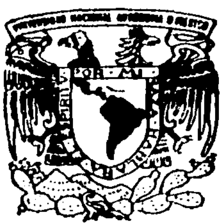
24021
20



UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
ACATLAN

"MODELOS ESTANDAR DE OBJETOS DISTRIBUIDOS APLICADOS EN LA ARQUITECTURA CLIENTE/SERVIDOR MULTICAPAS"

T E S I N A

QUE PARA OBTENER EL TITULO DE

LICENCIADA EN MATEMATICAS APLICADAS Y COMPUTACION

P R E S E N T A

JEZABEL ISAIAS ANGEL

ASESOR: LIC. ALEJANDRO ROBERTO RUBIO PEREZ

JULIO 2003

Autorizo a la Dirección General de Bibliotecas UNAM a difundir en formato electrónico e impreso el contenido de mi trabajo recepcional.
NOMBRE: Jezabel Isaias Angel
FECHA: 17-Jul-2003
FIRMA:

A

TESIS CON
FALLA DE ORIGEN

Universidad Nacional
Autónoma de México

Biblioteca Central

Dirección General de Bibliotecas de la UNAM

UNAM – Dirección General de Bibliotecas
Tesis Digitales
Restricciones de uso

DERECHOS RESERVADOS ©
PROHIBIDA SU REPRODUCCIÓN TOTAL O PARCIAL

Todo el material contenido en esta tesis esta protegido por la Ley Federal del Derecho de Autor (LFDA) de los Estados Unidos Mexicanos (México).

El uso de imágenes, fragmentos de videos, y demás material que sea objeto de protección de los derechos de autor, será exclusivamente para fines educativos e informativos y deberá citar la fuente donde la obtuvo mencionando el autor o autores. Cualquier uso distinto como el lucro, reproducción, edición o modificación, será perseguido y sancionado por el respectivo titular de los Derechos de Autor.

A la memoria de mi padre

Le agradezco todo el amor, apoyo y confianza que depositó en mi, sin ellos, este logro no hubiera sido posible.

Dios lo bendiga y lo tenga en su Gloria.

AGRADECIMIENTOS

A mi familia

A mi mamá por las enseñanzas de toda la vida y por su incondicional apoyo y amor. Te amo.

A mi hermana por apoyarme en la realización de este trabajo y por confiar siempre en mi. Te quiero.

A mi abuelita María por ser la mujer más admirable que he conocido. Gracias por tu cariño.

A mis sobrinos Javier y Erick por ser la más grande de mis alegrías.

A Xavier por sus enseñanzas, apoyo y amistad.

A mis amigos

A Carolina y Jennefer, por haber permanecido a mi lado durante todos estos años y seguir siendo parte importante de mi vida.

A Marisol por brindarme siempre una sonrisa y momentos tan agradables.

A Gabriel, Jaime e Isidro, porque representan para mi amistad, lealtad, confianza, alegría y lo invaluable que es el estar, siempre estar. Hicieron de la universidad la mejor de mis etapas de estudiante. Los quiero mucho.

A todos los que participaron en la realización de este trabajo especialmente a mi asesor Lic. En MAC Alejandro Rubio y a Gabriel García por su gran cariño y ayuda.

CONTENIDO

INTRODUCCIÓN 4

CAPITULO I. INTRODUCCIÓN A LA ARQUITECTURA MULTICAPAS 7

A. El modelo Cliente/Servidor *7*
1. Servidores de archivos 8
2. Servidores de datos 9
3. Servidores de transacciones 9
4. Servidores Groupware 10
5. Servidores de Aplicación de objetos 10
6. Servidores de Aplicación Web 10

B. Las capas en el modelo Cliente/Servidor *11*
1. Arquitectura Cliente/Servidor de dos capas 13
2. Arquitectura Cliente/Servidor multicapas 14
3. Comparación entre 2-capas y multicapas 16

C. Requerimientos de la arquitectura multicapas *17*

CAPITULO II. LOS MÉTODOS ESTÁNDARES DE OBJETOS DISTRIBUÍDOS 23

A. Principios básicos *23*

B. Objetos distribuidos *23*
1. Introducción 23
2. Beneficios 24

C. Objetos distribuidos y componentes *25*
1. Objetos 25
2. Componentes 26
3. Componentes del servidor. 29
4. Objetos comerciales. 30

D. Objetos distribuidos en la arquitectura multicapas. *31*

CAPITULO III. LA CAPA INTERMEDIA ORIENTADA A OBJETOS 36

A. Redes de computadoras *36*
1. Modelo ISO/OSI 36
2. Capa de transporte. 39

B. Tipos de capa intermedia en la arquitectura multicapas. *41*
1. Capa intermedia orientada a transacciones. 42
2. Capa intermedia orientada a mensajes. 43
3. Llamadas de procedimiento remoto (RPC) 43

C. La capa intermedia orientada a objetos *46*

D. Definición de la interfaz *48*

CAPITULO IV. EL MODELO COM EN LA ARQUITECTURA MULTICAPAS **49**

A. Los componentes COM *49*
1. Historia de la capa intermedia de Microsoft 49
2. El Modelo de distribución. 53

B. El modo de ejecución COM+ *59*
1. Introducción de los componentes al ambiente COM+ 60
2. Características de activación del componente de ejecución. 60

C. Los servicios de COM+ *61*
1. Diferencia entre los componentes síncronos y asíncronos. 62
2. Arquitectura DCOM. 65

D. Interoperabilidad COM *70*
1. Interoperabilidad de la capa cliente 70
2. Interoperabilidad de la capa intermedia 73
3. Interoperabilidad de la base de datos 75

CAPITULO V. EL MODELO CORBA EN LA ARQUITECTURA MULTICAPAS **77**

A. Introducción *77*

B. Historia de la OMG *78*

C. Arquitectura *81*
1. Comunicación entre objetos 84
2. Lenguaje de Definición de Interfaces (IDL) 86
3. El modelo de comunicaciones CORBA 88
4. El modelo de objetos CORBA 89
5. Enlaces entre la interfaz y el lenguaje 91
6. Servicios CORBA y facilidades CORBA. 92

D. Orbix y Visibroker *96*

E. Las capas del modelo CORBA *100*
1. Capa Superior. Arquitectura Básica de Programación 100
2. Capa Intermedia. Arquitectura de Proceso Remoto 101
3. Capa Inferior: Arquitectura del Protocolo de Comunicación 102

CONCLUSIONES **103**

APÉNDICE **106**

BIBLIOGRAFIA **110**

ANEXO I. Comparación DCOM vs. CORBA mediante un ejemplo **113**

A. Capa Superior. Arquitectura Básica de Programación *118*

B. Capa Intermedia. Arquitectura de Proceso Remoto *120*

C. Capa Inferior: Arquitectura del Protocolo de Comunicación *121*
D. Interoperabilidad CORBA/DCOM *121*
E. Resumen *122*

INTRODUCCIÓN

En la última década, la informática en línea (por ejemplo consultas de catálogos y ventas en internet) ha desempeñando un papel muy importante dentro de empresas de negocios y otras organizaciones no lucrativas. Además de ser un elemento de apoyo para la realización de operaciones básicas, también se ha constituido en un medio para obtener ventajas competitivas o de incremento de servicios.

Es por ello, que las aplicaciones deben ser desarrolladas con la misma velocidad con la que los requerimientos del negocio cambian. Hoy en día se está dando mucho énfasis en la importancia de contar con información accesible, oportuna, confiable y que esté disponible cuando se necesita.

Las nuevas aplicaciones deben basarse en tecnologías que disminuyan los costos de desarrollo y mantenimiento relacionados al hardware, software, operación y entrenamiento de personal. Una de las arquitecturas que responden a esas necesidades actuales es la ***Cliente/Servidor.***

La arquitectura Cliente/Servidor es un modelo para el desarrollo de sistemas de información, en el que las transacciones se dividen en procesos independientes que cooperan entre sí para el intercambio de información, recursos y servicios. En este modelo, las aplicaciones se dividen de forma tal que el servidor contiene los datos que deben ser compartidos por varios usuarios, y en el cliente permanece sólo lo particular de cada usuario.

La arquitectura Cliente/Servidor varía dependiendo de dónde se ejecutan los diferentes elementos involucrados:

- Administración de los datos.
- Lógica de la aplicación.
- Lógica de la presentación.

Dependiendo de la distribución de los elementos anteriormente citados se definen dos variantes conocidas como arquitectura Cliente/Servidor de 2 capas y 3 capas. A principios de los 80's, algunos proveedores de minicomputadoras introdujeron el término 3 capas para describir la división física de una aplicación a través de terminales (capa 1), minicomputadoras (capa 2) y mainframes (capa 3), permitiéndoles así vender sus computadoras de medio alcance como front-ends de mainframes.

En la actualidad, se utiliza el concepto ***capas*** para describir la división lógica de una aplicación entre clientes y servidores. 2 capas divide el procesamiento en dos partes, en la mayoría de los casos, la lógica de la aplicación se ejecuta en el cliente, quien por lo general envía solicitudes SQL al servidor de base de datos. En 3 capas (o **multicapas** como será nombrada en este trabajo) la interfaz gráfica del usuario corre en el cliente (capa 1), la lógica de la aplicación en el servidor (capa 2) y la administración de la base de datos puede encontrarse en el mismo servidor de la capa 2 o bien en otro servidor (capa 3).

La arquitectura multicapas puede ser diseñada con diferentes tecnologías, ya sean monitores de transacciones, sockets, RPCs, objetos distribuidos, etc. El crecimiento explosivo de internet, así como la popularidad alcanzada por los equipos PC y los avances en redes de alta velocidad, han hecho del proceso distribuido un objetivo prioritario para los desarrolladores de sistemas, ubicando a los objetos distribuidos como una excelente alternativa para el desarrollo de aplicaciones Cliente/Servidor multicapas.

En el presente trabajo se introducen los conceptos fundamentales para entender los dos **modelos estándares de objetos distribuidos** más populares que existen en el mercado: COM+ y CORBA. También se examinan las diferencias entre estos dos modelos para así apreciar mejor las características de cada uno de ellos.

En el capítulo I se definirán conceptos de Cliente/Servidor, 2-capas, multicapas y los requerimientos que conducen a la adopción de la arquitectura Cliente/Servidor multicapas.

En el capítulo II se introducirá al mundo de los objetos y los componentes a través de la definición de sus características, esto con la finalidad de entender los conceptos generales que involucran a COM+ y CORBA.

En el capítulo III se describirán los tipos de capas intermedia que existen en el modelo de objetos distribuidos y en la arquitectura multicapas. En este capítulo se presenta una definición básica del modelo de referencia OSI de ISO para entender mejor el funcionamiento de los modelos estándar de objetos distribuidos que trabajan fundamentalmente en las capas de sesión y de transporte.

En los capítulos IV y V se definirán los modelos COM+ y CORBA respectivamente. Estos dos últimos capítulos se complementan con la presentación de un programa ejemplo escrito en el lenguaje C++ presentado en el Anexo I, en él se muestran las principales similitudes y diferencias entre ambos modelos.

En la última parte a manera de apéndice, se exponen una serie de definiciones como consulta a aquellas personas que les pudiera surgir alguna duda a lo largo de la lectura de este trabajo. Principalmente son términos comúnmente utilizados en la Programación Orientada a Objetos, tales como Stub, instancia, marshal, etc. También el apéndice incluye la definición de acrónimos como COM, CORBA, EJB, GUI, etc.

Para una mejor comprensión de este trabajo es importante que el lector esté familiarizado con los conceptos del modelo OSI, TCP/IP, ActiveX y Programación Orientada a Objetos. Para la lectura de los ejemplos que se incluyen en los capítulos III, IV, V y en el Anexo I, se recomienda conocer algún lenguaje de programación, de preferencia orientado a objetos (Java, C++ o en su defecto Delphi).

CAPITULO I

INTRODUCCIÓN A LA ARQUITECTURA MULTICAPAS

A. El modelo Cliente/Servidor

A pesar de que cliente/servidor es una expresión estereotipada y popular en la industria, aún no se ha llegado a un acuerdo para definir exactamente este termino[1]. Como su nombre lo dice, clientes y servidores son entidades lógicas por separado que trabajan en conjunto dentro de una red con la finalidad de terminar una tarea. Las características más importantes que tienen todos los sistemas cliente/servidor son:

- *Servicio:* cliente/servidor es una relación de procesos corriendo en máquinas por separado, estos procesos se encuentran disponibles en el servidor para que el cliente haga uso de ellos cuando sea necesario. Es decir, el proceso servidor es un proveedor de servicios y el cliente es un consumidor de servicios.
- *Recursos Compartidos:* Un servidor puede trabajar con muchos clientes al mismo tiempo y controlar su acceso a los recursos compartidos.
- *Protocolos asimétricos*: Hay una relación muchos-a-uno entre los clientes y el servidor. Los clientes siempre inician la comunicación a través de la solicitud de un servicio. Los servidores esperan la petición de los clientes. Un componente es cliente de algún otro componente y éste puede ser también un servidor de otro componente, a esta característica se le conoce como asimetría.
- *Transparencia en la ubicación*: El servidor es un proceso que puede residir en la misma máquina o en una máquina diferente dentro de la red. El software cliente/servidor por lo general enmascara la ubicación del servidor a través del

---

[1] Para una definición más detallada del modelo cliente/servidor recomiendo ***Client/Server Survival Guide***, Robert Orfali et al. (Wiley Computer Publishing, 1999)

redireccionamiento de las llamadas de servicio. Bajo este contexto, una máquina puede tener un cliente, un servidor, o ambos.

- *Intercambio de mensajes:* Los clientes y los servidores interactúan por medio de un mecanismo de envío de mensajes. El mensaje es el mecanismo de intercambio para los servicios de petición y respuesta.
- *Encapsulamiento de servicios:* Un mensaje le informa a un servidor que servicio es el solicitado; es trabajo del servidor el determinar como responderá la petición. Los servidores pueden ser actualizados sin afectar a los clientes mientras no se cambie la interfaz de mensajes.
- *Escalabilidad:* Los sistemas cliente/servidor pueden expandir su capacidad de funcionamiento horizontalmente o verticalmente para mejorar su rendimiento. La escalación horizontal significa añadir o remover estaciones cliente teniendo un pequeño impacto en el funcionamiento del sistema. La escalación vertical significa ya sea migrar a un servidor más grande y rápido o distribuir el procesamiento a través de varios servidores.
- *Integridad:* El código y datos del servidor son administrados centralmente, lo que resulta más económico de mantener y protege los datos a compartir conservando su entereza. Al mismo tiempo que los clientes permanecen en forma independiente.

La arquitectura cliente/servidor ha sido implementada por una gran diversidad de vendedores que han desarrollado diferentes tecnologías: Servidores de archivos, Servidores de datos, Servidores de Transacciones, Servidores Groupware, Servidores de Aplicación de objetos y Servidores de Aplicación Web. A continuación se define cada uno de ellos.

1. Servidores de archivos

Un servidor de archivos es una combinación de hardware y software que permite a los usuarios compartir programas y datos dentro de una red, tales como documentos, imágenes, sonidos, etc. Por lo general, un servidor de archivos tiene significativamente un mejor procesador, una tarjeta de red más rápida, más memoria y mayor capacidad de almacenamiento que la mayoría de sus clientes en una red. En este tipo de servidor, el cliente (por lo general una PC) pasa las solicitudes de registros de archivos a través de

una red al servidor. Se trata de una forma primitiva de servicio de datos que necesita mucho intercambio de mensajes sobre la red para encontrar los datos solicitados.

2. Servidores de datos

En los servidores de datos, el cliente pasa solicitudes SQL como mensajes al servidor de bases de datos. Los resultados de cada comando SQL son regresados a través de la red. El código que procesa la solicitud SQL y los datos son almacenados en la misma máquina. El servidor utiliza su propio poder de procesamiento para encontrar los datos solicitados en vez de permitirle ver todos los registros al cliente y luego dejarlo encontrar por su propia cuenta los datos, como es el caso de los servidores de archivos. El resultado es mayor eficiencia en el uso del poder de procesamiento distribuido. Los servidores de datos juegan un papel muy importante en el data-warehousing.

3. Servidores de transacciones

En este tipo de servidor, el cliente invoca procedimientos remotos (o servicios) que residen en el servidor con una ingeniería de base de datos SQL. Estos procedimientos remotos en el servidor ejecutan un grupo de sentencias SQL. Las sentencias SQL fracasan o tienen éxito como una unidad (todo o nada), a todas estas sentencias agrupadas se les conoce como *transacciones*.

Con un servidor de transacciones, se puede crear una aplicación cliente/servidor codificando en ambos lados: cliente y servidor. El cliente por lo general incluye una Interfaz gráfica y el servidor consiste en transacciones SQL. Este tipo de aplicaciones son llamadas *On Line Transaction Processing* (Procesamiento transaccional en línea), u OLTP. Éstos tienden a ser aplicaciones de misión crítica que requieren de un tiempo de respuesta de 1-3 segundos el 100% de las veces. Las aplicaciones OLTP también requieren controlar la seguridad e integridad de la base de datos.

OLTP se divide en varias capas. La primera es la capa de datos y está compuesta por todos los datos requeridos por la organización de la empresa para completar las transacciones que ésta requiera. La capa de datos puede estar almacenada en SQL Server, Oracle, en manejadores de datos no relacionales como Microsoft Exchange, así

como cualquier otro tipo de servidor de datos. La siguiente capa es la que involucra las reglas y la lógica del negocio, abarca tecnologías que facilitan las transacciones de una empresa, como Exchange Server, Internet Information Server (IIS), entre otras. La última capa es la de presentación, ésta consiste en una interfaz gráfica de la funcionalidad actual de la capa de negocios y de datos.

4. Servidores Groupware

Los servidores Groupware ponen a la gente en contacto directo con otras personas. Estos sistemas cliente/servidor direccionan el manejo de información semiestructurada como el texto, imágenes, correos, carteles de anuncios, y el flujo del trabajo (workflow). Lotus Notes y Microsoft Exchange son ejemplos de este tipo de sistemas.

5. Servidores de Aplicación de objetos

Este tipo de servidor está construido por un objeto servidor, donde la aplicación cliente/servidor está escrita como un conjunto de objetos que se comunican. Los objetos clientes se comunican con los objetos servidores utilizando un ORB[2] (Object Request Broker), esto es, el cliente invoca un método en un objeto remoto donde el ORB localiza una instancia de la clase del objeto servidor e invoca el método solicitado regresando los resultados al objeto cliente. Los objetos servidores deben proporcionar soporte para la concurrencia y compartición. Ejemplos de ORBs comerciales son Orbix de IONA, VisiBroker de Inprise, DAIS de ICL, Kava IDL de JavaSoft, ObjectBroker de BEA, SOM de IBM y PowerBroker de Expertsoft.

6. Servidores de Aplicación Web

Este tipo de servidores es todavía un modelo nuevo de cliente/servidor que consiste en un cliente delgado, portable y "universal" que interactua con *servidores pesados*[3]. Los clientes y los servidores se comunican utilizando un protocolo tipo RPC llamado HTTP.

[2] En el capítulo V se definirá a detalle éste tipo de servidores.

[3] Término que se define en el inciso B del presente capítulo.

Este protocolo define un conjunto simple de comandos con parámetros que son pasados como cadenas.

La Web y los objetos distribuidos se están convirtiendo en un fuerte eslabón para proporcionar una forma interactiva de computación cliente/servidor. A ésta nueva convergencia se le llama Objeto Web. Dentro del universo de Microsoft, el objeto Web es un objeto basado en componentes ActiveX que se comunican vía COM+[4] o HTTP. También existen los Java applets y los browsers CORBA.

B. Las capas en el modelo Cliente/Servidor

Los modelos cliente/servidor pueden distinguirse por el servicio que proveen. Las aplicaciones cliente/servidor pueden ser diferenciadas por la manera en como la lógica se encuentra distribuida entre el cliente y el servidor.

Existen dos modelos de servidor principalmente. El *modelo de servidor pesado* le añade mayor funcionalidad al servidor. El *modelo de cliente pesado* realiza lo contrario. Groupware y servidores Web son ejemplos de servidores pesados, servidores de base de datos y servidores de archivos (file servers) son ejemplos de clientes pesados. Los objetos distribuidos pueden ser cualquiera de ellos.

Los *clientes pesados* son la forma más tradicional de cliente/servidor. El cuerpo de la aplicación corre en el cliente. En los modelos servidor de archivos y servidor de base de datos, los clientes saben como están organizados y almacenada la información en el servidor. Ellos proporcionan flexibilidad y oportunidades para crear herramientas front-end que permiten a los usuarios finales crear sus propias aplicaciones.

[4] En el capítulo IV se definirá a detalle ésta tecnología.

Las aplicaciones de *servidores pesados* son más fáciles de administrar y desarrollar en una red porque la mayoría del código corre en los servidores. Los servidores pesados intentan minimizar el tráfico de red por medio de la creación de niveles abstractos de servicio. Los servidores de transacciones y de objetos, por ejemplo, encapsulan la base de datos. En vez de exportar los datos, ellos exportan los procedimientos (o métodos en la terminología de orientación a objetos) que trabajan con esos datos. El cliente en el modelo de servidores pesados provee la interfaz gráfica (GUI) e interactua con el servidor a través de la llamada de procedimientos remotos (o invocación de métodos).

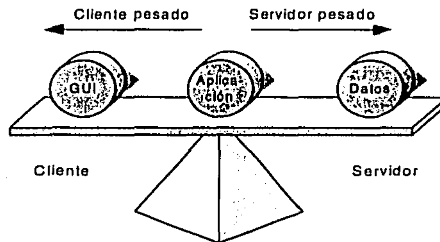


Fig. 1.1 Cliente pesado contra Servidor pesado

Los expertos en cliente/servidor prefieren utilizar los términos arquitectura de 2-capas y multicapas en vez de clientes pesados y servidores pesados, pero son básicamente la misma idea. Todo depende de la división de la aplicación cliente/servidor en unidades funcionales que puedan ser distribuidas tanto en el cliente como en uno o más servidores. Las unidades funcionales más comunes son, la interfaz del usuario, las reglas del negocio[5] y los datos compartidos. Existe una gran variedad de arquitecturas multicapas que dependen de la distribución de las aplicaciones y de la capa intermedia. En 2-capas la mayor parte de la lógica de la aplicación se ejecuta en el cliente, quien normalmente envía peticiones SQL a un servidor de base de datos, se le llama a esta arquitectura *cliente pesado* porque una gran parte de la aplicación se ejecuta en el cliente. En multicapas por lo regular el servidor ejecuta las reglas del negocio y los datos

[5] Restricciones que determinan la manera en que los datos serán capturados, relacionados y representados dentro de un sistema de base de datos.

TESIS CON
FALLA DE ORIGEN

compartidos, mientras los clientes ejecutan la interfaz gráfica, es por estas razones que en algunas ocasiones es llamado *servidor pesado.*

1. Arquitectura Cliente/Servidor de dos capas

En los sistemas cliente/servidor de 2-capas, la lógica de la aplicación se puede encontrar en el cliente o en el servidor de datos (o en ambos). En el cliente se corre una interfaz gráfica que envía las llamadas al sistema de datos, que consisten generalmente en comandos SQL o HTTP, dentro de una red al servidor (véase figura 1.2). El servidor procesa las peticiones del cliente y regresa los resultados. Para accesar a los datos, los clientes deben saber la forma en como están organizados y almacenados en el servidor. Una variación de 2-capas es el uso de stored procedures. En vez de enviar peticiones SQL a través de la red, los stored procedures permiten invocar una función que corre dentro de la base de datos.

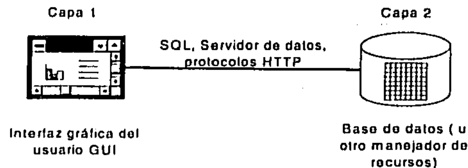


Fig. 1.2 Arquitectura Cliente/Servidor de 2 capas

2-capas está caracterizado por su facilidad de desarrollo utilizando herramientas visuales tales como Visual Basic de Microsoft, Delphi de Borland y Power Builder de Sybase. Por lo general, se trata de aplicaciones departamentales[6] o simples aplicaciones Web cuyo número de usuarios oscila entre la docena y 100 dentro de una LAN. Cuando las aplicaciones departamentales empezaron a expandirse, los arquitectos comenzaron a depender en 2-capas para aplicaciones de misión crítica. Pronto descubrieron que las arquitecturas y herramientas que utilizaban para 2-capas no eran escalables. Las aplicaciones que trabajaban perfectamente en prototipos y pequeñas instalaciones no funcionaban en la producción a gran escala.

[6] Aplicaciones desarrolladas en su mayoría por herramientas visuales que se caracterizan por ser útiles a ciertas áreas de una empresa determinada, tales como prototipos y groupwares a pequeña escala.

TESIS CON
FALLA DE ORIGEN

Con el crecimiento de internet, los requerimientos de desarrollo se vieron influenciados en el desarrollo de aplicaciones para el comercio electrónico. Consecuentemente, se tuvo que cambiar del tradicional cliente/servidor de 2-capas a una arquitectura que divide las aplicaciones en componentes y los distribuye en varios procesadores, surgiendo así la arquitectura multicapas.

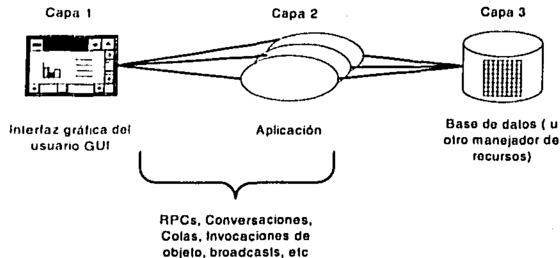
2. Arquitectura Cliente/Servidor multicapas

La respuesta de la industria ante las limitaciones de la arquitectura de 2-capas fue la de añadir una capa intermedia, ubicada entre el dispositivo de entrada-salida y el servidor de datos. La capa intermedia puede realizar varias funciones como ejecución de aplicaciones y comunicarse con la base de datos. El cliente puede enviar sus peticiones a la capa intermedia, liberarse y asegurarse de que recibirá la respuesta apropiada a su petición. La capa intermedia añade planificación y jerarquización de la tarea en proceso. El uso de una arquitectura con esta capa intermedia es llamada 3-capas o multicapas. Estos dos últimos términos son sinónimos en este contexto.

Una aplicación multicapas es un programa que está organizado en al menos tres partes, donde cada una de ellas están distribuidas en un lugar o lugares diferentes dentro de una red. Las tres partes principales son:

- ✓ La estación de trabajo (cliente)
- ✓ La lógica del negocio (aplicación servidora)
- ✓ La base de datos y los programas relacionados para su administración.

En una aplicación multicapas típica, la estación de trabajo contiene el programa que proporciona la interfaz gráfica (GUI) y las formas de entrada específicas o ventanas interactivas, así como algunos datos locales que son únicos de la estación de trabajo y que pueden estar almacenados en el disco duro. Una ventaja de multicapas es que provee mejor seguridad al no exponer el esquema de base de datos al cliente.



Las reglas del negocio están localizadas en un servidor dentro de una red de área local (LAN) o en cualquier otra computadora compartida. Esta capa determina que datos son necesarios (y donde se encuentran almacenados) y actúa como un cliente en relación a una tercer capa que pudiera estar localizada en otra computadora, servidor o mainframe. La arquitectura Multicapas sustituye pocas llamadas al servidor por muchas consultas SQL y actualizaciones, por lo que se desempeña mejor que 2-capas.

La tercer capa incluye la base de datos y los programas que administran los accesos de lectura y escritura a la misma.

Una aplicación multicapas usa el modelo cliente/servidor con al menos tres capas o partes, cada parte puede ser desarrollada por diferentes equipos de programadores en diferentes lenguajes. Debido a que la programación de una capa puede ser cambiada o redireccionada sin afectar otras capas, el modelo multicapas facilita escalar las aplicaciones continuamente ajustándose de manera fácil a las nuevas necesidades que se vayan presentando. Las aplicaciones existentes o partes críticas pueden ser permanentemente o temporalmente guardadas y encapsuladas dentro de una nueva capa convirtiéndose así en un componente. La arquitectura multicapas es consistente con las ideas de la programación orientada a objetos.

