29/3/99
9 (nueve)

Trabajo de Grado

Director: Doctor Gustavo Rossi

Codirector: Lic. Xavier Alvarez

Alumnos: Mariana Perstossi
Gabriel P. A. Casarini

Título: "Objetos Distribuidos con CORBA"

Director: Doctor Gustavo Rossi.

Codirector: Lic. Xavier Alvarez

Alumnos: Mariana Pertossi, N° Legajo: 34292/8, DNI: 23.724.300

Gabriel P. A. Casarini, N° Legajo: 34535/8, DNI: 22.798.606

Fecha: 30 de noviembre de 1998

Lugar: Facultad de Ciencias Exactas, Universidad Nacional de La Plata

Queremos agradecer muy especialmente a nuestro director y codirector. A Gustavo por responder los e-mails y alentarnos a seguir adelante continuamente. A Xavi por la generosidad de su tiempo, paciencia e ideas durante las interminables reuniones que mantuvimos a lo largo del año.

Mariana y Gabriel

CONTENIDO

Contenido de los diskettes:

- CORBA.DOC: está comprimido (CORBAARJ) y es el documento del trabajo de grado, escrito en el procesador de textos Word 97.
- INDICE.DOC: es el indice del documento
- CORBA.TXT: es el documento del trabajo de grado, en ascii.
- INDICE.TXT: es el indice del documento, en ascii.
- ODYSSEY: es una carpeta que guarda todos los fuentes de la aplicación desarrollada.
- GRAPHLT.TTF: es un font utilizado para el documento generado en Word 97.
- GRAPHLTN.TTF: es un font utilizado para el documento generado en Word 97.
- ARJ.EXE: el utilitario para comprimir y descomprimir archivos.

Documentación escrita:

Parte 1. Modelo Cliente/Servidor con Objetos Distribuidos

Parte 2. CORBA: El Bus de Objetos Distribuidos

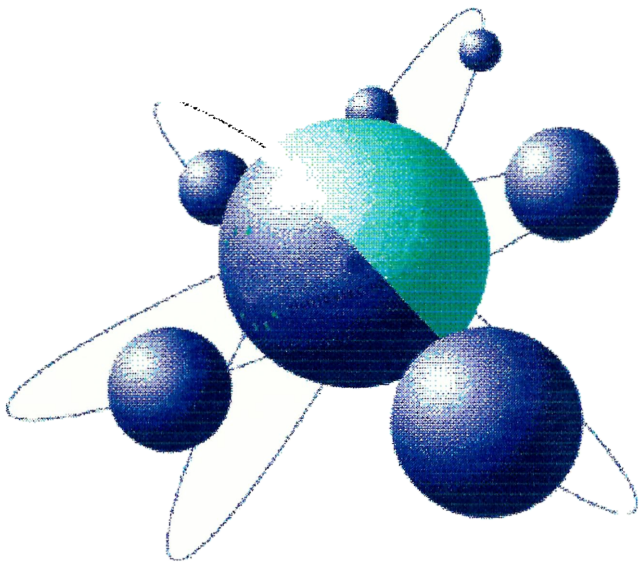
Parte 3. Frameworks para Business Objects y Componentes

Parte 4. Desarrollo de una aplicación distribuida con CORBA

Parte 5. Apéndices

Parte 1

El Modelo Cliente/Servidor con Objetos Distribuidos



MOTIVACION

La arquitectura cliente/servidor ha provocado un profundo cambio de paradigmas en la industria informática. Está reemplazando a las aplicaciones monolíticas de los mainframes por otras que están organizadas en un esquema de clientes y servidores. El cliente normalmente suministra una interface gráfica, mientras que el servidor administra el acceso a los recursos compartidos, como una base de datos.
Los objetos distribuidos producen una revolución dentro de ese cambio de paradigmas. Dividen a los clientes y servidores de las aplicaciones en componentes capaces de colaborar e interoperar a través de las redes.
CORBA, *Common Object Request Broker Architecture*, es una infraestructura emergente y abierta para la computación con objetos distribuidos. Define un estándard creado por el OMG, *Object Management Group*, que automatiza muchas tareas de programación sobre redes, facilitando la interoperabilidad de las aplicaciones a través de las distintas plataformas y lenguajes de programación.

Este trabajo tiene por objetivo explorar la tecnología de CORBA aplicada a los objetos distribuidos. En la *Parte 1* presentamos las ventajas de los objetos distribuidos comparados con otras tecnologías existentes. La *Parte 2* describe en profundidad todos los detalles del ORB y los servicios disponibles. La *Parte 3* cubre los business objects y frameworks para el desarrollo de aplicaciones. En la *Parte 4* presentamos el análisis, diseño e implementación de una aplicación distribuida con CORBA. Y finalmente, en la *Parte 5*, hay un glosario de vocabulario y una lista de la bibliografía consultada durante la investigación.

INTRODUCCION: LA PROXIMA REVOLUCION

Actualmente, la industria está en el umbral de una nueva etapa debido a dos razones principales: 1) el crecimiento exponencial del ancho de banda de bajo costo en WANs (Wide Area Netwoks)–por ejemplo Internet; y 2) una nueva generación de sistemas operativos de escritorio con capacidades de multithread y conexión a redes–por ejemplo OS/2 Warp, Win 95, Win 98, Mac OS8.
El comienzo de esta nueva etapa define la transición de las aplicaciones cliente/servidor hacia otro tipo de aplicaciones más sofisticadas. El centro de gravedad se está desplazando desde las aplicaciones cliente/servidor departamentales basadas en LANs hacia otra forma más evolucionada: entidades capaces de cumplir ambos roles al mismo tiempo. La adopción de la tecnología de objetos está motivada por una demanda creciente de respuestas a los cambios de requerimientos; los objetos facilitan la administración de la complejidad.
Las tecnologías necesarias al nivel de las aplicaciones cliente/servidor para cumplir con este desafío son:

- ***Procesamiento rico de transacciones:*** poder procesar transacciones anidadas y de larga duración, que se expandan sobre varios servidores.
- ***Agentes móviles (roaming agents):*** el nuevo entorno estará poblado de agentes electrónicos de todas las clases. Los consumidores tendrán sus propios agentes que harán las búsquedas de su interés. La tecnología de agentes incluye motores para ejecutar scripting y un ambiente para que los agentes sobrevivan en cualquier máquina de la red.
- ***Manejo de datos complejos:*** documentos compuestos con multimedia que se pueden almacenar, copiar, mover, consultar y editar en cualquier lugar de la red. La mayoría de los nodos deberá suministrar apoyo para la tecnología OpenDoc y OLE. Por supuesto que debe darse soporte para la información almacenada en tablas SQL.
- ***Entidades inteligentes independientes:*** con la llegada de la siguiente generación de sistemas operativos se vislumbra un mundo donde millones de máquinas podrán ser al mismo tiempo servidores o clientes. Se requiere software distribuido que sepa autoconfigurarse y administrarse.
- ***Middleware inteligente:*** el entorno distribuido debe ser percibido como un sistema único. No deben hacerse evidentes las máquinas subyacentes, protocolos, o la red que las conecta.

¿Quién liderará esta revolución cliente/servidor?

Actualmente tenemos cuatro paradigmas o tecnologías para desarrollar aplicaciones cliente/servidor: 1) bases de datos SQL; 2) monitores de procesamiento de transacciones (MPT); 3) tecnología de trabajo en grupos (groupware); y 4) objetos distribuidos.

1) Cliente/Servidor con bases de datos SQL

Los servidores de bases de datos SQL son el modelo dominante para el desarrollo de aplicaciones cliente/servidor. Aunque SQL comenzó como un lenguaje declarativo para manipular datos, con el tiempo se hizo evidente la necesidad de manipular también las funciones que manipulan o trabajan sobre tales datos. La forma más eficiente de lograr esto es encapsulando los comandos en funciones especiales que corren en el servidor.
Este mecansimo se denomina *stored procedures* y los clientes los invocan con RPC. Un stored procedure es formalmente un conjunto de sentencias SQL y lógica procedimental que se compila y almacena en el servidor de la base de datos.

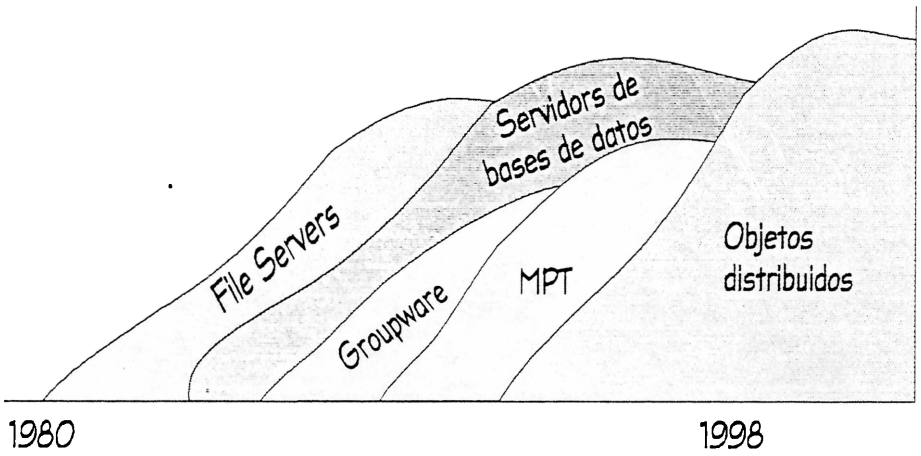


Figura 1-1. Evolución cliente/servidor a objetos distribuidos

Aunque el modelo SQL permite el desarrollo rápido de aplicaciones cliente/servidor, incluso con herramientas y paletas gráficas, no es lo suficientemente robusto para armar una infraestructura (middleware) como la que buscamos. Las razones:

- ***Administración pobre del control de procesos:*** SQL junta la lógica de la aplicación en un stored procedure o en herramientas front-end (Power Builder). Los programas son ciudadanos *"de segunda clase"* en el mundo SQL y aunque la generación actual de servidores SQL puede manejar los procesos en un mismo servidor, no hay escalabilidad hacia muchos servidores.

