (incluido POA).

```
//Iniciar ORB
CORBA::ORB_var orb = CORBA::ORB_init(argc,argv);
//conseguir una referencia a un contexto de nombres
CORBA::Object_var obj =
          orb->resolve_initial_references("NameService");
CosNaming::NamingContext_var inc =
        CosNaming::NamingContext::_narrow(obj);
```

Con esto ya se tendría una referencia a objeto para el contexto de nombres inicial. El siguiente paso sería unir un nombre a una referencia a objeto. Esto se realiza con el método `bind()`. Por ejemplo, si se supone que se tiene un objeto de la clase `Caja` y se quiere exportar utilizando el servicio de nombres.

```
//Se supone que ya se tiene una referencia al contexto inicial, inc
//Se crea un objeto de tipo Caja
Caja_impl *registradora = new Caja_impl();
```

```
Caja_var rg = registradora->_this();
//Se crea el nombre Super
CosNaming::Name nombre;
nombre.length(1);
nombre[0].id = CORBA::string_dup("Super");
//Se establece la relación entre nombre y objeto
inc->bind(nombre,rg)
```

En este momento ya se tendría el objeto de tipo `Caja` dado de alta en el servicio de Nombres, listo para ser utilizado por un cliente. La parte del cliente es muy sencilla, para resolver un nombre en una referencia a objeto utiliza el método `resolve()`. Los pasos a seguir son similares a los realizados por el servidor. También debe obtener una referencia a un objeto contexto de nombres.

```
//Se supone que ya se tiene una referencia a un contexto
//de nombres llamada inc
CosNaming::Name nombre;
nombre.length(1);
nombre[0].id = CORBA::string_dup("Super");
//Se resuelve el objeto
CORBA::Object_var obj = inc->resolve(name);
//Se hace un casting
Caja_var cliente = Caja::_narrow(obj);
```

El cliente ya tendría una referencia a un objeto de tipo `Caja` y podría acceder a los distintos servicios ofrecidos por el objeto.

El servicio de nombres ofrece más posibilidades que aquí no se detallan, y que se pueden encontrar en la especificación CORBA [OMG, 2001a] o en [Siegel, 2000]. Se ha visto que proporciona un mecanismo muy sencillo para que los clientes puedan localizar los objetos ofrecidos por un servidor.

8.2. Servicio de eventos

El servicio de eventos [OMG, 2001b] permite a las aplicaciones utilizar comunicaciones desacopladas en vez utilizar invocaciones síncronas entre el cliente y el servidor. Cuando un cliente realiza una invocación síncrona se queda bloqueado hasta que el servidor le responde. Para muchas aplicaciones distribuidas el modelo de invocación síncrono es demasiado restrictivo. Generalmente esas aplicaciones necesitan desacoplar los productores de información de los consumidores interesados en ella.

Este servicio permite desacoplar las comunicaciones entre objetos, es decir, actúa como mediador entre productores y consumidores. Los productores proporcionan eventos y los consumidores los reciben. Ambos tipos se conectan a un canal de eventos. Dicho canal conecta a los elementos de forma transparente, sin necesidad que tengan un conocimiento entre ellos. Es el elemento más importante del servicio y se encarga del registro de los productores y consumidores, de manejar los mensajes, de comunicar los errores...

El servicio de eventos proporciona dos modelos: *push* y *pull*.

En el modelo *push*, los productores colocan los eventos en el canal de eventos y éste se los envía a los consumidores. Éste se puede ver en la Figura 16.

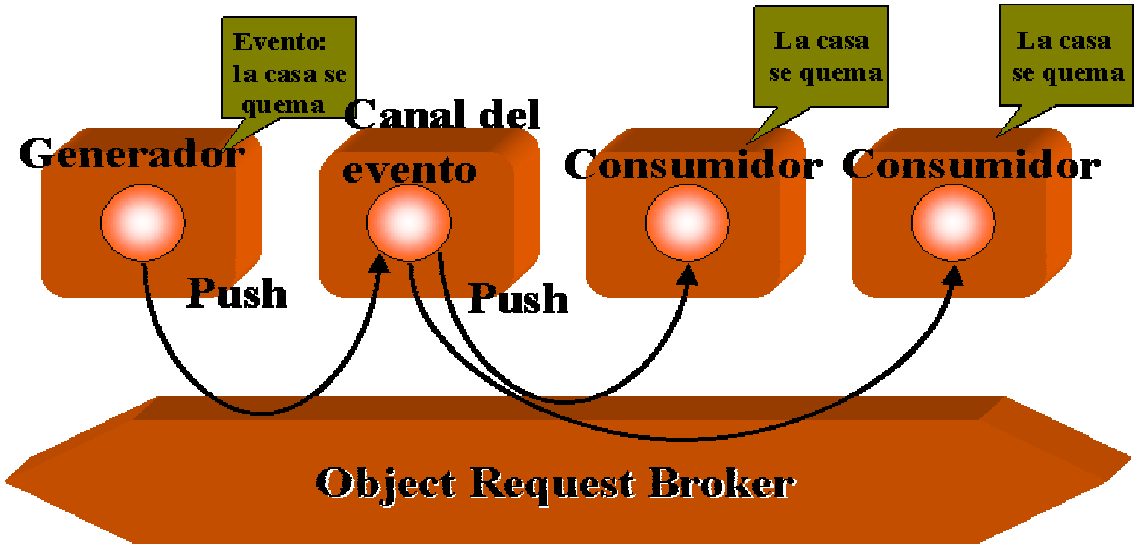

Figura 16. Modelo de eventos *push*

En el modelo *pull* el flujo de eventos ocurre en el sentido contrario, es decir, los consumidores solicitan eventos al canal de eventos y éste los solicita a los productores. Se puede observar el esquema en la Figura 17.

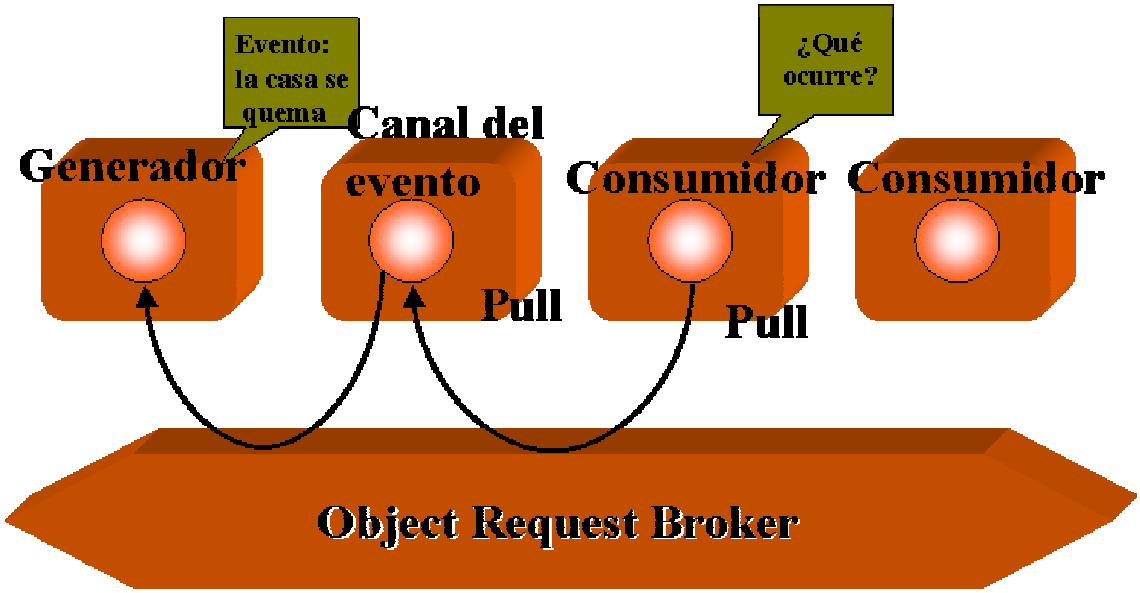

Figura 17. Modelo de eventos *pull*

Se pueden utilizar modelos híbridos *pull/push*. Lo más usual es utilizar productores *push* y consumidores *pull*, es decir, que ambos sean elementos activos, y que el canal de eventos tenga el papel de una cola que almacena eventos. Un único canal de eventos puede soportar varios tipos de modelo simultáneamente, a pesar de eso cada consumidor recibirá todos los eventos proporcionados por todos los productores. El canal sirve para desacoplar a los productores de los consumidores, es decir, no conocen el estado del resto de productores o consumidores.