TESIS CON
FALLA DE ORIGEN

3. Comparación entre 2-capas y multicapas

Las diferencias entre 2-capas y multicapas han sido resumidas en la tabla[7] que se presenta a continuación, para el desarrollo de aplicaciones cliente/servidor no departamentales:

Característica	2-capas	Multicapas
Administración del sistema	Complejo (Requiere mayor lógica en el cliente para administrar el sistema)	Menos complejo (la aplicación puede ser administrada en forma centralizada dentro del servidor)
Seguridad	Poca (a nivel de datos)	Alta (a nivel del servicio)
Encapsulamiento de datos	Bajo (las tablas están expuestas)	Alta (el cliente utiliza métodos o servicios)
Desempeño	Pobre (exceso de tráfico de red)	Bueno (las peticiones de servicio y las respuestas son el tráfico principal)
Escalabilidad	Pobre (administración limitada de la comunicación de los clientes)	Excelente (puede distribuir el trabajo a través de servidores múltiples)
Reuso	Pobre (aplicaciones monolíticas en el cliente)	Excelente (puede reusar los servicios y objetos)
Facilidad de desarrollo	Alta (gran diversidad de lenguajes de programación visuales: Visual Basic, Delphi, Power Builder, etc.)	Mejorando (Las herramientas estándar pueden ser usadas para la creación de los clientes, y las aplicaciones en el servidor están haciéndose cada vez más fáciles)
Infraestructura servidor a servidor	No	Si (Capa intermedia en el servidor)
Integración de Legado de aplicaciones	No	Si (gateways encapsulados por servicios u objetos)
Soporte Internet	Pobre (el ancho de banda limitado hace difícil bajar clientes pesados)	Excelente (clientes delgados son más fáciles de bajar)
Soporte de Base de Datos heterogéneo	No	Si (puede usar varias bases de datos dentro de la misma transacción)

[7] Tabla extraída del libro ***3-Tier Client/Server At Work***, Robert Orfali et al. (Wiley Computer Publishing, 1999)

TESIS CON
FALLA DE ORIGEN

Característica	2-capas	Multicapas
Amplias alternativas de comunicación	No (solo síncrona, conexión orientada a llamadas RPC)	Sí (soporta llamadas RPC, pero puede soportar mensajes sin conexión, reparto de colas, emisión y publicación-y-suscripción de mensajes)
Flexibilidad en la arquitectura de Hardware	Limitada (se tiene un cliente y un servidor)	Excelente (todas las capas pueden estar en diferentes equipos o no, la capa intermedia puede estar en múltiples servidores)
Disponibilidad	Pobre (pude fallar cuando se encuentre en proceso de respaldo el servidor)	Excelente (puede reiniciar los componentes de la capa intermedia en otros servidores de ser necesario)

C. Requerimientos de la arquitectura multicapas

Para identificar los factores que influyen en la elección de la arquitectura multicapas, me voy a basar en el proceso de ingeniería de software. El propósito de aplicar el proceso de ingeniería de software es el encontrar lo que el cliente realmente quiere y necesita. Los resultados de la conceptualización y las fases de análisis determinan fases futuras en el proceso. Para este propósito me voy a basar en un proceso de desarrollo de sistemas propuesto por Booch en 1995[8] que incluye una fase de requerimientos donde el sistema es conceptualizado y donde los requerimientos del usuario son determinados (Véase Figura 1.4). El método de Booch es uno de los más reconocidos en el ámbito del desarrollo de sistemas orientados a objetos porque está enfocado al desarrollo de sistemas asociados con el paradigma de objetos. Booch propuso *una metodología de implementación de sistemas* enfocada al desarrollo de sistemas de objetos distribuidos que ha sido bien aceptada por los programadores de esta tecnología, además de ser coautor de la tecnología UML (Unified Modeling Language).

TESIS CON
FALLA DE ORIGEN

[8] Booch, G. (1995). Object-Solutions:Managing the Object-Oriented Project. Addison Wesley.

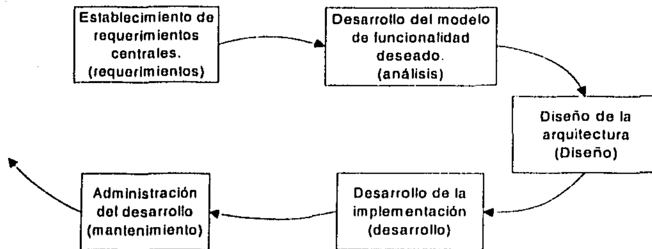


Fig. 1.4 Proceso de desarrollo de Booch

La fase de requerimientos involucra la identificación de las propiedades *funcionales* y *no funcionales* que se demandan en el sistema. Los requerimientos *funcionales* competen con las funciones que los sistemas pueden ejecutar para sus usuarios, y pueden estar localizados en componentes. Siguiendo el método de Booch, los requerimientos funcionales son formalizados durante el análisis de orientación a objetos. Los requerimientos *no funcionales* están relacionados con las cualidades del sistema. Los requerimientos no funcionales tienen un fuerte impacto sobre la arquitectura a elegir para el sistema, esto se hace durante la fase de diseño.

Los requerimientos no funcionales, que por lo general conducen a la adopción de sistemas multicapas son:

1. Escalabilidad.
2. Apertura
3. Heterogeneidad
4. Compartición de recursos
5. Manejo de errores

TESIS CON FALLA DE ORIGEN

Escalabilidad

Cuando se diseña un sistema, un factor muy importante que los ingenieros tienen que tomar en cuenta es la carga[9] que los usuarios del sistema van a generar. La carga que el sistema va a soportar puede ser determinada en varias dimensiones. Puede ser expresada por el número máximo de usuarios concurrentes; también puede ser definida por el número de transacciones que el sistema necesita ejecutar en un período dado y la carga también puede ser determinada por medio de la estimación del volumen de datos que deberá manipular el sistema.

En un sistema deseamos ciertas cualidades en el servicio, éstas se refieren a la manera en como se desempeña el sistema aún cuando exista una gran carga de trabajo. Por ejemplo, en un banco se desea que una transacción sea completada dentro de un par de segundos, de lo contrario el servicio será considerado como inapropiado. Las arquitecturas de software deben ser diseñadas de tal modo que sean estables durante el tiempo de vida del sistema, es decir, no solamente deben estar disponibles para soportar la carga cuando el sistema está en producción, sino que se debe realizar una predicción de cómo se comportará la carga durante el tiempo de vida del sistema. Esta predicción puede resultar un poco difícil; por ejemplo, cuando se diseñó Internet, nadie estimó el número de nodos concurrentes que necesitaría, una de las razones del éxito de internet fue que puede crecer al mismo tiempo que la carga de trabajo y esto es posible debido a su arquitectura **escalable**. En general, se dice que una arquitectura es escalable si los sistemas pueden crecer con la misma facilidad que la carga.

Los requerimientos de escalabilidad por lo general conducen a la adopción de sistemas distribuidos. Los sistemas distribuidos son escalables porque pueden añadir computadoras que almacenan componentes adicionales; traducido esto a términos de la arquitectura multicapas, se pueden añadir tantas aplicaciones servidoras (capas intermedias) como sea necesario.

[9] Considérese la carga igual a la carga de trabajo del sistema: número de usuarios, número de transacciones y volumen.

Apertura

Otra propiedad no funcional que se busca en todo software es que éste sea abierto. La apertura significa que el sistema puede ser fácilmente ampliado y modificado. Para facilitar la apertura, los componentes del sistema necesitan tener bien definidas y bien documentadas las interfaces. La construcción del sistema necesita adecuarse a estándares para que los componentes del sistema puedan ser substituidos sin convertirse en dependientes de un proveedor en particular.

Al integrar nuevos componentes al sistema se desea que ellos puedan comunicarse con otros componentes ya existentes, para esto, los componentes deben contar con interfaces bien definidas y la declaración de los servicios (operación que un componente ejecuta a petición de un usuario u otro componente). Los servicios son comúnmente parametrizados y los parámetros de los servicios necesitan también especificarse dentro de la interfaz. Un componente cliente utiliza a la interfaz para solicitar un servicio a un componente servidor, nótese que un componente que es cliente de algún otro componente puede ser así mismo un servidor de cualquier otro componente. Los componentes deben ser reactivos y tener que responder a la ocurrencia de eventos que son detectados en otros componentes.

La apertura y la distribución están relacionadas, los componentes dentro de un sistema distribuido tienen que declarar sus servicios para que otros componentes puedan utilizarlos. En un sistema centralizado, los componentes también deben declarar sus interfaces de tal manera que otros sistemas puedan usarlos, este uso, sin embargo está limitado a llamadas de procedimiento. Esto restringe la solicitud de los componentes de varias maneras, una solicitud en un sistema distribuido es más flexible porque pueden ser utilizados desde cualquier otra máquina. Pueden ser ejecutados de manera síncrona o asíncrona[10]. Finalmente, el solicitante y el servidor pueden ser construidos de manera heterogénea.

[10] Ésta característica será descrita en el siguiente capítulo.

Hetereogeneidad

La heterogeneidad de los componentes surge del uso de diferentes tecnologías para la implementación de los servicios, para el manejo de los datos y para la ejecución de los componentes en el hardware. Esto significa que los componentes pueden ser construidos utilizando diferentes lenguajes de programación y operados en diferentes plataformas.

La implementación de componentes heterogéneos, implica la construcción de sistemas distribuidos. Si el sistema incluye componentes que necesitan ser ejecutados en sus plataformas nativas, los componentes deben permanecer en ellas. Para que los usuarios utilicen el conjunto de componentes tienen que comunicarse a través de la red y la heterogeneidad tiene que ser resuelta durante esa comunicación.

Compartición de recursos

La palabra recursos, denota hardware, software y datos. El compartir recursos ocurre como consecuencia de comunicación y colaboración entre usuarios. El hecho de que los componentes no sean utilizados por un solo usuario en una máquina tiene muchas consecuencias de seguridad. Se tiene que definir quien tiene permitido el acceso a la información compartida dentro de un sistema multicapas. Como segundo requerimiento, se debe controlar el acceso al recurso que necesita ser compartido. El derecho a accesar un recurso es implementado por los manejadores de recursos, que no son más que componentes que tienen gran acceso al recurso compartido, los usuarios de estos manejadores pueden actuar en diferentes formas dentro de una arquitectura multicapas. En la arquitectura cliente/servidor, existen servidores que administran y proporcionan ciertos recursos y clientes que los utilizan. Un objeto distribuido representa y encapsula un recurso que se utiliza para proveer servicios a otros objetos. Los objetos que proveen servicios, a su vez, dependen de otros servicios proporcionados por otros objetos. Esto conduce a la construcción de una arquitectura que tiene varias capas (multicapas).

Manejo de errores

Un requerimiento no funcional muy importante es el manejo de errores, lo que significa que un sistema continúe trabajando, aún en caso de error. Este tipo de requerimiento por

lo general conduce a la adopción de la arquitectura multicapas. Los sistemas multicapas por lo general son mejores en el manejo de errores que los sistemas centralizados. Por ejemplo, si a un cliente le falla alguno de sus procesos, esto no significa, que los otros clientes no puedan continuar trabajando, esto se logra a través de la integración redundante de componentes que trabajan igual a la replicación. Si un componente falla, la réplica de ese componente puede entrar en su lugar y continuar el trabajo. Dado esto, existen más hosts en un sistema multicapas, donde cada uno de ellos corre en varios procesos.

CAPITULO II

LOS MÉTODOS ESTÁNDARES DE OBJETOS DISTRIBUÍDOS

A. Principios básicos

Se dice que la computación es *distribuida* cuando la programación y los datos con los que las computadoras trabajan son separados en más de una computadora, usualmente dentro de una red. En la actualidad la tendencia es moverse a la arquitecura cliente/servidor la cual, como se mencionó en el capítulo anterior, ésta se logra cuando una computadora cliente puede suministrar ciertas capacidades a un usuario y demandar otras desde otra computadora que provee servicios a los clientes.

B. Objetos distribuidos

1. Introducción

Hoy en día, personas asociadas con el mundo de la computación conocen o han oido hablar acerca del término "objeto". Actualmente la ingeniería de software se encuentra orientada hacia las metodologías de objetos y todo lo relacionado con la Programación Orientada a Objetos (OOP). El propósito de este capítulo es describir lo que los objetos pueden hacer por los sistemas cliente/servidor partiendo desde el concepto de "objetos distribuidos", y definiendo los conceptos de objeto y componente. Éstos términos son la base para entender los siguientes capítulos donde se definirán COM+ y CORBA, modélos

estándar que se pueden aplicar en el desarrollo de aplicaciones multicapas orientadas a objetos.

La tecnología de objetos permite unir sistemas de información cliente/servidor complejos simplemente ensamblando y extendiendo componentes de software reusables. Cualquiera de los objetos podrá ser modificado o reemplazado sin afectar el resto de los componentes del sistema o su manera de interactuar.

2. Beneficios

La tecnología de objetos distribuidos está diseñada para crear sistemas cliente/servidor flexibles, los datos y las reglas del negocio están encapsulados en objetos, lo que permite que éstos puedan ser colocados en cualquier parte dentro de un sistema distribuido. Además los objetos distribuidos (CORBA, Java RMI y COM+)[11] son altamente introspectivos, lo que significa que pueden informar cualquier cosa sobre sí mismos. Esta capacidad permite que herramientas visuales y otras aplicaciones descubran interfaces de objetos, eventos y propiedades.

Los objetos distribuidos tienen su potencial en permitir que varios componentes puedan ser ensamblados visualmente por medio de herramientas, interoperar dentro de redes, correr en diferentes plataformas, transitar en la red y administrarse a sí mismos y a los recursos que ellos controlan. Los objetos son entidades intrínsecamente automanejables. Los objetos nos permiten trabajar con sistemas muy complejos por medio de la difusión de instrucciones y alarmas. Cada objeto receptor deberá reaccionar de manera distinta al mensaje basado en su tipo de objeto.

Las aplicaciones cliente/servidor orientadas a objetos pueden ser más flexibles que las aplicaciones tradicionales. La estructura de las aplicaciones permiten a los usuarios finales mezclar componentes sin hacer la aplicación distribuida pesada. Además, los objetos distribuidos tienen la propiedad única de separar sus interfaces de la

[11] Métodos estándares de objetos distribuidos. En el caso de COM+ se profundizará en el capítulo IV y CORBA en el capítulo V del presente trabajo.

implementación. Esto significa que se pueden usar interfaces de objetos para cubrir las aplicaciones existentes y hacerlas parecer objetos ordinarios.

C. Objetos distribuidos y componentes

1. Objetos

Un objeto, ya sea diseñado en el lenguaje de programación C++, Java o Smalltalk, es una estructura de programación que agrupa información (datos) junto con las operaciones que la manipulan. Todo objeto tiene un conjunto de atributos que representan el estado del mismo. La información de un objeto se almacena en estructuras de datos que se conocen como variables de instancia (globales) y conforman el estado interno del objeto. Las operaciones que un objeto es capaz de realizar se denominan métodos de instancia y determinan su comportamiento.

El concepto de objeto de software es similar al del objeto del mundo real. Por ejemplo, una pelota consta de:

Un ***estado*** o atributos particulares como son el diámetro, color y elasticidad.

Un ***comportamiento*** o conjunto de acciones que puede realizar como: ser lanzada, rebotada o rodada.

También el objeto de software tiene un estado y un comportamiento:

El ***estado*** es guardado y mantenido en un estructura de datos o variables (de instancia).

El ***comportamiento*** es implementado por métodos (de instancia). Un concepto de método es equivalente al de función y muy próximo al de procedimiento.

Los objetos son dinámicos, pueden crearse y destruirse mientras se ejecuta el programa. Todo objeto debe tener un identificador único que lo distinga de otros objetos.

Formación de un objeto:

- Un objeto es formado a partir de una clase.
- Una clase, se puede decir, es el molde, el patrón, el template o el blueprint de un objeto. Una clase es un colección de estructuras de datos (variables globales) y operaciones (métodos).
- Se pueden tener varios objetos de una misma clase.
- Un objeto es una instancia (copia) de una clase.

Los objetos tienen atributos que necesitan ser accesibles a otros objetos. Es importante que los objetos puedan ser cambiados sin afectar otros objetos, porque pueden ser construidos por diferentes organizaciones. Los objetos clásicos porporcionan código que se puede volver a utilizar por medio de la herencia y encapsulamiento. Sin embargo, los objetos clásicos solo pueden ser funcionales dentro de un programa, es decir, solo el compilador del lenguaje en el que fue creado el objeto conoce de su existencia.

En contraste, un *objeto distribuido* es una estructura de programación que puede ser reutilizado en cualquier parte dentro de una red. Los objetos distribuidos estan empaquetados como piezas independientes de código que pueden ser accesadas invocando sus métodos. El lenguaje y el compilador usados para crear los objetos distribuidos son totalmente transparentes para sus clientes. Los clientes no necesitan saber donde reside el objeto distribuido o que sistema operativo lo está ejecutando, también puede estar en la misma máquina o en cualquier máquina dentro de la red.

2. Componentes

Cuando se habla de objetos distribuidos, realmente se habla de *componentes* de software independientes. Estos son programas ejecutables que pueden participar en diferentes redes, sistemas operativos y herramientas. Consisten en interfaces múltiples, donde cada interfaz es una colección de métodos. Un objeto, es una instancia de un componente. Un componente es un objeto que no está comprometido con un programa en particular, lenguaje de programación o implementación.

Los objetos distribuidos son, por definición, componentes por la manera en la que se encuentran empaquetados. En los sistemas distribuidos, la unidad de trabajo y la distribución son un componente. La infraestructura de un objeto distribuido hace que sea más fácil para los componentes ser autónomos, autosuficientes y colaborativos. Los componentes son también objetos en el sentido que soportan la herencia de interfaces, el polimorfismo y el encapsulamiento. A diferencia de los objetos clásicos, los componentes hoy en día no pueden expandirse utilizando la herencia.

La definición de componente que a continuación se describe es una composición de CORBA/EJB y ActiveX/COM+.[12] Por lo menos un componente debe tener las siguientes propiedades:

- *Es una entidad de valor comercial.* Un componente es una pieza binaria de software que puede ser fácilmente adquirida en el mercado.
- *No es una aplicación completa.* Un componente puede ser combinado con otros componentes para formar una aplicación completa. Está diseñado para ejecutar un conjunto limitado de tareas dentro del dominio de una aplicación. Los componentes pueden ser pequeños tal como el tamaño de un objeto en C++; objetos medianos tal como un control GUI o EJB; o un objeto grande tal como un módulo ERP.
- *Puede ser utilizado en combinaciones impredecibles.* Tal como los objetos del mundo real, un componente puede ser usado en formas totalmente imprevistas por el desarrollador original. Típicamente, los componentes pueden estar combinados con otros componentes de la misma familia utilizando plug-and-play.
- *Tiene una interfaz específica.* Como los objetos clásicos, un componente puede ser manipulado solo por su interfaz. Sin embargo, la interfaz de los componentes es separada desde su implementación. Se puede implementar un componente usando objetos, código procedural, o encapsulando el código existente. CORBA y COM+ también proporcionan un Lenguaje de definición de interfaz (IDL *Interface Definition Language*) que puede ser utilizado para especificar las interfaces de los componentes; Java soporta interfaces como parte del lenguaje.
- *Ser una herramienta.* Esto significa que un componente debe permitir ser importado dentro de una paleta estándar de herramientas donde pueda ser reutilizado y

[12] Definición extraída del libro ***Client/Server Survival Guide***, Robert Orfali et al. (Wiley Computer Publishing, 1999)

adecuado. La mayoría de las herramientas visuales hoy en día soportan ActiveX o JavaBeans. La paleta es simplemente el contenedor para los componentes. Además, la mayoría de las herramientas proporcionan *formas* que se pueden usar para unir a los componentes utilizando drag-and-drop y otras técnicas visuales de ensamblaje. La *forma* es también un contenedor visual de componentes. (Un ejemplo de esta técnica es la interfaz de Delphi y Visual Basic).

- *Notificación de eventos.* Un componente debe estar disponible para informar si algo de interés está sucediendo. Un componente hace esto enviando un evento. Otros componentes que tienen interés en este evento son notificados.
- *Configuración y administración de propiedades.* Las propiedades definen las características de un componente. Una propiedad es un atributo que se puede utilizar para leer y modificar el estado del componente. Algunos componentes pueden incluir asistentes (wizards) para personalizar las propiedades. La idea es poder configurar los componentes ajustando sus atributos.
- *Escritura de guiones.* Un componente debe permitir que su interfaz sea controlada por medio de lenguajes de escritura de guiones (scripts). Esto significa que la interfaz pueda autodescribirse.
- *Introspección y metadatos.* Un componente debe proveer y solicitar información sobre sí mismo. Esto incluye una descripción de sus interfaces, propiedades, eventos, etc.
- *Interoperabilidad.* Un componente puede ser invocado como un objeto a través de espacios de dirección, redes, lenguajes, sistemas operativos y herramientas. Es una entidad de software independiente.
- *Fácil de usar.* Un componente debe proveer un número limitado de operaciones que promuevan la utilización y reutilización. Es decir, el nivel de abstracción debe ser tan alto como sea posible para hacer que el componente invite a ser utilizado.

En resumen, un componente es una pieza de software reusable que es independiente a cualquier aplicación. Los componentes son objetos en el sentido que soportan encapsulamiento, herencia y polimorfismo. Sin embargo, los componentes del lado del servidor también deben proveer todas las características asociadas a un objeto independiente.

3. Componentes del servidor.

Los componentes del servidor, son componentes con características que les permiten dar un servicio a múltiples clientes dentro de una red. Por consecuencia, los componentes necesitan suministrar las facilidades asociadas a las entidades de red independientes, incluyendo;

- *Seguridad.* Un componente debe protegerse a sí mismo y a sus recursos del peligro exterior. También debe autentificarse ante sus clientes y viceversa. Debe suministrar controles de acceso y dar seguimiento de revisión para su uso.
- *Permiso.* Un componente debe ser capaz de implementar políticas de permiso incluyendo licencia de uso y contadores.
- *Administración del ciclo de vida.* Todo componente debe proteger sus recursos y colaborar con otros componentes para asegurar su integridad. Además, debe suministrar candados para el acceso a los recursos compartidos.
- *Persistencia.* Los componentes deben poder salvar su estado para después restablecerse.
- *Interconexión.* Todo componente debe permitir asociaciones dinámicas o permanentes con otros componentes.
- *Autocomprobación.* Un componente debe ser capaz de proporcionar al desarrollador la capacidad de diagnosticar y determinar dónde se encuentra un problema dado.
- *Autoinstalación.* Cualquier componente debe poder instalarse y automáticamente registrarse en el sistema operativo. También debe tener la capacidad de removerse a sí mismo.

Esto nos debe dar una idea del nivel de calidad y funcionalidad que se espera de un componente del servidor. Tanto en COM+ como en CORBA/EJB existen estas facilidades donde el contenedor por elección es el Monitor de transacción de Objetos (OTM Object Transaction Monitor)[13].

[13] Véase más a detalle la definición de OTM y ORBs en la sección D del presente capítulo.

4. Objetos comerciales.

En general, es fácil hacer que dos componentes trabajen conjuntamente si se conocen las características de ambos, sin embargo, lo difícil es hacer que dos componentes coolaboren sin tener el conocimiento previo de cada uno de ellos. Para ello se necesitan de estándares que establezcan reglas de enlace para definir los límites de interacción entre los componentes. Esta interacción define una *infraestructura* de componentes distribuidos.

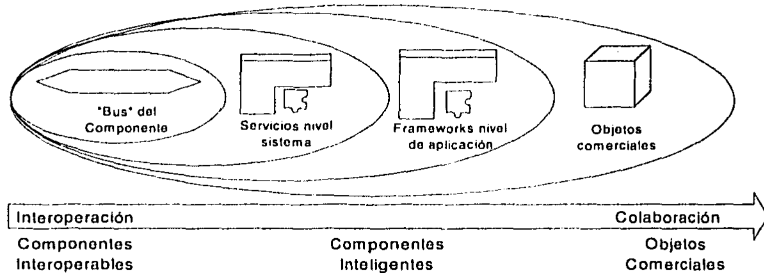


Fig. 2.3 Evolución de los componentes y sus límites de infraestructura

En su nivel más basico, una infraestructura de componentes proporciona un objeto bus (el ORB) que permite que los componentes interoperen entre direcciones, lenguajes, sistemas operativos y redes. El bus, también proporciona mecanismos para que los componentes intercambien metadatos. En el siguiente nivel, la infraestructura aumenta el bus con servicios de sistema que ayudan a crear componentes inteligentes. Ejemplos de estos servicios incluyen permisos, seguridad, control de versiones, persistencia, escritura de guiones y transacciones.

La última meta es crear componentes comericales, estos son componentes que por lo general, desarrollan funciones comerciales específicas. Por ejemplo, se pueden tener componentes para hoteles, aerolíneas, etc.

Los objetos comerciales son ideales para la creación de soluciones multicapas escalables porque son intrínsecamente divisibles. Un objeto comercial no es una pieza monolítica de

TESIS CON
FALLA DE ORIGEN

código, en vez de eso, es más como una pieza de Lego que ensambla códigos multicapas (Véase la figura 2.4). La primer capa representa aspectos visuales del objeto comercial. Estos objetos visuales por lo general se ubican en el cliente. En la capa intermedia hay objetos servidores que representan los datos persistentes y reglas del negocio. En una tercer capa se pueden encontrar las fuentes de datos, por ejemplo bases de datos SQL, archivos HTML, Lotus Notes, monitores TP, etc.

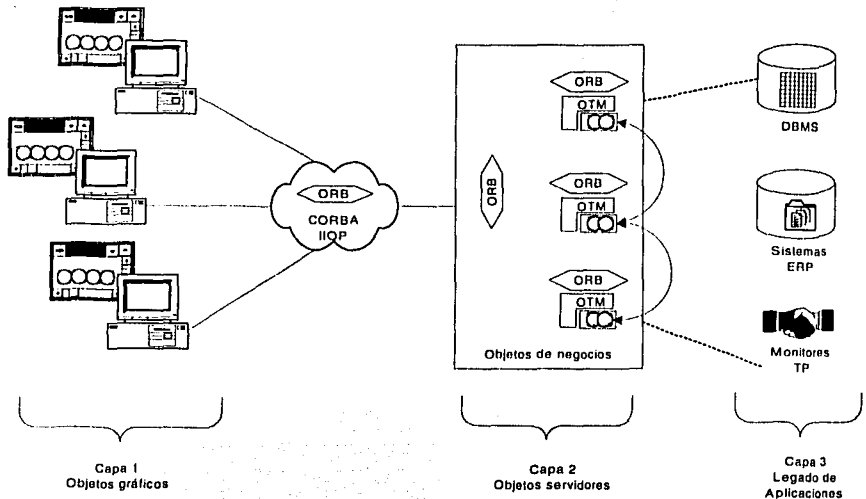


Fig. 2.4 Cliente/Servidor de 3 capas, Estilo objetos.

D. Objetos distribuidos en la arquitectura multicapas.

Para comprender mejor el trabajo de los objetos distribuidos en la arquitectura multicapas iniciaré definiendo los monitores TP[14]. Un TPM (Transaction Processing Monitor) es un

[14]Para una descripción detallada acerca de monitores TP recomiendo el libro ***Principles of Transaction Processing***, Philip A. Bernstein et al. (Morgan Kaufmann Publishers, 1997)

TESIS CON
FALLA DE ORIGEN

producto de software utilizado para crear, ejecutar y administrar aplicaciones de proceso de transacciones. Su trabajo principal es proporcionar un ambiente que toma aplicaciones para procesar una petición y las escala para que se ejecuten de manera eficiente y distribuida. Un monitor TP por lo general incluye software en tres áreas principalmente:

- Un programa de aplicación de interfaz (API) para establecer funciones en tiempo de ejecución que soportan la ejecución de transacciones, la comunicación entre los programas dentro de una transacción y la comunicación front-end que recibe las peticiones.
- Herramientas de desarrollo para construir programas, tales como compiladores que traducen operaciones de alto nivel en un lenguaje de programación específico a un API.
- Una interfaz de administración del sistema y funciones que monitorean y controlan la ejecución de los programas.

La estructura de una aplicación TP divide sus componentes de forma tal que se asemejan a los tres componentes principales de un modelo multicapas.

La infraestructura de objetos distribuidos suministra monitores TP con una gran variedad de servicios incluyendo metadatos, invocaciones dinámicas, consistencia, relaciones, eventos, colecciones, etc. Los objetos hacen más fácil a los Monitores TP crear y administrar modelos transaccionales grandes.