- ***Middleware no estándar:*** las implementaciones SQL de los diferentes desarrolladores son incapaces de interoperar. No hay un protocolo estándar para la comunicación de estos sistemas a través de las redes.
- ***SQL no es apropiado para el manejo de datos complejos:*** SQL se basa en el manejo de tipos simples de datos, pero no es apropiado para tratar con datos complejos, heterogéneos o esparcidos entre varios servidores.

2) Monitores de Procesamiento de Transacciones (Transaction Processing Monitors)

Esta tecnología proviene de los mainframes y se usa para administrar las bases de datos.
Los desarrolladores de los mainframes se dieron cuenta hace bastante tiempo que no es posible desarrollar aplicaciones críticas sin controlar los programas (o procesos) que operan sobre los datos.
Los monitores usan una *arquitectura cliente/servidor de tres niveles* (three-tired cliente/server architecture), lo que significa que el proceso está separado de los datos y de la interface del usuario.
Esta tecnología se introdujo para correr aplicaciones que pudieran atender a miles de clientes simultáneamente. Para lograrlo, ofrecen un ambiente de ejecución que opera como intermediario entre el cliente y el servidor. Esto les permite manejar las transacciones, rutearlas, balancear la carga de ejecución (load balancing) y arrancarlas automáticamente después de una caída. Los monitores también son capaces de funcionar cooperativamente.
Todas estas características son altamente deseables en los entornos cliente/servidor que involucran a miles de transacciones diarias que corren en cualquier lugar de la red.

3) Trabajo en grupo (groupware)

Por *groupware* entendemos un conjunto de tecnologías que permiten representar procesos complejos que se centran alrededor de actividades humanas colaborativas. Las tecnologías involucradas son: manejo de documentos multimediales, manejo de transacciones complejas y flujo de trabajo, email, conferencias y scheduling. Este esquema de trabajo permite recolectar datos (imágenes, texto, faxes, boletines, etc) y organizarlos en un *documento* que se puede ver, almacenar y duplicar en cualquier lugar de la red. Podemos decir que los documentos multimediales son para groupware lo que las tablas son para SQL: definen la unidad básica de manejo. La comunicación se basa principalmente en email.
Actualmente, el producto más importante que integra todas las caracterísctcas es Lotus Notes. Notes permite distribuir el trabajo sobre distintas rutas de trabajo en las que participan muchos usuarios, de una forma eficiente y natural. Sin embargo, es una tecnología propietaria y quizás no sea escalable a transacciones más sofisticadas.

4) Objetos Distribuidos

Como dijimos más arriba, el esquema cliente/servidor fue un gran cambio para las aplicaciones centralizadas y monolíticas, pues las dividió en dos mitades: cliente y servidor.

Pero desafortunadamente, cada una de esas mitades siguió siendo monolítica (construidas como un todo).
Los objetos distribuidos cambian esto. Con la infraestructura apropiada, los objetos permiten construir aplicaciones a partir de componentes capaces de interoperar y cooperar a través de los sistemas operativos y redes. La tecnología de objetos distribuidos es extremadamente apropiada para desarrollar sistemas flexibles, pues los datos y la lógica del problema están encapsulados en objetos disponibles desde cualquier lugar de un sistema distribuido.

Ventajas de los objetos distribuidos:

- Permiten compartir información entre varias aplicaciones y usuarios.
- Permiten sincronizar actividades a través de varias máquinas.
- Se pueden usar para aumentar el rendimiento asociado a cada tarea en particular.
- Se pueden usar para conectar aplicaciones que corren en una PC con información manejada por procesos UNIX o bases de datos de un mainframe.
- Son la manera de distribuir el poder de procesamiento sobre una red de computadoras, lo que hace manejable un crecimiento imprevisto. Los enfoques centralizados normalmente fallan en tales condiciones.
- Permiten que personas de diferentes ciudades contribuyan y colaboren en el mismo proceso o actividad
- Permiten que los procesos o actividades sean modificados o reimplementados sin necesidad de alterar las aplicaciones que usan tales objetos distribuidos.

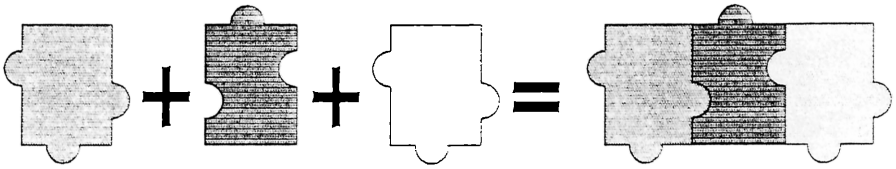
El la figura de la siguiente hoja se muestran las principales ventajas de los objetos distribuidos.

Los estándares llegaron primero

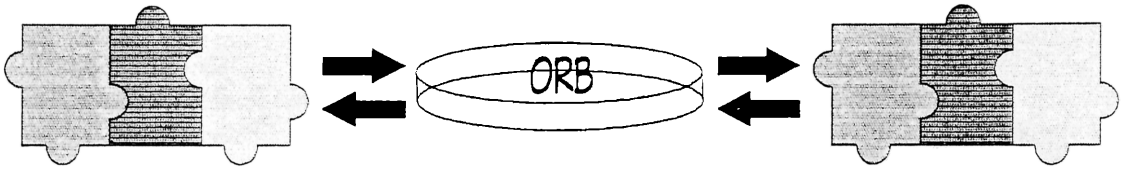
Desde 1989 un consorcio formado por varias empresas, OMG (Object Management Group), ha estado especificando la arquitectura de un bus de software abierto capaz de dar soporte a objetos de distintos desarrolladores de software para que interoperen sobre las redes y sistemas operativos. Hoy hay más de 500 miembros adheridos a OMG.

El bus de objetos provee un bus, ORB (Object Request Broker), que permite que los clientes invoquen métodos de objetos remotos en forma dinámica o estática.
A fines de 1994, OMG aprobó un conjunto de especificaciones denominadas CORBA, que definen un soporte inter–ORB basado en TCP/IP y también sobre el DCE de OSF.
El bus también ha sido extendido con servicios adicionales para crear, almacenar, definir y nombrar los objetos. Recientemente se incorporaron servicios de manejo de eventos, transacciones, control de concurrencia, relaciones, etc.

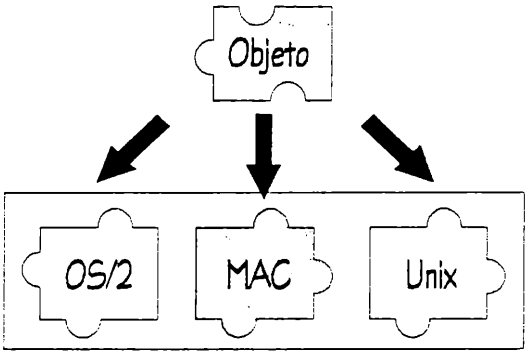
A) Plug-and-play de componentes



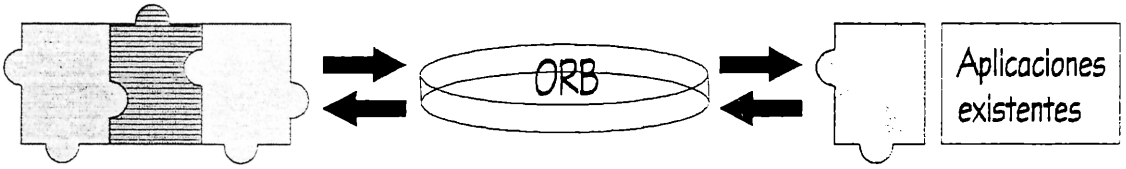
B) Interoperabilidad



C) Portabilidad



D) Coexistencia



E) Entidades independientes, capaces de autoadministrarse

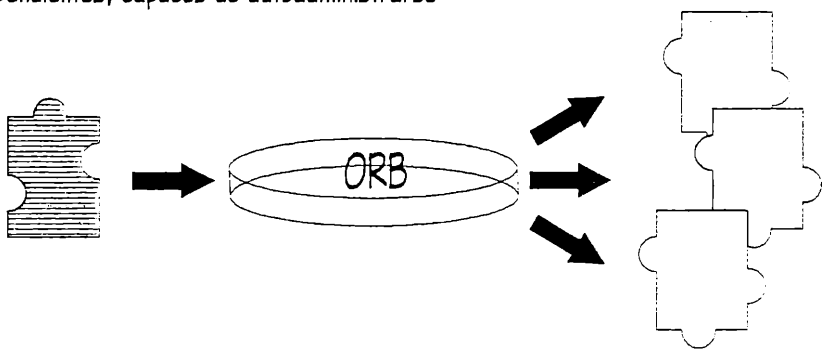
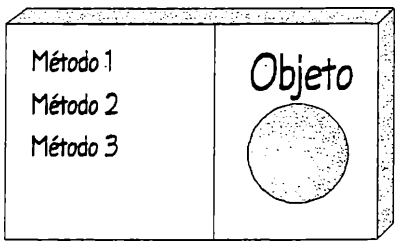


Figura 1-2. Evolución cliente/servidor a objetos distribuidos

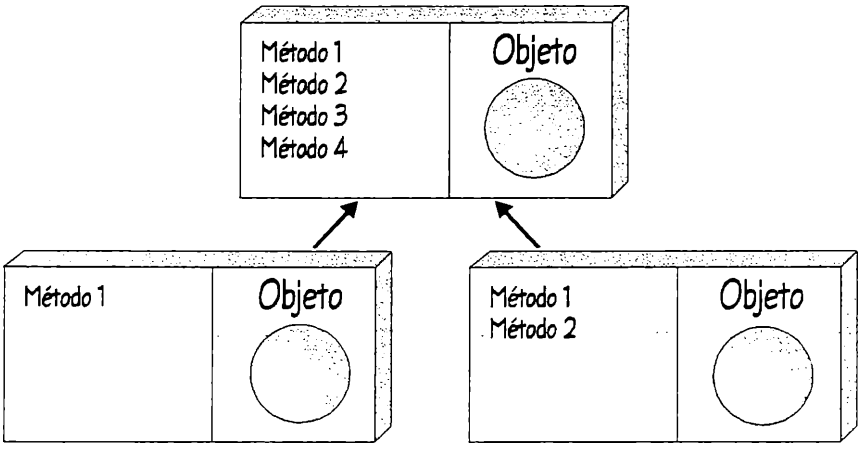
CONVENCIONES DE GRAFICOS Y SIMBOLOS

- ***Los objetos*** se representan usando un ícono con forma de esfera:

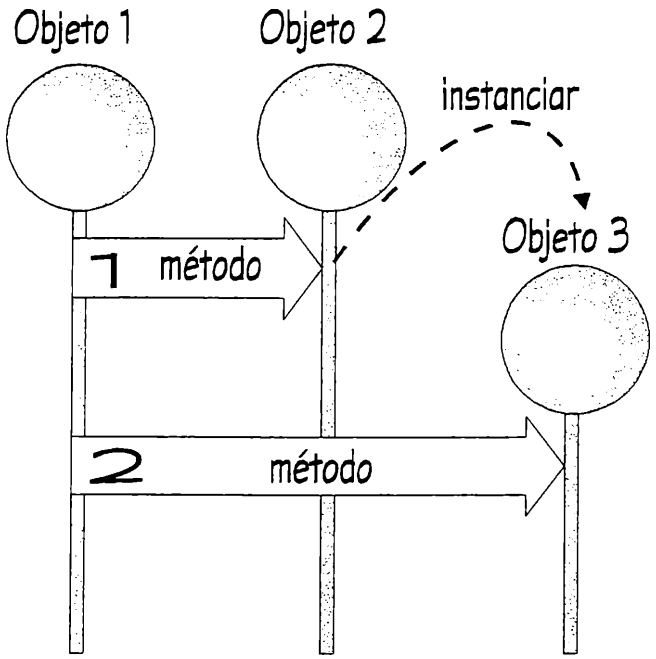
- ***El rectángulo*** representa un objeto con sus métodos; esto define una interface:



- ***Jerarquías de clases:*** la flecha representa la relación de herencia que las clase inferiores mantienen con la clase del nivel superior.



- ***Diagramas de interacción de objetos:*** muestran la ejecución de escenarios que describen las colaboraciones entre los objetos. Cada objeto se representa con un ícono con forma de esfera y una linea vertical. Las flechas horizontales representan el envío de mensajes entre los objetos . Las flechas punteadas describen otro tipo de interacciones, por ejemplo: instanciación de objetos. Los números indican el orden secuencial de las interacciones (el tiempo aumenta hacia abajo).



Finalmente hemos decido incorporar al texto los comentarios y opiniones de diferentes investigadores y personas vinculadas a la industria informática para ilustrar el panorama con una perspectiva realista.

DE LOS OBJETOS A LOS SUPERCOMPONENTES

La tecnología de objetos

> "El problema con el software, más que ser una forma de arte medieval, es que todo lo que construimos hoy es monolítivo."
>
> - Steve Mills,
> General Manager of Software, IBM
> (Marzo de 1995)

Un objeto es un fragmento de código encapsulado que tiene un nombre y una interface que describe qué cosas puede hacer el objeto. Otros programas y sistemas pueden invocar las funciones que describe la interface o simplemente reusar la función misma. Los objetos deberían permitir escribir programas más rapidamente incorporando fragmentos de código disponibles en otros objetos ya existentes–esto se denomina *herencia*. Por otro lado, un objeto maneja recursos o sus propios datos que son accesibles sólo mediante la interface pública del objeto. Esto significa que los objetos *encapsulan* los recursos y contienen toda la información necesaria para operar.

También es posible desarrollar objetos con la *composición* de otros objetos. Eventualmente, podríamos crear objetos que modelen entidades del mundo real–llamados *business objects*. Los business objects pueden ser objetos locales o distribuidos. El término business se refiere a un objeto que resuelve un conjunto de tareas asociadas a un proceso de negocios en particular (business process).

¿Qué es un objeto distribuido?

Es un objeto que puede acederse remotamente. Esto significa que un objeto distribuido puede ser usado como un objeto *tradicional*, pero desde cualquier lugar de una red. La localización del objeto no es crítica para el usuario de ese objeto.
La mayoría de la veces, los desarrollos con objetos distribuidos se basan en una configuración cliente/servidor. En cierto sentido los objetos son servidores, pues atienden pedidos de servicios. En otro son clientes, pues piden servicios a otros objetos.
Para establecer la diferencia de roles, nos referimos a los objetos que viven del *lado del servidor* y a los objetos que viven *del lado del cliente*. Los objetos que están del lado del servidor ofrecen servicios y recursos. Los objetos que están del lado del cliente piden servicios y recursos.

Podemos extender el concepto y decir que los objetos distribuidos son componentes de software independientes; piezas inteligentes de software que funcionan sobre distintas redes, sistemas operativos y herramientas. Un componente es un objeto que no está ligado a un

programa en particular, lenguaje o implementación. Los objetos desarrollados como componentes permiten la construcción rápida de aplicaciones distribuidas.

BREVE TUTORIAL DE OBJETOS

Un *objeto* es una pieza de código que posee *atributos* y ofrece servicios a través de *métodos* (también denominados operaciones o funciones). Normalmente, los métodos operan sobre el estado interno del objeto, que es privado–variables de instancia.
Un conjunto de objetos similares constituye una *clase* (a veces se conoce como *type*). Una clase actúa como template que describe el comportamiento de esos objetos similares. Técnicamente hablando, un objeto es una instancia de una clase. Cada objeto es identificado por un ID único (también conocido como *referencia*).
Los objetos tienen tres propiedades mágicas que los hacen increíblemente útiles para trabajar: *encapsulamiento*, *herencia* y *polimorfismo*.

Encapsulamiento

> "Uno de los principios fundamentales de la tecnología de objetos es que el estado interno de un objeto es privado a ese objeto y no puede ser accedido, ni siquiera consultado desde el exterior."
>
> - David Taylor, Autor de
> Business Engineering with Object Technology
> (Wiley, 1995)

Encapsulamiento significa "no me digas cómo, simplemente hazlo". Los objetos logran esto administrando sus propios recursos y limitando la visibilidad externa. En este sentido, los objetos son átomos autocontenidos. Un objeto publica una interface que define cómo otros objetos y aplicaciones pueden interactuar con él. La implementación del objeto está *encapsulada*–es decir oculta de la visión externa. La interface pública es un contrato común establecido entre los desarrolladores de una clase y sus potenciales clientes.

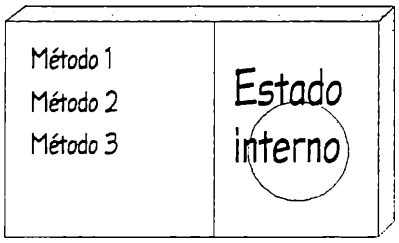


Figura 1-3. El primer pilar de OO: Encapsulamiento

Polimorfismo

"Un verdadero objeto siempre puede ser reemplazado."

- Christine Comaford, Columnista de PC Week

Polimorfismo es una manera de decir que el mismo método puede hacer diferentes cosas, dependiendo de la clase que lo implemente. Así, los objetos de diferentes clases reciben el mismo mensaje y reaccionan de manera diferente. El objeto que envía el mensaje no puede establecer la diferencia, es el receptor quien suministra el comportamiento apropiado. El polimorfismo permite entonces ver como similares a dos objetos que tienen la misma *interface*. El *overloading* es una variante del polimorfismo que permite definir diferentes versiones de un método, cada una con diferentes tipos de parámetros.
La figura muestra dos clases distintas que implementan el mismo método, *acelerar*

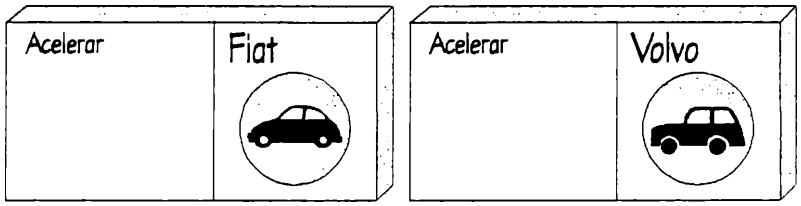


Figura 1-4. El segundo pilar de OO: Polimorfismo

Herencia

"Las clases hijas pueden mejorar el comportamiento de sus clases padres. Y si el comportamiento de las clases padres es mejorado, es para el beneficio de las hijas."

Steve Jobs
NeXT, Apple
(Febrero de 1995)

La *herencia* es un mecanismo que permite crear clases hijas–subclases–a partir de clases existentes. Las clases hijas heredan de sus padres los métodos y estructuras de datos. Es posibles agregar nuevos métodos o sobreescribir–modificar–los métodos heredados para definir nuevo comportamiento. Usamos la herencia para extender los objetos.
Algunos modelos de objetos soportan *herencia simple*–cada clase tiene exáctamente una clase padre. Otros modelos soportan *herencia múltiple*–una clase puede tener más de una clase padre.
Las *clases abstractas* son aquellas cuyo principal propósito es ser heredadas por otras. Los *mixins* son clases cuyo principal objetivo es ser heredadas en forma múltiple.

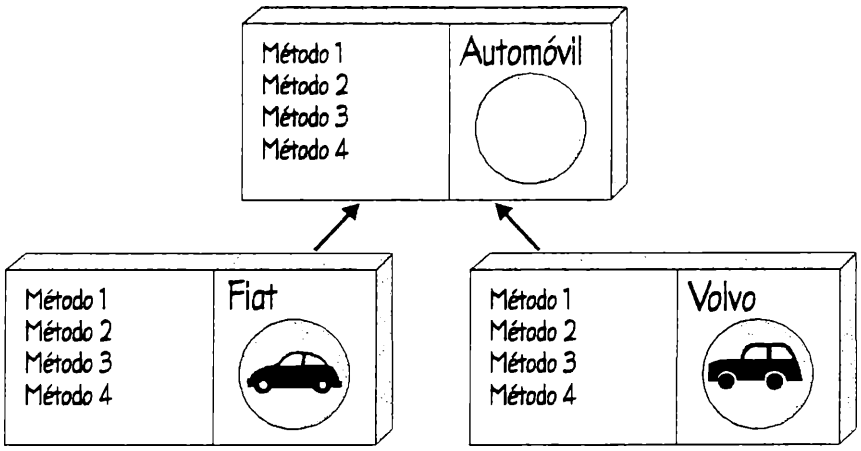


Figura 1-5. El tercer pilar de OO: Herencia

Frameworks y bibliotecas de clases

Los tres pilares de la programación orientada a objetos permiten crear, ensamblar y reusar objetos. La primera generación de orientación a objetos se basó en el uso de *bibliotecas de clases* para empaquetar objetos reusables. Un enfoque más evolucionado está constituido por los *framworks*. Son bibliotecas preensambladas de clases que se empaquetan para proveer una funcionalidad específica. Los frameworks facilitan el ensamblado de objetos; levantan el nivel de abstracción.