Para utilizar el servicio de eventos se debe manejar el módulo *CosEventcomm* que proporciona las definiciones IDL necesarias.

```
module CosEventComm{
      exception Disconnected{};
      interface PushConsumer{
            void push(any data) raises(Disconnected);
```

```
                void disconnected_push_consumer();
            };
            interface PushSupplier{
                void disconnect_push_supplier();
            };
            interface PullConsumer{
                void disconnected_pull_consumer();
            };
            interface PullSupplier{
                any pull() raises(Disconnected);
                any try_pull(out boolean has_event)
                    raises(Disconnected);
                void disconnected_pull_consumer();
            };
        //...
};
```

También existe un módulo IDL con las definiciones del canal de eventos. Para más información se puede consultar la documentación dada por OMG o la implementación de algún ORB comercial.

Este servicio no se encuentra implementado en todos los ORBs comerciales y tiene algunas limitaciones que no siempre recomiendan su uso. Por una parte, al poder conectarse muchos productores a un canal de eventos, algunos consumidores podrían recibir eventos que no les interesan. Esto es debido a que los canales comunican todos los eventos a todos los consumidores que estén conectados a él, por lo que se despilfarran recursos. La única solución sería separar los tipos de eventos en distintos canales, con lo que consumidores interesados en distintos eventos podrían conectarse a distintos canales. Otro problema consiste en la falta de seguridad en el servicio, no se puede proporcionar una comunicación punto a punto segura.

Finalmente, en algunos casos las aplicaciones no requieren comunicaciones desacopladas si no que buscan comunicaciones asíncronas, es decir, poder realizar una petición sin que haya un bloqueo esperando la respuesta. Más tarde recibiría la respuesta mediante un *callback*. También podría desearse realizar peticiones independientes del tiempo, es decir, dar la posibilidad a un cliente de realizar una petición, desconectarse, conectarse de nuevo y conseguir la respuesta. Para cumplir estos requisitos OMG ha desarrollado el *CORBA Messaging Service* (6-5-1998) que no se encuentra implementado en su totalidad en ningún ORB de los manejados en el grupo de investigación. Esto es sólo un ejemplo de la distancia que existe entre las normas CORBA aprobadas por OMG y los servicios implementados por los distintos ORB.

9. RT-CORBA

La inclusión de la extensión a CORBA de tiempo real (Real-Time CORBA) viene justificada por la necesidad de cumplimiento de especificaciones de calidad de servicio (*QoS* o *Quality of Service*) en aplicaciones distribuidas en las que se pretende mantener el cumplimiento del comportamiento en tiempo real. Estas especificaciones aparecen en sistemas críticos como pueden ser los pertenecientes al campo de la aviónica, del control de procesos o de las telecomunicaciones. En este apartado se van a analizar los antecedentes que llevan a la definición de la especificación, para posteriormente señalar los aspectos fundamentales de RT-CORBA. Finalmente, se presentan algunas implementaciones.

9.1. Antecedentes

El problema planteado básicamente consiste en mantener el cumplimiento de las restricciones temporales *end-to-end* y no solamente en cada uno de los equipos que constituyen el sistema distribuido. Es evidente que este cumplimiento ha de estar sustentado por los recursos sobre los

que se apoya la implementación de CORBA correspondiente, es decir, el sistema operativo, que ha de seguir el perfil de tiempo real (estándar POSIX 1003.1b), y el subsistema de comunicaciones, donde los protocolos implicados deben garantizar la exactitud temporal.

No obstante, aunque dichas especificaciones sean cumplidas por los recursos indicados, el problema se presenta porque las especificaciones de CORBA 2.X no ofrecían soporte para estas restricciones **[Schmidt et al., 1999]**. No se proporcionaban interfaces que especifiquen el comportamiento punto a punto. Así, no existía una interfaz que permitiera especificar las prioridades relativas de las peticiones al ORB. Además, sería necesario que los clientes pudieran informar al ORB de cómo se han de ejecutar operaciones periódicas que tengan un plazo de finalización (*deadline*).

Otras limitaciones de la especificación son la imposibilidad para notificar a los clientes sobre las operaciones de control de flujo en el nivel de transporte y la carencia de un soporte para llevar a cabo operaciones temporizadas.

Asimismo, en lo que se refiere a las implementaciones, la mayor parte de los ORB planifican y atienden las peticiones de los clientes siguiendo un orden FIFO y, está demostrado que este tipo de estrategias incurre en problemas de inversión de prioridades. Además, no se dispone de un control de "grano fino" de la ejecución del servidor, debido a su caracter apropiativo, siendo esta característica un fuerte limitante para lograr sistemas predecibles. Por otra parte, aunque el GIOP soporta el paso de mensajes asíncronos, no existe un mapeo en un lenguaje de programación para la transmisión asíncrona de los mensajes.

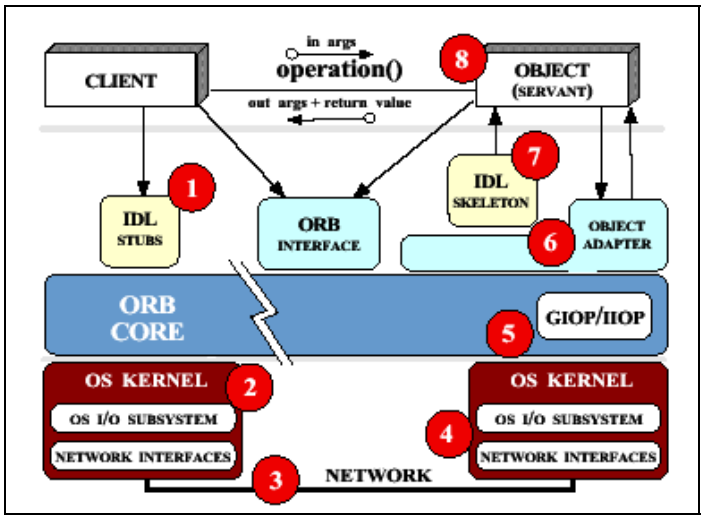

Figura 18. Problemas en el cumplimiento de requisitos de Tiempo Real en una implementación de CORBA

Finalmente, se pueden destacar los aspectos de funcionamiento de CORBA que pueden provocar problemas a la hora de cumplir las restricciones de tiempo real. En la Figura 18 se muestra la localización de los principales problemas que producen una pérdida de rendimiento y son:

- El *marshaling* puede llevar a desbordamientos en la caché (1,7)
- El tamaño variable de los mensajes exige *buffering* no uniforme (1,5)
- Los algoritmos de comunicación ineficientes (3)
- Las llamadas entre ORB's producen cadenas de un tamaño muy elevado(4)
- Ejecución del método(8)

9.2. *El estándar RT-CORBA*

El objetivo de esta especificación es resolver los problemas planteados que en buena parte eran debidos a una especificación incompleta. Así, formando parte del estándar CORBA 2.2, en 1999 se define la especificación de RT-CORBA **[OMG, 1999b]** que aparece también en la especificación de CORBA 2.6 **[OMG, 2001e]** y que ya es soportada por algunas implementaciones como TAO **[TAO, 2001]**.