Dentro del modelo de objetos distribuidos, los clientes utilizan los objetos por medio de sus interfaces sin conocer donde en realidad se están ejecutando dichos objetos, permitiendo que el desarrollador cambie los objetos sin tener que modificar a los clientes que los llaman. En el modelo multicapas, por lo general, los clientes interactuan con los objetos servidores que se encuentran dentro de la capa intermedia a través de un ORB (Object Request Broker). Además, los objetos de la capa intermedia se pueden comunicar entre sí por medio de un OTM (Object Transaction Monitor) que pueden ser utilizados para balancear las cargas e intercambiar los eventos. Finalmente, los objetos del servidor se comunican con la tercer capa utilizando el tradicional middleware.

Un ORB (Object Request Broker) es, como se mencionó anteriormente, un objeto bus. Con él, los objetos deben determinar cuándo y cómo llamar los servicios del ORB, por ejemplo, seguridad, transacciones y ciclo de vida. Los objetos deben proporcionar todas las llamadas a estos servicios, por lo que los objetos deben contener mucho código específico que hace difícil enviarlo por los contenedores. Además, cada objeto debe dirigir las llamadas a los diferentes servicios, haciéndolo propenso a errores. Se requieren de programadores expertos que conocen a fondo el trabajo de un ORB y que se encuentra altamente relacionado con los servicios en el middleware.

Un OTM (Object Transaction Monitor) es fundamentalmente una combinación de un Monitor TP en cuanto a su ambiente de ejecución y un ORB en cuanto a su estilo de programación y comunicación. Un OTM maneja un conjunto de contenedores que ejecutan los componentes del servidor. Las propiedades de los componentes del servidor se definen y administran estableciendo sus atributos, por lo general utilizando una herramienta visual. El programador, únicamente se encarga de definir la lógica del sistema. En tiempo de corrida, el OTM intercepta todas las llamadas de entrada, invoca los objetos de comunicación en el contenedor, y envía la petición al objeto. En otras palabras, los ORBs suministran los pipes para los objetos distribuidos y los OTMs la plataforma.

Un OTM suministra un ambiente organizado para ejecutar los componentes en el servidor, se encarga de hacer las llamadas a los componentes en el momento adecuado y con la secuencia adecuada, también llama los servicios del sistema en nombre de los componentes basándose en sus atributos. El principio de un OTM es "no nos llames, nosotros te llamamos".

Como los OTMs, los monitores TP se convierten en el esqueleto (frameworks) para manejar componentes inteligentes. Los monitores TP hacen posible que los ORBs administren millones de objetos.

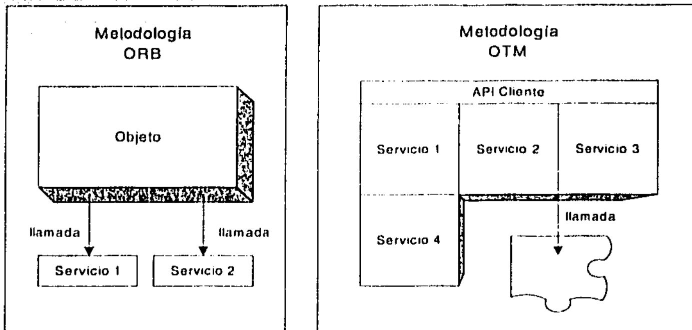


Fig 2.5 Metodología de desarrollo de objetos / ORB vs. OTM

Algunas de las opciones para diseñar aplicaciones distribuidas se presentan a continuación, el objetivo de esta tésis es explicar aquellos que se relacionen con los modélos estándar de objetos distribuidos, así que más adelante se hablará de ellos en específico:

- *Programación con sockets.* Las aplicaciones se comunican entre sí a través de un "canal". Es la forma más directa de comunicar componentes de una aplicación. Se escriben o leen los datos del socket. Al ser una forma de programación de bajo nivel, no se introduce mucha sobrecarga a la aplicación, pero no es muy adecuada para manejar tipos de datos complejos, especialmente cuando los módulos de la aplicación se encuentran en diferentes tipos de máquinas o están implementados en diferentes lenguajes.
- *Llamada a procedimiento remoto (RPC).* El programador define una función que usa sockets para comunicarse con el servidor remoto que ejecuta la función y devuelve el resultado de nuevo mediante sockets. El RPC es un mecanismo bastante potente como para ser la base de muchas aplicaciones cliente/servidor.
- *Entorno de computación distribuida de la OSF (DCE).* Se trata de un conjunto de estándares iniciados por el Open Software Foundation (OSF). Nunca ha tenido demasiada aceptación.

TESIS CON
FALLA DE ORIGEN

- *Modelo CORBA.* Proporciona un estándar para poder definir interfaces entre módulos, así como algunas herramientas para facilitar la implementación de dichas interfaces en el lenguaje de programación escogido. Este modelo es independiente tanto de la plataforma como del lenguaje de la aplicación. El capítulo V definirá a detalle este modelo.
- *Modelo de objetos para componentes distribuidos de Microsoft (DCOM).* Ofrece capacidades similares a las de CORBA. Es un modelo de objetos relativamente robusto. Su defecto es que está disponible casi exclusivamente en el entorno Windows aunque para aplicaciones específicas de Windows es muy buena solución. El capítulo IV definirá a detalle este modelo.
- *Invocación de métodos remotos de Java (RMI).* Su principal ventaja es que soporta pasar objetos por valor y su principal desventaja es que es una solución solamente para Java.

CAPITULO III

LA CAPA INTERMEDIA ORIENTADA A OBJETOS

A. Redes de computadoras

1. Modelo ISO/OSI

El modelo de referencia Open Systems Interconnection (OSI) fue definido en 1977 por la International Organization for Standarization (ISO). Define la necesidad de estandarización de la comunicación entre los hosts construidos por diferentes distribuidores. No es objetivo de esta tesis presentar el modelo de referencia ISO/OSI a gran detalle, sin embargo, se tocarán sus principales características de las cuales hablaré más adelante.

El modelo de referencia ISO/OSI es una arquitectura de capas en la cual cada capa se complementa por la capa de abajo. La figura 3.1 muestra una vista gráfica de este modelo. Se trata de un modelo conceptual que intenta explicar la arquitectura de las redes y en el que actualmente muchos vendedores se basan para explicar el funcionamiento de su propio software de red.

7. Aplicación
6. Presentación
5. Sesión
4. Transporte
3. Red
2. Enlace de Datos
1. Física

Figura 3.1 Capas del Modelo ISO/OSI

TESIS CON
FALLA DE ORIGEN

En el momento que fue desarrollado este modelo ya existian varias arquitecturas de red tales como TCP/IP, XNS y SNA, que todavía subsisten. Como cada una de estas arquitecturas había sido construida con un sello propio, el conocer una de ellas no implicaba el conocer cualquiera de las otras restantes. De allí que el modelo OSI pretende dar un concepto genérico a las redes para su mejor entendimiento y que en la actualidad ha llevado al establecimiento de estándares.

Para que un usuario o una aplicación de usuario pueda comunicarse de un Host A a un Host B, su mensaje (comando o información) tiene que pasar por el software de comunicación que tiene instalado el Host. Este software consta de siete instancias o capas.

Así un mensaje, enviado desde el Host A, primero pasa por la capa 7, luego por la 6 y así sucesivamente hasta llegar a la capa No. 1 (Física). Cada capa, le va agregando aditivamente información propia a lo que le llega de la capa inmediata superior. Así lo que llegue de la capa 5, la capa 4 le agrega más información de control, propia de esta capa.

A partir del hardware de comunicaciones del Host A, toda la información agregada por las capas incluyendo el mensaje del usuario, es enviado a través del medio de transmisión.

Después, el hardware de comunicación del Host B, recibe del medio de transmisión la información enviada por la capa Física del Host A, iniciando ahora el proceso inverso de ir quitando información propia de cada capa, desde la capa 1 hasta la 7 (véase la figura 3.2).

El propósito de cada capa (N), es ofrecer un servicio específico a la capa superior (N + 1), protegiendo a esta capa superior de cómo el servicio es implementado (independientemente de cómo sea la computadora y/o red física). Un objetivo de esta división por capas, además de hacer comprensible la arquitectura de la red, pretende hacer independientes a las capas, de tal manera que un cambio efectuado en una capa se haga por separado sin que el resto se entere. En teoría podría intregar un software de red con capas elaboradas por siete vendedores distintos.

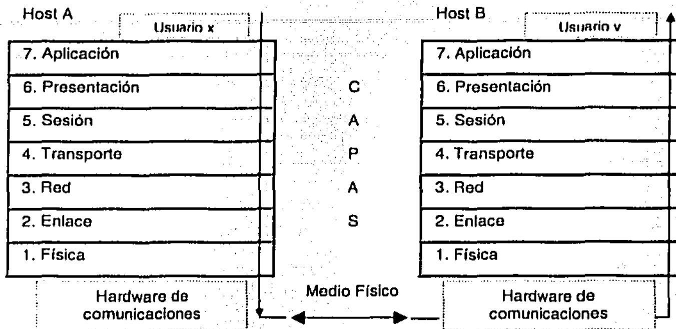


Figura 3.2 Modelo de comunicaciones del modelo OSI

Funciones de las capas:

- *Capa física.* Especifica el comportamiento mecánico, eléctrico y óptico de los conectores y sockets de una interfaz física a la red. Las implementaciones de la capa física realizan funciones de señalización, modulación y sincronización de la transmisión de bits en la red física.
- *Capa de datos.* Conceptualmente, una red es una gráfica de nodos interconectados, la capa de datos implementa aristas en estas gráficas. Estos nodos pueden ser ruteadores, switches o computadoras. Delinea la conexión física proporcionada por la capa física en una capa que puede transmitir paquetes de un nodo a otro de una forma relativamente libre de errores. Esta capa proporciona corrección de errores basados en la detección de errores físicos primitivos proporcionados por la capa física.
- *Capa de Red.* Implementa la conexión de muchos nodos en una red. Rutea los mensajes a lo largo de aristas múltiples dentro de la gráfica de nodos.
- *Capa de Transporte.* Añade la capacidad de mapear mensajes largos o secuencias continuas de datos entre dos hosts.
- *Capa de Sesión.* Establece y mantiene conexiones entre dos o más componentes distribuidos. Si la capa de transporte es orientada a la conexión, entonces la capa de sesión utiliza mecanismos de conexión de la capa de transporte.
- *Capa de Presentación.* Resuelve diferencias de representación de los datos y proporciona la habilidad de transportar datos complejos tales como registros, arreglos y secuencias.

TESIS CON
FALLA DE ORIGEN

➲ *Capa de Aplicación.* Proporciona funcionalidad en la aplicación. Está relacionada con componentes distribuidos y su interacción.

La capa de transporte proporciona las bases suficientemente poderosas para la implementación de la distribución en el software de la capa intermedia. Los sistemas middleware por consiguiente son puestos sobre la capa de transporte e implementan las capas de presentación y sesión. En esta capa tanto los procesos clientes como los procesos servidores, le piden la transportación del mensaje de solicitud del cliente y de respuesta del servidor.

2. Capa de transporte.

En general, se distinguen dos tipos de implementaciones de la capa de transporte[15], una de ellas es la *capa de transporte orientada a la conexión (Connection-oriented),* la cual mantiene una conexión entre dos componentes distribuidos en representación de las capas superiores. Los componentes realizan operaciones para abrir y cerrar una conexión, para escribir datos hacia la conexión y para leer datos desde la conexión. Además, proveen capas superiores con un mecanismo de intercambio de datos a través de cadenas de bytes.

La *capa de transporte sin conexión (connection-less)* proporciona capas superiores con la capacidad de enviar mensajes de tamaño fijo entre los host distribuidos. Estos mensajes son referidos como datagramas. Si un protocolo de transporte sin conexión es utilizado, las implementaciones de la capa de sesión necesitan asociar el envío y recepción de mensajes con ciertas aplicaciones. Este tipo de conexión solo puede atender un paquete a la vez.

Para proporcionar un ejemplo de implementación de ambos tipos de capas de transporte, me voy a basar en dos protocolos que utiliza el sistema operativo Unix y que son los más populares. Estos son TCP (Transmission Control Protocol) y UDP (User Datagram Protocol). TCP es orientado a conexión y proprociona un medio confiable de transporte,

[15] Es recomendable leer algún libro especializado en TCP/IP para comprender mejor la capa de transporte. Recomiendo ***Internetworking with TCP/IP Vol.1: Principles, Protocols, and Architecture***, Douglas Comer (Prentice Hall, 2000).

este tipo también está disponible para los sistemas operativos de Microsoft. UDP es sin conexión y proporciona un medio no confiable de transporte de datos. El utilizar TCP o UDP depende de que tanto la aplicación requiere retener la conexión por largos periodos de tiempo. Las aplicaciones, como *rwho*, la cual verifica los logins en todos los host de una red de área local, utilizan UDP para evitar tener conexiones abiertas de un gran número de computadoras. Aplicaciones como *ftp*, que necesitan contar con una conexión confiable por períodos prolongados de tiempo, establecen conexiones TCP.

Como se mencionó en el párrafo anterior, Unix implementó dos tipos de capa de transporte, la primera TCP es orientada a conexión. En TCP, cada vez que un cliente solicita conectarse con el servidor concurrente (padre), este se encarga de crear un servidor hijo con el cual el cliente establece una conexión definitiva.

Así el cliente del host X se conecta con el servidor hijo, formandose la conexión C1 dándose la asociación (TCP, dirX, ptoX, dirY, ptoY), véase figura 3.3.

Casi al mismo tiempo, el cliente del host Z, que solicitó conexión con el servidor, también recibe una conexión con el servidor hijo, estableciéndose la conexión C2 con asociación (TCP, dirZ, ptoZ, dirY, ptoY).

De esta manera en este momento se tienen dos servidores hijos atendiendo concurrentemente a dos clientes. En este caso el servidor concurrente original o padre actúa como un gran despachador al cual los clientes hacen la solicitud inicial de conexión.

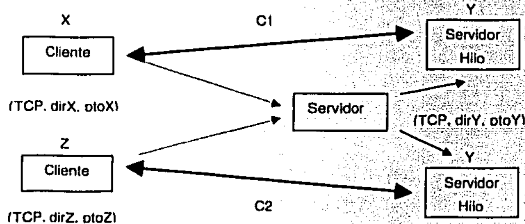


Figura 3.3 Capa de transporte orientada a la conexión basada en TCP

La otra implementación de la capa de transporte de Unix es UDP y está clasificada como *capa de transporte sin conexión.* Una desventaja de este protocolo es que debido a que no se establece una conexión, cualquier datagrama puede perderse sin que el cliente o el servidor se den cuenta. (Véase la figura 3.4)

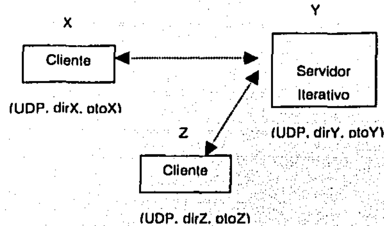


Figura 3.4 Capa de transporte sin conexión basada en UDP

B. Tipos de capa intermedia en la arquitectura multicapas.

La capa intermedia[16] delinea parámetros complejos que son transferidos a las peticiones de servicio, establecen hetereogeneidad, determinan las direcciones lógicas de los componentes para los nombres de dominio de Internet y las direcciones de los puertos, además de sincronizar los objetos del cliente y del servidor. Mientras hace esto, la capa intermedia implementa la sesión y presentación de capas del modelo de referencia ISO/OSI.

La capa intermedia puede ser clasificada en cuatro categorías: orientada a transacciones, orientada a mensajes, llamadas a procedimiento remoto y orientada a objetos.

[16] Dentro del contexto multicapas, capa intermedia se refiere a la aplicación servidora en la que se encuentran ubicadas la lógica y las reglas del negocio.

1. Capa intermedia orientada a transacciones.

Este tipo de capa intermedia soporta transacciones a través de diferentes sistemas de bases de datos distribuidas. Este tipo de capa es comunmente utilizado en arquitecturas donde los componentes son aplicaciones de bases de datos. Los productos en esta categoría incluyen CICS de IBM, Tuxedo de BEA y Encina de Trnsarc.

Para explicar mejor este tipo de capa, mencionaré de manera superficial a Tuxedo[17]. Tuxedo proporciona una infraestructura distribuida de capa intermedia que permite escribir aplicaciones como una colección de servicios, cada uno implementando una función. Tuxedo administra estos servicios y coordina su enfrascamiento en transacciones. Los servicios pueden accesar varias bases de datos y estar seguros de una completa integridad de transacciones.

Tuxedo proporciona el siguiente soporte para transacciones:

- Crea un identificador global de transacciones cuando un cliente o un servicio inicializa una transacción.
- Rastrea los componentes que están involucrados en una transacción y después necesita ser coordinado cuando la transacción está lista para cometer (commit).
- Notifica a los administradores de recursos (por lo general bases de datos) cuando son accesados por una transacción. Estos administradores bloquean los registros accesados hasta que se finalice la transacción.
- Dirige la cometida de dos fases (two-phase commit) cuando la transacción es completada, lo cual asegura que todos los participantes en la transacción consignen sus actualizaciones simultáneamente. Coordina la cometida con cualquier base de datos que haya sido actualizada utilizando el protocolo XA de X/Open.
- Ejecuta el procedimiento rollback (deshacer) cuando la transacción es abortada.
- Ejecuta el procedimiento de recuperación cuando una falla ocurre.

[17] Para mayor información sobre Tuxedo, recomiendo ***The Tuxedo System***, Andrade et al. (Addison Wesley, 1996)

2. Capa intermedia orientada a mensajes.

Este tipo de capa soporta la comunicación entre los componentes de un sistema distribuido facilitando el intercambio de mensajes. Los productos en esta categoría incluyen MQSeries de IBM, ToolTalk de Sun y TopEnd de NCR. Los componentes del cliente utilizan estos sistemas para enviar un mensaje y solicitar la ejecución de un servicio a un componente del servidor. El contenido del mensaje incluye los parámetros del servicio. Otro mensaje es enviado al cliente desde el servidor para transmitir el resultado del servicio.

La fortaleza de la capa orientada a mensajes es que este paradigma soporta la entrega de mensajes asíncronos de manera natural. El cliente continúa procesando tan pronto como la capa intermedia toma el mensaje. El servidor, eventualmente envía mensajes incluyendo el resultado y el cliente puede recolectar ese mensaje en el tiempo que considere apropiado.

Otra característica importante de este tipo de capa es que soporta el multi-casting, puede distribuir el mismo mensaje a varios receptores de manera transparente a los clientes. La capa orientada a mensajes consigue la tolerancia de errores a través de la implementación de colas de mensajes que almacenan temporalmente los mensajes en un medio de almacenamiento constante. El mensajero escribe el mensaje dentro de la cola de mensajes y si el receptor no esta disponible a causa de alguna falla, la cola de mensajes retiene el mensaje hasta que el receptor está disponible nuevamente.

3. Llamadas de procedimiento remoto (RPC)

RPC (Remote Procedure Calls) fue inventado a principios de los 80's por Sun Microsystems como parte de la platafoma del ONC (Open Network Computing). Los RPCs son operaciones que pueden ser invocadas remotamente a través de diferente hardware y distintas plataformas de sistemas operativos.

Las llamadas de procedimiento remoto son el origen del middleware orientado a objetos. En algunos sistemas, los RPCs también son utilizados para implementar las peticiones de los objetos distribuidos.

Los componentes del servidor que ejecutan los RPCs son llamados programas RPC. Los programas RPC tienen una definición de interfaz que establece los procedimientos que pueden ser llamadas remotamente. También definen tipos de datos de los argumentos que pueden ser pasados a los procedimientos remotos.

Capa de presentación

La capa de implementación de los RPCs mapean las estructuras de datos de las aplicaciones en forma tal que puede ser transmitida por la capa de transporte. Esto involucra dos tareas primordialmente: la resolución de los datos heterogeneos y el mapeo de estructuras de datos complejas en bloques de cadenas de bytes. Véase la figura 3.5.

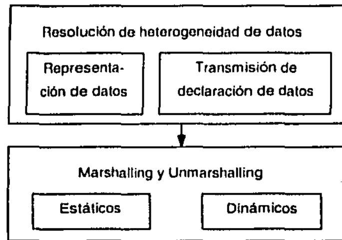


Figura 3.5 Tareas efectuadas en la capa de presentación

Una vez resuelta la heterogeneidad, la capa de presentación transforma estructuras de alto nivel que son pasados como parametros de manera que puedan ser transferidos por la capa de transporte. El proceso de transformación requerido, es comúnmente referenciado como *marshalling* (organizador). La capa de presentación también implementa el mapeo de regreso, esto es, la reconstrucción de estructuras de datos complejas de una secuencia o bloque de bytes. A este proceso se le conoce como *unmarshalling* (desorganizador). En el ejemplo 3.1 se muestra un ejemplo de marshalling y unmarshalling escritos en C++. En la práctica, este código es generado por la definición de interfaz del lenguaje en vez de escribirlo a mano.

TESIS CON
FALLA DE ORIGEN

```
char * marshal() {
  char *msg;
  msg = new char [4*(sizeof(int)+1) + strlen(nombre)+1];
  sprintf(msg, "%d %d %d %d %s", dob.dia, dob.mes, dob.anio, strlen(nombre), nombre);
  return(msg);
};

void unmarshal(char *msg) {
  int nombre_long;
  sscanf(msg, "%d %d %d %d", &dob.dia, &dob.mes, &dob.anio, &nombre_long);
  nombre = new char[nombre_long+1];
  sscanf(msg, "%d %d %d %d %s", &dob.dia, &dob.mes, &dob.anio, &nombre_long, nombre);
};
```

La operación marshal transforma los datos en una secuencia de bytes y la operación unmarshal realiza la operación contraria. Operaciones similares son realizadas por cada cliente y servidor.

Ejemplo 3.1

Capa de sesión

La implementación de la capa de sesión necesita permitir a los clientes localizar un servidor RPC. Esto se puede hacer de manera estática o dinámica. El ejemplo 3.2 es estático porque el nombre del host está asignado en tiempo de compilación. La asignación estática es muy simple, pero no es transparente. Con asignación dinámica, la selección del servidor es en tiempo de ejecución. Esto puede hacer que la migración y la replicación sean de manera transparente.

```
print_persona(char * host, jugador * pers) {
  CLIENTE *clnt;
  clnt = clnt_crear(host, 105040, 0, "udp");
  if (clnt == (CLIENTE *) NULL) exit(1);
  if (print_0(pers, clnt) == NULL)
  clnt_perror(clnt, "Error de llamada");
  clnt_destuir(clnt);
}
```

Este código en C utiliza una conexión UDP para conectarse al host. Si la creación es exitosa, llama al servidor con print_0.

Ejemplo 3.2

TESIS CON
FALLA DE ORIGEN

C. La capa intermedia orientada a objetos

Este concepto se involucra en parte con la idea de llamadas de procedimiento remoto. El primero de estos sistemas fue CORBA de la OMG, después Microsoft se integró al concepto de distribución y lanzó COM y Sun proporcionó un mecanismo en Java llamado RMI.

Una desventaja de las llamadas a procedimiento remoto es que no son reflexivas, lo que significa que los procedimientos exportados por un programa RPC no pueden regresar otro programa RPC. La capa intermedia orientada a objetos proporciona una interfaz de definición de lenguaje (IDL), donde las interfaces definen tipos de objetos e instancias de estos tipos que son objetos. Esto significa que un objeto servidor que implementa un tipo objeto puede regresar otro objeto servidor.

La capa orientada a objetos, extiende llamadas de procedimiento remoto para el manejo de errores. RPC de Sun regresa punteros nulos cuando el procedimiento remoto falla. Esto no permite al objeto servidor informar al cliente el porqué de la falla.

A pesar de que los procesos RPC parecen ser similares a la arquitectura de componentes distribuidos, estos procesos no soportan polimorfismo. Esto significa que el código debe estar definido en el lado del cliente.

Implementación de la capa de presentación

La capa de presentación del middleware orientado a objetos es muy similar a la de los RPCs. También trabaja bajo el concepto de marshalling y resuleve la heterogeneidad de la representación de los datos. La única diferencia con respecto a los RPCs está relacionada con el manejo de referencias a objetos. El protocolo de la capa de transporte de una capa orientada a objetos necesita definir representaciones de objetos referenciados. Necesitan ser organizados (marshalled) y desorganizados (unmarshalled) de manera similar a las estructuras de datos a nivel de la aplicación.

Implementación de la capa de sesión

La implementación involucra la conexión entre varios objetos sobre una o varias conexiones establecidas por la capa de transporte.

Los objetos clientes siempre identifican al objeto del cual están respondiendo el servicio por medio de un objeto de referencia. Este objeto de referencia es una parte fundamental en la petición y siempre es transmitido con una identificación de la operación solicitada y los parametros de petición. El lado del cliente utiliza la referencia al objeto para localizar el host donde el objeto servidor reside. La capa de sesión se conecta con la implementación en el servidor que reside en ese host. El servidor del objeto solicitante contiene un componente que es conocido como objeto adaptador. Hay por lo menos un objeto adaptador en cada host, estos objetos implementan la activación y desactivación de los objetos.

El objeto adaptador implementa el mapeo de objetos de referencia a implementaciones activas de objetos. Antes de la petición de un objeto, el objeto adaptador verifica si el objeto está activo , esto es, cuando la implementación del objeto está siendo ejecutada por un proceso del sistema operativo. Si no, el objeto adaptador busca iniciar la implementación del objeto en un repositorio o registro de implementación. El objeto adaptador debe lanzar un proceso de sistema operativo por separado para el objeto, crear un nuevo hilo (thread) de un proceso existente o ligar de manera dinámica una libreria hacia si mismo. Una vez que el adaptador ha iniciado o identificado el proceso del sistema operativo ejecutándose, regresa la petición a ese objeto.

La capa de sesión necesita implementar un despachador de operaciones, éste invoca la operación requerida, para lograr esto la implementación del objeto necesita decodificar parcialmente la información solicitada que ha sido transmitida por la red y pasada a través del objeto adaptador. La información solicitada deberá incluir un identificador de la operación requerida y así el objeto sabe que operación llamar en la implementación del objeto servidor.

La capa de sesión en ambos lados necesita llevar a cabo la sincronización entre los objetos cliente y servidor. El objeto solicitante usa por omisión formas síncronas de

comunicación. Lo que significa que el middleware tiene que forzar al objeto cliente a esperar hasta que el objeto servidor haya terminado de ejecutar la solicitud y ha producido un resultado o notifica a través de una excepción que es imposible completar la petición. El lado del cliente necesita esperar la respuesta antes de darle el control de regreso al cliente; el lado del servidor necesita confirmar la finalización de la petición al lado del cliente.

D. Definición de la interfaz

La tecnología de componentes es un medio de empaquetamiento de software que puede ser utilizado por un cliente. No es una tecnología de implementación. Los lenguajes de programación orientados a objetos son tecnologías de implementación. Las tecnologías de componentes permiten implementar los componentes con Visual Basic o COBOL, a elección del programador. Las tecnologías de componentes por lo general tienen una forma de describir una colección de procesos a los cuales el componente responde. En su mayoría esto es a través del lenguaje de definición de interfaces (IDL). Los dos modelos de componentes más dominantes durante los 90s fueron CORBA (Common Object Request Broker Architecture) de la OMG (Object Management Group) y COM+/DCOM de Microsoft.

CAPITULO IV

EL MODELO COM EN LA ARQUITECTURA MULTICAPAS

A. Los componentes COM

Microsoft introdujo el primer ambiente COM en 1995 con el Servidor de Transacciones de Microsoft (MTS). Hoy en día Microsoft se encuentra publicando la nueva generación: **COM+**. COM+ combina el tradicional COM, DCOM y MTS (Conocido como Viper), e introduce algunas nuevas capacidades en los servicios.

1. Historia de la capa intermedia de Microsoft

Microsoft introdujo COM en 1993, en ese tiempo COM (acrónimo de Component Object Model), era un modelo de lenguaje "neutral" para componentes. COM fue diseñado para resolver dos problemas que los programadores de Windows comunmente se enfrentaban:

- Define una especificación por la cual se pueden crear objetos que pueden ser utilizados por múltiples lenguajes o ambientes de programación.
- Define una forma por la cual aplicaciones clientes en una máquina pueden interactuar con servidores en otra máquina.