Binding de objetos

El *binding* se refiere al enlace de la interface de software entre dos objetos dados. Si el binding es *estático*, ambos–cliente y servidor–tienen una interface que se determina en el momento de la compilación.
El binding *dinámico*–llamado también *late binding*–ocurre cuando se envía el mensaje. Requiere que los objetos tengan una manera de determinar la interface al momento del envío. Esto permite crear sistemas más flexibles, pues el código que se ejecuta se determina en tiempo de ejecución. Sirve para reemplazar código dinámicamente e introducir nueva funcionalidad sin recompilar el software existente.

Relaciones entre objetos

Hay tres maneras de relacionar los objetos: 1) *especialización*, en la cual las clases de definen como casos especiales de otras clases; 2) *colaboración*, los objetos se envían mensajes para pedirse servicios; y 3) *composición*, los objetos se construyen a partir de otros objetos.
Un objeto que se compone de otros objetos se denomina *compuesto*. Un objeto compuesto mantiene referencias a los objetos que contiene, en lugar de contener al objeto mismo. Luego, el mismo objeto puede estar contenido en varios objetos compuestos.

Hacia un estándar de componentes

Los componentes son objetos independientes con capacidad de plug & play para redes, aplicaciones, lenguajes, herramientas de desarrollo y sistemas operativos. Por definición, los objetos distribuidos son componentes, por la manera en que están empaquetados. En los sistemas de objetos distribuidos la unidad de trabajo son los componentes. Sin embargo no todos los componentes son objetos. Y menos distribuidos. Por ejemplo, un OCX (OLE Custom Control) es un componente, pero no es ni un objeto ni es distribuido.
La tecnología de componentes–en todas sus formas–promete alterar radicalmente la forma de desarrollar software. Por ejemplo, los objetos distribuidos permiten integrar sistemas de información cliente/servidor simplemente ensamblando y extendiendo componentes. El objetivo de los componentes es ofrecer a los usuarios finales y desarrolladores los mismos niveles de plug & play que actualmente están disponibles para los consumidores y fabricantes de circuitos y partes electrónicas (hardware).
Sin embargo, los componentes por si mismos no ofrecen una infraestructura para integrar en el mismo espacio software creado por distintos fabricantes–menos aún a través de redes y sistemas operativos. La solución es aumentar las capacidades de los objetos clásicos con una *infraestructura estándar de componentes.*
OpenDoc y OLE son actualmente los estándares líderes para integración de componentes en aplicaciones de escritorio; CORBA ofrece un estándar para integrar componentes en el siguiente nivel y a gran escala.
CORBA y OpenDoc se complementan mutuamente y permitirán reemplazar las aplicaciones monolíticas de hoy con paquetes de componentes (suites).
Ventajas de la combinación de estas tecnologías:

- ***Los usuarios finales*** podrán ensamblar sus propias aplicaciones personales usando componentes ya desarrollados. Usarán scripts para unir las partes y ajustar el comportamiento.
- ***Los pequeños desarrolladores*** notarán que el uso de componentes reduce los costos y baja las barreras para entrar al mercado de software. Podrán desarrollar sus propios componentes a partir de otros ya disponibles.
- ***Los desarrolladores de gran envergadura e integradores de sistemas*** usarán los paquetes de componentes para crear o ensamblar aplicaciones cliente/servidor de gran escala. Tales sistemas serán mucho más fáciles de probar por la alta confiabilidad de los componentes pretesteados. El hecho de que muchos de los componentes integrados sean de caja negra reduce la complejidad del proceso de desarrollo. Los componentes–especialmente en la variedad CORBA–serán diseñados para trabajar juntos en redes con aplicaciones cliente/servidor.
- ***Los fabricantes de aplicaciones*** de escritorio usarán los componentes para ensamblar aplicaciones dirigidas a segmentos específicos del mercado (por ejemplo "WordPerfect for Legal Firms"). En lugar de vender aplicaciones gigantes–que integran toda la funcionalidad posible–ofrecerán a los consumidores lo que realmente necesitan..

En resumen, los componentes reducen la complejidad de las aplicaciones, costos de desarrollo y tiempo de venta. También mejoran la reusabilidad del software, mantenimiento, independencia de la plataforma y distribución cliente/servidor. Finalmente, los componentes ofrecen mayor flexibilidad y libertad para elegir .

¿Qué es exactamente un componente?

"Si los componentes vienen con una mala reputación, nadie los usará. Así que deberán ser una calidad extraordinaria. Necesitan estar testeados, ser eficientes y estar bien documentados... Los componentes deberían invitar al reuso."

- Ivar Jacobson, Autor
Object-Oriented Software Engineering
(Addison-Wesley, 1993)

Dijimos que los componentes interoperan usando modelos de interacción cliente/servidor neutrales (independientes del lenguaje). A diferencia de los objetos tradicionales, los componentes pueden interoperar a través de los lenguajes, herramientas, sistemas operativos y redes. Pero los componentes son parecidos a los objetos en el sentido de que soportan herencia, encapsulamiento y polimorfismo. Como los componentes significan realmente cosas diferentes para distintas personas, definiremos las funciones mínimas que deberían ofrecer. Nuestra noción de componente se refiere a lo que CORBA, OpenDoc y OLE ofrecen. Así, un componente debe contar con las siguientes propiedades:

- ***Es una entidad comercializable.*** Un componente es una pieza de software autocontenida disponible en el mercado.
- ***No es una aplicación completa.*** Los componentes deben combinarse con otros componentes para formar aplicaciones. Se desarrollan para realizar un conjunto limitado de tareas. Los objetos pueden ser básicos–objetos C++, Smalltalk; objetos de tamaño medio–por ejemplo un control de interface, GUI; o bien objetos altamente complejos–por ejemplo un applet.
- ***Se pueden usar de maneras no previstas.*** Como ocurre con los objetos del mundo real, los componentes se pueden usar de maneras que no fueron previstas originalmente por su desarrollador.
- ***Tienen una interface bien definida.*** Como los objetos tradicionales, los componentes pueden manipularse sólo a través de su interface. Así es como los componentes exponen su funcionalidad al mundo externo. Es importante notar que la interface del componente debe estar separada de la implementación. El mismo componente puede estar implementado usando objetos, código procedural o bien encapsulando e integrando código ya existente.
- ***Son objetos interoperables.*** Un componente puede ser invocado como un objeto a través de diferentes espacios de direcciones, redes o sistemas operativos. Es una entidad independiente del sistema subyacente.

- ***Son objetos extensibles.*** Como los objetos, soportan herencia, encapsulamiento y polimorfismo.

¿Qué es un supercomponente?

Los *supercomponentes* son componentes con habilidades extras, más inteligencia. Estas habilidades son necesarias para crear objetos autónomos, desacoplados, capaces de navegar entre máquinas y redes. Estos componentes necesitan ofrecer el tipo de facilidades que se asocian a entidades conectadas en red e indenpendientes, incluyendo:

- ***Seguridad.*** Un componente debe protegerse a si mismo y a sus recursos de las amenazas externas. Debe ser capaz de indentificarse ante un cliente y viceversa. Debe ofrecer control de acceso. Debe contar con mecanismo de auditoría de su uso.
- ***Licencias.*** Un componente de estar preparado para forzar políticas de licenciamiento y medición de su uso por parte de los clientes. Es importante proteger a los fabricantes de componentes del uso de los componentes que venden.
- ***Versiones.*** Un componente debe ofrecer alguna forma de control de versiones; debe garantizar a los clientes que están usando la versión correcta.
- ***Manejo del ciclo de vida.*** Un componente debe ser capaz de manejar su creación, destrucción y almacenamiento. Debe ser capaz de clonarse, exportar su estado interno y moverse de un lugar a otro.
- ***Soporte para herramientas y paletas de integración.*** Un componente debe de permitir ser importado o integrado en una herramienta de desarrollo de aplicaciones. Un ejemplo son las paletas que soportan la integración de componentes OLE y partes OpenDoc.
- ***Notificación de eventos.*** Un componente debe ser capaz de notificar a las partes interesadas (clientes) cuando algo de interés le ocurra.
- ***Configuración y manejo de propiedades.*** Un componente debe ofrecer una interface que permita a los usuarios configurar sus propiedades.
- ***Scripting.*** Un componente debe permitir que su interface sea controlada desde lenguajes de scripting. Esto significa que la interface debe ser capaz de autodescribirse y soportar dinamic binding.
- ***Metainformación e instropección.*** Un componente debe proveer, bajo demanda, información sobre si mismo. Esto incluye una descripción de sus interfaces, atributos y suites que soporta.
- ***Control de transacciones y bloqueo.*** Un componente debe proteger sus recursos con esquemas de transacciones, esto es, cooperar con otros componentes para ofrecer integridad de *todo o nada*. Además debe ofrecer mecanismos de bloqueo para serializar el acceso a recursos compartidos.
- ***Persistencia.*** Un componente debe ser capaz de salvar su estado en un medio de almacenamiento no volátil y luego recuperarlo.
- ***Relaciones.*** Un componente debe ser capaz de formar asociaciones dinámicas y/o permanentes con otros componentes. Ejemplo: un componente puede contener a otros componentes.

- ***Facilidad en el uso.*** Un componente debe presentar un número limitado de operaciones para invitar al uso (y reuso). En otras palabras, el nivel de abstracción debe ser tan alto como sea posible.
- ***Autotesteo.*** Un componente debe ser capaz de autoverificar su estado y configuración. Debe ser posible correr mecanismos de diagnóstico provistos por el componente para determinar problemas.
- ***Mensajes semánticos.*** Un componente debe ser capaz de entender el vocabulario de las suites y extensiones específicas del dominio que soporta.
- ***Autoinstalable.*** Un componente debe estar preparado para instalarse a si mismo en forma automática y registrarse en el sistema operativo o registro de componentes (registry). Luego, cada componente debe ser capaz de desinstalarse si fuera necesario.