Como elemento central se define una entidad planificable de forma que:

- Se proporciona una interfaz para el control de prioridades que se caracteriza por ser dependiente del cliente y es declarado por el servidor
- Dispone de mecanismos para evitar la inversión de prioridades y los bloqueos que sean causados por el carácter apropiativo del servidor

Asimismo, se plantea que los sistemas no sean estrictamente CORBA, es decir, que otros tipos de señales del sistema puedan disparar hilos que estén o no relacionados con los servicios de CORBA.

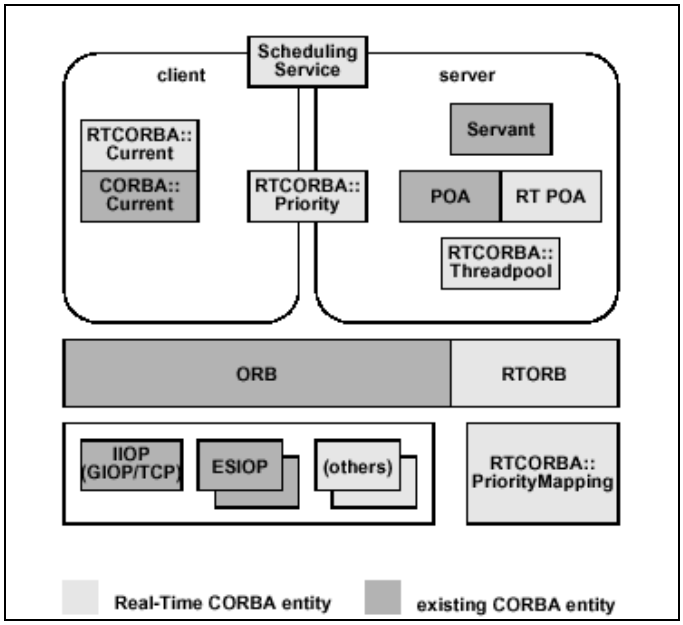

Figura 19.- Arquitectura de RT-CORBA

En la Figura 19 se presentan los elementos de la arquitectura de RT-CORBA y su relación con CORBA. Como se puede observar, con objeto de mantener la interoperatividad de los sistemas CORBA, en la especificación de RT-CORBA no se realizan definiciones nuevas (por ejemplo del IOP) sino que se realizan extensiones sobre definiciones existentes. Así, para el interfaz CORBA::ORB se define una extensión RTCORBA::ORB. Respecto al adaptador de objetos, de forma similar se define el RT-POA (Figura 20).

```
// IDL
module RTPortableServer {

    // locality constrained object
    interface POA : PortableServer::POA {

        ...

    };

};
```

Figura 20.- Definición del RT-POA

Como característica más sobresaliente, se puede señalar que el RT-POA ha de soportar las prioridades y las políticas de tiempo real que se establezcan. Como es evidente, los sistemas operativos de tiempo real que soportan las implementaciones del RT-ORB han de disponer de las correspondientes políticas de prioridades. Pero ésta no es uniforme para todos los sistemas operativos, por lo que para garantizar la interoperatividad se definen dos prioridades, para las cuales es necesario definir un procedimiento de mapeo:

- Nativa que está asociada a cada sistema operativo. Se definen dos tipos:
  - *Base* en la creación de un hilo.
  - *Activa* que se puede corresponder con la prioridad original, para un hilo ya existente, o con la heredada, cuando se aplica la herencia de prioridades.
- CORBA donde se realiza una representación uniforme, independiente de la plataforma.

Se definen dos modelos básicos de prioridades: el primero que es propagado por el cliente y el segundo en que las prioridades son declaradas en el servidor. El modelo se define a través del *PriorityModelPolicy* que es aplicado en el RT-POA desde el momento de su creación. Asimismo, se pueden definir transformaciones del modelo de prioridades utilizado.

Otras características adicionales vendrían dadas con objeto de poder definir exclusiones mutuas. Para ello, es necesario definir mecanismos que eviten problemas de inversión de prioridades, como puede ser el de herencia de prioridades. Asimismo, se definen *Threadpools* en el servidor como bancos de hilos de ejecución controlables por el usuario que están organizados en "calles" (semejantes a las de las piscinas) con prioridades. Finalmente, se plantea la incorporación de *time-outs* con el objetivo de incorporar predicibilidad.

9.3. Implementaciones

A continuación se muestran las implementaciones que se consideran más interesantes y que se adecuan a las especificaciones de RT-CORBA:

- **Visibroker [Highlander, 2002].** Su fabricante (*Highlander*) afirma que se trata de la primera implementación de CORBA con características de tiempo real. Especialmente pensada para aplicaciones de tiempo real, funciona con la mayor parte de los sistemas operativos con estas características, como Lynx, VxWorks, etc. Asimismo, está disponible para otras plataformas con menores prestaciones en este sentido, pero mucho más extendidas como Windows o Linux. Su potencial reside en que la plataforma está pensada para su utilización en sistemas empotrados (*embedded systems*).
- **TAO** (Proyecto ACE) **[TAO, 2001]**. Esta implementación será comentada en el apartado siguiente por ser de distribución gratuita. Aparte de incorporar las especificaciones del estándar de CORBA, implementa nuevas características encaminadas a mejorar la relación de los sistemas de E/S con el ORB.

10. ORBs de carácter gratuito

En este apartado se mencionan algunos de los ORBs de uso libre que se utilizan en los desarrollos del grupo de investigación. La reseña a éstos es somera, indicando la URL adecuada donde ampliar la información sobre ellos.

- **ORB MICO**. MICO **[MICO, 2001]** es un ORB orientado a la educación que no necesita equipos de grandes prestaciones, desarrollado por la Universidad de Frankfurt. Aunque su rendimiento no es muy bueno, es sumamente adecuado para

los primeros desarrollos. Es uno de los ORBs con más servicios implementados. Utiliza C++ como lenguaje de implementación. Ya existe MICO CCM, un ORB con el Modelo de Componentes de CORBA – CCM (*Corba Component Model*) [OMG, 1999] propio de CORBA 3.0.

- **ORB BACUS**. ORBACUS [IONA, 2001] un ORB de libre distribución para fines no comerciales. Su rendimiento es bastante bueno. Ya está actualizado a la norma CORBA 2.3. También implementa muchos servicios, aunque menos que MICO. Una ventaja es que ofrece *mapping* tanto a JAVA como a C++. Se puede instalar en diversas plataformas (Unix, linux, NT...).
- **ORB TAO.** TAO [TAO, 2001] es un ORB desarrollado en la Universidad de Washington por Douglas Schmidt y su equipo. Es un ORB con grandes prestaciones, es el que más servicios implementa. Permite la creación de servidores multihilo y tiene servicios para tiempo real. Su principal problema es la cantidad de espacio que ocupa y los recursos que consume. El manejo es más complicado que el de los anteriores. La implementación es en C++.
- **JACORB.** JACORB [JACORB, 2001] es un ORB desarrollado en la Universidad de Berlín. Está implementado en Java, pero ofrece menos servicios que el ORBACUS.

11. Desarrollo de una aplicación

Como se ha visto en el apartado anterior hay varios ORBs, tanto comerciales como de libre distribución, disponibles en este momento. Para el ejemplo que se va a desarrollar en el presente apartado se va a utilizar el ORB MICO, desarrollado en la Universidad de Frankfurt.

Los pasos, como se ilustra en la Figura 21, para crear una aplicación son los siguientes:

1. Definir las interfaces que participan en la aplicación mediante el IDL.
2. Compilar estas interfaces con el compilador IDL.
3. Crear la implementación de los objetos, el servidor y el cliente.
4. Compilar el código.
5. Ejecutar la aplicación.