La tecnología COM comenzó como un objeto OLE (Object Linking and Embedding). OLE era basicamente una forma elegante de efectuar el Intercambio dinámico de datos (DDE). Permite a una aplicación cliente almacenar datos de una aplicación servidora con información suficiente del servidor para activar al servidor cuando el usuario lo solicite (por lo general por medio de la ejecución de doble click en una imagen que representa los datos del servidor).

Cuando Microsoft mejoró OLE, añadieron características adicionales tales como Automatización y controles OLE. Entonces el término de Object Linking and Embedding se volvió obsoleto. COM es tecnicamente definido como la implementación de OLE, sin embargo ambos terminos se convirtieron tan confusos que la mayoría de la gente los utilizaba de forma intercambiable.

En la actualidad COM se ha convertido en COM+ añadiendole dos características:

- El contador de referencia es ahora automático.
- Los objetos COM pueden ser examinados en tiempo de corrida.

En el tiempo que fue lanzado COM, no tenía la capacidad de distribución y por consecuencia no se encontraba en el mismo nivel con el trabajo desarrollado por la OMG (Desarrollador de CORBA). Microsoft introdujo los componentes distribuidos a mediados de 1995 con la introducción de DCOM (Distributed COM). DCOM era una tecnología que permitía a los componentes COM ser separados por los clientes que los utilizaban. En ese momento Microsoft entra a competir con la OMG en el desarrollo de componentes distribuidos, sobre todo cuando Microsoft tomó muchos de los servicios de los objetos que la OMG había definido para CORBA. Sin embargo en ese momento COM y DCOM no representaban ninguna competencia importante. Al mismo tiempo surgió MTS de Microsoft (Microsoft Transaction Server), se trataba de un ambiente en tiempo de ejecución para los componentes en la capa intermedia. Los componentes MTS son equivalentes a los recursos en el servicio de transacciones CORBA[18], con la diferencia de que los recursos de CORBA no tienen que implementar una interfaz en particular.

CORBA diseñó los componentes de manera tal que contenían instancias con datos y estados asociados a ellos. Microsoft afirmaba que las instancias de los componentes no eran un lugar adecuado para almacenar datos siendo las bases de datos una mejor opción.

El modelo CORBA no solo asumió que las instancias de los componentes guardaran el estado, sino que también la forma adecuada para compartir información entre los clientes era a través de compartir las instancias, e indirectamente, el estado que ellos contenían. Hubo muchas técnicas para realizar esto, la más común fue asociar una instacia con un

nombre bien conocido, registrar el nombre y la instancia con un servicio bien conocido, de esta manera permitir regresar proxies a cualquier cliente que deseara el acceso a la instancia y el estado que contenía.

Microsoft propuso que los clientes deberían compartir la información no através de proxies que compartieran instancias, sino a través de instancias que compartieran bases de datos back-end. Microsoft afirmó que este modelo era mejor debido a dos razones. La primera era, que debido a la naturaleza de las bases de datos para compartir datos no era necesario inventar nuevos mecanismos para las instancias de los componentes. La segunda, cuando cualquier capa del componente en una máquina fallara, la instancia del componente que está almacenada en bases de datos robustas no se perdería. Las figuras 4.1 y 4.2 muestran la diferencia entre el modelo CORBA y el modelo MTS para compartir estados entre los clientes, así como la manera en como comparten el mismo estado.

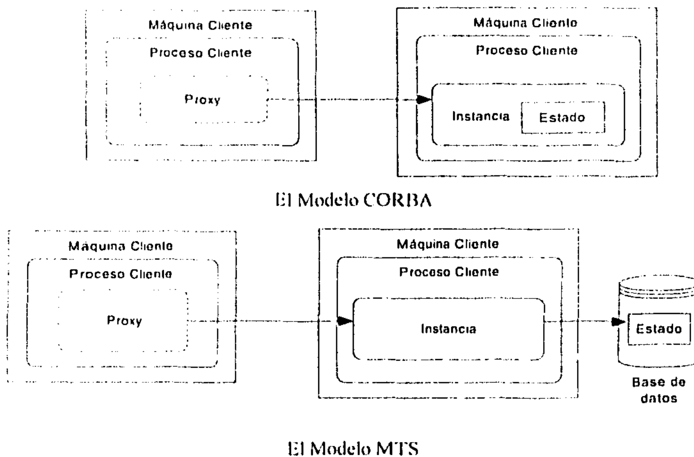
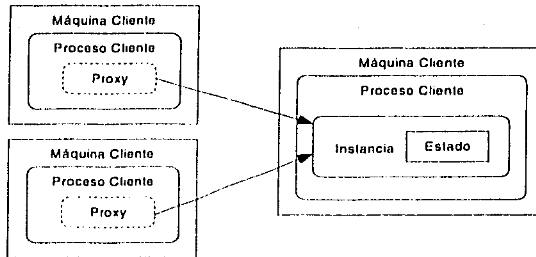


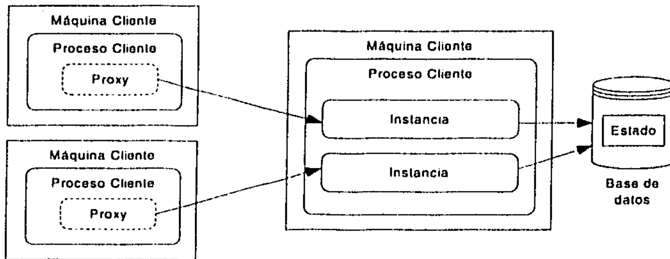
Figura 4.1 Diferencia entre CORBA y MTS

[18] Véase servicios CORBA y facilidades CORBA en el siguiente capítulo.

TESIS CON
FALLA DE ORIGEN



El Modelo CORBA



El Modelo MTS

Figura 4.2 La diferencia entre CORBA y MTS compartiendo el estado

El modelo CORBA asumía que un API extenso era requerido por los programadores para controlar las reglas no relacionadas al negocio, tales como el inicio y final de las transacciones, seguridad y ubicación de instancias específicas de los componentes. Mientras la OMG se enfocaba en definir e incluir nuevos servicios de objetos a través de APIs, Microsoft introdujo un intermediario entre el proxy cliente y la instancia del componente. Este intermediario lee los parametros de configuración de los componentes y automáticamente hace los ajustes necesarios de ambiente.

Además de COM, MTS y DCOM existían otros productos MS-DTC (Microsoft Distributed Transaction Coordinator), MSMQ (Microsoft Message Queue), inicialmente conocido

TESIS CON
FALLA DE ORIGEN

como "Falcon" y MSCS (Microsoft Cluster Server). En Windows 2000, estas tres tecnologías continúan jugando un papel muy importante en el soporte de COM+.

MS-DTC permite bases de datos múltiples, o múltiples conexiones de bases de datos, para trabajar de manera conjunta en una transacción coordinada. COM+ notifica a MS-DTC que la transacción ha comenzado, MS-DTC coordina las distintas bases de datos en el protocolo de cometida de dos fases y en general se lleva a cabo el siguiente flujo. El método deja el proxy cliente y guía a la instancia del componente. Antes de que la instancia sea alcanzada, es interceptada por el ambiente en tiempo de ejecución de COM+. Asume que este componente ha sido configurado para requerir una transacción, el interceptor lanza un mensaje a MS-DTC indicando que la transacción ha comenzado. MS-DTC inicia la primer fase del protocolo de cometida llamada *contemplativa*. Las reglas del negocio del método se ejecutan y actualizan algunas bases de datos. MS-DTC, está al tanto de qué bases de datos fueron actualizadas para una referencia futura.

Cuando las reglas del negocio son completadas, el método regresa al proxy. Mientras es completado COM+ se entera que el metodo está regresando desde el método que inició la transacción. Si todo sale bien la transacción finaliza en una cometida, de lo contrario se efectua un rollback a MS-DTC.

2. El Modelo de distribución.

En su modo más superficial, un componente COM+ es una colección de datos binarios usados para almacenar programas y fragmentos de código. Bajo este contexto, se puede definir a un componente COM+ como "un *blob*[19] de software". Todo componente COM+ se apega a las siguientes reglas:

- El blob está compuesto por una o más interfaces.
- La interfaz define los métodos que el blob soporta.
- Para cada interfaz el soporte del blob debe derivar de una interfaz llamada IUnknown.

---

[19] Binary Large Object. Colección de datos binarios almacenados en una entidad dentro de un sistema manejador de base de datos. También pueden ser usados para almacenar programas y fragmentos de código, tales como componentes.

- Al menos una de las interfaces debe (aunque no es requerido) derivar de una interfaz llamada IDispatch. IDispatch es derivada de IUnknown.
- Las interfaces están descritas por un archivo llamado libreria de tipos del componente (Type Library). Las librerías de tipo son programas de computación muy sofisticados que contienen información, como las interfaces que son implementadas por un objeto, que propiedades y métodos están definidos por la interfaz, y el número y tipos de argumentos para cada método. Las librerías de tipo deben integrarse dentro de los servidores COM como un recurso.
- El blob vive en uno o más procesos dentro de uno más sistemas.

a) Definición de interfaces

La manera más simple de entender una interfaz es a través de la comparación con una clase abstracta, debido a que son más o menos equivalentes. Esto es, son clases que definen un comportamiento, pero no implementan ese comportamiento por si mismos. En vez de ello, dependen de clases descendientes que proveen dicha implementación. Considerese la siguiente declaración de clase en el lenguaje Delphi:

```
type
  TNumeroFormateado = class
  public
    function CadenaFormateada: string; virtual; abstract;
  end;
```

El ejemplo anterior muestra una clase abstracta, se reconoce como el establecimiento de la base para una serie de clases derivadas que todas tienen algo en común. Todas tienen la capacidad para formatear algún tipo de número por medio del método llamado CadenaFormateada.

Nunca se debe crear una instancia de una clase abstracta. La clase es creada solo para establecer las reglas para sus clases descendientes. Considerese el siguiente fragmento de código:

```
var
  MiNumero : TNumeroFormateado;
begin
  MiNumero := TNumeroFormateado.Create;
  MiNumero.Free;
end;
```


TESIS CON
FALLA DE ORIGEN

Este código, regresa un warning (advertencia):

```
Constructing instance of 'TNumeroFormateado' containing abstract methods
```

El código, compila y corre perfectamente. Sin embargo, al intentar utilizar MiNumero el compilador regresa una excepción de tipo EabstractError que indica que se ha intentado llamar un método abstracto. Para que TNumeroFormateado sea utilizado, se necesita derivar una clase de la misma tal como se presenta a continuación:

```
type
  TNumeroFormateado = class
  public
    function CadenaFormateada: string; virtual; abstract;
  end;

TEnteroFormateado = class(TNumeroFormateado)
private
  FValor : Integer;
public
  function CadenaFormateada: string; override;
end;

TFlotanteFormateado = class(TNumeroFormateado)
private
  FValor : Double;
public
  function CadenaFormateada: string; override;
end;
```

Una vez definidas las clases, se pueden utilizar como a continuación se presentan:

```
procedure MostrarNumeroFormateado (F: TNumeroFormateado);
begin
  ShowMessage(CadenaFormateada);
end;
```

El procedimiento MostrarNumeroFormateado no le importa si se le pasa un TEnteroFormateado o TFlotanteFormateado. Simplemente llama el método CadenaFormateada de la clase apropiada.

Ahora, veamos la misma estructura utilizando una interfaz. Una interfaz tiene los mismos efectos pero además tiene algunos otros adicionales.

```
type
  INumeroFormateado = interface
    function CadenaFormateada: string;
  end;
```

TESIS CON
FALLA DE ORIGEN

Conceptualmente, una interfaz no es más que un contrato entre el implementador de la interfaz y el usuario de la misma. El usuario de la interfaz puede codificar la especificación de la interfaz sin preocuparse de que esa especificación cambie.

A pesar de las similitudes entre clases e interfaces, también existen diferencias importantes. Las interfaces tienen las siguientes características:

- Una interfaz es declarada como un tipo de interfaz, y no como un tipo de clase.
- Todas las interfaces heredan, directa o indirectamente, de IUnknown, siendo ésta la raíz de todas las interfaces en COM.
- No se puede crear una instancia de una interfaz.
- Todos los métodos definidos por una interfaz son públicas.
- Una interfaz no puede declarar variables, solo determina que funcionalidad será proporcionada.
- Todas las funciones y procedimientos declarados en una interfaz son por definición funciones y procedimientos virtuales y abstractos.

Una vez que se define y publica una interfaz, no se puede modificar. Obviamente, en el proceso de desarrollo de la interfaz, se puede realizar cualquier tipo de modificación, pero no cuando haya sido liberada a otros para su uso. Si se desea mejorar la intefraz, se deberá realizar una nueva versión de la misma.

Al declarar una función se observa de la siguiente manera:

```
InumeroFormateado = interface
 ['{2DE825C1-EADF-11D2-B39F-0040F67455FE}']
 function CadenaFormateada: string;
end;
```

Nótese la cadena justo debajo del nombre de la interfaz. Esta cadena es conocida como "Identificador Único Global" (Globally Unique Identifier - GUID). Todas las interfaces COM, así como aquellas que son usadas internamente en la aplicación requieren un GUID para que funcionen de manera adecuada.

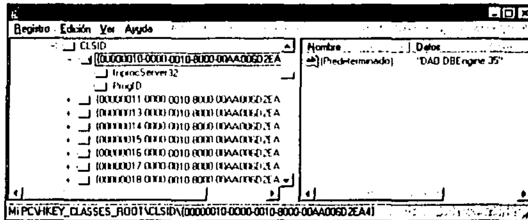
TESIS CON
FALLA DE ORIGEN

b) Definición de GUID

Un GUID es un número único de 16 bytes. El algoritmo para crear un GUID es bastante complejo y fue definido por Open Software Foundation. El algoritmo toma en cuenta la fecha y hora actual, el número de proceso del programa utilizado para generar el GUID y el ID único almacenado en la tarjeta de red. Todas las tarjetas de red tienen un número serial único conocido como dirección MAC. Si la PC donde se declara la interfaz no tiene tarjeta de red, el GUID garantiza ser unico debido al número de sus dígitos y el algoritmo que lo genera.

Dentro del registro de Windows, existe una llave llamada HKEY_CLASSES_ROOT/CLSID, cada CLSID o GUID, representa una implementación de una interfaz COM.

Por ejemplo, en la figura 4.3 se presenta el CLSID de la interfaz de la ingeniería DAO de Microsoft (Data Access Objects). La llave InprocServer32 contiene información que es usada por Windows para localizar la DLL de DAO en la máquina. La llave ProgID especifica el nombre entendible para los seres humanos por la cual esta interfaz puede ser referenciada para propósitos de automatización.



4.3 Registro de Windows con la lista de clases COM

c) Interfaces y COM

Un objeto COM no es más que un objeto que implementa una o más interfaces. Los objetos COM están implementados ya sea en una DLL o en un archivo ejecutable.

TESIS CON
FALLA DE ORIGEN

Los objetos COM que residen en una DLL se conocen como *servidores dentro del proceso.* Los objetos COM de los EXEs se conocen como *servidores fuera del proceso.* Los servidores COM deben contener uno o más objetos COM.

Los servidores dentro del proceso toman su nombre por implementarse dentro de una DLL. El servidor ocupa el mismo espacio de dirección que utiliza la aplicación. Los servidores de este tipo exportan cuatro funciones estandar: DllRegisterServer, DllUnregisterServer, DllGetClassObject, y DllCanUnloadNow.

- *DllRegisterServer.* Registra los objetos COM con el registro de Windows.
- *DllUnregisterServer.* Revierte el proceso de DllRegisterServer.
- *DllGetClassObject.* Es el responsable de proveer un mecanismo para que el servidor COM implemente una fábrica de clase (class factory) por cada objeto COM que exporte.
- *DllCanUnloadNow.* COM es el responsable de llamar a DllCanUnloadNow para ver si está bien descargar el servidor COM de la memoria. Si cualquier aplicación hace una referencia a cualquier objeto COM en este servidor, DllCanUnloadNow regresa FALSE. Si no hay referencias abiertas a cualquier objeto COM en este servidor, DllCanUnloadNow regresa TRUE, y COM remueve el servidor COM de la memoria.

Los servidores fuera del proceso son implementados en un ejecutable, consecuentemente, corren en un espacio de dirección separado de la aplicación cliente. Este tipo de servidores COM no exportan las cuatro funciones requeridas en un servidor dentro del proceso. Utilizan un método diferente para registrarse dentro de Windows. Para realizar este registro, solo se requiere ejecutar el servidor a través de un comando que depende del lenguaje de programación en el que está siendo diseñado, por ejemplo en el caso de Delphi se ejecuta el servidor con el parámetro /regserver en la línea de comando. Para eliminar el registro se utiliza la misma linea de comando con el parámetro /unregserver.

Un servidor COM fuera del proceso soporta uno de tres métodos de instancia:

- *Instancia simple.* Significa que solo una instancia del objeto COM está permitida por aplicación. Cada aplicación que solicita una instancia del objeto COM deberá activar una copia por separado del servidor COM.
- *Instancia múltiple.* Significa que el servidor COM está disponible para crear múltiples copias de un objeto COM. Cuando un cliente solicita una instancia del objeto COM, un nuevo servidor no es iniciado (al menos que el servidor aún no esté ejecutándose). En vez de eso, el servidor que actualmente se encuentra ejecutándose crea una instancia del objeto COM.
- *Únicamente Interno.* Es utilizado por objetos COM y no esta disponible para las aplicaciones clientes. La única aplicación que puede crear objetos COM es el servidor COM en donde residen.

B. El modo de ejecución COM+

COM+ soporta el desarrollo de componentes de alto rendimiento que pueden ser escalados a aplicaciones Web. También soporta una arquitectura distribuida donde el cliente reside en un proceso (conocido como *proceso cliente*) y una instancia de componente que reside en otro proceso (conocido como *proceso componente*). Usualmente estos dos procesos se encuentran en equipos diferentes.

Los clientes y las instancias no se comunican directamente entre sí, utilizan intermediarios. El proceso cliente tiene un sustituto de la instancia actual, llamado proxy. El proceso componente tiene un sustituto para el cliente, llamado stub[20]. Existe, al menos conceptualmente, un proxy por cada instancia del componente. El cliente se comunica con el proxy suponiendo que habla con la instancia. La instancia toma la respuesta del stub completamente convencido de que ésta proviene directamente del cliente.

Una vez que es creado un componente es necesario ponerlo en el ambiente COM+ a través de MMC (Microsoft Management Console). MMC es una herramienta administrativa que integró Microsoft y que permite administrar muchos de los servicios de la

[20] Término utilizado para indicar el llamado de una instancia desde otra instancia. En COM+ stub es sinónimo de proxy. En CORBA cuando se trata de una instancia del servidor también se le conoce como *esqueleto*.

computadora. Uno de estos servicios es el ambiente de ejecución de COM+ que se encuentra administrado por los servicios del componente. MMC tiene cuatro categorías de funciones de administración para COM+:

- Introduce componentes COM+ dentro del ambiente de ejecución COM+.
- Define características del componente en tiempo de ejecución.
- Crea paquetes de componentes que pueden ser instalados en otro cliente o ambientes COM+ en otros equipos.
- Monitorea los componentes.

1. Introducción de los componentes al ambiente COM+

Para poder inciar con este tema, es conveniente hablar de lo que Microsoft define como aplicación. Una aplicación no es mas que una colección de componentes que correrán de manera conjunta dentro de un proceso componente.

Cuando se utiliza MMC para crear una nueva aplicación COM+, lo que en realidad está haciendo es definiendo una nueva colección de componentes que correrán juntos. Una vez definida la aplicación, se introducen los componentes COM+ a la misma.

Por lo general los componentes son creados y guardados en forma de DLLs que incluyen información de librerías de tipo. Son estas DLLs las que son añadidas a una aplicación COM+. Cada vez que una DLL es introducida a una aplicación COM+, la base de datos administrativa se actualiza para incluir la información de clase del componente y la ubicación de la DLL, además incluye la aplicación que ha sido añadida.

2. Características de activación del componente de ejecución.

La segunda función administrativa de MMC es permitir a los administradores de la aplicación definir las características del componente en tiempo de ejecución. El objetivo del software distribuido es simplificar la vida de los programadores permitiéndoles enfocarse en problemas del negocio en vez de problemas de ambiente. Elimina la necesidad de utilizar APIs para el diseño de aplicaciones multicapas.

MMC permite definir características en tiempo de ejecución desde lo más complejo (colección de componentes o "aplicaciones") a lo más simple (el método). Las características en tiempo de ejecución mas importantes que se pueden administrar son las siguientes:

- Seguridad.
- Transacciones
- Hilos y concurrencia
- Administración de instancias

C. Los servicios de COM+

Con el surgimiento de COM+, Microsoft añadió nuevas capacidades a la colección de COM, DCOM y MTS en una categoría general conocida como *servicios* COM+:

- *Componentes asíncronos.* Permiten que las invocaciones de método sean almacenadas y procesadas en el futuro. Por ejemplo, una tienda en línea debe configurar su componente de compra de manera asíncrona, así las instancias de compra pueden procesar ordenes por la noche, cuando la carga del sistema es menor.
- *Eventos.* Los eventos permiten que las instancias de un componente respondan a eventos inicializados por otras instancias. Por ejemplo, en la tienda en línea se debe tener un disparador de seguridad en el componente de ordenes para ser activado en caso de que el costo de cualquier orden exceda algún monto especificado.
- *Componente de balance de carga.* Este componente permite distribuir la carga de trabajo a un gran número de equipos. Por ejemplo, en la tienda en línea se puede configurar el componente de ordenes para ser dinámicamente balanceados en tres equipos, de esta manera el sistema puede tomar ordenes en un índice que cualquier máquina pueda procesar.
- *Base de Datos en memoria.* (IMDB In-Memory Database) tiene una región de memoria compartida por equipo que puede ser utilizada para compartir información entre las instancias de los componentes que se encuentran ejecutándose en esa máquina. Por ejemplo, una aplicación de una tienda en línea puede almacenar un catálogo de los artículos y precios en la IMDB.

1. Diferencia entre los componentes síncronos y asíncronos.

La comunicación entre componentes síncronos y asíncronos es muy diferente, mientras que los componentes síncronos se comunican a través de la invocación de métodos, la comunicación asíncrona lo hace a través de mensajes en cola. Hoy en día la mayoría de los programadores están más familiarizados con métodos debido a la programación orientada a objetos, y no están dispuestos a comprender por completo un nuevo API cuando pueden invocar un método, en otras palabras lo que quieren es un componente que trabaje de manera asíncrona.

Para entender mejor el concepto de componentes síncronos y asíncronos considérese el siguiente fragmento de código.

```
De_Cuenta.extraer(IDDeCuenta, monto);
A_Cuenta.depositar(IDACuenta, monto);
```

En este fragmento de código no se puede saber si los metodos son invocaciones síncronas o asíncronas. Con la siguiente modificación de código, se elimina la posibilidad de invoación asíncrona:

```
error := De_Cuenta.extraer(IDDeCuenta, monto);
If (no error) then A_Cuenta.depositar(IDACuenta, monto);
```

El segundo segmento del código debe ser síncrona porque el código de invocación utiliza el valor de regreso del método *extraer* para decidir si procesar o no el método *depositar*. Si el método *extraer* fuera asíncrono en vez de síncrono, entonces no habría manera de saber cuando es ejecutado el código que está detrás de el método *extraer*.
En algún nivel, los requerimientos para los componentes asíncronos son un subconjunto de requerimientos para los componentes síncronos. Todo componente asíncrono se puede convertir potencialmente en componente síncrono. La inversa no es posible. Por ejemplo, los componentes síncronos pueden regresar información a través de parámetros, los componentes asíncronos no lo pueden hacer, si un componente regresa información por medio de un parámetro, está siendo síncrono, pero si no, puede ser síncrono o asíncrono.

TESIS CON
FALLA DE ORIGEN

La arquitectura de un componente asíncrono, como se muestra en la figura 4.4, es solamente una capa entre COM+ y MSMQ. Esta capa está compuesta por tres piezas: un receptor, un ejecutor y un oyente.

Un receptor es un proxy análogo. Es específico a un componente asíncrono, y soporta la misma interfaz que el componente asíncrono. Reside en el proceso cliente y es la instancia del componente. Refiriendo la figura 4.4, el receptor es el que recibe la solicitud del método desde el cliente (1). Tal como un proxy, el receptor no contiene las reglas del negocio, pero contiene la lógica necesaria para transportar la solicitud del método (2) hacia el proceso del componente. Diferente a un proxy, el transporte no está basado en los mecanismos de transportación de COM+, en vez de eso está basado en mensajes en cola (conocidos también como datagramas).

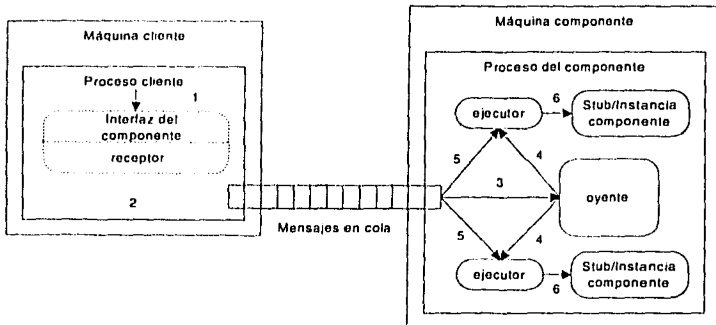


Figura 4.4 Arquitectura de componentes asíncronos

El oyente generalmente reside en el proceso del componente. Se encarga de monitorear los datagramas, esperando hasta que exista un mensaje disponible. Como puede haber cientos de mensajes en el sistema, el oyente junto con el ejecutor deciden cual es el mensaje que está siendo utilizado. Una vez que el oyente notifica que hay una solicitud de método en el datagrama (3), solicita al ejecutor por medio de un método (4), trabajar con el mensaje. El ejecutor recibe el mensaje de la cola de mensajes (5), traslada el mensaje por medio de una invocación al método e invoca ese método directamente en.el stub al

TESIS CON
FALLA DE ORIGEN

componente asíncrono (6). Debido a que el ejecutor y la instancia stub residen en el mismo proceso y comparten todas las características en tiempo de ejecución de COM+, el uso de un proxy puede ser eliminado, permitiendo la comunicación entre el ejecutor y el stub.

Por lo tanto, hay tres piezas distintas en la arquitectura de los componentes asíncronos. Están el oyente/receptor/ejecutor, quienes trasladan desde la solicitud del método a las colas de mensajes y luego regresan nuevamente. MSMQ es proveedor de los servicios de transporte asíncronos. Y COM+ procesa la solicitud del método utilizando su procesamiento normal síncrono una vez que el mensaje ha recibido asíncronamente por medio del ejecutor de MSMQ.

Con el comportamiento de la arquitectura síncrona y asíncrona de los componentes que se explicó anteriormente, la desición de que sea un componente síncrono o asíncrono depende más que nada de su configuración. El que un componente dado se comporte algunas veces de manera síncrona y otras veces de forma asíncrona depende de varias razones, todas estas están relacionadas con el uso de MSMQ por ser el mecanismo de interproceso para los componentes asíncronos en vez de COM+ que es utilizado para el transporte síncrono. Para entender mejor, explicaré las diferencias inherentes entre estos dos mecanismos de transporte.

Para comenzar, en ambos casos un bloque de información de la invocación de un método es transportada. Por lo menos, ese bloque debe incluir información que identifica al método y todos los parámetros que son pasados al método. El bloque es creado en el proceso cliente, es decir, el proceso que originalmente invocó al método. El bloque debe ser transportado al proceso componente donde reside el stub y donde la instancia será asignada para procesar el método utilizando algoritmos de administración de instancias.

Una vez que el bloque ha sido creado, se necesita de algún mecanismo para moverlo del proceso cliente al proceso componente. Se utiliza un componente síncrono en el mecanismo de transportación de COM+ y es conocido como DCOM.

DCOM es una tecnología que permite transportar todas las características de COM dentro de una red. De esta manera se puede desarrollar un servidor COM en una máquina, y el cliente accesa al servidor en otra máquina.