Business objects

Para llegar al punto anterior de integración y maduración de los componentes es necesario contar con estándares que definan las reglas para la interacción. Juntas, todas estas reglas definen los que denominamos *infraestructura de componentes distribuidos*.

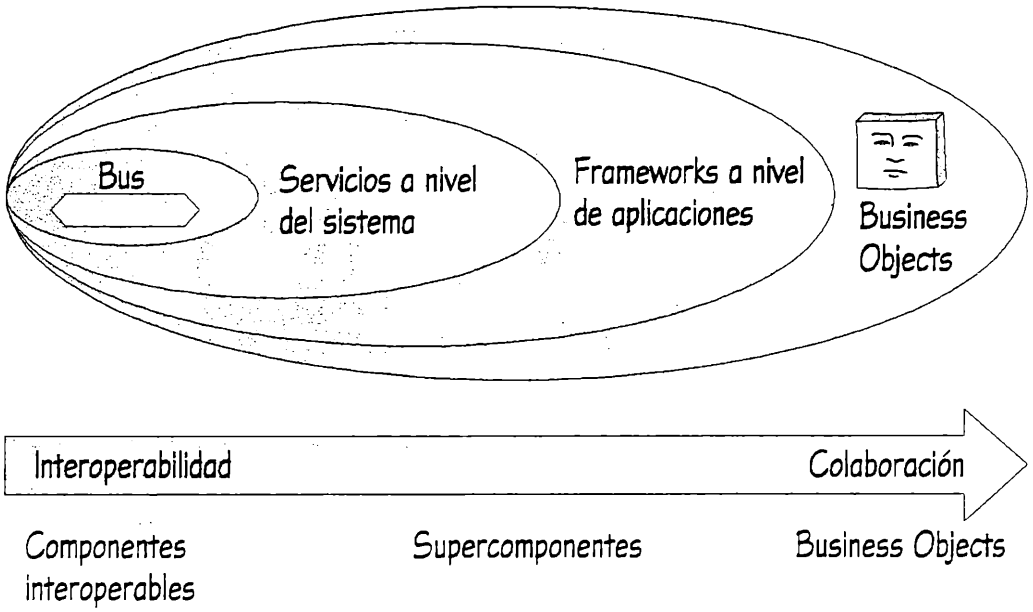


Figura 1-6. Evolución hacia los componentes

- En el nivel más básico, la infraestructura necesaria se reduce al bus de objetos (el ORB, Object Request Broker) que permite que los componentes interoperen sobre espacios de objetos, sistemas operativos, redes, etc. Este bus también provee mecanismos para que los objetos intercambien *metainformación* y se descubran mutuamente.

- En el siguiente nivel agregamos *servicios a nivel de sistema* que permiten crear *supercomponentes*. Estos servicios incluyen: administración de licencias, seguridad, control de versiones, persistencia, semantic messaging, scripting (cada objeto debe poder manejarse con algún lenguaje de script), transacciones, etc.
- El siguiente nivel corresponde a los *frameworks a nivel de aplicaciones*. Esta es la infraestructura necesaria al nivel de las aplicaciones. Estos frameworks fortalecen las reglas de ensamblado entre distintos componentes para que formen *suites*.
- Finalmente llegamos al nivel más alto. Allí están los componentes que modelan entidades del mundo real en algunas aplicaciones de dominio específico. Estos componentes realizan funciones específicas en sus dominios–por ejemplo, un cliente, un auto, un hotel son ejemplos de *business objects*. Los agentes también son un buen ejemplo de business objects.

CORBA: LA LLEGADA DE UN ESTANDAR

"Para ensamblar objetos, se requiere que éstos sean compatibles unos con otros. Rara vez esto es un problema cuando se escribe un programa, porque todos los objetos se desarrollan en el mismo lenguaje, corren en la misma máquina y usan el mismo sistema operativo. Pero el desarrollo de sistemas completos a partir de objetos, es una cuestión completamente diferente. Los objetos tienen que interactuar entre si aún cuando hayan sido desarrollados en lenguajes diferentes y corran sobre plataformas de hardware y software distintas."

- David Taylor, Autor de
Business Engineering with Object Technology
(Wiley, 1995)

"Los estándares son más importantes para los objetos distribuidos que para cualquier otra tecnología en cualquier otra industria. Los objetos desarrollados por una compañía deben ser capaces de comunicarse y cooperar con objetos desarrollados por otras compañías."

- Roger Sessions, Autor
Object Persistence
(Prentice Hall, 1996)

CORBA (Common Object Request Broker Architecture), es la respuesta del OMG (Object Management Group) a la necesidad de interoperabilidad entre los productos de hardware y software disponibles en la actualidad. Dicho de manera simple, CORBA permite que las aplicaciones se comuniquen unas con otras sin importar la localización ni quién las haya diseñado.
CORBA 1.1 se introdujo en 1991 y definió el *Lenguaje de Definición de Interfaces* (IDL) y las APIs que para la interacción cliente/servidor dentro de una implementación específica de un ORB (Object Request Broker). CORBA 2.0, adoptado en diciembre de 1994, define interoperabilidad verdadera especificando cómo los ORBs de diferentes fabricantes deben interoperar.

El ORB es el middleware que establece las relaciones cliente/servidor entre los objetos. Usando un ORB, un cliente puede invocar de manera transparente un método en un objeto servidor que puede estar en la misma máquina o ser accesible a través de una red. El ORB intercepta la llamada y es responsable de encontrar el objeto que implementa el servicio pedido, le pasa los parámetros, invoca el método y retorna los resultados.
El cliente no tiene noción de dónde está localizado el servidor, su lenguaje de programación, su sistema operativo o cualquier otro aspecto que no forme parte de algún aspecto de la interface del objeto. Con esto, el ORB ofrece interoperabilidad entre aplicaciones en

máquinas distintas, sobre entornos heterogeneos distribuidos. Permite interconectar múltiples sistemas con objetos de manera transparente.

En la construcción de aplicaciones cliente/servidor típicas, los desarrolladores usan su propio diseño o un estándar reconocido para definir el protocolo que se usará entre los dispositivos de comunicación. La definición del protocolo depende del lenguaje de implementación, características de la red de transporte y una docena de otros factores.
Los ORBs simplifican este proceso. Con un ORB, el protocolo se define a través de las interfaces de las aplicaciones, mediante una especificación en un lenguaje independiente de la implementación, el IDL. Y los ORBs ofrecen flexibilidad. Permiten que los programadores elijan el sistema operativo más apropiado, el entorno de ejecución e encluso el lenguaje de programación de cada componente del sistema. Más importante aún, permiten la integración de componentes ya existentes. En una solución basada en ORBs, los desarrolladores simplemente modelan los componentes usando el mismo IDL que usan para crear componentes nuevos; escriben un *wrapper* que actúa de intermediario adaptando las interfaces del componente al bus estandarizado.

Tal vez, el secreto del éxito del OMG es que crea especificaciones de interfaces, no de código. Las interfaces que especifica siempre se derivan de tecnología ya desarrollada por sus miembros. Las especificaciones son escritas en un lenguaje neutral, IDL, que define los límites de los componentes, es decir, las interfaces contractuales con potenciales clientes. Los componentes escritos en IDL deberían ser portables entre diferentes lenguajes, herramientas, sistemas operativos y redes.

¿Qué es un objeto distribuido CORBA?

Un objeto CORBA es un objeto que obedece a ciertas reglas y es accesible a través de un protocolo predeterminado. Un objeto distribuido no es necesariamente un objeto CORBA. Por ejemplo: un objeto distribuido puede ser un objeto implementado en C++ que se accede a través de un socket o por RPC. Para que los objetos sean CORBA deben tener declarada una interface IDL.

Los objetos CORBA son masas inteligentes que pueden vivir en cualquier lugar de una red. Están empaquetados como componentes binarios a los cuales los clientes pueden acceder remotamente con invocaciones de métodos. Los lenguajes y compiladores utilizados para crear los objetos servidores son completamente transparentes para los clientes. Los clientes no necesitan saber dónde residen los objetos distribuidos o sobre qué sistema operativo están ejecutando. Pueden estar en el mismo proceso o en una máquina remota accesible a través de una red intergaláctica. Además, los clientes no necesitan saber nada sobre la implementación de los servidores. Por ejemplo, un objeto servidor podría estar implementado con clases C++ o con miles de lineas de COBOL–los clientes no podrán establecer la diferencia. Lo único que los clientes necesitan conocer es la interface pública de los servidores. Esta interface actúa como contrato que vincula a los clientes con los servidores.