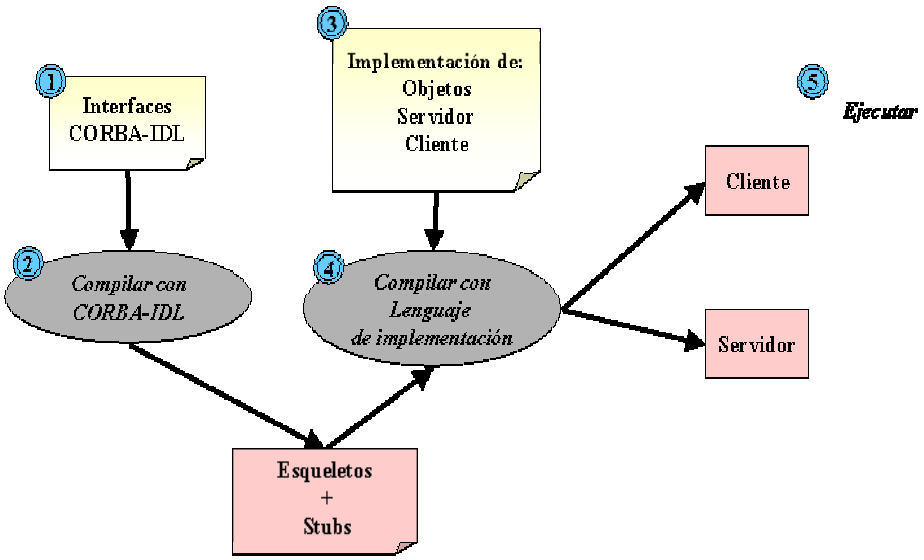

Figura 21. Pasos para el desarrollo de una aplicación CORBA

11.1. Un primer ejemplo

Como primer ejemplo se va tomar desarrollar el esqueleto de una caja registradora. Se tendría un fichero `caja.idl`, en el que se declararían todas las funciones miembro que el objeto `caja` ofrecerá a los clientes, indicando los parámetros y los tipos de retorno.

```
// IDL
interface  caja {
     long resultado ( );
     void meterdinero (in long cantidad);
     void sacardinero (in long cantidad2);
};
```

Utilizando el compilador IDL se obtendrían dos ficheros `caja.cc` y `caja.h`. En `caja.cc` estarán el *skeleton* y el *stub*. En el *servidor.cc* se definen la implementación del objeto y el código del servidor.

```
// servidor.cc
#include <CORBA.h>
#include <fstream.h>
#include "caja.h"
class caja: virtual public POA_caja {
 protected:
     CORBA::Long dinero;
   public:
     caja_impl ( ) { CORBA::Long dinero =0 ;}
     CORBA::Long   caja_impl::resultado ( ){return dinero;}
     void  caja_impl::meterdinero (CORBA::Long cantidad) {
       dinero =+ cantidad;
      }
     void caja_impl::sacardinero(CORBA::Long cantidad) {
       dinero=- cantidad;
     }
};

void main(int argc, char *argv[] ) {
  // Iniciar el ORB
  CORBA::ORB_var  orb = CORBA::ORB_init (argc, argv, "mico-local-orb");
  // Obtener una referencia al POA y al manager
  CORBA::Object_var poaobj = orb-> resolve_initial_references ("RootPOA");
  PortableServer: :POA_var poa = PortableServer: :POA: :_narrow(poaobj);
  PortableServer: :POAManager_var poaManager = poa-> the_POAManager ( );
  // Creación de la implementacion
  caja_impl *cajero= new caja_impl ( );
  // Activar la implementacion
  PortableServer::ObjectId_var identificador=poa->activate_object(cajero);
  // Almacenar la referencia
  ofstream fichero( "caja.ref");
  fichero << orb->object_to_string(poa->id_to_reference(identificador))<<endl;
  fichero.close ( ) ;
  poaManager->activate ( );  // Activar el POA
  orb->run ( ) ;  // Ciclo de recepción de peticiones
}
```

Un servidor siempre comienza localizando el ORB y el adaptador de objetos. Mediante `resolve_initial_references` aplicada a `RootPOA` se obtiene una referencia a la raíz del adaptador de objetos POA. También se obtiene una referencia al manager del adaptador POA que será el que gestione el estado del adaptador POA.

Existen diversas formas para exportar la referencia. En el ejemplo se utiliza un fichero para simplificar el servidor al máximo. En una aplicación real se utilizaría el servicio de nombres para publicar las referencias.

Lo primero que hace el cliente es obtener una referencia a un objeto con el que desea trabajar. En este caso, la referencia a objeto se le ha pasado a través de un archivo. Una vez que tiene la referencia objeto sólo tiene que acceder a los métodos definidos por ese objeto, como si estuviera ejecutando una invocación local.

```
//cliente.cc

#include <fstream.h>
#include "caja.h"
void main ( int argc, char *argv[] ) {
  CORBA::ORB_var  orb = CORBA: :ORB_init (arg,argv,"mico-local-orb");
  // Se lee la referencia
  ifstream fichero("caja.ref);
  char cadena[1000];
  fichero >> cadena;
  CORBA::Object_var  obj=orb->string_to_object(cadena);
  assert (!CORBA::is_nil (obj));
  caja_var  cajero = caja::_narrow(obj);
  assert (!CORBA::is_nil (caja) );
  fichero.close ();
  //Ejecutar metodos
  caja->meterdinero (1000);
  caja->sacardinero (500);
  cout << "Nos queda " << caja->resultado() << endl;
}
```

11.2. Un segundo ejemplo

En este caso se muestra una aplicación completa para observar como se aplica el *mapping* de tipos. La aplicación en sí misma no tiene utilidad, pero su interfaz ha sido elegido cuidadosamente para mostrar un ejemplo de *mapping* no trivial.

El primer paso es crear un fichero .idl, en el que se definan los tipos que se van a utilizar, así como la interfaz de la clase que será generada por el compilador IDL.

Fichero: ejemplo2.idl

```
module Ejemplo
{
   const long tamano=256;
   typedef string Array_entradas[tamano];
   interface Leer
   {
         Array_entradas Lee_directoeio(in string directorio);
   };
};
```

En este fichero se ha definido una constate `tamano` que se utiliza para indicar el tamaño del `array`, el cual es del tipo `string`, y tiene un tamaño variable ya que una cadena de caracteres puede ser lo larga que uno quiera. También se define la interfaz `Leer` que será traducida en una clase con un método que utilizará el `array` definido como tipo de retorno.

También comentar que el `Module Ejemplo` será traducido en C++ como un `namespace`. El compilador IDL generará dos ficheros: `ejemplo2.h`, en el que se definen los tipos y las clases, y `ejemplo2.cc`, en el que se implementan los métodos.