2. Arquitectura DCOM.

DCOM es una extensión del Modelo COM. COM define la manera en que los componentes y sus clientes interactúan. Esta interacción está definida de tal manera que el cliente y el componente puedan conectarse sin la necesidad de ningún componente de sistema intermediario. En la figura 4.5 se ilustra ésta en la notación de COM:

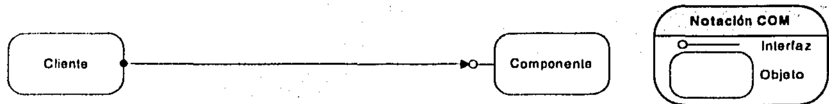


Figura 4.5 Componentes COM en el mismo proceso

En los sistemas operativos de hoy en día, los procesos están protegidos entre sí. Un cliente que necesita comunicarse con un componente en otro proceso no puede llamar al componente directamente, tiene que utilizar alguna forma de comunicación interprocesos proporcionada por el sistema operativo. COM proporciona esta comunicación de manera transparente: intercepta las llamadas del cliente y las envía al componente que se encuentra en otro proceso. La figura 4.6 ilustra como las librerías en tiempo de ejecución COM/DCOM proporcionan la liga entre el cliente y el componente.

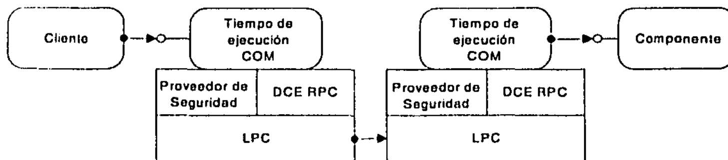


Figura 4.6 Componentes COM en diferentes procesos

TESIS CON
FALLA DE ORIGEN

Cuando el cliente y el componente residen en diferentes equipos, DCOM reemplaza la comunicación interprocesos local con el protocolo de red. Tanto el cliente como el componente están concientes de que el medio que los conecta se encuentra en otro lado.

La figura 4.7 muestra por completo la arquitecutra DCOM: El objeto en tiempo de ejecución COM proporciona servicios orientados a objetos hacia los clientes y los componentes, además utiliza RPC y el proveedor de seguridad para generar paquetes de red estándar que conforman el protocolo DCOM estándar.

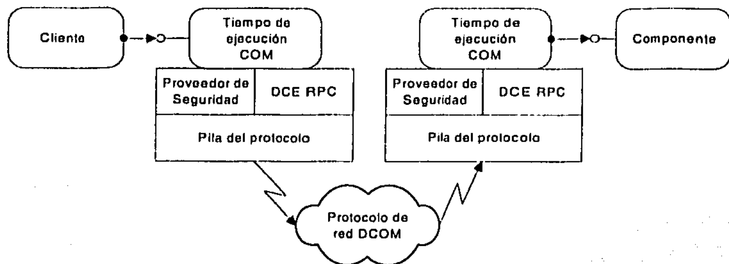


Fig. 4.7 DCOM: Componentes COM en diferentes equipos

Beneficios DCOM:

- *Neutralidad de lenguaje.* Si una aplicación es escrita en el lenguaje Delphi, puede ser llamada y utilizada por otra aplicación hecha en Visual Basic. Esto permite que los desarrolladores no se encierren en una sola herramienta permitiéndoles seleccionar la mejor herramienta para trabajar.
- *Administración de conectividad.* DCOM utiliza un contador de referencia para determinar cuántos clientes están conectados en un servidor en particular. De este modo, cuando un cliente nuevo es añadido a la lista, el contador se incrementa, e inversamente, cuando el cliente se desconecta de la aplicación servidora, la referencia se decrementa. Cuando el contador de referencia se hace cero, la aplicación se cierra. Cuando una aplicación servidora no está ejecutándose, DCOM activa la aplicación y actualiza el contador de referencia apropiadamente. Si un cliente

TESIS CON
FALLA DE ORIGEN

es desconectado sin decrementar el contador de referencia, debe existir una forma de que el servidor automáticamente decremente el contador de referencia. Esto es llevado a cabo por el servidor haciendo un "ping" al cliente. Si el servidor falla al "ping" por tres ocasiones, la conexión es considerada como cerrada y el contador de referencia es ajustado.

➲ *Interfaces múltiples.* DCOM es implementado por el uso de interfaces. Las interfaces son los métodos expuestos y propiedades de un objeto en particular. Pueden haber muchas interfaces en el mismo objeto lo que permite la habilidad de adaptar un objeto sin la necesidad de rediseñarlo. Esto se convierte en una característica importante cuando el objeto requiere de funcionalidad adicional que sólo un pequeño conjunto de clientes lo solicitan. En este caso, el desarrollador puede utilizar la misma implementación con interfaces diferentes.

➲ *Integración con otros protocolos de red.* Debido a que DCOM está basado en TCP/IP y protocolos RPC, las aplicaciones servidoras pueden ser llamadas desde cualquier parte en Internet utilizando el básico TCP/IP o cualquier otro protocolo basado en TCP (como PPTP (Point to Point Tunneling Protocol)) para usuarios dial-in[21], o para usuarios HTTP de internet . La red de comunicación de DCOM es transparente a las aplicaciones debido al bajo nivel de interoperabilidad con la red, lo que proporciona varias opciones de red al desarrollador para que decida la mejor conectividad para la aplicación.

➲ *Independencia de ubicación y balanceo de carga.* La forma en que una máquina cliente "encuentra" una aplicación servidora es a través de la localización del registro apropiado en el archivo de registro del cliente. Cambiando la información de registro del Servidor A al Servidor B causará que el cliente cargue la aplicación en el Servidor B. Cuando una aplicación tiene muchos usuarios, pueden ser divididos en grupos de usuarios. Por ejemplo, el Grupo 1 puede utilizar una aplicación servidora de inventario que está ejecutándose en el Servidor A, mientras que el Grupo 2 puede usar la misma aplicación pero corriendo en el Servidor B. Al hacer esto, la carga de trabajo puede estar balanceada entre varias máquinas. Este tipo de balance de carga es conocido como *estático* porque los usuarios se conectan a la misma máquina servidora, sin importar cuantos usuarios estén conectados. A pesar que esta situación es buena para situaciones simples, la mayoría de las veces la carga necesita estar balanceada

[21] Usuarios que se conectan remotamente vía módem a un servidor o sistema.

de manera *dinámica*. Considerando el ejemplo anterior, si a cierta hora del día, el Grupo 1 requiere utilizar más la aplicación servidora. Al mismo tiempo, las necesidades del Grupo 2 son menores. Si el balance de carga estático es utilizado, el Servidor 2 será poco utilizado, mientras que el Servidor 1 está sobre utilizado. Si hubiera una manera de intercambiar dinámicamente algunos usuarios del Grupo1 al Servidor 2 en vez del Servidor 1, basado en el tráfico existente de red para cada servidor, la utilización sería balanceada entre dos servidores mejorando el rendimiento. DCOM proporciona una infraestructura para implementar el balanceo de carga dinámicamente. Componentes de remisión simple pueden ser utilizados para implementar de manera transparente localidades de servidor dinámicas en tiempo de conexión. Mecanismos más sofisticados para el re-enrutamiento de los métodos de invocación hacia los diferentes servidores pueden ser fácilmente implementados, pero requieren de un conocimiento más a fondo de la interacción entre los clientes y los componentes.

➲ *Tolerancia de errores.* DCOM proporciona un fácil soporte para reconocer las fallas debido a sus facilidades de "ping".

DCOM hace fácil la implementación de la tolerancia de errores. Cuando el cliente detecta la falla de un componente, se reconecta al mismo componente de remisión con quien estableció la primer conexión. El componente de remisión tiene información acerca de cuales servidores ya no están disponibles y automáticamente proporciona al cliente una nueva instancia del componente que está ejecutándose en otra máquina. Las aplicaciones, por supuesto, aún tendrán que tratar con la recuperación de error a niveles más altos (consistencia, pérdida de información, etc.).

Con la habilidad de DCOM para dividir un componente en el lado del servidor y uno en el lado del cliente, la conexión y la reconexión a los componentes, así como la consistencia, puede hacerse completamente de manera transparente para el cliente.

Existe otra técnica, comúnmente conocida como *respaldo novedoso.* Consiste en dos copias del mismo componente servidor corriendo en paralelo en diferentes equipos y procesando la misma información. Los clientes pueden conectarse explícitamente a ambas máquinas simultáneamente. Los componentes distribuidos DCOM, hacen esta acción completamente transparente a la aplicación cliente por medio de la inserción de código servidor en el lado del cliente, quien maneja la tolerancia a errores. Otro método puede utilizar un componente de coordinación ejecutándose en una máquina

diferente quien se encarga de publicar las solicitudes del cliente a ambos componentes servidores en nombre del cliente.

En la figura 4.8 se ilustra el esquema cliente/servidor de la arquitectura DCOM, donde lo que es visible para todos los desarrolladores de los programas cliente y servidor es la "Arquitectura básica de programación[22]". La capa intermedia es la "Arquitectura remota[23]" que es transparente a los desarrolladores y está encargada de crear los apuntadores de la interfaz o las referencias de los objetos de manera significativa entre los procesos. La última capa es la "Arquitectura del protocolo[24]", la cual se encarga de distribuir la arquitectura entre las diferentes máquinas.

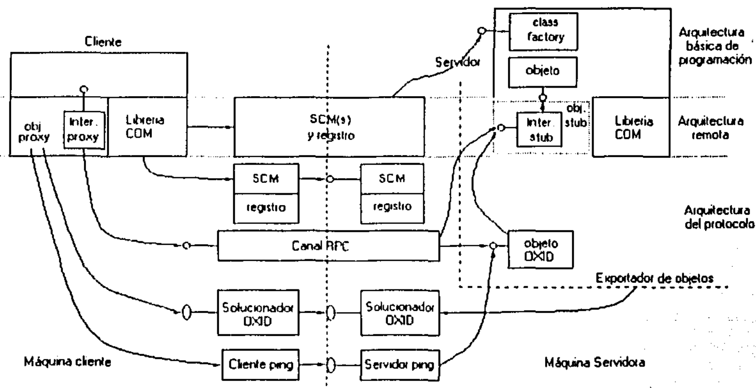


Fig. 4.8 Arquitectura del modelo DCOM

[22] Representa la vista de los programadores. Describe como un cliente solicita un objeto e invoca sus métodos, y cómo el servidor crea una instancia de un objeto y la hace d· ,ponible al cliente.

[23] Consiste en la infraestructura necesaria para ofrecer al cliente y al servidor la ilusión de encontrarse en el mismo espacio de memoria.

[24] Especifica el protocolo para soportar la comunicación entre clientes y servidores y para que corran en diferentes máquinas.

TESIS CON
FALLA DE ORIGEN

D. Interoperabilidad COM

Microsoft diseñó Windows 2000 y COM+ para resolver ciertos problemas específicos, así como existen sistemas operativos que también resulven otros problemas en específico. Ante esta situación, los usuarios deben elegir sistemas que mejor se adapten a sus problemas para luego centrarse en encontrar la forma para hacer que ambos sistemas trabajen de manera conjunta. A esta forma de trabajo es a la que se conoce como interoperabilidad.

Los sistemas de Microsoft y sus aplicaciones pueden interoperar con casi cualquier máquina virtual existente (UNIX, Apple, sistema IBM AS/400) y en la mayoría de ellos de muchas maneras.

En el desarrollo de este tema me voy a limitar al área en la que más puede impactar la interoperabilidad entre sistemas, en específico estoy hablando de la Web. Me centraré en tres diferentes áreas de la interoperabilidad de la capa intermedia: con clientes No-Microsoft, con aplicaciones de la capa intermedia no-Microsoft y con bases de datos no-Microsoft.

1. Interoperabilidad de la capa cliente

Cuando se piensa en interoperar con clientes no-Microsoft, en general se está hablando de navegadores de Web. Esto hace un poco más complicado hablar de la relación que existe entre la capa componente y el navegador, porque se tiene una capa en medio de ambos, para este contexto la llamaré "capa Web". Para Microsoft, la capa Web es el IIS (Internet Information Service). Asúmase entonces que la configuración es el componente COM+ como cliente interoperando com Microsoft IIS como la capa Web, e IIS interoperando con navegadores no-Microsoft. Véase la figura 4.9.

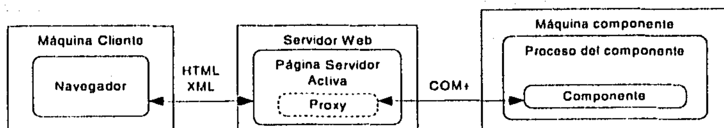


Fig. 4.9 Interoperabilidad del cliente

La interoperabilidad con los navegadores no-Microsoft es complicada debido a su rápida evolución de estándares en el área de navegadores. A pesar de que se tienen estándares para XML (eXtensible Markup Language) y su tecnología relacionada para describir datos, y DHTML (Dynamic HyperText Markup Language) para describir interfaces dinámicas de usuarios, si queremos escribir sistemas que interoperen con la mayoría de los navegadores posibles, hoy solo pudieramos asumir que existe soporte para HTML (HyperText Markup Language).

La mayoría de los desarrolladores están más familiarizados con HTML. El mayor problema de HTML es que es estático. El texto es creado en la capa Web y enviada al navegador donde es desplegada. Pero, para la solicitud de nuevas páginas Web se necesita regresar a la capa Web, la cual genera las nuevas descripciones HTML y las envía de vuelta al navegador. Esto resulta en una interfaz lenta para el usuario, la mayoría de las actualizaciones de despliegue requieren de esperar de una nueva descripción de HTML desde la capa Web.

DHTML está diseñado para eliminar, lo más que se pueda, los viajes de regreso a la capa Web para las actualizaciones de despliegue en la máquina cliente. La mayoría de las nuevas desiciones de despliegue pueden ser hechas por la máquina navegador.

XML establece una manera estándar de describir los datos. En un sentido, XML no es nada nuevo, es solamente una cadena de texto que contiene datos. Los estándares de XML establecen que la cadena de datos contienen una serie de nombres de campos y datos. Los nombres de los campos son identificados por la subcadena *<nombre-campo>*. Cualquier cadena subsecuente al nombre del capo es asumida como los datos para ese

TESIS CON
FALLA DE ORIGEN

campo, hasta encontrar la subcadena *</nombre-campo>*. La siguiente subcadena XML describe el titulo de un libro:

```
"<titulo>Romeo y Julieta</titulo>"
```

Y la siguiente subcadena XML describe el autor:

```
"<autor>William Shakespeare</autor>"
```

También se necesita mostrar la manera en como estas dos subcadenas se relacionan. Para ello se encierran ambos rubros dentro de uno más grande que determine que ambos son parte de un concepto más amplio:

```
"<libro>
<titulo>Romeo y Julieta</titulo>
<autor>William Shakespeare</autor>
</libro>"
```

¿Cómo impacta esto a la capa intermedia? En un sentido, en nada. Es responsabilidad de la capa Web de crear cadenas XML y regresarlas al navegador. Pero en otro sentido, XML tiene un alto impacto en los mismos componentes, debido a la espectativa de cómo los componentes regresarán la información. Hoy en día se encuentran algunos programadores diseñando componentes para regresar cadenas XML. Los desarrolladores que se encuentran haciendo la transición de la programación orientada a objetos a programación de componentes se encuentran con la dificultad de diseñar sus propias interfaces. Ellos están acostumbrados a diseñar interfaces con métodos individuales que regresan datos específicos, pudiendo concentrar la información en un componente, que para el ejemplo especificado seria un componente libro como este:

```
type
 Ilibro   interface
   Procedure GuardaTitulo(ISBN: integer, Titlo: string);
   Procedure GuardaAutor(ISBN: integer, Autor: string);
   Procedure GuardaEditor(ISBN: integer, Editorial: string);
   Function LeeTitulo(ISBN: integer): string;
   Function LeeAutor(ISBN: integer): string;
   Function LeeEditor(ISBN: integer): string;
 End;
```

La llave en éste ejemplo es ISBN que es un número único de referencia del libro, sin embargo, ésta interfaz se vuelve obsoleta cuando deseamos agregar otra característica

TESIS CON
FALLA DE ORIGEN

del libro, como sería el caso de guardar el precio, para esto se puede diseñar una interfaz como la que a continuación se describe, donde se pasa información vía cadenas XML:

```
type
  ILibro = interface
    Procedure GuardaLibro(ISBN: integer, Info: string);
    Function LeeLibro(ISBN: integer): string;
  End;
```

El utilizar una cadena XML es una forma genérica de invocar un método de un componente sin tener que conocer nada sobre la tecnología del componente utilizada para implementarlo. El cliente solo realiza su solicitud a través de cadenas XML las cuales contienen el nombre del componente, el nombre del método, y los parámetros. La arquitectura general para este tipo de sistemas se muestran en la figura 4.10.

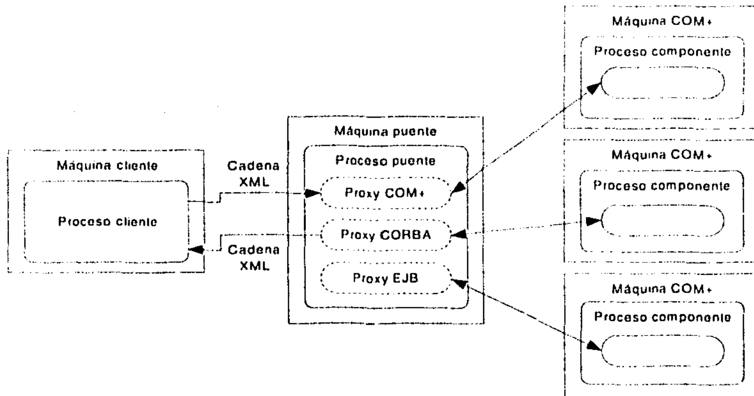


Fig. 4.10 Arquitecura basada en XML

2. Interoperabilidad de la capa intermedia

Es en la capa intermedia donde la interoperabilidad se hace más compleja. El uso de la arquitectura basada en XML en la capa intermedia, puede ser una forma muy efectiva de enlazar al cliente con la capa intermedia. Cuando los componentes en la capa intermedia, invocan un método u otro componente, el componente que los invoca se convierte en un

TESIS CON
FALLA DE ORIGEN

cliente. La figura 4.11 muestra una arquitectura similar a la descrita en la figura 4.10 siendo utilizada como puente entre COM+ y un componente EJB.

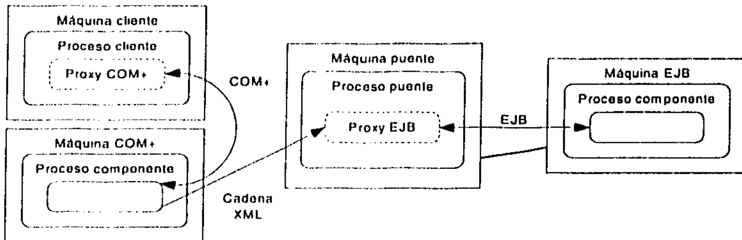


Fig. 4.11 Arquitectura de la capa intermedia basada en XML como puente

Existe también, otra tecnología que puede servir como puente y es específica para unir componentes COM+ con los componentes CORBA. CORBA se describirá más a detalle en el siguiente capítulo. La tecnología puente que une a COM con CORBA es *COM/CORBA bridge.*

El COM/CORBA bridge es la metodología oficial de OMG para interoperar entre COM y CORBA. Ésta metodología fue diseñada en 1997 y puede ser utilizada para construir componentes intermediarios que puedan comunicarse tanto con COM como con CORBA. La tecnología de puente basada en COM/CORBA se muestra en la figura 4.12.

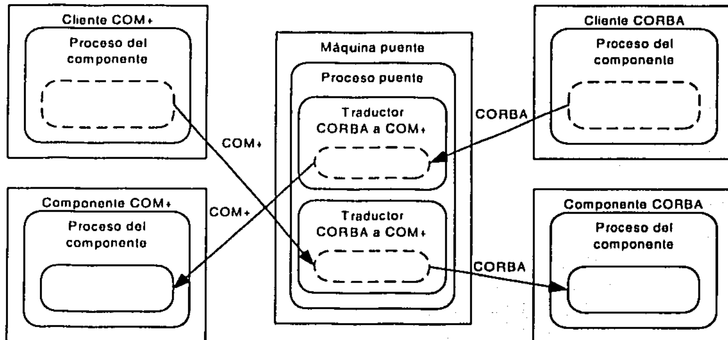


Fig. 4.12. Puentes COM/CORBA basados en la tecnología OMG

TESIS CON
FALLA DE ORIGEN

Existen problemas con esta tecnología. Primero, como los puentes COM/CORBA están basados en interfaces de componentes, el puente solo puede ser utilizado siempre y cuando se apliquen las interfaces apropiadas para así poder unir los dos sistemas. En segundo lugar, estos puentes son por lo general síncronos. Un método CORBA es recibido, traducido como método COM+ y es invocado de inmediato. Estos puentes por lo general no tienen la capacidad de almacenar métodos para un procesamiento posterior. Los puentes, consecuentemente, están limitados a trabajar con aplicaciones que se encuentran ejecutándose al mismo tiempo.

A pesar que COM+ corre únicamente bajo Windows 2000, Microsoft ha diseñado componentes COM+ básicos y tecnología de distribución (el COM original y la tecnología DCOM) disponible para otras plataformas. Esto hace posible el ejecutar una aplicación en UNIX, por ejemplo.

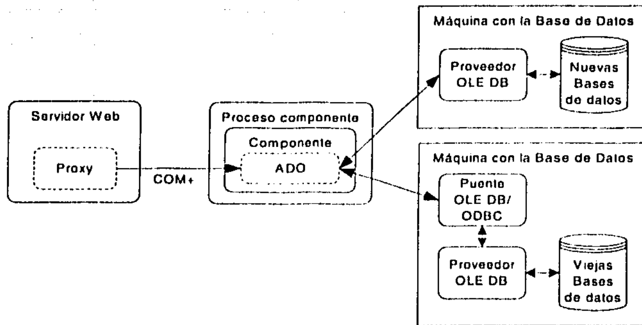
Otro producto que merece ser mencionado es el servidor SNA, la nueva versión de Babylon. Ésta es una tecnología de conectividad de Microsoft para mainframes, el más importante es probablemente la tecnología COMTI que permite aplicaciones back-end para CICS e IMS IBM para trabajar dentro de un marco COM+. Para la prespectiva de CICS o IMS, COMTI es únicamente otro cliente.

3. Interoperabilidad de la base de datos

En la figura 4.13, se muestra una visión general de la interoperabilidad de bases de datos de Microsoft. Las reglas del negocio utilizan un conjunto de componentes de acceso a datos llamados ActiveX Data Objects (ADO).

ADO está diseñado para ser una interfaz general para el acceso a datos, y para trabajar bien con una gran variedad de bases de datos. ADO es la interfaz a nivel del cliente. Los productos de datos pueden conectarse con ADO por medio de dos Interfaces proveedores de Servicio (SPI Service Provider Interfaces): ODBC y OLE DB.

TESIS CON
FALLA DE ORIGEN



4.13 Interoperabilidad de bases de datos Microsoft

ODBC es el API original de Microsoft y es utilizado para el acceso a datos, es proyectado en su mayoría a bases de datos relacionales. OLE DB es un nuevo SPI (Service Provider Interface), enfocado a todos los productos de almacenamiento de datos, sean relacionales o no. Existen providers de OLE DB para Oracle, Sybase, Informix, SQL Server, RDB, Ingres, Non-Stop SQL/MP, Non-Stop SQL/MX. RMS, DBMS, MUMPS, Adabas, IBM RS/6000 y ObjectStore, entre otros productos de bases de datos.

TESIS CON
FALLA DE ORIGEN

CAPITULO V

EL MODELO CORBA EN LA ARQUITECTURA MULTICAPAS

A. Introducción

CORBA es acrónimo de Common Object Request Broker Architecture. Es el producto de Object Management Group (OMG). Las especificaciones de CORBA definen un objeto bus llamado Object Request Broker (ORB) el cual suministra la infraestructura que permite que objetos interoperen entre direcciones, lenguajes, sistemas operativos y redes, es decir, permite la comunicación entre plataformas por medio de objetos distribuidos y programas clientes. En el mercado actual, CORBA es el estándar de objetos distribuidos más aceptado. CORBA especifica los estándares necesarios para la invocación de métodos sobre objetos en entornos heterogéneos.

Los entornos heterogéneos son aquellos en los que las arquitecturas que los constituyen pueden ser sistemas Microsoft Windows, máquinas Unix de diferentes fabricantes (GNU/Linux entre otros) e incluso sistemas como MacOS y OS/2. Y es más, dentro de la heterogeneidad también se incluyen los sistemas de comunicaciones (protocolos de comunicación como TCP/IP, IPX, etc.) o los lenguajes de programación utilizados en las diferentes arquitecturas.

CORBA define su propio modelo de objetos, basado en la definición de las interfaces de los objetos mediante el lenguaje IDL. De esta forma se logra una abstracción de la heterogeneidad que permite que el uso de CORBA no sea nada complejo. CORBA sigue una metodología concreta y fácil de seguir.

Para la implementación de los objetos se puede utilizar cualquier lenguaje de programación que proporcione enlaces con el lenguaje IDL. Para que un lenguaje de

programación se pueda utilizar desde CORBA, debe tener definida la forma de enlazarse con IDL.

De esta forma, y a partir de una especificación de las interfaces en IDL, se generan unos cabos (stubs) en el lenguaje elegido que permiten el acceso a la implementación de los objetos desde la arquitectura CORBA.

CORBA es un estándar creado con la idea de una distribución de los sistemas basada en objetos. Con CORBA se pretende definir una arquitectura que especifique cómo se crean los objetos y cómo se accede a sus funcionalidades.

B. Historia de la OMG

La OMG empezó en 1989 con ocho compañías (Data General, Hewlett-Packard, Sun Microsystems, Cannon, American Airlines, Unisys Corporation, Philips Telecommunications N.V., y 3COM Corporation) se formó con la misión de crear un mercado de programación basada en componentes e impulsando la introducción de objetos de programación estandarizados. La OMG es una organización internacional cuya sede principal se encuentra en Framingham, MA, Estados Unidos. Tiene oficinas internacionales en el Reino Unido, Alemania, Japón, Australia, Brasil, Italia e India. Hoy, la OMG tiene alrededor de 800 vendedores de software, desarrolladores y usuarios finales, tales como Sun, JavaSoft, IBM, Netscape, Apple, Oracle, BEA Systems, Hewlett Packard, etc. La excepción notable es Microsoft, el cual tiene su propio producto COM+ definido en el capítulo anterior.

Los estatutos de la organización incluyen el establecimiento de guías para la industria y especificaciones detalladas para la gestión de objetos a fin de suministrar un marco de trabajo común para el desarrollo de un entorno de computación distribuida que abarque las principales arquitecturas de máquina y sistemas operativos.

Su proposito principal es desarrollar una arquitectura única basada en componentes de programación disponibles comercialmente, utilizando la tecnología de objetos, para la

integración de aplicaciones distribuidas que garantice la reusabilidad de los componentes, la interoperabilidad y la portabilidad.

Mientras que las primeras propiedades de la arquitectura propuesta por la OMG están basadas en los beneficios del uso de la tecnología, la última corresponde a una decisión estratégica del consorcio orientada a evitar las dificultades que han tenido los productos de otros organismos de normalización como la ISO para encontrar una adecuada respuesta en el mercado. Su objetivo, más allá de generar normas, es el de asegurarse de que estas normas sean utilizadas ampliamente, por lo que ninguna propuesta es adoptada como una norma a menos que describa una tecnología que ya se encuentre en el mercado o asegure una rápida disponibilidad. La arquitectura de la OMG está constituida por componentes (objetos) interoperables con interfaces normalizadas, ofrecidos en el mercado por diversas empresas, que los usuarios adquieren para construir sus propias aplicaciones.

A pesar de que en su nombre se menciona a los objetos en general, la OMG no pretende cubrir todos los ámbitos de aplicación de la tecnología de objetos, los cuales incluyen los lenguajes de programación, los métodos de análisis y diseño, las interfaces gráficas de usuario, las bases de datos, y los componentes. Su interés está centrado en la generación de normas con la especificación (de las interfaces) de los componentes requeridos para el desarrollo de aplicaciones distribuidas, o lo que es lo mismo, para la integración de aplicaciones.