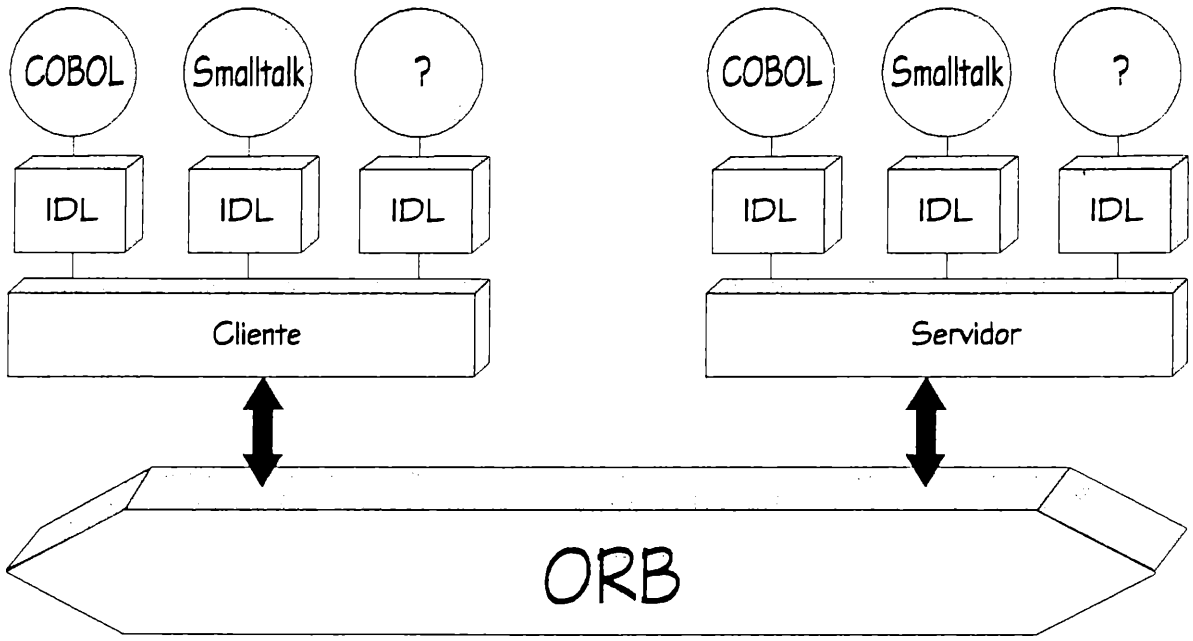


Figura 1-7. Bindings IDL hacia lenguajes para ofrecer interoperabilidad cliente/servidor

Todo en IDL

"El IDL del OMG es la mejor notación estándar disponible para definir los límites y alcances de los componentes. Ofrece una notación universal para especificar APIs. IDL soporta interfaces de bibliotecas de funciones de la misma manera que objetos distribuidos en una red."

- Tom Mowbray et al., Autor
The Essential CORBA
(Wiley, 1995)

Como dijimos antes, el OMG usa contratos en IDL (Interface Definition Languaje) para especificar los alcances de los componentes y las interfaces contractuales con los potenciales clientes. El IDL es puramente declarativo. Esto significa que no especifica detalles sobre la implementación de los componentes. Luego, los métodos definidos en IDL pueden implementarse en C++, Smalltalk, ObjectiveC, Ada, etc. El IDL ofrece independencia del lenguaje de programación y del sistema operativo para todos los componentes y servicios disponibles a través del bus de CORBA. Podemos usar IDL para especificar los atributos de los componentes, las clases de las que hereda, las excepciones que dispara, los tipos de

eventos que activa y los métodos que soportan las interfaces–incluyendo el tipo de los parámetros de entrada y salida.
La ambición de CORBA es *IDL-izar* todo el middleware disponible actualmente para las aplicaciones cliente/servidor y todos los componentes que viven en un ORB.

Componentes CORBA: de los objetos al nivel de sistema a los business objects.

> "Los objetos pueden variar tremendamente en número y tamaño. Pueden representar todo: desde hardware hasta aplicaciones de diseño completas. ¿Cómo podemos decidir qué cosas deberían ser objetos?"
>
> - Erich Gamma et al., autores de
> Design Patterns
> (Addison Wesley, 1994)

Hemos estado utilizando sin distinciones los términos componentes y objetos distribuidos. Los objetos distribuidos CORBA son, por definición, componentes por la manera en que son empaquetados. En los sistemas de objetos distribuidos, la unidad de trabajo y distribución es el componente. La infraestructura de objetos distribuidos de CORBA facilita que los componentes sean autónomos, se autoadministren y colaboren.
El objetivo final es lograr la colaboración en el nivel semántico de las aplicaciones. El truco es lograr la colaboración entre componentes que no se conocen previamente (no se desarrollaron en forma conjunta). Para llegar a eso ya dijimos que se requieren estándares de definan los alcances de la colaboración.

Arquitectura para la administración de objetos

A fines de 1990, el OMG publicó una guía con la primera definición de OMA (Object Management Architecture). Esa definición de sometió a una revisión en 1992.
Los cuatro elementos principales de OMA son: 1) el ORB (Object Request Broker) que define el bus de obejtos de CORBA; 2) Los servicios a nivel de objetos (Common Object Services) definen frameworks a nivel del sistema para extender el bus; 3) Facilidades comunes (Common Facilities) definen frameworks horizontales y verticales a nivel de aplicaciones que son usandos directamente por los business objects; y 4) Objetos para desarrollar aplicaciones (Application Objects) son los business objects y las aplicaciones–son los consumidores finales de la infraestructra CORBA. Veremos una presentación global de los cuatro elementos:

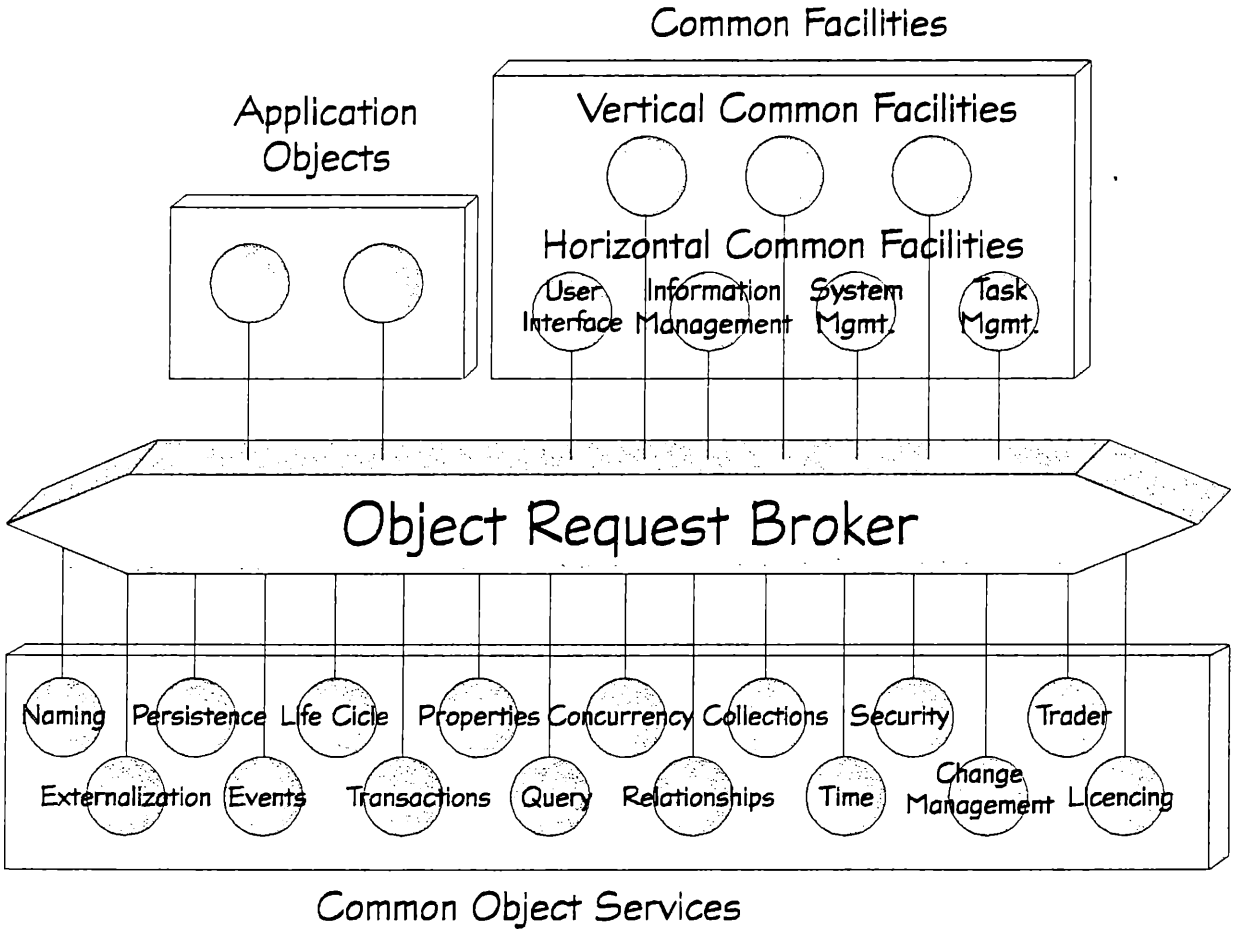


Figura 1-8. Arquitectura de CORBA, OMA

1) El ORB (Object Request Broker)

El ORB (Object Request Broker) es el *bus de objetos*. Permite que los objetos hagan pedidos de servicios en forma transparente–y reciban una respuesta–a otros objetos locales o remotos. El cliente no tiene noción de los mecanismos usados para la comunicación, activación y almacenamiento de los objetos que actúan como servidores.

Un ORB CORBA ofrece un conjunto muy rico de servicios adicionales. Permite que los objetos se descubran y conozcan mutuamente en tiempo de ejecución. Un ORB es mucho más sofisticado que otras formas alternativas de middleware para aplicaciones cliente/servidor–incluyendo RPC (Remote Procedure Call), MOM (Message-Oriented Middleware, stored procedures para bases de datos, etc. En teoría, CORBA es el mejor middleware alguna vez definido. En la práctica, CORBA resulta tan bueno como los productos que lo implementan.
Beneficios que ofrece el ORB CORBA:

- ***Invocación dinámica y estática de métodos.*** Un ORB CORBA permite definir las invocaciones de métodos estáticamente en compilación o bien determinarlas durante la ejecución, dinámicamente. Esto nos permite obtener chequeo fuerte de tipos en compilación o la máxima flexibilidad asociada al binding dinámico.
- ***Enlaces con lenguajes de alto nivel.*** Un ORB CORBA permite invocar métodos en objetos usando cualquier lenguaje de programción de alto nivel–Smalltalk, C++, ObjectiveC, etc. No importa realmente en qué lenguaje se escribió el código de los objetos. CORBA separa la especificación de la interface de la implementación y ofrece tipos neutrales de datos que hacen posible el envío de mensajes a otros objetos a través de lenguajes y sistemas operativos.
- ***Sistemas autodescriptivos.*** CORBA ofrece metainformación disponible durante la ejecución para describir la interface pública de cualquier objeto conocido en el sistema. Todos los ORBs de CORBA deben soportar un Interface Repository que registre toda la información necesaria para describir la interface de un objeto, incluyendo sus métodos y parámetros. Los clientes usan la metainformación para determinar cómo deben invocar los métodos. También se permite tener herramientas de generación de código *al vuelo*, (on the fly). La metainformación se genera con precompilares de IDL o a partir de código escrito en lenguajes de programación (se requieren compiladores especiales).
- ***Transparencia de localización.*** Un ORB CORBA puede estar funcionando sólo en una laptop o bien interconectado con otros ORBs del universo (usando los servicios de interconexión de CORBA 2.0). Un ORB puede atender las llamadas entre objetos de un mismo proceso, de varios procesos corriendo en la misma máquina o varios procesos corriendo en una red. Todo ocurre de manera transparente para los objetos.
- ***Incluye seguridad y manejo de transacciones.*** Un ORB incluye información de contexto en sus mensajes para el manejo de la seguridad y transacciones.
- ***Mensajes polimórficos.*** En contraste a otras formas de middleware, un ORB no invoca simplemente funciones remotas, más bien invoca funciones en un objeto remoto. Esto significa que la misma función puede producir efectos distintos dependiendo del objeto sobre el que se invoca. Por ejemplo un mensaje *configure_yourself* se comporta de manera diferente cuando se aplica a un objeto base de datos y a un objeto impresora.