Fichero: ejemplo2.h

Este fichero se genera automáticamente por el compilador IDL y en el se definen cuatro clases. La clase `Leer`, de la que se derivan el resto y que contiene los métodos virtuales definidos por el usuario además de algunos extras. La clase `Leer_skel`, de la que se deriva la implementación que haga el usuario, tiene los métodos necesarios para obtener la referencia del objeto. La clase `Leer_stub`, es la utilizada por el cliente y tiene todos los métodos necesarios para traducir (*marshalling*) los argumentos de las funciones miembro del objeto remoto. Por último, la clase `Leer_var` tiene la particularidad de tener el manejo de memoria automático, liberando la memoria reservada para un objeto cuando este ya no se utilice.

```
    /*
     *  MICO --- a free CORBA implementation
     *  Copyright (C) 1997-98 Kay Roemer & Arno Puder
     *
     *  This file was automatically generated. DO NOT EDIT!
     */

    #if !defined(__EJEMPLO2_H__) || defined(MICO_NO_TOPLEVEL_MODULES)
    #define __EJEMPLO2_H__

    #ifndef MICO_NO_TOPLEVEL_MODULES
    #include <CORBA.h>
    #include <mico/throw.h>
    #endif

    #ifndef MICO_NO_TOPLEVEL_MODULES
    MICO_NAMESPACE_DECL Ejemplo {
    #endif

    #if !defined(MICO_NO_TOPLEVEL_MODULES) || defined(MICO_MODULE_Ejemplo)


    MICO_EXPORT_CONST_DECL const CORBA::Long tamano;
    typedef CORBA::String_var Array_entradas[ 256 ];
    typedef CORBA::String_var Array_entradas_slice;
    typedef       ArrayVarVar<CORBA::String_var,Array_entradas_slice,Array_entradas,256>
Array_entradas_var;
    typedef Array_entradas_var Array_entradas_out;
    enum _dummy_Array_entradas { _dummy_Array_entradas_0 };
    typedef
ArrayVarForAny<CORBA::String_var,Array_entradas_slice,Array_entradas,256,_dummy_Array_en
tradas> Array_entradas_forany;
    #undef MICO_ARRAY_ARG
    #define MICO_ARRAY_ARG CORBA::String_var
    DECLARE_ARRAY_ALLOC(Array_entradas,MICO_ARRAY_ARG,Array_entradas_slice,256)
    DECLARE_ARRAY_DUP(Array_entradas,MICO_ARRAY_ARG,Array_entradas_slice,256)
    DECLARE_ARRAY_FREE(Array_entradas,MICO_ARRAY_ARG,Array_entradas_slice,256)
    DECLARE_ARRAY_COPY(Array_entradas,MICO_ARRAY_ARG,Array_entradas_slice,256)
    class Leer;
    typedef Leer *Leer_ptr;
    typedef Leer_ptr LeerRef;
    typedef ObjVar<Leer> Leer_var;
    typedef Leer_var Leer_out;


    // Common definitions for interface Leer
    class Leer :
      virtual public CORBA::Object
    {
      public:
        virtual ~Leer();

        #ifdef HAVE_TYPEDEF_OVERLOAD
        typedef Leer_ptr _ptr_type;
        typedef Leer_var _var_type;
        #endif

        static Leer_ptr _narrow( CORBA::Object_ptr obj );
        static Leer_ptr _narrow( CORBA::AbstractBase_ptr obj );
```

```
        static Leer_ptr _duplicate( Leer_ptr _obj )
        {
          CORBA::Object::_duplicate (_obj);
          return _obj;
        }

        static Leer_ptr _nil()
        {
          return 0;
        }

        virtual void *_narrow_helper( const char *repoid );
        static vector<CORBA::Narrow_proto> *_narrow_helpers;
        static bool _narrow_helper2( CORBA::Object_ptr obj );

        virtual Array_entradas_slice* Lee_directoeio( const char* directorio ) = 0;

      protected:
        Leer() {};
      private:
        Leer( const Leer& );
        void operator=( const Leer& );
    };

    // Stub for interface Leer
    class Leer_stub:
      virtual public Leer
    {
      public:
        virtual ~Leer_stub();
        Array_entradas_slice* Lee_directoeio( const char* directorio );

      private:
        void operator=( const Leer_stub& );
    };

    class Leer_skel :
      virtual public StaticMethodDispatcher,
      virtual public Leer
    {
      public:
        Leer_skel( const CORBA::BOA::ReferenceData & = CORBA::BOA::ReferenceData() );
        virtual ~Leer_skel();
        Leer_skel( CORBA::Object_ptr obj );
        virtual bool dispatch( CORBA::StaticServerRequest_ptr _req, CORBA::Environment
&_env );
        Leer_ptr _this();

    };

    #endif // !defined(MICO_NO_TOPLEVEL_MODULES) || defined(MICO_MODULE_Ejemplo)

    #ifndef MICO_NO_TOPLEVEL_MODULES

    };
    #endif


    #ifndef MICO_CONF_NO_POA

    #endif // MICO_CONF_NO_POA

    #if !defined(MICO_NO_TOPLEVEL_MODULES) || defined(MICO_MODULE__GLOBAL)

    extern CORBA::StaticTypeInfo *_marshaller_Ejemplo_Leer;

    extern CORBA::StaticTypeInfo *_marshaller__a256_string;

    #endif // !defined(MICO_NO_TOPLEVEL_MODULES) || defined(MICO_MODULE__GLOBAL)


    #if !defined(MICO_NO_TOPLEVEL_MODULES) && !defined(MICO_IN_GENERATED_CODE)
    #include <mico/template_impl.h>
    #endif
```

```
#endif
```

En las siguientes líneas de código se observa como se traduce los tipos de acuerdo con lo que se explicó anteriormente en el apartado 7.2.1.

```
MICO_EXPORT_CONST_DECL const CORBA::Long tamano;
typedef CORBA::String_var Array_entradas[ 256 ];
typedef CORBA::String_var Array_entradas_slice;
typedef       ArrayVarVar<CORBA::String_var,Array_entradas_slice,Array_entradas,256>
Array_entradas_var;

typedef
ArrayVarForAny<CORBA::String_var,Array_entradas_slice,Array_entradas,256,_dummy_Array_en
tradas> Array_entradas_forany;
```

La constante se traduce en una constante y además se sustituye por su valor numérico. También se observa como se crea el tipo `Array_entradas` como un `array` y el tipo `Array_entradas_slice` con una dimensión menos. Además, se crean los tipos `Array entradas_var` que tiene manejo automático de memoria y el tipo `Array_entradas_forany` para el almacenamiento del `array` en un tipo `any`.

```
class Leer :
  virtual public CORBA::Object
{
  public:
   ...
       virtual Array_entradas_slice* Lee_directoeio( const char* directorio ) = 0;
   ...
}
```

Este código pertenece a la definición de la clase `Leer` que hereda de la clase `CORBA::Object` y se ve como se define el método `Lee_directorio()` como virtual puro para que sus clases derivadas puedan implementarlo. En este método el tipo de retorno es un puntero `Array_entradas_slice` que es un elemento de `Array_entrasdas` ya que este sólo tiene una dimensión. Por su parte, la cadena que se utiliza como argumento de entrada se ha convertido en un `const char`.