La primer especificación adoptada por la OMG fue la *Arquitectura de gestión de objetos* (Object Management Architecture OMA). La OMA, figura 5.1, define un modelo de referencia para los objetos distribuidos y clasifica a los objetos en diferentes categorías. La comunicación entre objetos es lograda por medio del agente ORB el cual tiene la finalidad de ocultar la heterogeneidad y la distribución.

ESTA TESIS NO SALE

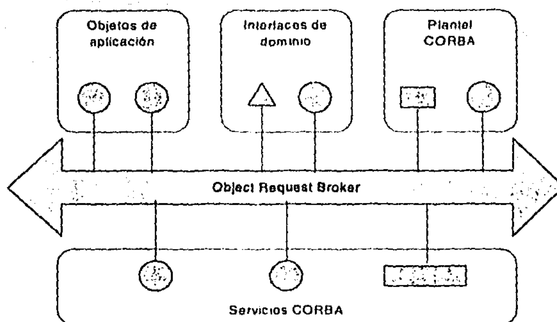


Figura 5.1 Arquitectura OMG

La OMG estableció una base común de partida para el trabajo con la OMA, y está constituida por los siguientes elementos:

- Un conjunto único de términos y definiciones para los conceptos a manejar en relación con la orientación a objetos: objeto, clases, mensaje, etc.
- Un marco de abstracción común, o modelo de objetos, que define un modelo matemático riguroso para las aplicaciones en términos de objetos y para la compartición de información entre aplicaciones en términos de mensajes.
- Un modelo de referencia común o arquitectura.
- Un conjunto común de interfaces, protocolos y lenguajes.

La segunda especificación substancial que la OMG produjo es CORBA (Common Object Request Broker Architecture). Esta especificación define el lenguaje de definición de interfaces para CORBA e identifica un número de ligas de lenguaje de programación. Estas ligas determinan como los objetos servidores y clientes pueden ser implementados en diferentes lenguajes de programación. La especificación de CORBA también define adaptadores de objeto que controlan la activación de los objetos, su desactivación y la generación de referencias de objetos.

Después de la primer versión de la especificación CORBA, la OMG comenzó trabajando con la especificación de los servicios CORBA (CORBA*services*). Un servicio, en

TESIS CON
FALLA DE ORIGEN

terminología de OMG, es un mecanismo básico que hace posible la construcción de sistemas distribuidos. En la actualidad la OMG ha adoptado más de una docena de CORBAservices. Éstos soportan la localización de objetos, la creación y el traslado de objetos, el control de acceso concurrente a objetos, manejo de transacciones distribuidas entre los objetos y asegura el acceso a los objetos.

Más recientemente, la OMG ha adoptado especificaciones de interface que son útiles pero, a diferencia de CORBAservices, no mandatorios para la construcción de un sistema distribuido. La ORB se refiere a estas interfaces como CORBAfacilities. Éstos incluyen interfaces para un sistema de ayuda distribuida, para facilitar la entrada de conjuntos de caracteres asiáticos, y la facilidad en el establecimiento de impresión y spooling independiente del sistema operativo.

En 1998 la OMG se centró en interfaces específicas para dominios (Domain Interfaces). La OMG las desarrolló para ser usadas en el mercado de los dominios, tal como el comercio electrónico, finanzas, telecomunicaciones, transporte y cuidado de la salud.

C. Arquitectura

CORBA es una arquitectura orientada a objetos. Por eso exhibe muchas de las características comunes a otros sistemas orientados a objetos, incluyendo la herencia de interfaces y el polimorfismo. Lo que diferencia a CORBA haciéndolo más interesante es que proporciona esta capacidad incluso cuando se utiliza con lenguajes no orientados a objetos como C y COBOL.

CORBA es una especificación normativa que resulta de un consenso entre los miembros de la OMG. Esta norma cubre cinco grandes ámbitos que constituyen los sistemas de objetos distribuidos:

➲ Un lenguaje de definición de interfaces IDL (Interface Definition Language), traducciones del lenguaje IDL a lenguajes de implementación (tales como C++, Java, ADA, etc.) y una infraestructura de distribución de objetos llamada Object Request

Broker (ORB) que ha dado su nombre a la propia norma: Common Object Request Broker Architecture (CORBA).

- Una descripción de servicios, conocidos con el nombre de Servicios CORBA (*CorbaServices*), que complementan el funcionamiento básico de los objetos que dan lugar a una aplicación. Estas especificaciones cubren los servicios de nombrado, persistencia, eventos, transacciones, etc.
- Una descripción de servicios orientados al desarrollo de aplicaciones finales, estructurados sobre los objetos y servicios CORBA que llevan el nombre de Facilidades Corba (*CorbaFacilities*), estas especificaciones cubren servicios de alto nivel, como las interfaces de usuario, los documentos compuestos, la administración de sistemas y redes, etc. CORBAfacilities pretende definir colecciones de objetos prefabricados para aplicaciones habituales en la empresa: creación de documentos, administración de sistemas informáticos, etc.
- Una descripción de servicios verticales denominados Interfaces de Dominio (*CorbaDomains*), que proveen funcionalidad de interés para usuarios finales en campos de aplicación particulares. Por ejemplo, existen proyectos en curso en sectores como: telecomunicaciones, finanzas, medicina, etc.
- Un protocolo genérico de intercomunicación, el Protocolo General Inter-ORB GIOP (General Inter-ORB Protocol), que define los mensajes y el empaquetado de los datos que se transmiten entre los objetos. Además define implementaciones de ese protocolo genérico sobre diferentes protocolos de transporte, lo que permite la comunicación entre los diferentes ORBs consiguiendo la interoperabilidad de elementos de diferentes vendedores. Por ejemplo el IIOP para redes con la capa de transporte TCP.

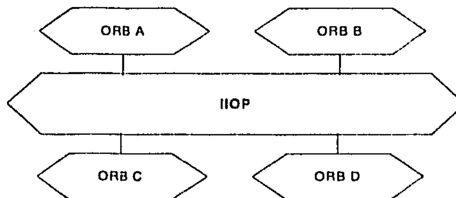


Figura 5.2. IIOP

En la figura 5.3 se ilustra el esquema cliente y servidor de la arquitectura CORBA, donde lo que es visible para todos los desarrolladores de los programas cliente y servidor es la "Arquitectura básica de programación". La capa intermedia es la "Arquitectura remota" que es transparente a los desarrolladores y está encargada de crear los apuntadores de la interface o las referencias de los objetos de manera significativa entre los procesos. La última capa es la "Arquitectura del protocolo", la cual se encarga de distribuir la arquitecutra entre las diferentes máquinas.

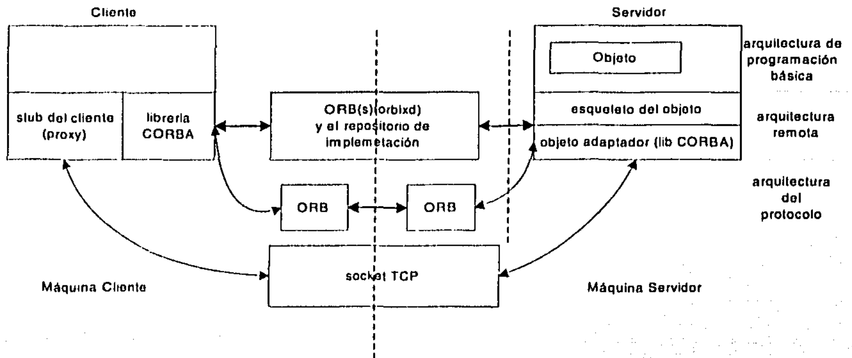


Fig. 5.3 Arquitectura Cliente/Servidor CORBA

Objetos

Todo objeto CORBA tiene un identificador único y múltiples referencias a objetos. Los clientes no conocen el contenido de las referencias de los objetos, para solictar la ejecución de una operación del objeto servidor, un cliente tiene que referenciar al objeto servidor. Los clientes pueden solicitar operaciones a objetos referenciados sin tener que saber donde se encuentran ubicados físicamente.

Las referencias de los objetos CORBA son persistentes, más adelante se definirá el concepto de persistencia.

TESIS CON
FALLA DE ORIGEN

1. Comunicación entre objetos

En el transcurso de este trabajo se ha definido el término ORB (Object Request Broker), considerado uno de los pilares fundamentales de CORBA. En esta sección complementaré la información acerca de este componente. Un ORB es un componente software cuyo propósito es facilitar la comunicación entre objetos. Esto lo hace proporcionando una serie de facilidades, una de las cuales es localizar un objeto remoto dada una referencia a ese objeto y otra es la ordenación de los parámetros y valores de retorno hacia y desede invocaciones a métodos remotos.

Con la la utilización de un ORB, un cliente puede invocar transparentemente un método de un objeto servidor que se encuentre en la misma máquina o en otra distinta. El ORB intercepta la llamada realizada por el objeto que implementa la petición, pasa los parámetros, invoca el método y regresa los resultados. El cliente no necesita saber dónde se localiza el objeto que ejecutará el método, el lenguaje en el que está programado, el sistema operativo, ni cualquier otro aspecto que no sea su interfaz. Cabe resaltar que las definiciones de cliente y servidor que se dan a los objetos son simplemente para coordinar las interacciones entre ambos; estas definiciones pueden cambiar ya que un objeto puede ser cliente o servidor dependiendo de la ocasión, tal y como se mencionó en el capítulo II.

En la figura 5.4 se muestra la estructura de un sistema distribuido basado en CORBA. En él se pueden observar el cliente y el servidor. En la parte cliente existe un programa cliente propiamente dicho al que CORBA le añade cierta infraestructura que permite la comunicación con el servidor a través de la red. La parte servidora está formada por el objeto que exporta su funcionalidad, integrado en el servidor (proceso en el que se ejecuta) y diversos elementos que permiten que las invocaciones realizadas por el cliente a los métodos del objeto lleguen a éste, sean procesadas y sus resultados devueltos.

TESIS CON
FALLA DE ORIGEN

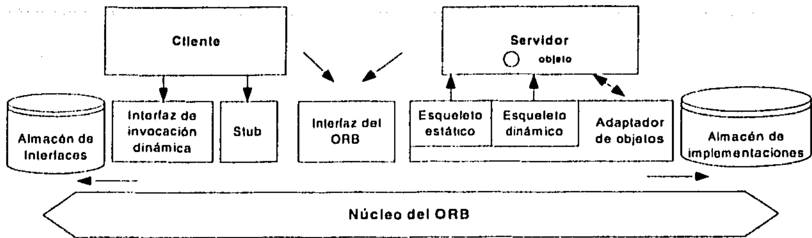


Fig. 5.4 Estructura de un sistema distribuido basado en CORBA

El funcionamiento del ORB es el siguiente: cuando un módulo de una aplicación quiere usar un servicio proporcionado por otro módulo obtiene una referencia del objeto que provee ese servicio. Después de obtenerla, el módulo puede invocar métodos en ese objeto. La primera responsabilidad del ORB es resolver peticiones de referencias de objetos, permitiendo a los módulos de la aplicación establecer conexión entre ellos.

Otra de las tareas del ORB es el marshalling y el unmarshalling. Después de que un módulo de la aplicación obtiene una referencia del objeto, cuyos servicios quiere usar, ese módulo ya puede invocar métodos en ese objeto. Por lo general estos métodos necesitan parámetros de entrada y devuelven otros parámetros de salida. El ORB recibe los parámetros de entrada del módulo que llama al método y los une (marshal). Es decir, el ORB traduce los parámetros a un formato (on-the-wire format) que puede ser transmitido por la red hasta el objeto remoto. El ORB también hace el proceso contrario de unir (unmarshals) los parámetros devueltos, convirtiéndolos a un formato que el módulo que lo solicitó entienda.

Todo el proceso anteriormente citado se hace de forma transparente al programador. La aplicación cliente invoca un método remoto y recibe los resultados como si el método fuese local. Es gracias a este proceso que se consigue la independencia de plataforma, debido a que los parámetros se traducen durante la transmisión a un formato totalmente independiente de la plataforma (el on-the-wire format es parte de las especificaciones CORBA) y en el momento de la recepción se convierten al formato específico de la plataforma. Por ejemplo, un cliente que se ejecuta en un sistema basado en Microsoft

TESIS CON
FALLA DE ORIGEN

podrá invocar métodos de un servidor que se ejecuta en un sistema UNIX. También las diferencias de hardware entre equipos (como el ordenamiento de los bytes, la longitud de las palabras, etc.) son irrelevantes puesto que el ORB hace las conversiones necesarias automáticamente.

En resumen, las responsabilidades del ORB son:

1. Dada una referencia a un objeto por un cliente, el ORB localiza la correspondiente implementación del objeto (el servidor).
2. Cuando el servidor está localizado, el ORB asegura que el servidor esté preparado para recibir la petición.
3. El ORB del lado del cliente acepta los parámetros del método que se está invocando y marshal los parámetros a la red.
4. El ORB del lado del servidor unmarshal los parámetros de la red y se los entrega al servidor.
5. Los parámetros de retorno, si existen, realizan el marshal/unmarshal del mismo modo.

La mayor ventaja que ofrece el ORB es el tratamiento de los datos independientemente de la plataforma.

2. Lenguaje de Definición de Interfaces (IDL)

Todo proceso de diseño de software orientado a objetos comienza definiendo en forma clara las interfaces de comunicación de los objetos de la aplicación. Estas se describen en el mundo de CORBA utilizando el leguaje IDL. Se trata de un lenguaje de *especificación* y no puede sustituir el papel de lenguajes de programación como pueden ser C++ y Java, ya que no es capaz de implementar las interfaces que especifica. CORBA utiliza un lenguaje de especificación, como el CORBA-IDL, para ocultar la implementación del objeto a los posibles clientes. Se sitúa la especificación IDL entre el cliente y la implementación de manera que éste nunca llegue a interactuar directamente sobre la misma. Con ello se consigue la independencia del lenguaje de implementación, lo que permitirá al programador elegir el que más se adapte a sus conocimientos y necesidades.

El conjunto de condiciones IDL es el responsable de asegurar que los datos sean intercambiados correctamente entre lenguajes diferentes. Por ejemplo, el tipo long en IDL es un entero de 32 bits con signo, que corresponde a un long de C++ o a un int de Java.

La independencia de lenguaje se consigue gracias al lenguaje de mapeo, que es un conjunto de condiciones que iguala las construcciones hechas en IDL con las de algún lenguaje de programación particular. Por ejemplo en el mapeo de C++ la interface de IDL corresponde a una clase de C++. La OMG ha definido un número de languaje de mapeo estándar para muchos lenguajes, tales como C, C++, COBOL, Java, Smalltalk, etc.

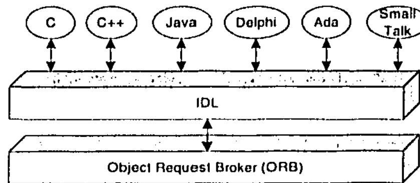


Fig. 5.5 **Interacción entre IDL y los lenguajes**

Otro aspecto a resaltar de IDL es que no es un lenguaje de implementación (no se pueden escribir aplicaciones en IDL), más bien es un lenguaje de definición de interfaces. Haciendo una comparación con C++, se podría decir que las definiciones IDL son semejantes a los ficheros de cabecera ("*.h" o "*.hh" en C++) para las clases (las cuales no contienen la implementación de la clase, sino más bien la definición de su interfaz).

CORBA al igual que COM, es independiente de lenguaje, por lo que deja a los diseñadores de la aplicación la elección del lenguaje que más se ajuste a sus necesidades. Se pueden utilizar varios lenguajes para implementar cada parte de la aplicación. Por ejemplo, el cliente puede estar escrito en Java que asegura que pueda funcionar en cualquier máquina, mientras que el servidor se puede implementar en C++ para obtener buenas prestaciones.

TESIS CON
FALLA DE ORIGEN

3. El modelo de comunicaciones CORBA

En esta sección será necesario entender el papel de CORBA en una red de computadoras. Típicamente una red consiste en sistemas que están físicamente conectados. Como se definión en el capítulo III, esta capa física proporciona el medio a través del cual tiene lugar la comunicación. Más allá de la capa física se encuentra la capa de transporte, que incluye los protocolos responsables de mover los paquetes de datos desde un punto hacia otro. CORBA es neutral respecto estos protocolos de red, es independiente del protocolo utilizado y podría funcionar con cualquiera.

El estándar CORBA especifica lo que se conoce como el Protocolo General Inter-ORB (GIOP) que, en un nivel alto, establece un estándar para la comunicación entre varios ORBs de CORBA. GIOP es tan sólo un protocolo general, por lo que el estándar CORBA también especifica otros protocolos que detallan GIOP para usar un protocolo de transporte en particular (como TCP/IP o DCE). El utilizado para redes TCP/IP se denomina Internet Inter-ORB Protocol (IIOP). Los fabricantes deben implementar este protocolo, esto es con la finalidad de asegurar la interoperabilidad entre productos CORBA de diferentes fabricantes.

Los ORBs de CORBA normalmente se comunican usando el IIOP debido, en parte, a que este protocolo es el correspondiente al protocolo TCP/IP, que es el usado en Internet, aunque un ORB puede soportar cualquier otro protocolo. En el dialecto CORBA/IIOP las referencias a objeto se pasan a denominar Referencias a Objetos Interoperables o IORs.

La especificación del protocolo GIOP consiste en los siguientes elementos:

- El CDR (representación común de datos). Define cómo deben codificarse los datos para su transferencia entre clientes y servidores.
- El formato de los mensajes GIOP. Define los mensajes que se intercambian entre el ORB del cliente y el del servidor cuando se realizan invocaciones sobre objetos, se localizan implementaciones y se controlan los canales de comunicación.
- Condiciones de la capa de transporte. La especificación GIOP describe condiciones que la capa de transporte debe cumplir para poder transmitir mensajes GIOP. La especificación también describe cómo se han de gestionar las conexiones y las restricciones en el orden de los mensajes GIOP.

La especificación IIOP añade los siguientes elementos a la especificación GIOP:

➲ El transporte de mensajes IOP en Internet. La especificación IIOP (Internet IOP) describe cómo los agentes abren conexiones TCP/IP y las usan para transferir mensajes GIOP.

➲ El IIOP no es una especificación separada, sino que es una especialización o mapeado del GIOP al protocolo especifico TCP/IP. La especificación GIOP se considera una base para futuras implementaciones a otros protocolos de transporte.

Resumiendo, las aplicaciones CORBA se construyen encima de los protocolos derivados de GIOP (como el IIOP). Estos protocolos, a su vez, están encima de los protocolos de transporte (TCP/IP, DCE). Las aplicaciones CORBA no están limitadas a usar sólo uno de estos protocolos, sino que se puede usar un puente para interconectar aplicaciones situadas en redes que trabajen con diferentes protocolos. La arquitectura CORBA, más que reemplazar los protocolos de transporte, crea otra capa (la capa del protocolo Inter-ORB) que utiliza estos protocolos como soporte. CORBA no se limita al uso de un protocolo de transporte en particular por lo que lo hace interoperable con culaquier otro protocolo.

4. El modelo de objetos CORBA

En CORBA, todas las comunicaciones entre objetos se hacen a través de referencias de objeto. La visibilidad de los objetos se obtiene únicamente pasando referencias a esos objetos, los objetos no pueden ser pasados por valor. En otras palabras, los objetos remotos en CORBA permanecen remotos, no hay actualmente ninguna forma de mover o copiar un objeto de un sitio a otro.

Las arquitecturas orientadas a objetos ofrecen un modelo de objetos, que describe cómo se representan los objetos en el sistema. CORBA se diferencía de los modelos tracicionales en tres aspectos fundamentalmente: en la forma semitransparente de distribuir los objetos en CORBA, el tratamiento de las referencias a objetos y el uso de los llamados adaptadores de objetos (como el BOA -Basic Object Adapter).

Para un cliente CORBA es lo mismo una llamada a un método remoto que una llamada a un método local, esto se debe a la transparencia que ofrece la distribución de los objetos en CORBA. El hecho que los objetos se encuentren distribuidos permite que aumenten las posibilidades de error, CORBA resuelve esta situación ofreciendo un conjunto de excepciones de sistema, que pueden ser lanzadas por cualquier método remoto.

En una aplicación distribuida, existen dos métodos para obtener el acceso a un objeto desde otro proceso. El primero se conoce como "*paso por referencia*" y puede ser explicado mediante un ejemplo. Supongase dos procesos, A y B. El primer proceso (A) pasa una referencia de un objeto al segundo proceso (B). Cuando el proceso B invoca un método dentro de ese objeto, el método ejecuta el proceso A puesto que él posee el objeto. El proceso B sólo tiene visibilidad del objeto (a través de la referencia al objeto) y por lo tanto sólo puede pedirle al proceso A que ejecute los métodos en su lugar. En otras palabras, pasar un objeto por referencia significa que un proceso concede visibilidad de uno de sus objetos a otro proceso mientras retiene la propiedad del objeto.

El segundo método de pasar un objeto se denomina "*paso por valor*". El estado actual del objeto es pasado al módulo que lo pide donde se crea una nueva copia del objeto. Cuando el proceso B invoque métodos en el objeto se ejecutarán en dicho proceso (en la copia) en vez de en el proceso A, donde el objeto original reside. Por lo tanto el estado del objeto original no cambia.

Como se mencionó con anterioridad, en el modelo de objetos de CORBA todos los objetos se pasan por referencia. Si se quiere pasar un objeto por valor es necesario asegurarse de que el módulo que recibe el objeto tiene implementaciones para los métodos soportados por dicho objeto. Si los objetos son pasados por referencia, el paso anterior no es necesario.

El problema de pasar los objetos por referencia es que todas las llamadas a los métodos deberán ser remotas, por lo que si un componente invoca repetidas veces métodos en un mismo objeto remoto, se producirá una gran cantidad de sobrecarga en la comunicación entre los dos componentes. Es más eficiente pasar el objeto por valor debido a que el componente puede manipular el objeto localmente.

El estándar CORBA define un par de adaptadores de objetos (object adapters) cuyo primer propósito es conectar la implementación de un objeto con su ORB, consiguiendo que las peticiones que llegan a través del ORB sean procesadas por la implementación del Objeto. Entre sus responsabilidades se encuentran:

- *Registro de objetos.* Los Adaptadores de Objetos deberán proporcionar funciones para registrar implementaciones para los objetos CORBA.
- *Generación de referencias a objetos.* Los Adaptadores de Objetos deberán generar referencias para los objetos que tengan registrados.
- *Activación de procesos servidores.* Los Activadores de Objetos deberán activar los procesos (servidores) donde los objetos puedan ser activados.
- *Activación de objetos.* Los Activadores de Objetos deberán activar los objetos registrados.
- *Multiplexión de peticiones a los objetos registrados.* Los Adaptadores de Objetos deberán asegurar que todas la peticiones sean recibidas por los objetos, aunque tengan múltiples conexiones, sin que ninguna se bloquee indefinidamente.
- *Gestión de las invocaciones.* Los Adaptadores de Objetos deben despachar las peticiones de objetos registrados.

En la norma CORBA han aparecido 2 adaptadores de objetos: el actual adaptador POA (Portable Object Adaptor), y el original y ya desaparecido (en la versión CORBA2.2) BOA (Basic Object Adaptor).

5. Enlaces entre la interfaz y el lenguaje

Una vez creadas las definiciones de la interfaz de un módulo usando IDL (esta definición es guardada en archivos), se ejecutan los archivos IDL resultantes a través de un compilador IDL, generando así lo que se conoce como *client stubs* y *server skeletons.* Éstos sirven como una especie de "pegamento" que conecta las especificaciones de la interfaz IDL independientes del lenguaje con la implementación en un código de un lenguaje específico.

Un *stub cliente* es una parte de código precompilado que define la forma en como los clientes invocaran los servicios correspondientes en los servidores. Los stubs cliente

proporcionan una capa intermedia entre el cliente y el núcleo del ORB. Definen cómo los clientes invocan los servicios que proporcionan los objetos servidores. Desde la perspectiva del cliente, el stub actúa como una especie de proxy, dando la impresión de que la invocación se está realizando sobre un objeto local como en cualquier aplicación orientada a objetos. El stub incluye código para ejecutar el marshaling, lo que significa que codifica y decodifica la operación y sus parámetros en forma de mensajes para poder ser enviados al servidor.

Un *skeleton servidor* es una parte de código precompilado que proporciona interfaces estáticas para cada servicio exportado por el servidor. Para cada método de una interfaz el compilador IDL genera un método vacío en el skeleton servidor. El programador debe entonces proporcionar una implementación para cada uno de estos métodos.

6. Servicios CORBA y facilidades CORBA.

Existen capacidades tales como el manejo de eventos, licenciamiento, seguridad, nombramiento, manejo de la interfaz del usuario, intercambio de datos entre otras que han sido estandarizadas a manera de interfaces por la OMG con la finalidad de que las aplicaciones puedan usar estos servicios tal como usan cualquier otro objeto CORBA. A estas capacidades se les conoce como servicios y facilidades CORBA.

Servicios

Los servcios CORBA son conjuntos de objetos preimplementados que complementan la funcionalidad de otros objetos. Su propósito es general, lo que significa que son independientes del dominio de aplicación. Pueden ser usados por los programadores para crear componentes, nombrarlos e introducirlos al ambiente.

La norma CORBA especifica los siguientes servicios:

Consulta	Externalización	Colección	Concurrencia
Relación	Seguridad	Eventos	Licencia
Nombrado	Propiedades	Persistencia	Intercambio
Ciclo de vida	Transacciones	Tiempo	

a) Servicio de Nombre

Provee la capacidad a los objetos CORBA de encontrar otros objetos CORBA usando una convención de nombre fácilmente distinguible. También permite asociar un nombre lógico con un objeto en tiempo real, lo que provee aun más flexibilidad en cómo se configura el sistema de objetos distribuidos.

b) Servicio de Eventos

Provee un canal de comunicaciones decodificado entre los objetos de CORBA. Los objetos en un sistema de software envían y reciben eventos, los cuales son notificaciones de un cambio en alguna parte del sistema. En programas CORBA, un servidor dado puede estar interesado en eventos generados por otro servidor, en alguna otra parte del sistema. El servicio define un objeto bien conocido llamado *canal del evento* (event channel) que colecciona y distribuye eventos entre componentes que no saben nada uno del otro.

c) Servicio de Objetos Persistentes.