2) Servicios disponibles para los objetos

Estos servicios son conjuntos de componentes que ofrecen servicios a nivel de sistema, especificados con una interface IDL. Pueden verse como funcionalidad extendida para el ORB. Se usan para crear, nombrar e introducir componentes al entorno del ORB. Actualemente tenemos disponibles once servicios:

- El ***Servicio del Ciclo de Vida*** (Life Cycle Service) define operaciones para crear, copiar, mover y eliminar objetos del bus.
- El ***Servicio de Persistencia*** (Persistence Service) suministra una interface común para almacenar objetos en forma persistente en diferentes medios–incluyendo bases de datos orientadas a objetos (ODBMSs), sistemas de bases de datos relacionales (RDBMSs) y archivos.

- El ***Servicio de Nombres*** (Naming Service) permite que cualquier componente del bus encuentre a otros componentes usando un nombre. Utiliza contextos de nombres, directorios de redes ya existentes, etc.
- El ***Servicio de Eventos*** (Event Service) permite que los componentes del bus registren o quiten su interés en ciertos eventos disponibles. Este servicio define un objeto denominado *canal de eventos* (event channel) que se encarga de recolectar y distribuir los eventos entre los componentes suscriptos.
- El ***Servicio de Control de Concurrencia*** (Concurrency Control Service) suministra un administrador de bloqueos (locks) para las transacciones.
- El ***Servicio de Transacciones*** (Transaction Service) ofrece un esquema de coordinación de dos fases para componentes involucrados en transacciones simples o anidadas.
- El ***Servicio de Relaciones*** (Relationship Service) suministra un mecanismo para crear asocianes dinámicas (links) entre objetos que no se conocen mutuamente. También cuenta con mecanismos para navegar por estas redes de relaciones entre componentes. Estas relaciones son útiles para forzar restricciones de integridad entre componentes, definir relaciones de contención ("contiene a") y enlazar componentes por cualquier criterio.
- El ***Servicio de Exportación*** (Externalization Service) ofrece una manera estándar de poner y sacar información de los componentes con mecanismos basados en streams.
- El ***Servicio de Consultas*** (Query Serice) sirve para formular consultas sobre los objetos. Se basa en la especificación del OQL (Object Query Languaje) definido por el ODMG (Object Database Managemente Group). OQL es un estándar para el manejo de bases de datos orientadas a objetos. ODMG es un subgrupo del OMG.
- El ***Servicio de Licencias*** (Licencing Service) permite medir y controlar el uso de los componentes. Fue creado para la explotación comercial de los componentes de software. Permite el cobro por el uso de los componentes en todos los niveles y momentos: por sesión de conexión, por nodo, por creación de instacias, etc.
- El ***Servicio de Propiedades*** (Property Service) suministra operaciones para asociar atributos (propiedades) a cualquier componente. Con este servicio es posibles agregar nuevas propiedades dinámicamente a cualquier componente. Por ejemplo: un título o una fecha.

3) Common Facilities

Son un conjunto de componentes con interface IDL que proveen servicios para los objetos de las aplicaciones. Pueden verse como el siguiente nivel en la jerarquía semántica que hemos estado definiendo. Hay dos categorías: *facilidades horizontales* y *facilidades verticales*.

Las *facilidades horizontales* pueden ser de cuatro tipos: 1) *Servicios de interface del usuario para la edición* (in place editing) similares a los ofrecidos por OpenDoc y OLE; 2) *Servicios para el manejo de información*–incluye almacenamiento de documentos compuestos; 3) *Servicios para el manejos del sistema*: interfaces para administrar, configurar, reparar y operar componentes; y 4) *Servicios para la administración de tareas*–transacciones de larga duración, agentes, scripting, email.

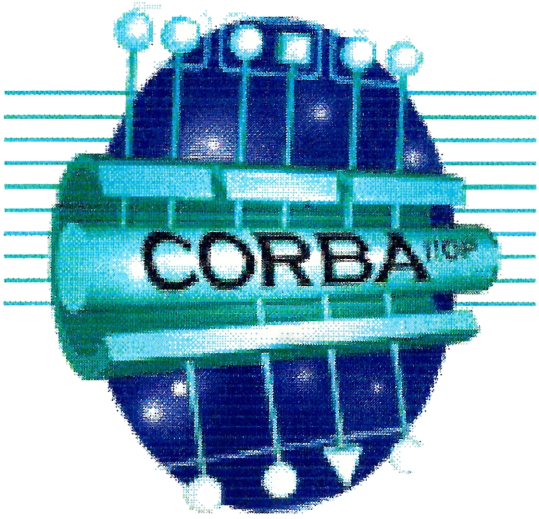
Las *interfaces verticales* suministran interfaces IDL para segmentos verticales del mercado, como por ejemplo: salud, finanzas, telecomunicaciones, etc.

4) Objetos para las aplicaciones: business objects

Los objetos a nivel de aplicación son específicos a las aplicaciones del usuario. Estos objetos deben definirse usando IDL si queremos que participen en intercambios de mensajes con el ORB. Estos objetos se arman y basan en los servicios disponibles en el ORB y las facilidades comunes.

Parte 2

El Bus de Objetos Distribuidos



EL BUS INTERGALACTICO

"Los ORBs facilitan el reuso de clases suministrando un capa que funciona por debajo del nivel de las aplicaciones pero está por encima de los lenguajes de programación, sistemas operativos y hardware. En este piso intermedio, crece el reuso y ventajas de la tecnología de objetos."

- John Gidman, Fidelity Investments

Un ORB CORBA ofrece mucho más que los mecanismos necesarios para enviar mensajes que permiten a los objetos se comunicarse entre si a través de lenguajes heterogeneos, plataformas y redes. También provee un entorno para manejar esos objetos, hacer pública su presencia y describirlos con metainformación. Un ORB CORBA es un bus autodescriptivo de objetos.

¿Qué es exactamente un ORB CORBA?

Un ORB actúa de intermediario entre las aplicaciones clientes que necesitan servicios y las aplicaciones servidoras capaces de ofrecerlos.

- Richard Adler, Coordinated Computing
(Abril de 1995)

Un ORB CORBA define el middleware que establece las relaciones cliente/servidor entre los objetos. Usando un ORB, un objeto cliente puede invocar de manera transparente un método en un objeto servidor, que puede estar disponible en la misma máquina ser accesible a través de una red. El ORB intercepta la llamada y es resposable de encontrar al objeto que implementa el método, pasarle los parámetros, hacer la invocación y devolver el resultado.
El cliente no necesita conocer la localización física del servidor, el lenguaje que se usó para implementarlo, el sistema operativo que usa o cualquier otro aspecto que quede afuera de la interface pública del objeto.
Es muy importante notar que los roles de cliente y servidor se usan solamente para coordinar la interacción entre los objetos. Los objetos de un ORB pueden actuar como clientes o como servidores, dependiendo de la ocasión.

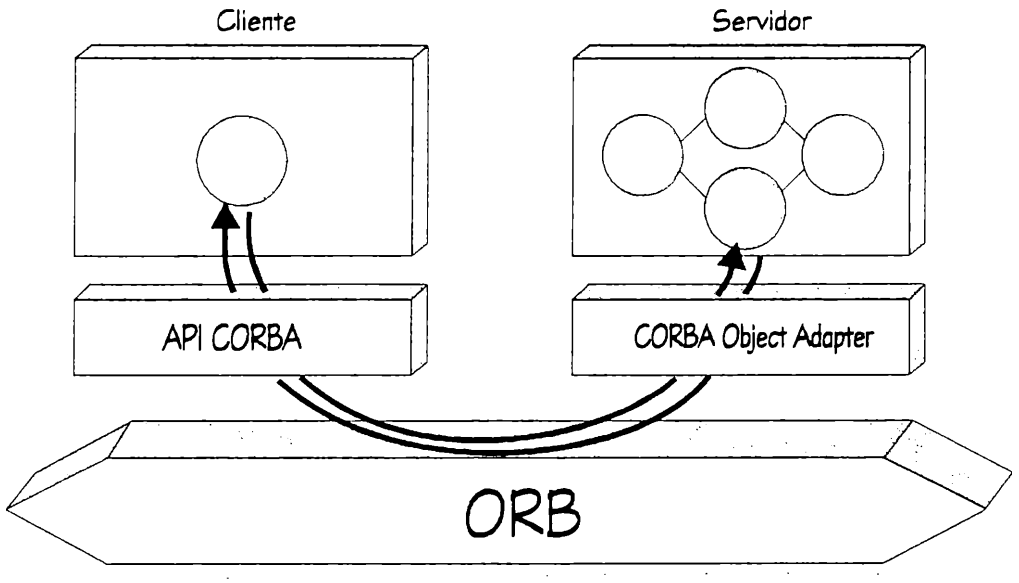


Figura 2-1. Cliente/Servidor usando un ORB

Anatomía de un ORB CORBA

En la siguiente figura mostramos el lado del cliente y el lado del servidor de un ORB CORBA. ¡Aunque hay muchas cajitas y flechas, no es tan complicado como parece!
Lo fundamental es entender que CORBA, como SQL, ofrece interfaces dinámicas y estáticas para los servicios. Comencemos analizando el lado del cliente.