Fichero: ejemplo2.cc

En este fichero se implementan los métodos de las clases definidas en el fichero ejemplo2.h.

```
/*
 *  MICO --- a free CORBA implementation
 *  Copyright (C) 1997-98 Kay Roemer & Arno Puder
 *
 *  This file was automatically generated. DO NOT EDIT!
 */

#include "ejemplo2.h"

//-------------------------------------------------------
//  Implementation of stubs
//-------------------------------------------------------
#ifdef HAVE_NAMESPACE
namespace Ejemplo { const CORBA::Long tamano = 256; };
#else
const CORBA::Long Ejemplo::tamano = 256;
#endif


// Stub interface Leer
Ejemplo::Leer::~Leer()
{
}

void *Ejemplo::Leer::_narrow_helper( const char *_repoid )
{
  if( strcmp( _repoid, "IDL:Ejemplo/Leer:1.0" ) == 0 )
    return (void *)this;
```

```
  return NULL;
}

bool Ejemplo::Leer::_narrow_helper2( CORBA::Object_ptr _obj )
{
  if( strcmp( _obj->_repoid(), "IDL:Ejemplo/Leer:1.0" ) == 0) {
    return true;
  }
  for( vector<CORBA::Narrow_proto>::size_type _i = 0;
       _narrow_helpers && _i < _narrow_helpers->size(); _i++ ) {
    bool _res = (*(*_narrow_helpers)[ _i ])( _obj );
    if( _res )
      return true;
  }
  return false;
}

Ejemplo::Leer_ptr Ejemplo::Leer::_narrow( CORBA::Object_ptr _obj )
{
  Ejemplo::Leer_ptr _o;
  if( !CORBA::is_nil( _obj ) ) {
    void *_p;
    if( (_p = _obj->_narrow_helper( "IDL:Ejemplo/Leer:1.0" )))
      return _duplicate( (Ejemplo::Leer_ptr) _p );
    if( _narrow_helper2( _obj ) ||
       ( _obj->_is_a_remote( "IDL:Ejemplo/Leer:1.0" ) ) ) {
      _o = new Ejemplo::Leer_stub;
      _o->MICO_SCOPE(CORBA,Object::operator=)( *_obj );
      return _o;
    }
  }
  return _nil();
}

Ejemplo::Leer_ptr
Ejemplo::Leer::_narrow( CORBA::AbstractBase_ptr _obj )
{
  return _narrow (_obj->_to_object());
}

Ejemplo::Leer_stub::~Leer_stub()
{
}

Ejemplo::Array_entradas_slice*   Ejemplo::Leer_stub::Lee_directoeio(   const   char*
directorio )
{
  CORBA::StaticAny _directorio( CORBA::_stc_string, &directorio );
  CORBA::StaticAny __res( _marshaller__a256_string );

  CORBA::StaticRequest __req( this, "Lee_directoeio" );
  __req.add_in_arg( &_directorio );
  __req.set_result( &__res );

  __req.invoke();

  mico_sii_throw( &__req,
    0);
  return (Ejemplo::Array_entradas_slice*) __res._retn();
}


#ifdef HAVE_NAMESPACE
namespace Ejemplo { vector<CORBA::Narrow_proto> * Leer::_narrow_helpers; };
#else
vector<CORBA::Narrow_proto> * Ejemplo::Leer::_narrow_helpers;
#endif

class _Marshaller_Ejemplo_Leer : public CORBA::StaticTypeInfo {
    typedef Ejemplo::Leer_ptr _MICO_T;
  public:
    StaticValueType create () const;
    void assign (StaticValueType dst, const StaticValueType src) const;
    void free (StaticValueType) const;
```

```
        CORBA::Boolean demarshal (CORBA::DataDecoder&, StaticValueType) const;
        void marshal (CORBA::DataEncoder &, StaticValueType) const;
    };


    CORBA::StaticValueType _Marshaller_Ejemplo_Leer::create() const
    {
      return (StaticValueType) new _MICO_T( 0 );
    }

    void _Marshaller_Ejemplo_Leer::assign( StaticValueType d, const StaticValueType s )
const
    {
      *(_MICO_T*) d = ::Ejemplo::Leer::_duplicate( *(_MICO_T*) s );
    }

    void _Marshaller_Ejemplo_Leer::free( StaticValueType v ) const
    {
      CORBA::release( *(_MICO_T *) v );
      delete (_MICO_T*) v;
    }

    CORBA::Boolean    _Marshaller_Ejemplo_Leer::demarshal(    CORBA::DataDecoder    &dc,
StaticValueType v ) const
    {
      CORBA::Object_ptr obj;
      if (!CORBA::_stc_Object->demarshal(dc, &obj))
        return FALSE;
      *(_MICO_T *) v = ::Ejemplo::Leer::_narrow( obj );
      CORBA::Boolean ret = CORBA::is_nil (obj) || !CORBA::is_nil (*(_MICO_T *)v);
      CORBA::release (obj);
      return ret;
    }

    void _Marshaller_Ejemplo_Leer::marshal( CORBA::DataEncoder &ec, StaticValueType v )
const
    {
      CORBA::Object_ptr obj = *(_MICO_T *) v;
      CORBA::_stc_Object->marshal( ec, &obj );
    }

    CORBA::StaticTypeInfo *_marshaller_Ejemplo_Leer;

    class _Marshaller__a256_string : public CORBA::StaticTypeInfo {
        typedef CORBA::String_var _MICO_T;
      public:
        StaticValueType create () const;
        void assign (StaticValueType dst, const StaticValueType src) const;
        void free (StaticValueType) const;
        CORBA::Boolean demarshal (CORBA::DataDecoder&, StaticValueType) const;
        void marshal (CORBA::DataEncoder &, StaticValueType) const;
    };


    CORBA::StaticValueType _Marshaller__a256_string::create() const
    {
      return (StaticValueType) new _MICO_T[ 256 ];
    }

    void _Marshaller__a256_string::assign( StaticValueType d, const StaticValueType s )
const
    {
      for( int i = 0; i < 256; i++ )
        ((CORBA::String_var *) d)[ i ] = ((CORBA::String_var *) s)[ i ];
    }

    void _Marshaller__a256_string::free( StaticValueType v ) const
    {
      delete[] (_MICO_T *) v;
    }

    CORBA::Boolean    _Marshaller__a256_string::demarshal(    CORBA::DataDecoder    &dc,
StaticValueType v ) const
    {
```

```
      if( !dc.arr_begin() )
        return FALSE;
      for( CORBA::ULong i = 0; i < 256; i++ ) {
        if( !CORBA::_stc_string->demarshal( dc, &((_MICO_T*)v)[i].inout() ) )
          return FALSE;
      }
      return dc.arr_end();
    }

    void _Marshaller__a256_string::marshal( CORBA::DataEncoder &ec, StaticValueType v )
const
    {
      ec.arr_begin();
      for( CORBA::ULong i = 0; i < 256; i++ )
        CORBA::_stc_string->marshal( ec, &((_MICO_T*)v)[i].inout() );
      ec.arr_end();
    }

    CORBA::StaticTypeInfo *_marshaller__a256_string;

    struct __tc_init_EJEMPLO2 {
      __tc_init_EJEMPLO2()
      {
        _marshaller_Ejemplo_Leer = new _Marshaller_Ejemplo_Leer;
        _marshaller__a256_string = new _Marshaller__a256_string;
      }
    };

    static __tc_init_EJEMLO2 __init_EJEMPLO2;

    //--------------------------------------------------------
    //  Implementation of skeletons
    //--------------------------------------------------------

    Ejemplo::Leer_skel::Leer_skel( const CORBA::BOA::ReferenceData &_id )
    {
      CORBA::ImplementationDef_var _impl =
        _find_impl( "IDL:Ejemplo/Leer:1.0", "Leer" );
      assert( !CORBA::is_nil( _impl ) );
      _create_ref( _id,
        0,
        _impl,
        "IDL:Ejemplo/Leer:1.0" );
      register_dispatcher( new StaticInterfaceDispatcherWrapper<Leer_skel>( this ) );
    }

    Ejemplo::Leer_skel::Leer_skel( CORBA::Object_ptr _obj )
    {
      CORBA::ImplementationDef_var _impl =
        _find_impl( "IDL:Ejemplo/Leer:1.0", "Leer" );
      assert( !CORBA::is_nil( _impl ) );
      _restore_ref( _obj,
        CORBA::BOA::ReferenceData(),
        0,
        _impl );
      register_dispatcher( new StaticInterfaceDispatcherWrapper<Leer_skel>( this ) );
    }

    Ejemplo::Leer_skel::~Leer_skel()
    {
    }

    bool     Ejemplo::Leer_skel::dispatch(     CORBA::StaticServerRequest_ptr     _req,
CORBA::Environment & /*_env*/ )
    {
      #ifdef HAVE_EXCEPTIONS
      try {
      #endif
        if( strcmp( _req->op_name(), "Lee_directoeio" ) == 0 ) {
          CORBA::String_var directorio;
          CORBA::StaticAny _directorio( CORBA::_stc_string, &directorio.inout() );

          Array_entradas_slice* _res;
          CORBA::StaticAny __res( _marshaller__a256_string );
```

```
          _req->add_in_arg( &_directorio );
          _req->set_result( &__res );

          if( !_req->read_args() )
            return true;

          _res = Lee_directoeio( directorio );
          __res.value( _marshaller__a256_string, _res );
          _req->write_results();
          Array_entradas_free( _res );
          return true;
        }
      #ifdef HAVE_EXCEPTIONS
      } catch( CORBA::SystemException_catch &_ex ) {
        _req->set_exception( _ex->_clone() );
        _req->write_results();
        return true;
      } catch( ... ) {
        assert( 0 );
        return true;
      }
      #endif
      return false;
    }

    Ejemplo::Leer_ptr Ejemplo::Leer_skel::_this()
    {
      return Ejemplo::Leer::_duplicate( this );
    }
```