El Servicio de Objetos Persistentes (POS). La persistencia de objetos se refiere a la capacidad del objeto de mantener su estado fuera del rango del cliente u objeto que originalmente lo creó. Este servicio provee la capacidad para un objeto de ser guardado en algún tipo de almacén de datos y de ser llamado en un tiempo posterior, cuando se le necesite.

d) Servicio de Transacciones

CORBA permite construir sistemas distribuidos a gran escala para el control de transacciones y provee de un grupo de transacciones definidas en la especificación del servicio. Hay dos tipos de transacciones soportadas por este servicio: Las transacciones planas, que son las transacciones básicas, las que no son contenidas por otras transacciones por lo que son llamadas transacciones de alto nivel. Las otras son las transacciones anidadas, que son subtransacciones embebidas dentro de una mayor (una transacción padre), lo que recuerda una topología de árbol jerárquico.

e) Servicio de Control de Concurrencia

En un sistema complejo, es posible que múltiples clientes necesiten acceso concurrente a un objeto en particular. Si un cliente está cambiando un objeto, mientras otro está tratando

de leerlo, o si dos clientes tratan de cambiar un objeto en el mismo instante, el sistema rápidamente cae en críticos problemas de concurrencia. Este problema es resuelto por el Servicio de Control de Concurrencia, que se basa en el concepto de clientes y su interacción con recursos compartidos.

f) Servicio de Relaciones

El Servicio de Relaciones provee la capacidad de relacionar un objeto con otro, o un objeto con múltiples objetos. También provee una variedad de maneras descriptivas de definir una relación. Este servicio puede ser usado para implementar las restricciones de integridad referencial o cualquier otro tipo de vínculo entre componentes.

g) Servicio de Ciclo de Vida

El Servicio de Ciclo de Vida define la convención usada cuando se crea, copia, borra y mueve objetos en el sistema CORBA. Este servicio se preocupa de asuntos relacionados con el estado de los objetos y gráficos de objetos, pero descansa en el Servicio de Relaciones para proveer los medios para mantener las relaciones en el grafo.

h) Servicio de Externalización

El Servicio de Externalización describe la convención para guardar y restaurar el estado de un objeto. Hay dos conceptos principales que están relacionados con este servicio: Externalización e internalización. La Externalización se refiere a la capacidad de guardar un estado de un objeto en un stream de datos. La internalización se refiere a la capacidad de recuperar los datos de un objeto de un stream de datos.

i) Servicio de Consulta

El Servicio de Consulta fue diseñado para proveer operaciones de consultas en colecciones de objetos. La arquitectura de este Servicio permite el uso de múltiples consultas anidadas, las cuales pueden ser útiles en sistemas más complejos.

j) Servicio de Licenciamiento

El Servicio de Licenciamiento fue creado para resolver muchos de los problemas que surgen del licenciamiento de los componentes de software de CORBA. La creación del Servicio de Licenciamiento fue hecho con la visión de proveer capacidades de

licenciamiento que puedan ajustarse a una amplia variedad de requerimientos de licenciamiento.

k) Servicio de Propiedades

Provee operaciones que permiten agregar atributos nombrados (propiedades) dinámicamente a componentes encapsulados existentes sin introducir cambios al componente. Por ejemplo, un título o una fecha.

l) Servicio de Tiempo

El Servicio de Tiempo fue creado para proveer soporte a aplicaciones CORBA distribuidas que necesiten información actualizada de tiempo con una indicación del nivel asociado de error. El concepto de tiempo puede causar confusión en sistemas distribuidos debido a que pueden estar corriendo en computadoras con diferentes fuentes de tiempo, e incluso posiblemente en computadores con diferentes zonas horarias. Además de proveer una base de tiempo común, este servicio provee la capacidad de registrar el tiempo en el cual los eventos ocurrieron, la capacidad de generar eventos ordenados en tiempo, basados en timers y alarmas, y finalmente la capacidad de calcular el intervalo entre dos eventos.

m) Servicio de seguridad

El Servicio de Seguridad fue diseñado para cubrir todos los hoyos de seguridad que puedan ser encontrados por intrusos maliciosos o por usuarios inexpertos que usen los sistemas de manera incorrecta.

n) Servicio de Colección

Una colección es un agrupamiento de objetos que soporta operaciones de navegación y manipulación de objetos en un grupo. El Servicio de Colección define un amplio rango de colecciones que se pueden usar. Estas colecciones saben como pasar a través de la conexión CORBA y pueden ser usadas por el cliente y el servidor en diferentes plataformas escritas en diferentes lenguajes.

o) Servicio de Intercambio

Permite localizar objetos basado en los servicios que estos objetos ejecutan. El Servicio de Intercambio funciona como los avisos clasificados de algún periódico. Los objetos registran (avisan) un servicio que ellos proveen, con el Servicio de Intercambio. Este

proceso de aviso se llama exportación porque un objeto exporta algunas capacidades que tiene y le avisan al Servicio de Intercambio de la ubicación de la interfaz que provee esa capacidad. Clientes que busquen una capacidad especifica pueden descubrir o importar este servicio con la ayuda del Servicio de Intercambio.

Facilidades

Las facilidades CORBA son servicios de alto nivel, a manera de funciones de aplicación genéricas. Se conocieron inicialmente como Facilidades Comunes Horizontales, para diferenciarlas de las Facilidades Comunes Verticales (llamadas actualmente Interfaces de Dominio) que son específicas para determinados ámbitos de aplicación; hoy en día se demoninan *CORBAfacilities.*

Mientras que los *servicios* están orientados a la manipulación de los objetos distribuidos, las *facilidades Comunes* están orientadas al usuario final, brindándole servicios tales como impresión de documentos, correo eletrónico, etc.

D. Orbix y Visibroker

Basándose en el estándar de CORBA, algunos vendedores de software han creado sus propias ORBs. En este momento las principales ORBs dentro del mercado son Orbix, Visibroker y EXPERSOFT. Se dice que los ganadores en el mercado de las ORBs serán IONA, y Visigenic. ORBs creadas con IONA y Visigenic son Orbix y VisiBroker. En la actualidad Visigenic es parte de Borland.

a) Orbix

Orbix es el middleware líder del mercado para la construcción de aplicaciones distribuidas. Orbix está basado en el estándar CORBA.

Las dos claves principales de la potencia del estándar de CORBA son el compilador IDL e IIOP. El compilador IDL permite la compilación de interfaces definidos en el lenguaje estándar IDL, que es un lenguaje de definición similar a la sintaxis de C++, realizando el mapeo a un lenguaje determinado (en el caso de Orbix, es C++).

El stub compilado puede enlazarse después con la aplicación. Orbix RunTime que proporciona las capacidades necesarias para tener aplicaciones distribuidas.

IIOP proporciona un protocolo a "nivel de conexión" que permite a un "componente" comunicar con otro sobre TCP/IP.

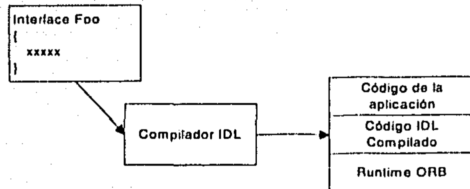


Fig. 5.6 Compilador IDL

Las especificaciones de la OMG definen IIOP como un protocolo para el uso de una red TCP/IP. Como se puede ver en el diagrama, "otro protocolo" puede emplearse con el ORB (como ejemplo, OrbixTalk utiliza el protocolo UDP).

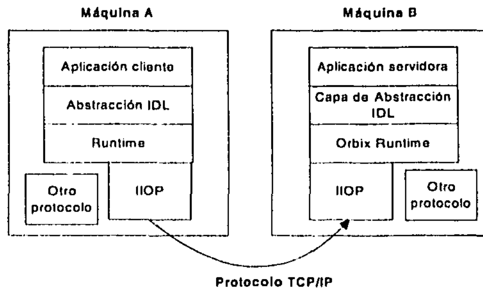


Fig. 5.7 Implemetación Orbix

TESIS CON
FALLA DE ORIGEN

Entre las características más importantes que ofrece Orbix de IONA son:

- *Integración de COM-CORBA con OrbixCOMet.* Combina características de CORBA y DCOM, ofreciendo a los desarrolladores capacidad para construir sistemas utilizando componentes COM y CORBA. Este enlace dinámico bidireccional de tan alto rendimiento, permite que las aplicaciones DCOM escritas con herramientas como Visual Basic, PowerBuilder, Delphi, MS Office o Active Server Pages, puedan acceder a las aplicaciones CORBA ejecutando en Windows, Unix, y MVS. IONA Technologies y Microsoft han llegado a un acuerdo mediante el cual, Microsoft se compromete a proporcionar a IONA el código fuente para el producto DCOM Microsoft.
- *Transparencia de Localización y Balance de Carga con OrbixNames.* OrbixNames es la implementación del Servicio de Nombres CORBA, que IONA ha llevado a cabo. El propósito de este servicio es actuar como repositorio de nombre de objetos que los clientes emplean para localizar las aplicaciones del servidor. Un servidor CORBA que contiene una referencia a un objeto puede registrar el mismo con OrbixNames, dándole un nombre fijo que posteriormente pueda utilizar cualquier cliente para encontrar el objeto. Si el servidor cambia su ubicación, se puede asociar su nueva referencia al objeto empleando el mismo nombre fijo, proporcionando así al cliente una total transparencia de localización. OrbixNames extiende el modelo de nombres de CORBA proporcionando balance de carga entre servidores, a través de replicación.
- *Desarrollo visual con OrbixCGT.* El paquete de herramientas para la generación de código que Orbix ofrece, es un potente elemento añadido para los programadores de CORBA. Los scripts ejecutables, llamados Genies, ayudan al desarrollo de Orbix y a las aplicaciones OrbixWeb. Esta rápida herramienta de desarrollo de aplicaciones (RAD) reduce considerablemente el número de horas empleadas en tiempo de desarrollo.
- *Seguridad en Internet Incorporada con Orbix Wonderwall.* Proporcionan control de acceso de los objetos finales, además de transparencia del lado del cliente.

b) VisiBroker

Visibroker es un ORB que cumple con la norma CORBA, dando soporte al desarrollo, distribución y administración de aplicaciones de objetos distribuidos en una variedad de plataformas y sistemas operativos. Provee los servicios de Nombrado y Eventos (Naming

Service y Event Service) del estándar de CORBA, el nombrado permite la asociación de uno o varios nombres lógicos a la implementación de un objeto específico, permitiendo su localización a través de este, el servicio de eventos provee y facilita la comunicación entre los objetos.

VisiBroker Gatekeeper. Este producto es ejecutado en un web server y habilita a los programas clientes localizar y utilizar objetos que no residen en el web server, pasando a través del firewall y accediendo a los servidores de aplicación. El Gatekeeper puede ser usado como un demonio de HTTP para evitar la necesidad de un server adicional durante la etapa de desarrollo.

VisiBroker SSL-Pack. Este producto provee los sistemas de una forma de comunicación segura, soportada sobre el protocolo SSL (Secure Sockets Layer) desarrollado por Nestcape Comunication Corporation, el cual permite la transmisión de datos sensibles a través de una red insegura como Internet, agregando a nuestras transmisiones las caracteristicas de Autenticidad, Privacidad e Integridad.

Visibroker se compone de lo siguiente:

1) Librerías de CORBA.
Las librerías de CORBA le ayudan a los programas a exponer métodos CORBA y a utilizar los métodos desde los programas Clientes. Las librerías de CORBA pueden ser de dos tipos:

- *ORB.* Como se mencionó anteriormente, el ORB es el exportador e importador de interfaces. Todos los clientes y servidores deben inicializar el ORB y utilizarlo para obtener interfaces. Básicamente el ORB es el que interpreta los punteros de interfaces y los traduce a mensajes de red, o recibe llamadas de red y los traduce a punteros locales para su ejecución.
- *Adaptador de Objetos Básico (BOA).* El adaptador de Objetos Básico (Basic Object Adaptor) es la librería que le ayuda a exportar el puntero de interfaz. Se utiliza en el servidor.

2) Servicio de Nombres "SmartAgent"

El servicio SmartAgent ayuda a exportar un objeto para su utilización en una red local.

3) Compilador de IDL

IDL es un lenguaje "neutral" para definición de interfaces. Como tal, tiene declaraciones pero no tiene sintaxis de control (for, do while, etc). Bajo CORBA, el desarrollador describe su interfaz utilizando IDL y utiliza el compilador de IDL para crear clases de apoyo a su implementación, y clases auxiliares para que los clientes puedan crear instancias del objeto.

E. Las capas del modelo CORBA

Una vez detallados cada uno de los conceptos involucrados en el modelo CORBA, se puede especificar su coolaboración en cada una de las divisiones del modelo multicapas. Esta sección del capítulo cinco se complementa con al Anexo I, el cual describe con un ejemplo las diferencias entre CORBA y DCOM.

1. Capa Superior. Arquitectura Básica de Programación

La conexión entre cliente y servidor es transparente, ofreciendo la imagen de estar ejecutándose en el mismo espacio de memoria en la misma máquina. En CORBA, un objeto puede activarse simplemente invocando a cualquiera de sus métodos. Algunos fabricantes proveen métodos especiales, como _bind, tal es el caso de Orbix, para activar un objeto servidor y obtener su referencia. Véase Anexo I tabla 5.

Activación del objeto

1. El cliente llama al método _bind para activar el objeto servidor. Ejemplo:
```
gridVar = grid::_bind(":grid").
```
2. ORB arranca un servidor que contiene un objeto que soporta la interfaz de la clase solicitada. En el ejemplo anterior llamada *grid.*
3. En cada constructor, se realizan llamadas para crear y registrar referencias para los objetos.

4. ORB devuelve la referencia del objeto al cliente. En el ejemplo anterior a través de la variable *gridVar*.

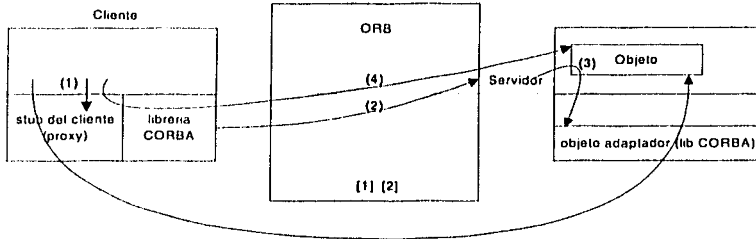


Fig. 5.7 **Capa superior CORBA**

Invocación del método

1. El cliente llama la referencia del objeto la cual invocará a la función correspondiete en el lado del servidor.

```
//En el cliente se invoca con la variable gridVar el método get()
gridVar->get()

//En el servidor se ejecuta el método get() de la clase grid a través de la invocación
del cliente
grid_i::get()
```

2. Del mismo modo con reset().

CORBA ofrece soporte para tratar excepciones estándar de C++ y algunas otras, además de permitir la declaración de excepciones definidas por el usuario.

2. Capa Intermedia. Arquitectura de Proceso Remoto

Esta capa contiene la infraestructura necesaria para ofrecer al cliente y al servidor la ilusión de encontrarse en el mismo espacio de direcciones.

Es en esta capa es donde el proceso de marshaling cobra importancia y es utilizado para enviar datos a través de distintos espacios de direcciones. Como se mencionó anteriormente, éste proceso empaqueta los parámetros de la llamada y devuelve un valor en el servidor en un formato de transmisión estándar, como una RPC normal. La

operación contraria se conoce como *unmarshaling*, y convierte los datos entrantes del formato estándar al propio del espacio de memoria destino.

Aquí aparece el **OA** (*Object Adaptor*). Se sitúa sobre el ORB, y se encarga de realizar la conexión con la implementación del objeto. Este mecanismo proporciona servicios como generación e interpretación de las referencias de objetos, invocación de métodos, activación y desactivación de objetos. El Adaptador Básico de Objeto (BOA) define un adaptador que puede emplearse para la mayoría de las implementaciones de objetos convencionales. Sin embargo, con el reemplazo de BOA, el Adaptador Portátil de Objeto (POA) se elimina la dependencia de BOA con respecto al fabricante, ya que en las especificaciones originales de CORBA no se hablaba de la implementación del enlace ORB/BOA, y existen tantos esquemas BOA distintos como productos CORBA comerciales.

3. Capa Inferior: Arquitectura del Protocolo de Comunicación

Esta capa especifica el protocolo para soportar la comunicación entre clientes y servidores. CORBA no especifica un protocolo determinado para la comunicación entre un cliente y un servidor ejecutándose en ORBs del mismo fabricante. El protocolo, se deja a la elección del fabricante. Sin embargo, para soportar la interoperabilidad de diferentes ORBs, sí se especifica un Protocolo General Inter-ORB (GIOP). Existe un GIOP basado en TCP/IP, denominado *Internet Inter-ORB Protocol* (IIOP).

CONCLUSIONES

Es fácil percatarse que el futuro de las aplicaciones informáticas se encuentra relacionado con los conceptos de Internet e Intranet. Actualmente el Internet, las intranets, los objetos distribuidos y los componentes están impulsando a la arquitectura **Multicapas** como la corriente principal de la arquitectura Cliente/Servidor. Esta tecnología permite empezar en pequeñas dimensiones (tanto en escala como en funcionalidad) y luego crecer las aplicaciones a grandes proporciones.

La arquitectura multicapas puede ser desarrollada con diferentes tecnologías, ya sean monitores de transacciones, sockets, RPCs, objetos distribuidos, etc. Los objetos distribuidos son una excelente opción, porque proporcionan soluciones escalables y flexibles para ambientes Cliente/Servidor de gran escala y para el internet e intranets. Los objetos comerciales pueden ser descompuestos y divididos en múltiples capas para darle a una aplicación lo que necesita adaptándose perfectamente al modelo multicapas.

A lo largo de este trabajo se han definido los dos modelos estándares más importantes que existen en el mercado actual de objetos distribuidos: COM+ y CORBA. Se han comentado las características fundamentales de cada uno de ellos (lenguaje de programación, sistema operativo, códigos IDL, declaración de objetos, creación de interfaces, etc.), se han comparado las dos soluciones y presentado un programa ejemplo (Anexo I) con el fin de comprender mejor la dinámica de ejecución de cada uno de los dos modelos.

A pesar que el análisis y comparación de ambos estándares conduce a preguntarse *qué* modelo es el mejor, esta respuesta siempre dependerá de las necesidades y recursos de la empresa que desea implementar o migrar a una solución multicapas.

Además, para poder hacer una comparación entre ambas tecnologías no basta en citar las características de DCOM y de CORBA, porque en el resultado de esta comparación CORBA tiene mayores ventajas. COM y DCOM entran en verdadera competencia con CORBA y sus ORBS con la combinación de MTS. El conjunto de estas herramientas han dado origen a lo que hoy Microsoft llama COM+.

La comparación de ambas tecnologías se puede resumir de la siguiente manera:

- Ambas tecnologías son independientes de lenguaje, pero CORBA se está centrando cada vez más en la interoperabilidad con Java, lo que lo hace no ser del todo independiente de lenguaje.
- Solamente CORBA es independiente de plataforma.
- MTS como ambiente de ejecución de COM proporciona seguridad, procesamiento de transacciones y escalabilidad, CORBA proporciona estas mismas características por especificación, no necesariamente por implementación.
- CORBA es administrado a través de programación sobre un API, COM lo hace administrando el ambiente.
- CORBA es una solución multivendedor, pero dadas sus incompatibilidades entre las diferentes implementaciones y del soporte de plataformas cruzadas de cada una de las mismas, se tiene que trabajar con un producto CORBA de un solo vendedor y terminar con un solo vendedor propietario del estándar, tal y como es el caso de COM.
- Para integrar ambientes múltiples la única opción es utilizar CORBA.
- Para ambientes basados en Windows la mejor solución es COM.
- CORBA proporciona un excelente mecanismo para unir MS Desktops y Servidores Unix.

Uno de los aspectos fundamentales que ayuda a decidir por la tecnología a utilizar en una aplicación multicapas, es la ubicación de la capa intermedia. Si la capa intermedia no va a estar situada en una computadora con NT o Windows 2000, COM+ no es la tecnología a elegir. DCOM y COM corren en otras plataformas además de NT, pero MTS no corre en ninguna otra que no sea basada en NT. En cambio CORBA permite que trabajen de forma "transparente" un gran número de máquinas de diferentes plataformas en la capa intermedia.

Como se ha podido apreciar, CORBA y COM+ no sólo representan una solución para el manejo transparente de objetos dinámicos en sistemas distribuidos como se pensó inicialmente, sino que se podrían considerar además como un completo paradigma de desarrollo de aplicaciones, ya que permiten definir fácilmente las interfaces de los objetos que componen la aplicación e implementarlas individualmente en distintos lenguajes, protocolos y arquitecturas, lo cual permite a los desarrolladores reutilizar componentes de Software y Hardware ya existentes.

Los desarrolladores hoy en día deben diseñar sistemas que cuenten con requerimientos en permanente transición, teniendo así que enfrentar la integración de diversas aplicaciones en redes heterogéneas. Con las capacidades y flexibilidad de COM y CORBA apoyando la unificación de la infraestructura, los desarrolladores se podrán enfocar a proveer soluciones sólidas para problemas de alto nivel, y tendrán que preocuparse menos acerca de cómo hacer que los problemas simples funcionen en los sistemas distribuidos e inevitablemente heterogéneos.

Finalizo mis conclusiones argumentando que debido a que los modelos estándar de objetos distribuidos aplicados a la arquitectura multicapas han estado jugando un papel importante durante los últimos años en el modo de diseñar sistemas, considero de gran importancia que los estudiantes de la carrera de Matemáticas Aplicadas y Computación se vean involucrados en el desarrollo de proyectos que utilicen objetos distribuidos, sobre todo en aquellas materias que involucren algún lenguaje Orientado a Objetos, esto no solo con la finalidad de saber como diseñar este tipo de aplicaciones, sino porque en la actualidad muchas de las empresas se encuentran migrando sus sistemas a esta tecnología y será necesario dar soporte, escalar y mantener dichos sistemas.

APÉNDICE

API (Application Program Interface). Conjunto de rutinas, protocolos y herramientas para la construcción de aplicaciones de software. En la mayoría de los ambientes operativos, como Microsoft Windows proporciona una API para que los programadores escriban aplicaciones consistentes al ambiente operativo.

Aplicaciones departamentales Aplicaciones desarrolladas en su mayoría por herramientas visuales que se caracterizan por ser útiles a ciertas áreas de una empresa determinada, tales como prototipos y groupwares a pequeña escala.

Back-end Programa de respaldo (programa de soporte en el sistema de computación, como banco de datos). Programa que efectúa las acciones de fondo.

Blob Binary Large Object. Se trata de una colección de datos binarios almacenados en una entidad dentro de un sistema manejador de base de datos (DBMS). Los Blobs son usados principalmente para almacenar objetos multimedia tales como imágenes, videos y sonido, también pueden ser usados para almacenar programas e inclusive fragmentos de código.

Capa intermedia Una colección de máquinas utilizadas para ejecutar las reglas del negocio. Cuando las reglas del negocio están empaquetadas en componentes, por lo regular es referido como capa componente.

COM (Component Object Model). Modelo de código binario desarrollado por Microsoft. OLE y ActiveX están basados en *COM*.

Cometer En inglés commit. Una solicitud a una base de datos para finalizar una transacción e intentar ejecutar todas las actualizaciones a la base de datos que fueron parte de una transacción. En contraste con deshacer (rollback).

Data warehouse Es un repositorio central para todas las partes (o partes significativas) de datos que son coleccionados por distintos sistemas de negocio.

Demonio Proceso que se ejecuta en el background y desempeña una operación específica en un tiempo predefinido o en respuesta a ciertos eventos. El término *demonio* es un término de UNIX, sin embargo otros sistemas operativos también proveen soporte para demonios. Windows refiere a los *demonios* como Agentes de Sistema y Servicios.

Deshacer En inglés rollback. Solicitud a una base de datos para terminar una transacción y cancelar todas las actualizaciones a la base de datos que fueron parte de una transacción.

Dial-in Término que se le da al proceso de accesar remotamente a un sistema ó servidor vía módem. Por ejemplo, se dice que un estudiante es un *usuario dial-in* cuando accesa a un sistema de alguna universidad desde su casa u oficina.

EJB (Enterprise Jave Beans). Tecnología asociada con el lenguaje de programación orientada a objetos.

ERP (Enterprise Resource Planing) Sistema de planificación complementada de los recursos de una empresa, incluye todos los aspectos y medios computarizados para planificar su administración eficiente.

Gateway Puerta de comunicaciones de red, es una combinación de programa y hardware que comunica dos tipos diferentes de redes.

Groupware Tipo de programa de grupo, que permite el trabajo en grupo y la cooperación entre los usuarios conectados a dicha red local (como concretar citas, utilización de servicios mutuos, etc.)

GUI (Graphical User Interface) Interfaz de usuario gráfica.

GUID (Globally Unique ID). Tipo de Identificador que es generado por un algoritmo que garantiza que será único.

HTTP (HyperText Transfer Protocol), es el protocolo utilizado por la Web. HTTP define la manera en como los mensajes serán formateados y transmitidos, y que acciones deberán tomar los Servidores Web y los browsers (buscadores) en respuesta a los comandos solicitados. Por ejemplo, cuando se introduce una URL en el browser, éste envía un comando HTTP al servidor Web direccionándolo para ir a buscar y transmitir la página Web solicitada.

IDL Lenguaje de definición de interfaces (IDL) que especifica las interfaces entre objetos *CORBA*

Instancia La información de un *objeto* se almacena en estructuras de datos que se conocen como variables de instancia (globales) y conforman el estado interno del *objeto*.

Interfaz Colección de métodos. Ejemplo: La interfaz de un empleado incluye los métodos para modificar el nombre del empleado.

Legado de Aplicaciones Aplicaciones heredadas de lenguajes, plataformas y técnicas anteriores a la tecnología actual. La mayoria de las empresas que utiliza computadoras tiene legado de aplicaciones y bases de datos que prestan servicio a las necesidades críticas del negocio. Una caracteristica importante de los nuevos productos de software es la habilidad para trabajar con aplicaciones de legado de las compañias.

Librería de tipos En inglés Type Library. En términos de Microsoft, es una descripción de un componente y sus interfaces.

Marshal Proceso de reunir datos y transformarlos en su formato estándar antes de ser transmitido por la red. Para que un *objeto* sea transportado en una red, primero debe ser convertido en una cadena de datos (data stream) que corresponde a la estructura del paquete del protocolo de transferencia. Esta conversión es conocida como "data marshalling" (ordenación de datos).

Middleware Software intermedio. Software que conecta dos aplicaciones separadas. En el contexto de multicapas, el middleware ocupa la capa intermedia. Las

categorías más comunes del middleware incluyen: MonitoresTP, Ambientes DCE, sistemas RPC, ORBs (Object Request Brokers) y Sistemas de acceso a Bases de Datos.

Multi-casting Transmisión de mensajes e información desde la computadora central al resto de las computadoras de una red.

Objeto Es una estructura de programación que agrupa información (datos) junto con las operaciones que la manipulan.

Objetos comerciales *Objeto* local o distribuido que desempeña un conjunto de actividades asociadas a un proceso comercial en específico.

OLTP (Online Transaction Processing). Sistemas de misión crítica que a menudo interactuan con Servidores de Transacciones en vez de Servidores de Base de Datos. Es un sistema que procesa transacciones inmediatamente a su solicitud, a diferencia de una solicitud batch que primero es almacenada para su procesamiento posterior. OLTP es esencial para llevar registros contables e inventarios.

OMG (Object Management Group). Corporación de software que definió los estándares de componentes distribuidos, especialmente *CORBA*.

ORB (Object Request Broker). Implementación de la arquitectura *CORBA*.

OTM (Object Transaction Monitor). Término derivado de *TPM*.

Proxy Servidor situado entre una aplicación cliente, como el Web browser, y un servidor real. Intercepta todas las solicitudes del servidor real para verificar si puede atender la solicitud por si mismo. En el contexto de *COM+*, proxy es sinónimo de *instancia* suplente.

Reglas del Negocio (Business Rules) Restricciones que delimitan la forma en que los datos deberán ser capturados, relacionados y representados dentro de un sistema de base de datos. El grado de complejidad en un proyecto de base de datos es proporcional al porcentaje de esfuerzo empleado en la definición de las reglas del negocio. Cada compañía tiene su propio conjunto de reglas de negocio.
En el establecimiento de las reglas del negocio se deberán reflejar las restricciones que existen en el negocio dado, de modo que nunca sea posible llevar a cabo acciones no válidas. Ejemplos de reglas del negocio son: no permitir crear facturas pertenecientes a clientes inexistentes, controlar que el saldo negativo de un cliente nunca sobrepase cierta cantidad, etc.

RPC (Remote Procedure Call). Una técnica para crear sistemas distribuidos basados en la invocación de procedimientos.

Servicio Operación que un componente desarrolla a petición de un usuario u otro componente. En el contexto de objetos distribuidos, a la aplicación que proporciona un conjunto de habilidades o capacidades a menudo se le da el nombre de *servicio*.

SPI (Service Provider Interface). Interfaz Proveedora de Servicio.

Stream Es un *objeto* que conoce como leer y escribir datos en un espacio dado de almacenamiento. Este almacenamiento puede ser en memoria o en un

archivo en un disco.

Stub (cabo) Sinónimo de *instancia* suplente. Rutina que no hace nada más que declararse a sí misma y los parámetros que acepta. Un Stub es comúnmente utilizado para almacenar rutinas que aún necesitan ser desarrolladas. El stub contiene el código suficiente para ser compilado y ligado con el resto del programa. Término utilizado para indicar el llamado de una *instancia* desde otra *instancia*. En *COM+* stub es sinónimo de *proxy*. En *CORBA* cuando se trata de una *instancia* del servidor se le da el nombre de *esqueleto*.

Tecnología de objetos Término relacionado con la programación orientada a objetos. Se encarga de unir sistemas de información cliente/servidor complejos ensamblando y extendiendo componentes de software reusables.

Tecnología de objetos distribuidos Término relacionado con la ingeniería de objetos distribuidos. Tecnología diseñada para crear sistemas cliente/servidor donde los datos y las reglas del negocio se encuentran encapsulados en objetos, permitiendo que éstos puedan ser colocados en cualquier parte dentro de un sistema distribuido.

TPM (Transaction Procesing Monitor). Infraestructura para la capa intermedia generalmente usada en sistemas comerciales de grandes dimensiones.

Unmarshal Proceso contrario al que realiza *marshal*.

XML (eXtensible Markup Language). Formato de datos neutral y autodescriptivo.