Fichero: Leer_impl.cc

En este fichero se implementa el método que lee el fichero, concretamente lo que hace es leer las 256 primeras entradas en el directorio que se le dé como argumento. Lo más interesante de este código es cómo se utilizan y declaran tipos CORBA, además de como se retornan dichos tipos dentro de una aplicación.

```
    #include "ejemplo2.h"
    #include <sys/stat.h>
    #include <sys/types.h>
    #include <unistd.h>
    #include <dirent.h>
    #include <fcntl.h>


    Ejemplo::Array_entradas_slice   *servidor::   Leer_impl::LeeDirectorio(const   char*
directorio)
    {
       int n_entrada,fin,i;
       struct stat ss;
       struct dirent *datos;
       char *buffer,*ptr_aux;
       int dir,tam,n_bytes;
       off_t pbase;
       Ejemplo::Array_entradas_var ficheros;
       ficheros=Ejemplo::Array_entradas_alloc();


       chdir(directorio);
       dir=open(directorio,O_RDONLY);
       if(dir==-1)
              cerr<<"Error fatal1"<<endl;
       if(stat(directorio,&ss)==-1)
              cerr<<"Error fatal2"<<endl;
       buffer=new char[ss.st_blksize];
       datos= new struct dirent;
       n_bytes=ss.st_blksize;
       do
       {
              tam=getdirentries(dir,buffer,n_bytes,&pbase);
              ptr_aux=buffer;
              datos=(struct dirent *)buffer;
              while(ptr_aux-buffer<tam&&n_entrad<256)
```

```
            {
                    //introducir datos en la estructura a devolver
                    ficheros[n_entrada++]=CORBA::string_dup(datos->d_name);
                    ptr_aux+=datos->d_reclen;
                    datos=(struct dirent*)ptr_aux;
            }
    }while(tam>0);
    delete[] buffer;
    return ficheros._retn();
}
```

En las siguientes líneas de código se muestra como se declara un tipo CORBA en este caso se ha escogido `Array_entradas_var` por las facilidades que ofrece en el manejo de memoria. Además, se utiliza la función `Array_entradas_alloc()` que permite la reserva de memoria para este `array`.

```
Ejemplo::Array_entradas_var ficheros;
ficheros=Ejemplo::Array_entradas_alloc();
```

Ahora se muestra una de las posibilidades que existen para introducir valores en el `array` a través del operador `[]`, ya que se definen para el tipo `Array_entradas` para acceder a sus elementos individuales y mediante la función `CORBA::string_dup()` se introducen valores a esos elementos ya que esta función duplica la cadena que recibe como argumento.

```
ficheros[n_entrada++]=CORBA::string_dup(datos->d_name);
```

Por último, resaltar como se realiza el retorno utilizando un método del tipo `Array_entradas_var`. Esto es debido a que todos los tipos CORBA son clases que tienen sus propios métodos, este en cuestión se utiliza para pasar el tipo o puntero de retorno adecuado.

```
return ficheros._retn();
```

Fichero: principal.cc

En este caso se ha utilizado BOA y en este fichero se inicia el ORB y el adaptador de objetos BOA, además de obtener la referencia del objeto que se introduce en un fichero para que el cliente pueda leerlo. También se pone en marcha el bucle de eventos del ORB.

```
#include "ejemplo2.h"
#include <fstream.h>
#include <iostream.h>

int main (int argc, char *argv[]) {
   // Se inicia BOA
   CORBA::ORB_var orb=CORBA::ORB_init(argc,argv,"mico-local-orb");
   CORBA::BOA_var boa=orb->BOA_init(argc,argv,"mico-local-boa");

   Ejemplo::Server_impl* instancia=new servidor::Server_impl;

   CORBA::String_var ref=orb->object_to_string(instancia);
   ofstream out("/tmp/prueba1.objid");
   out<<ref<<endl;
   out.close();

   boa->impl_is_ready(CORBA::ImplementationDef::_nil());
   orb->run();
   CORBA::release(instancia);
   return 0;
}
```

Fichero: cliente.cc

Este fichero inicia el adaptador BOA y el ORB, para a partir de la referencia que lee de un fichero utilizar el objeto mediante su interfaz. Por último, resaltar que la variable en la que se almacenan los datos retornados por `Leer->Lee_directorio()` se declara pero no se reserva memoria ya que ésta ha sido reservada en el servidor. Al tratarse del tipo `Array_entradas_var` tampoco es necesario liberarla.

```
#include <iostream.h>
#include <fstream.h>
#include "ejemplo2.h"
int main (int argc, char *argv[])
{
   // Se inicia el adaptador BOA
   CORBA::ORB_var orb=CORBA::ORB_init(argc,argv,"mico-local-orb");
   CORBA::BOA_var boa=orb->BOA_init(argc,argv,"mico-local-boa");

   ifstream in("/tmp/prueba1.objid");
   char ref[1000];
   in>>ref;
   in.close();

   cout<<"Referencia:"<<ref<<endl;

   CORBA::Object_var obj=orb->string_to_object(ref);
   Ejemplo::server_var client=servidor::server::_narrow(obj);

   Ejemplo::Array_entradas_var ficheros;
   cout<<"Antes"<<endl;
   ficheros=client->LeeDirectorio(".");
   int cont;
   cout<<"Despues"<<endl;
   for(cont=0;cont<4;cont++)
          cout<<"Fichero:"<<ficheros[cont]<<endl;

   return 0;
}
```

12. Conclusiones

En este documento se ha querido resumir los conceptos fundamentales que cualquiera que desee trabajar con CORBA debe conocer y manejar.

CORBA es hoy en día una opción tecnológica sumamente aceptada por la industria del software en lo referente al desarrollo de soluciones distribuidas. A CORBA le surgen competidores, especialmente desde el mundo de Microsoft y desde el mundo de Java, aunque CORBA presenta las ventajas de no ser propietario, no estar ligado a ningún lenguaje de programación concreto y además ser capaces de integrarse sus competidores mediante las pasarelas adecuadas, en un claro ejemplo de interoperabilidad entre sistemas heterogéneos.

Como punto final de este documento se vuelven a resaltar los aspectos más destacados de CORBA, que son

- **Heterogeneidad**. La utilización de OMG IDL para definir interfaces de objetos permite que esas interfaces sean usadas tanto por lenguajes de programación como por plataformas de computación diferentes.
- **Modelo de Objetos**. El modelo de objeto proporcionado por OMA define las reglas para la interacción entre objetos CORBA que sean independientes de los protocolos de red. Gracias a esta independencia pueden ser utilizados en una gran cantidad de entornos.
- **Integración**. Debido a que CORBA no especifica la implementación, un ORB bien diseñado no requiere que los componentes y las tecnologías en uso sean abandonados.
- **Aproximación orientada a objeto**. CORBA y las aplicaciones construidas en esa plataforma se diseñan más fácilmente siguiendo los principios de la orientación a objetos. El hecho de que las interfaces de los objetos sean definidas en IDL ayuda a los desarrolladores a pensar en sus aplicaciones en términos de componentes reutilizables que interaccionan.

Como una de las mayores desventajas de CORBA cabe citar el retraso existente entre las especificaciones publicadas por el OMG y su incorporación a los ORBs que se emplean para el desarrollo de aplicaciones. Esto es especialmente grave en el caso de los servicios CORBA donde se depende completamente del ORB elegido.

13. Referencias

[Bacon et al., 2000] Bacon, J., Moody, K., Bates, J., Hayton, R., Ma, C., McNeil, A., Seidel, O. y Spiteri, M. (2000). "*Generic Support for Distributed Applications*". IEEE Computer, 33(3):68-76. March, 2000.