BIBLIOGRAFIA

- 3-Tier Client/Server at work.
  Jeri Edwards
  Wiley Computer Publishing
  Estados Unidos, 1999

- Client/Server Survival Guide
  Robert Orfali, Dan Harkey, Jeri Edwards
  Wiley Computer Publishing
  Estados Unidos, 1999

- Delphi COM programming
  Eric Harmon
  Macmillan technical publishing
  Estados Unidos, 2000

- Principles of Transaction Procesing
  Philip A. Bernstein, Eric Newcomer
  Morgan Kaufmann
  Estados Unidos, 1997

- Distributed Computing: Implementation and Management Strategies
  Khanna, R.
  Prentice Hall
  Estados Unidos, 1996

- Integrating CORBA and COM Applications
  David Curtis, Michael Rosen
  Wiley Computer Publishing
  Estados Unidos, 1999

- COM+ and the Battle for the Middle Tier
  Sessions, Rogger
  Wiley Computer Publishing
  Estados Unidos, 2000

White Papers y Artículos de Internet

- COM versus CORBA: A Decision Framework
  http://www.quoininc.com/company/articles/COM_CORBA.pdf
  Por Owen Tallman y J. Bradford Kain
  Marzo 1998

- IIOP: OMG's Internet Inter-ORB Protocol: A Brief Description
  http://www.omg.org/library/iiop4.html
  Por Mike Bradley del Object Management Group
  Mayo 1997

- Comparing ActiveX and CORBA/IIOP
  http://www.omg.org/library/activex.html
  Por Andrew Watson, Richard Soley, y Mike Bradley del Object Mangement Group
  Febrero 1997

- Side by Side, Step by Step, and Layer by Layer
  http://www.research.microsoft.com/~ymwang/papers/HTML/DCOMnCORBA/S.html
  Por P. E. Chung, Y. Huang, S. Yajnik, D. Liang, J. C. Shih, C.-Y. Wang, y Y. M. Wang

- Java, RMI and CORBA
  http://www.omg.org/library/wpjava.html
  Por David Curtis del Object Management Group
  Mayo 1997

- A Detailed Comparison of CORBA, DCOM and Java/RMI
  http://my.execpc.com/~gopalan/misc/compare.html
  Por Gopalan Suresh Raj
  Septiembre 1998

ANEXO I

Comparación DCOM vs. CORBA mediante un ejemplo

Con la finalidad de reforzar los conceptos presentados en los capítulos IV y V he anexado este documento retomado de la página que recomienda el sitio oficial de Microsoft para la comparación de DCOM y CORBA :
"DCOM and CORBA Side by Side, Step by Step, and Layer by Layer"
http://www.research.microsoft.com/~ymwang/papers/HTML/DCOMnCORBA/S.html.

El documento presenta un programa ejemplo en C++ que sirve como punto de comparación entre las especificaciones DCOM y CORBA. En el caso de CORBA, hay cierta información que no proporciona el estándar, así que se ha optado por utilizar una implementación determinada, IONA Orbix, aunque en caso de elegir cualquier otra (Visigenic Visibroker, por ejemplo) el proceso sería similar.

El ejemplo que se va a emplear en la descripción se denomina Matriz. El servidor de Matriz mantiene una matriz de enteros y soporta dos grupos de métodos. El primero de ellos contiene dos métodos: get() y set(), que se invocan para leer y cambiar el valor de un elemento determinado de la matriz respectivamente. El segundo grupo tiene sólo un método: reset(), que coloca el valor dado en todos los elementos de la matriz. Como ejemplo de aplicación, se hará un programa que empleará get() para obtener el valor del elemento (0,0), incrementará ese valor en una unidad y después llamará a reset() para actualizar toda la matriz.

Para CORBA se definiran tres interfaces: grid1, que contiene a get() y a set(); grid2, que contiene a reset(); y grid, construida por herencia múltiple de las dos anteriores. En el caso de DCOM, se definirán dos interfaces, IGrid1 e IGrid2, para los dos grupos de métodos. La implementación del objeto Grid usa herencia para construirlo a partir de las dos interfaces IGrid1 e IGrid2.

TESIS CON
FALLA DE ORIGEN

Para simplificar, sólo se mostrará el código esencial. En la tabla 1 se puede observar el primer archivo, la especificación IDL. El correspondiente a DCOM asocia, asimismo, varias interfaces a una misma clase, como aparece en el bloque *coclass*. Este código pasaría por un compilador de IDL para generar el código del stub y la cabecera de la interfaz (grid.h o grid.hh) que usarán tanto el servidor como el cliente. En DCOM, como se mencionó en el capítulo IV, cada interfaz lleva asociada un **GUID** o identificador único global, llamado **Interface ID** (IID). Del mismo modo, a cada clase se le asigna un único identificador **Class ID** (CLSID). Por otro lado, cada interfaz COM debe heredar propiedades de la interfaz IUnknown, que consta de un método llamado QueryInterface() para navegar entre diferentes interfaces del mismo objeto, y otros dos métodos AddRef() y Release() para contar las referencias. Así, un objeto COM lleva cuenta de sus clientes, y puede descargarse a sí mismo cuando no se le necesita.

IDL DCOM

```
// uuid y definición de IGrid1
    [  object,
       uuid(3CFDB283-CCC5-11D0-BA0B-
00A0C90DF8BC),
       helpstring("Interfase Igrid1"),
       pointer_default(unique)
    ]
interface Igrid1 : IUnknown {
        import "unknwn.idl";
        HRESULT get([in] SHORT n, [in] SHORT m,
         [out] LONG *value);
        HRESULT set([in] SHORT n, [in] SHORT m,
         [in] LONG value);
    };

// uuid y definicion de IGrid2
    [  object,
       uuid(3CFDB284-CCC5-11D0-BA0B-
00A0C90DF8BC),
        helpstring("Interfase Igrid2"),
        pointer_default(unique)
    ]
    interface Igrid2 : IUnknown      {
        import "unknwn.idl";
        HRESULT reset([in] LONG value);
    };

// uuid y definición de la librería de tipos
[    uuid(3CFDB281-CCC5-11D0-BA0B-00A0C90DF8BC),
    version(1.0),
    helpstring("Libreria de tipos grid 1.0)
]
library GRIDLib
{
    importlib("stdole32.tlb");
    // uuid y definición de clase
    [uuid(3CFDB287-CCC5-11D0-BA0B-00A0C90DF8BC),
        helpstring("Clase Grid")
    ]
    // multiples interfaces
    coclass Cgrid
    {   [default] interface Igrid1;
         interface IGrid2;
    };
};
```

IDL CORBA

```
interface grid1
{
        long get(in short n, in short m);
        void set(in short n, in short m, in long
value);
};

interface grid2
{
        void reset(in long value);
};

interface grid: grid1, grid2
{
};
```


TESIS CON
FALLA DE ORIGEN

Tabla 1. Especificación IDL.

En la tabla 2 aparece el código que muestra la forma en cómo la clase del servidor se deriva de las interfaces. El código DCOM incluye la definición del método CClassFactory, que suele usarse aunque no es necesario. Como puede apreciarse, AddRef() incrementa la cuenta de referencia, mientras que Release() la decrementa. Cuando la cuenta llega a cero, el servidor se descarga a sí mismo.

En CORBA, el programador implementa la clase grid_i. Hay dos maneras de asociar la clase implementada con la de la interfaz: el enfoque basado en herencia y el basado en delegación. En este ejemplo, se ha optado por la herencia. Así, el compilador IDL para Orbix también genera una clase llamada gridBOAimpl que se encarga de instanciar la clase esqueleto. Esta clase es el Object Adapter que se mencionó al principio. La clase grid BOAimpl hereda las propiedades de la clase grid, que a su vez hereda las suyas de la clase CORBA:Object. La clase grid_i hereda de gridBOAimpl para completar el enlace entre las clases de la interfaz y de la implementación.

BOAimpl hereda las propiedades de la clase *grid*, que a su vez hereda las suyas de la clase CORBA::Object. La clase grid_i hereda de gridBOAimpl para completar el enlace entre las clases de la interfaz y de la implementación.

Definición del servidor - DCOM

```
#include "grid.h" //generar-IDL archivo cabecera

class CclassFactory : public IclassFactory {
 public:
  // Iunknown
  STDMETHODIMP QueryInterface(REFIID riid,
   void** ppv);
  STDMETHODIMP_(ULONG) AddRef(void)  { return 1;
   };
  STDMETHODIMP_(ULONG) Release(void) { return 1;
   }

  // IclassFactory
  STDMETHODIMP CreateInstance(LPUNKNOWN
   punkOuter,
                REFIID iid, void **ppv);
  STDMETHODIMP LockServer(BOOL fLock)
          { return E_FAIL; };
};

class CGrid : public Iqridl, public IGrid2 {
 public:
  // Iunknown
  STDMETHODIMP QueryInterface(REFIID riid,
   void** ppv);
  STDMETHODIMP_(ULONG) AddRef(void)
   { return InterlockedIncrement(&m_cRef); }
  STDMETHODIMP_(ULONG) Release(void)
   { if (InterlockedDecrement(&m_cRef) == 0)
```

Definición del servidor - CORBA

```
#Include "grid.hh" //genera-IDL archivo cabecera

class grid_i  : public gridBOAImpl (public:
  Virtual CORBA::Long get(CORBA::Short n,
  CORBA::Short m,
        CORBA::Environment &env);
  Virtual void set(CORBA::Short n, CORBA::Short
        m, CORBA::Long value, CORBA::Environment
        &env);

  Virtual void reset(CORBA::Long value,
        CORBA::Environment &env);
  Grid_i(CORBA::Short h, CORBA::Short w);
  Virtual ~grid_i();
  Private:
        CORBA::Long **m_a;
        CORBA::Short m_height, m_width;
};
```


TESIS CON
FALLA DE ORIGEN

```
      { delete this; return 0; }
    return 1; }
  // IGrid1
  STDMETHODIMP get(IN SHORT n, IN SHORT m,
          OUT LONG *value);
  STDMETHODIMP set(IN SHORT n, IN SHORT m,
          IN LONG value);
  // IGrid2
  STDMETHODIMP reset(IN LONG value);

  CGrid(SHORT h, SHORT w);
  ~CGrid();
 private:
  LONG  m_cRef, **m_a;
  SHORT m_height, m_width;
};
```

Tabla 2. Cabeceras del servidor.

En la tabla 3 se muestran los métodos de la clase del servidor. El código de DCOM también implementa algunos métodos de ClassFactory. En la tabla 4 aparece el programa principal del servidor. El realizado para DCOM crea un evento y espera a que sea señalado cuando todos los objetos servidores desaparezcan y así pueda terminar. De manera similar, el programa CORBA crea una instancia de la clase grid_i y queda bloqueado en impl_is_ready() para recibir las llamadas de los clientes. Si el servidor no recibe nada en un determinado periodo, se cierra.

Implementación del servidor - DCOM

```
#include "cgrid.h"

STDMETHODIMP
CClassFactory::QueryInterface(REFIID riid,
void** ppv) {
    if (riid == IID_IClassFactory || riid ==
IID_IUnknown) {
      *ppv = (IClassFactory *) this;
      AddRef(); return S_OK;
    }
    *ppv = NULL;
    return E_NOINTERFACE;
}

STDMETHODIMP
CClassFactory::CreateInstance(LPUNKNOWN p,
REFIID riid, void** ppv) {
    IGrid1* punk = (Igrid1*) new CGrid(100, 100);
    HRESULT hr = punk->QueryInterface(riid, ppv);
    punk->Release();
    return hr;
}

STDMETHODIMP CGrid::QueryInterface(REFIID riid,
void** ppv) {
    if (riid == IID_IUnknown || riid ==
IID_IGrid1)
      *ppv = (IGrid1*) this;
    else if (riid == IID_IGrid2) *ppv = (IGrid2*)
this;
    else { *ppv = NULL; return E_NOINTERFACE; }

    AddRef(); return S_OK;
}

STDMETHODIMP CGrid::get(IN SHORT n, IN SHORT m,
OUT LONG* value) {
     *value = m_a[n][m];
     return S_OK;
```

Implementación del servidor - CORBA

```
#include "grid_i.h"

CORBA::Long grid_i::get(CORBA::Short n,
CORBA::Short m, CORBA::Environment &) {
     return m_a[n][m];
}
```


TESIS CON
FALLA DE ORIGEN

```
}

STDMETHODIMP CGrid::set(IN SHORT n, IN SHORT m,
IN LONG value) {
     m_a[n][m] = value;
     return S_OK;
}

STDMETHODIMP CGrid::reset(IN LONG value) {
     SHORT n, m;
     for (n=0; n < m_height; n++)
       for (m=0; m < m_width; m++)
         m_a[n][m] = value;
     return S_OK;
}


CGrid::CGrid(SHORT h, SHORT w) {
  m_height = h;
  m_width= w;
  m_a = new LONG*[m_height];
  for (int i=0; i < m_height; i++)
    m_a[i] = new LONG[m_width];
  m_cRef = 1;
}

extern HANDLE hevtDone;

CGrid::~CGrid () {
  for (int i=0; i < m_height; i++)
    delete[] m_a[i];
  delete[] m_a;
  SetEvent(hevtDone);
}
```

```
void grid_i::set(CORBA::Short n, CORBA::Short m,
     CORBA::Long value, CORBA::Environment &) {
     m_a[n][m] = value;
}


void grid_i::reset(CORBA::Long value,
CORBA::Environment &) {
     short n, m;
     for (n = 0; n < m_height; n++)
       for (m = 0; m < m_width; m++)
         m_a[n][m]=value;
     return;
}

grid_i::grid_i(CORBA::Short h, CORBA::Short w) {
     m_height=h;    // establece la altura
     m_width=w;     // establece el ancho
     m_a = new CORBA::Long* [h];
     for (int i = 0; i < h; i++ )
       m_a[i] = new CORBA::Long[w];
}



grid_i::~grid_i () {
     for (int i = 0; i < m_height; i++)
       delete[] m_a[i];
     delete[] m_a;
}
```

Tabla 3. Implementación del servidor.

Programa principal servidor - DCOM

```
HANDLE hevtDone;
Void main()
{
     // Evento utilizado para señalar este thread
principal
     hevtDone = CreateEvent(NULL, FALSE, FALSE,
NULL);
     hr = CoInitializeEx(NULL,
COINIT_MULTITHREADED);
     CClassFactory* pcf   new CclassFactory;
     hr = CoRegisterClassObject(CLSID_CGrid, pcf,
          CLSCTX_SERVER,  REGCLS_MULTIPLEUSE ,
&dwRegister);
     // Espera hasta que el evento está
establecido para CGrid::~CGrid()
     WaitForSingleObject(hevtDone, INFINITE);
     CloseHandle(hevtDone);
     CoUninitialize();
}
```

Programa principal servidor - CORBA

```
int main()
{
  // crea un objeto grid utilizando la clase de
implementación grid_i


  grid_i ourGrid(100,100);

  try {
    // informa a Orbix que se ha completado la
inicialización del servidor:
CORBA::Orbix.impl_is_ready("grid");
  } catch (...) {
    cout << "Excepción inesperada" << endl;
    exit(1);
  }
}
```

Tabla 4:Programa principal servidor.

Por último, en la tabla 5 se muestra el código del cliente. Puede apreciarse que el código DCOM tiende a ser más largo que el de CORBA, debido a las llamadas al método IUnknown.

Después de compilar y antes de ser ejecutados, DCOM y CORBA requieren un proceso de registro para el servidor. En CORBA, la asociación entre el nombre de la interfaz y la localización del ejecutable se guarda en el **repositorio de implementación**. En DCOM, la

TESIS CON
FALLA DE ORIGEN

asociación entre el CLSID y la localización del ejecutable se guarda en el **registro** (regedit.exe) de Windows. Además, como la interfaz del proxy/stub es también un objeto COM, necesita ser registrada de la misma manera.

Cliente - DCOM

```
#include "grid.h"

void main(int argc, char**argv)
{
    IGrid1      *pIGrid1;
    IGrid2      *pIGrid2;
    LONG        value;

    CoInitialize(NULL); // inicializa COM
    CoCreateInstance(CLSID_CGrid, NULL,
CLSCTX_SERVER, IID_IGrid1, (void**) &pIGrid1);
    pIGrid1->get(0, 0, &value);
    pIGrid1->QueryInterface(IID_Igrid2, (void**)
&pIGrid2);
    pIGrid1->Release();
    pIGrid2->reset(value+1);
    pIGrid2->Release();
    CoUninitialize();
}
```

Cliente CORBA

```
#include "grid.hh"

void main (int argc, char **argv)
{
  grid_var gridVar;

  CORBA::Long value;

  // bind to "grid" object; Orbix-specific
  gridVar = grid::_bind(":grid");

  value = gridVar->get(0, 0);


  gridVar->reset(value+1);
}
```

Tabla 5: Programas cliente.

A. Capa Superior. Arquitectura Básica de Programación

En esta capa se pueden comprobar las diferencias para el programador entre DCOM y CORBA. En ambos casos, la conexión entre cliente y servidor es transparente, ofreciendo la imagen de estar ejecutándose en el mismo espacio de memoria en la misma máquina. Las diferencias fundamentales residen en la manera de especificar una interfaz. En la tabla 6 aparece un desglose paso a paso de las operaciones que hay que realizar en cada uno de los dos modelos, ilustradas en las figuras 4 y 5.

En CORBA, un objeto puede activarse simplemente invocando a cualquiera de sus métodos. Algunos fabricantes proveen métodos especiales, como _bind en Orbix, para activar un objeto servidor y obtener su referencia.

DCOM	CORBA
Activación del objeto	
1. El cliente llama a la función CoCreateInstance() con CLSID_Grid e IID_Grid1 2. COM arranca un objeto servidor para CLSID y llama a CoRegisterClassObject() para registrar cada una. El servidor se queda esperando un evento. 3. El servidor crea *class factories* para cada CLSID. y	1. El cliente llama a grid::bind() 2. ORB arranca un servidor que contiene un objeto que soporta la interfaz grid. 3. En cada constructor, se realizan llamadas para crear y registrar referencias para los objetos. 4. ORB devuelve la referencia del objeto en gridVar al cliente.

TESIS CON
FALLA DE ORIGEN

llama a CoRegisterClassObject() para registrar cada una. El servidor se queda esperando un evento. 4. COM obtiene el puntero IclassFactory para CLSID_Grid y llama a CreateInstance(). 5. En CreateInstance(), el servidor crea un objeto y hace una llamada a QueryInterface() para obtener n puntero a la interfaz IDD_Igrid1. 6. COM devuelve el puntero como pIGrid1 al cliente.	
Invocación del método	
1. El cliente llama a pIGrid1->get(), que luego llamará a Cgrid::get() en el servidor. 2. Para obtener un puntero a otra interfaz IID_Igrid2 de la misma instancia del objeto, el cliente llama a pIGrid->QueryInterface(), que llama a Cgrid::QueryInterface. 3. Cuando se acaba de usar pIGrid1, el cliente llama a pIGrid-ZRelease() 4. Del mismo modo con reset().	1. El cliente llama a gridVar->get() que llamara a grid_i::get() en el servidor. 2. Del mismo modo con reset().

Tabla 6. Capa Superior.

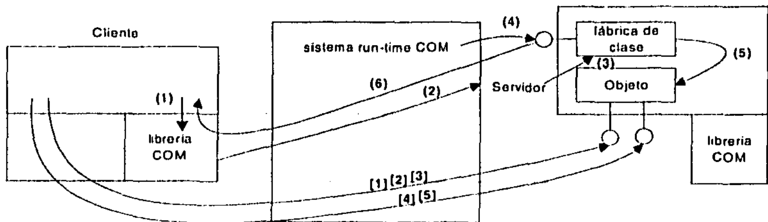


Figura 1. Capa superior DCOM.

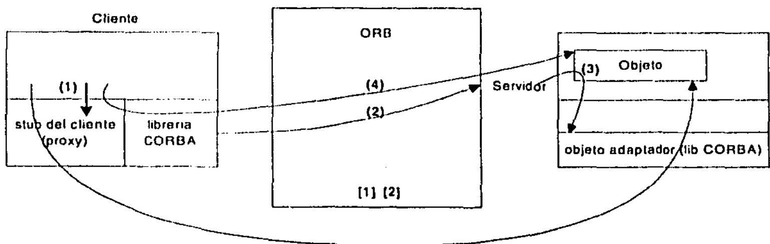


Figura 2. Capa superior CORBA.

Una diferencia a considerar entre los dos modelos es la forma de tratar las excepciones. CORBA ofrece soporte para tratar excepciones estándar de C++ y algunas otras, además

TESIS CON FALLA DE ORIGEN

de permitir la declaración de excepciones definidas por el usuario. Por el contrario, DCOM exige que todos lo métodos devuelvan un código de error de 32 bits llamado HRESULT, dejando a las herramientas de programación, como Visual C++, convertir los errores en excepciones.

B. Capa Intermedia. Arquitectura de Proceso Remoto

Esta capa contiene la infraestructura necesaria para ofrecer al cliente y al servidor la ilusión de encontrarse en el mismo espacio de direcciones.

Las diferencias fundamentales entre DCOM y CORBA en esta capa se encuentran en la manera de registrar los objetos servidores y cuándo se crean las instancias del proxy/stub/esqueleto. DCOM emplea el registro de Windows para lo primero, mientras que CORBA lo hace en el ORB. En cuanto a la creación de objetos, DCOM crea un stub después de llamar a CreateInstance(), mientras que el esqueleto de CORBA se crea en el constructor de la clase grid_i; ambos crean el objeto proxy cuando llega el puntero que referencia al servidor.

Para enviar datos a través de distintos espacios de direcciones se emplea el proceso denominado *marshaling*. Como se mencionó en el capítulo 5, éste proceso empaqueta los parámetros de la llamada y devuelve un valor en el servidor en un formato de transmisión estándar, como una RPC normal. La operación contraria se conoce como *unmarshaling*, y convierte los datos entrantes del formato estándar al propio del espacio de memoria destino. DCOM también proporciona un mecanismo para hacer este proceso según la definición del usuario. Este procedimiento, denominado *custom marshaling*, puede emplearse para soportar infraestructuras de comunicación específicas de la aplicación.

Aquí aparece un elemento que ya reseñamos al comienzo del anexo: el **OA** (*Object Adaptor*). Se sitúa sobre el ORB, y se encarga de realizar la conexión con la implementación del objeto. Este mecanismo proporciona servicios como generación e interpretación de las referencias de objetos, invocación de métodos, activación y desactivación de objetos.. El Adaptador Básico de Objeto (BOA) define un adaptador que puede emplearse para la mayoría de las implementaciones de objetos convencionales. Sin embargo, ha aparecido el reemplazo de BOA: el Adaptador Portátil de Objeto (POA).

Con este adaptador se elimina la dependencia de BOA con respecto al fabricante, ya que en las especificaciones originales de CORBA no se hablaba de la implementación del enlace ORB/BOA, y existen tantos esquemas BOA distintos como productos CORBA comerciales.

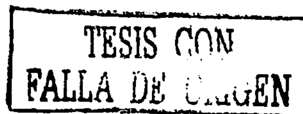
C. Capa Inferior: Arquitectura del Protocolo de Comunicación

Esta capa especifica el protocolo para soportar la comunicación entre clientes y servidores. Las diferencias principales entre DCOM y CORBA se encuentran en la representación de referencias a objetos y en el formato en el que los datos son "empaquetados" (*marshaled*) para su transmisión en un entorno heterogéneo. CORBA no especifica un protocolo determinado para la comunicación entre un cliente y un servidor ejecutándose en ORBs del mismo fabricante. Este protocolo, así, se deja a la elección del fabricante. Sin embargo, para soportar la interoperabilidad de diferentes ORBs, se especifica un Protocolo General Inter-ORB (GIOP). Existe un GIOP basado en TCP/IP, denominado *Internet Inter-ORB Protocol* (IIOP).

El protocolo empleado en DCOM se basa principalmente en la especificación OSF DCE RPC con algunas extensiones, como ya se ha comentado en capítulos anteriores de este documento.

D. Interoperabilidad CORBA/DCOM

Un tema importante entre ambas tecnologías es aquel al que se refiere al poder hacer que objetos CORBA funcionen en un cliente que espera objetos DCOM o viceversa. La OMG ha desarrollado un estándar para asegurar la interoperabilidad entre los dos sistemas. Este estándar se encuentra plasmado en el documento *"COM/CORBA Internetworking specification"*. Sin embargo, no todas las formas de interoperabilidad son iguales, ni todos los distribuidores soportan el estándar completo. El documento hace referencia a dos elementos diferentes: interoperabilidad entre OLE Automation y CORBA e interoperabilidad entre COM y CORBA. Actualmente Digital Equipment, Hewlett-Packard y Sun ofrecen completa interoperabilidad.


TESIS CON
FALLA DE ORIGEN

Para cada uno de los dos elementos mencionados anteriormente, la especificación describe dos niveles de interoperabilidad: *mapping* (interoperabilidad en una dirección) e *interworking* (bidireccional). La solución unidireccional hace que los objetos de un sistema sean visibles para el otro, pero no al revés. Por otro lado, dos enlaces de una dirección no constituyen uno bidireccional. En primer lugar porque el enlace es asimétrico, y en segundo lugar porque las clases suelen devolver objetos desde sus miembros o propiedades, lo que complica la portabilidad de tipos. Así, por ejemplo, CORBA soporta *Arrays* mientras que DCOM emplea *SafeArrays*; DCOM tiene definidos dos tipos de cadenas, mientras que CORBA sólo tiene uno; y así unas cuantas diferencias más.

Se pueden diferenciar dos esquemas de interoperación : *system-neutral* y *system-centric.* Con el primer esquema, cualquier clase CORBA puede instalarse en OLE y viceversa. Una vez instalada, aparece como si fuera propia de aquel. En el segundo esquema se consigue que los desarrolladores empleen un sólo modelo de programación. Actualmente esto está disponible tan sólo por parte de los distribuidores de CORBA, aunque la elección de un modelo u otro depende de las necesidades concretas de los desarrolladores y/o de los usuarios.

E. Resumen

Como se ha podido comprobar con el ejemplo, las arquitecturas de DCOM y CORBA/Orbix son muy similares. Ambas proporcionan infraestructura para la activación y acceso transparente a objetos remotos. En la tabla 7 se resumen los términos principales de las dos arquitecturas.

Las diferencias principales pueden resumirse como sigue. En primer lugar, DCOM soporta objetos con interfaces múltiples y proporciona un método QueryInterface() para navegar entre las interfaces. Con este método se puede conseguir también que un objeto proxy/stub cargue dinámicamente interfaces de proxies/stubs remotos. Estos conceptos no existen en CORBA. En segundo lugar, cada interfaz CORBA hereda sus propiedades de CORBA::Object, el constructor en el que se realizan tareas comunes, como el registro

TESIS CON
FALLA DE ORIGEN

del objeto, la generación de su referencia, la instanciación del esqueleto. En DCOM, estas tareas se realizan explícitamente por los programas servidores o se manejan dinámicamente por el sistema DCOM. Por último, el protocolo de comunicación de DCOM está estrechamente ligado a RPC, mientras que en CORBA no sucede así.

	DCOM	CORBA
Capa superior: Arquitectura de Programación Básica		
Clase común base	*Iunknown*	*CORBA::Object*
Identificador de clase	*CLSID*	Nombre de la interfaz
Identificador de la interfaz	*IID*	Nombre de la interfaz
Activación del cliente	*CoCreateInstance()*	Cualquier llamada a un método
Manejador del objeto	Puntero a la interfaz	Referencia del objeto
Capa intermedia: Arquitectura de Proceso Remoto		
Enlace con las implementaciones	Registro	*Implementation Repository*
Información sobre tipos	Librería de tipos (*Type library*)	*Interface Repository*
Localización	SCM (*Service Control Manager*)	ORB
Activación	SCM	OA
Stub del cliente	Proxy	Stub/proxy
Stub del servidor	Stub	Esqueleto
Capa inferior: Protocolo de Comunicación		
Server endpoint resolver	OXID resolver	ORB
Server endpoint **Referencia del objeto**	Object exporter OBJREF	OA IOR
Generación de la referencia	Object exporter	OA
Formato de marshalling	NDR	CDR
Identificador de la instancia de la interfaz	IPID	Object_key

Tabla 7. Correspondencia de términos y entidades.

TESIS CON
FALLA DE ORIGEN