[Balkovich et al., 1985] Balkovich, E., Lerman, S. y Parmelee, R. (1985). "*Computing in Higher Education: The Athena Experience*". Communications of the ACM, 28(11):1214-1224.

[Beck and Cunningham, 1989] Beck, Kent and Cunningham, Ward. (1989). "*A Laboratory for Teaching Object-Oriented Thinking*". In Proceedings of the 1989 OOPSLA - Conference proceedings on Object-Oriented Programming Systems, Languages and Applications (October 2 - 6, 1989, New Orleans, LA USA); Reprinted in Sigplan Notices, 24(10):1-6.

[Bohnhoff et al., 1994] Bohnhoff, P., Jansen, R. y Martín, R. (1994). "*Fundamentos Cliente/Servidor*". IBM.

[Brockschmidt, 1994] Brockschmidt, K. (1994). "*Inside OLE 2*". Microsoft Press.

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P, y Stal, M. (1996). "*Pattern Oriented Software Architecture: A System of Patterns*". John Wiley & Sons.

[Campagnoni, 1994] Campagnoni, F. R. (1994). "*IBM's System Object Model*". Dr. Dobb's Journal, Special Report, #225 Winter 1994/1995: 24-28.

[Coplien, 1998] Coplien, James O. "*A Pattern Definition - Software Patterns*". http://hillside.net/patterns/definition.html. 1998.

[Curtis et al., 1997] Curtis, D., Stone, C. y Bradley, M. (1997). "*IIOP: OMG's Internet Inter-ORB Protocol: A Brief Description*". OMG Whitepaper. http://www.omg.org/news/whitepapers/iiop.htm [Última vez visitado, 27-7-2001].

[Chappel, 1996] Chappel, D. (1996). "*Understanding ActiveX and OLE*". Microsoft Press.

[Chin y Chanson, 1991] Chin, R. S. y Chanson, S. T. (1991). "*Distributed Object-Based Programming Systems*". ACM Computing Surveys, 23(1):91-124. March, 1991.

[DRAE, 1995] Real Academia Española. "*Diccionario de Real Academia*". Vigésimo primera edición. Espasa-Calpe. Edición electrónica, versión 21.1.0. 1995.

[Geihs, 2001] Geihs, K. (2001). "*Middleware Challenges Ahead*". IEEE Computer, 34(6):24-31. June, 2001.

[Geihs y Hollberg, 1990] Geihs, K. y Hollberg, U. (1990). "*A Retroperspective on DACNOS*". Communications of the ACM, 33(4):439-448.

[González et al.,2000] González, J., Curto, B., García, F. J., Moreno, V. y Blanco, J. (2000). "*Diseño de un Sistema Distribuido para la Gestión de una Célula CIM*". Actas del Simposio Español de Informática Distribuida, SEID 2000. Editores S. Barro, J. Mª Busta, J. M. Corchado y P. Cuesta (Ourense, 25-27 de Septiembre de 2000). Páginas 127-134.

[Herbert, 1987] Herbert, A. (1987). "*Key Architectural Concepts*". ANSA Project, Report Nº AO-82-02.

[Highlander, 2002] Highlander (2002). http://www.highlander.com/ [Última vez visitado, 15-1-2002]

[IONA, 2001] IONA. (2001). "*ORBACUS*". http://www.orbacus.com/. [Última vez visitado, 27-7-2001].

[JACORB, 2001] JACORB. (2001). "*JAC ORB*". http://www.jacorb.org. [Última vez visitado, 27-7-2001].

[Lewis, 1995] Lewis, T. G. (1995). "*Where Is Client/Server Software Headed*". IEEE Computer, 28(4): 49-55. April, 1995.

[MICO, 2001] MICO. "*Mico Is CORBA*". http://www.mico.org. [Última vez visitado 27-6-2001]. 2001.

[Microsoft, 1996] Microsoft Corporation. (1996). "*DCOM Technical Overview*". White Paper, Microsoft Developer Network.

[Morris et al., 1986] Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S. y Smith, F. D. (1986). "*Andrew: A Distributed Personal Computing Environment*". Communications of the ACM, 29(3):184-201.

[OMG, 1995] Object Management Group. (1995). "*Common Object Request Broker: Architecture and Specification*". Revision 2.0.

[OMG, 1997] Object Management Group. (1997). "*Common Object Request Broker: Architecture and Specification*". Revision 2.1.

[OMG, 1998a] Object Management Group. (1998). "*Common Object Request Broker: Architecture and Specification*". Revision 2.3a.

[OMG, 1998b] Object Management Group. (1998). "*The Portable Object Adaptor*". Revision 2.2.

[OMG, 1999] OMG. (1999). "*CORBA Components - Volume I*". Object Management Group Inc. http://www.omg.org/cgi-bin/doc?orbos/99-07-01.

[OMG, 1999b] OMG. (1999). "*Real Time CORBA*". Object Management Group Inc. http://www.omg.org/cgi-bin/doc?ptc/99-05-03.

[OMG, 2001a] Object Management Group. (2001). "*Complete CORBA 2.4.2 Specification*". Document formal/01-02-33.

[OMG, 2001b] Object Management Group. (2001). "*Event Service Specification*". Version 1.1. March, 2001.

[OMG, 2001c] Object Management Group. (2001). "*Naming Service Specification*". Revised Edition: February 2001.

[OMG, 2001d] Object Management Group. (2001). "*OMG Unified Modeling Language Specification (draft) Version 1.4*". Object Management Group Inc. http://www.celigent.com/omg/umlrtf/artifacts.htm. [Última vez visitado, 12/2/2001]. February, 2001.

[OMG, 2001e] Object Management Group. (2001). "*CORBA 2.6 Specification*". Document formal/01-02-35. http://www.omg.org/cgi-bin/doc?formal/01-12-35. [Última vez visitado, 23/1/2002]. December, 2001.

[Ozsu y Valduriez, 1991] Ozsu, T. y Valduriez, P. (1991). "*Principles of Distributed Database Systems*". Prentice-Hall.

[Platt, 1999] Platt, D. S. (1999). "*Understanding COM+*". Microsoft Press.

[Rosenberry et al., 1992] Rosenberry, W., Kenney, D. y Fisher, G. (1992). "*Understanding DCE*". O'Reilly & Associates.

[Schmidt et al. 1999] Schmidt D. C., Levine D. L. y Cleeland C. (1999). "*Architectures and Patterns for Developing High-performance, Real-time ORB Endsystems*", Advances in Computers. Ed. Academic Press

[Sevilla, 1998] Sevilla Ruiz, D. (1998). "*Aplicaciones Distribuidas en Internet/Intranets: De los Sockets a los Objetos Distribuidos*". Proyecto de Final de Carrera. Universidad de Murcia.

[Siegel, 2000] Siegel, J. (editor). (2000). "*CORBA 3 Fundamentals and Programming*". 2nd edition. Jonh Wiley & Sons.

[Stroustrup, 1997] Stroustrup, B. (1997). "*The C++ Programming Language*". 3rd Edition, Addison Wesley.

[SUN, 2000] SUN Microsystems. (2000). "*Enterprise JavaBeans™ Specification, Version 2.0*". Version 2.0, Proposed Final Draft.

[SUN, 2001] SUN Microsystems. (2001). "*The Java Tutorial. A Practical Guide for Programmers*". http://java.sun.com/docs/books/tutorial/index.html. [Última vez visitado, 5/6/2001]. May, 2001.

[Tanenbaum, 1992] Tanenbaum, A. S. (1992). "*Modern Operating Systems*". Prentice-Hall.

[TAO, 2001] TAO. "*Real-Time CORBA with TAO*". http://www.cs.wustl.edu/~schmidt/TAO.html. [Última vez visitado 27-6-2001].

[Tari y Bukhres, 2001] Tari, Z. y Bukhres, O. (2001). "*Fundamentals of Distributed Object Systems. The CORBA Perspective*". John Wiley & Sons.