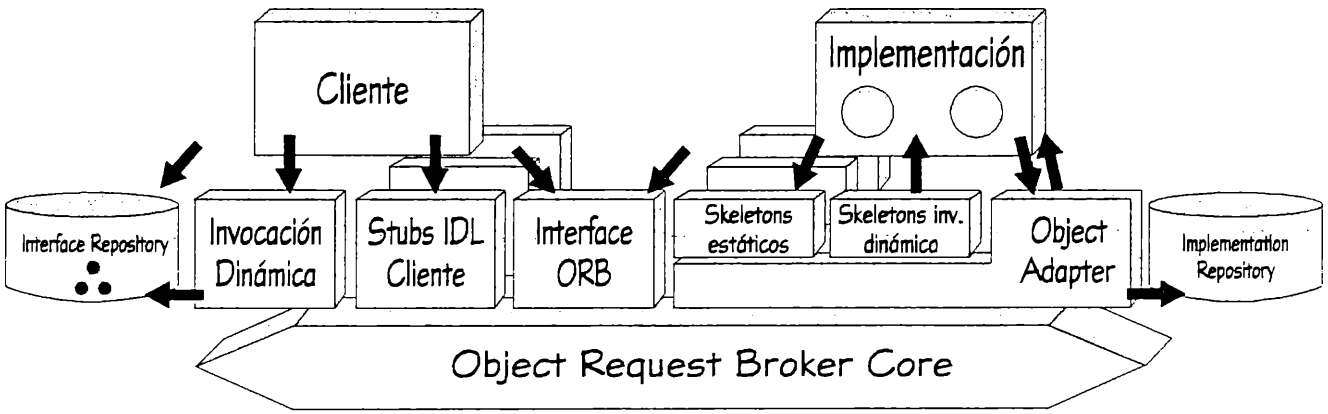


Figura 2-2. Estructura del ORB de CORBA

- Los ***stubs IDL del cliente*** proveen las interfaces estáticas para los servicios. Estos stubs precompilados definen cómo los clientes invocan los correspondientes servicios en los servidores. Desde el punto de vista de un cliente, los stubs se comportan como una llamada local–son *proxies* locales a objetos remotos. Estos servicios se definen con IDL

y los stubs se generan con un compilador IDL. Cada cliente debe tener definido un stub para cada interface que use del servidor. Los stubs incluyen código para las operaciones de *marshaling*. Esto significa que codifican (y decodifican) la operación y sus parámetros en mensajes de formato *plano* que se pueden enviar a un objeto servidor. También incluyen archivos de encabezado para permitir las invocaciones desde lenguajes de alto nivel (C, C++, Smalltalk) sin que el programador debe preocuparse por los protocolos de transmisión, formato de los datos, etc. Es tan simple como invocar un mensaje desde un programa para obtener un servicio remoto.

- Las ***Interfaces de Invocación Dinámica*** (DII, Dynamic Invocation Interfaces), permiten descubrir dinámicamente–en ejecución–los métodos que se invocarán. CORBA define una API estándar para buscar la metainformación que define la interface del objeto servidor, generar los parámetros, invocar el método y obtener los resultados.
- El ***Depósito de Interfaces*** (Interface Repository) permite obtener y modificar la descripción de todas las interfaces de los componentes registrados, los métodos que definen y los parámetros que requieren. El Interface Repository es una base de datos que contiene versiones legibles por el sistema de todas las interfaces definidas con IDL. Puede verse como un depósito dinámico de metainformación para los ORBs, que puede ser consultado, accedido y modificado por los componentes. El uso de la metainformación permite que cada componente sea capaz de autodescribirse con sus interfaces. El ORB mismo es un bus autodescriptivo.
- La ***Interface del ORB*** está formada por unas pocas APIs a servicios locales que podrían ser útiles para las aplicaciones. Por ejemplo, tenemos una API para codificar una referencia remota en un string y viceversa. Esto es muy útil si se deben almacenar las referencias a objetos remotos para luego restaurarlas.

Del lado del servidor no es posible establecer las diferencias entre una invocación dinámica y otra estática; semánticamente ambas son similares. En ambos casos, el ORB encuentra un *adaptador* (object adapter), transmite los parámetros y transfiere el control a la implementación a través de los stubs (o skeletons) IDL del servidor. Veamos las partes involucradas del lado del servidor.

- Los ***stubs IDL del servidor*** (el OMG los llama skeletons) suministran interfaces estáticas para cada servicio que ofrece el servidor. Estos stubs, como los definidos en los clientes, se crean usando un compilador IDL.
- La ***Interface Dinámica de Skeletons*** (Dynamic Skeleton Interface, DSI)–introducida en CORBA 2.0–provee un mecanismo de binding dinámico para los objetos que necesiten manejar mensajes entrantes de componentes que no tienen skeletons compilados en IDL. Un skeleton dinámico analiza los parámetros de entrada en un mensaje recibido para determinar a cuál componente está dirigido–es decir el componente y el método destino. Como contrapartida, los stubs compilados (estáticos) se definen para cada clase en particular y esperan una implementación para cada método especificado en IDL. Los skeletons son sumamente útiles para definir puentes entre distintos ORBs. También pueden ser utilizados por intérpretes y lenguajes de scripting para generar código en forma dinámica.

- El ***Adaptador de Objetos*** (Object Adapter) está ubicado por encima de los servicios básicos de comunicación del ORB y atiende los pedidos que van dirigidos a los objetos. Suministra el entorno de ejecución necesario para instanciar objetos, pasarles pedidos de servicios y asignarles IDs–CORBA los denomina *referencias remotas*. El adaptador también registra las clases que soporta y las instancias que maneja (objetos), en el *Depósito de Implementación* (Implementation Repository). CORBA especifica que todo ORB debe contar con la implementación de un adapter estándar denominado *Basic Object Adapter* (BOA). Los servidores de todos modos pueden contar con más de un adapter.
- El ***Depósito de Implementación*** (Implementation Repository) suministra un depósito dinámico de información sobre las clases, los objetos instanciados y sus IDs. También sirve como lugar común para almacenar información adicional asociada a la implementación de los ORBs. Algunos ejemplos pueden ser información de seguimiento (tracing), seguridad y otros datos sobre la administración del ORB.
- La ***Interface del ORB*** (ORB Interface) está formada por unas pocas APIs locales que son idénticas a las disponibles del lado del cliente.

Con esto concluimos una presentación general de los componentes del ORB y sus interfaces. Ahora profundizaremos un poco más en cada uno.

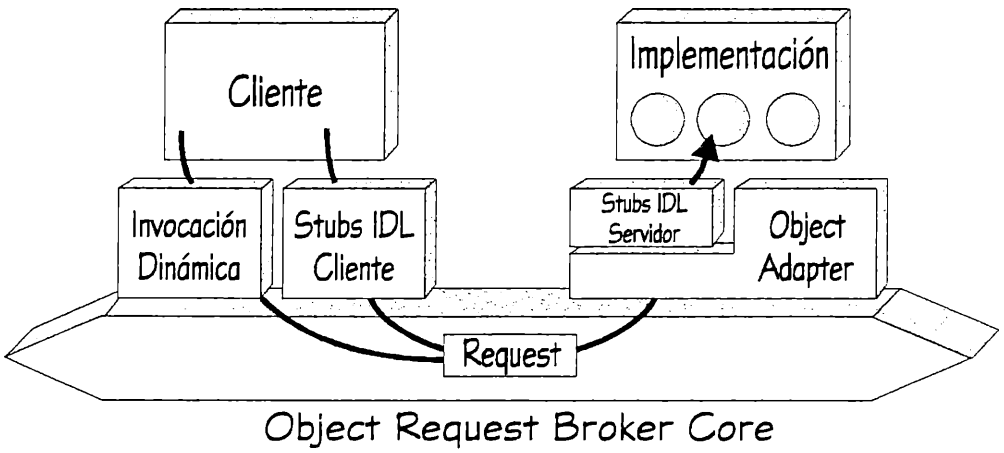


Figura 2-3. Invocaciones dinámicas y estáticas de métodos

Invocación de métodos en CORBA: estática vs. dinámica.

En ambos casos, el cliente realiza un pedido a través de una referencia a un objeto remoto (con un ID) invocando el método que cumple con el servicio requerido. Ambos tipos de interfaces, dinámicas y estáticas, cumplen con los mismos requisitos semánticos.

Los clientes acceden a las interfaces públicas de los objetos por un *mapeo* disponible al nivel de los lenguajes de programación (languaje mapping). Este mapeo permite acceder al ORB y todos sus servicios. Tanto la implementación de los objetos, el object adapter involucrado en

la invocación y el ORB utilizado para el acceso, son completamente transparente para los clientes dinámicos o estáticos.
La interface estática es generada en forma de stubs por un precompilador IDL. Es aconsejable para los programas que conocen en tiempo de compilación qué métodos deberán invocar exáctamente. Ofrece las siguientes ventajas:

- ***Es más fácil de programar***
- ***Ofrece un chequeo de tipos más robusto***
- ***Tiene buena performance***
- ***Es autodescriptiva–leyendo el código se entiende cómo funciona.***

Por su lado, una invocación dinámica a un método suministra un entorno de ejecución más flexible. Permite agregar nuevas clases al sistema sin afectar al cliente. Es muy útil para trabajar con herramientas que descubren servicios y objetos en forma dinámica

¿Qué es una referencia a un objeto?

Una *referencia a un objeto* suministra toda la información necesaria para identificar un objeto en un sistema distribuido en forma unívoca–es un identificador o nombre único. CORBA no hace ninguna especificación sobre la implementación de las referencias.
Dos ORBs que cumplan con el estándar CORBA (CORBA compliant) pueden usar representaciones distintas para las referencias remotas. Pero CORBA sí define *Referencias Interoperables de Objetos* (Interoperable Object References, IOR) que los distintos fabricantes deben usan para el manejo de las referencias a través de ORBs heterogéneos. Por esto, el acceso a objetos atravesando varios ORBs interconectados es transparente a los clientes, que referencian a sus objetos a nivel de lenguajes de programación.

Invocación estática de métodos en CORBA: desde la especificación en IDL hasta los stubs de la interface. Secuencia de pasos:

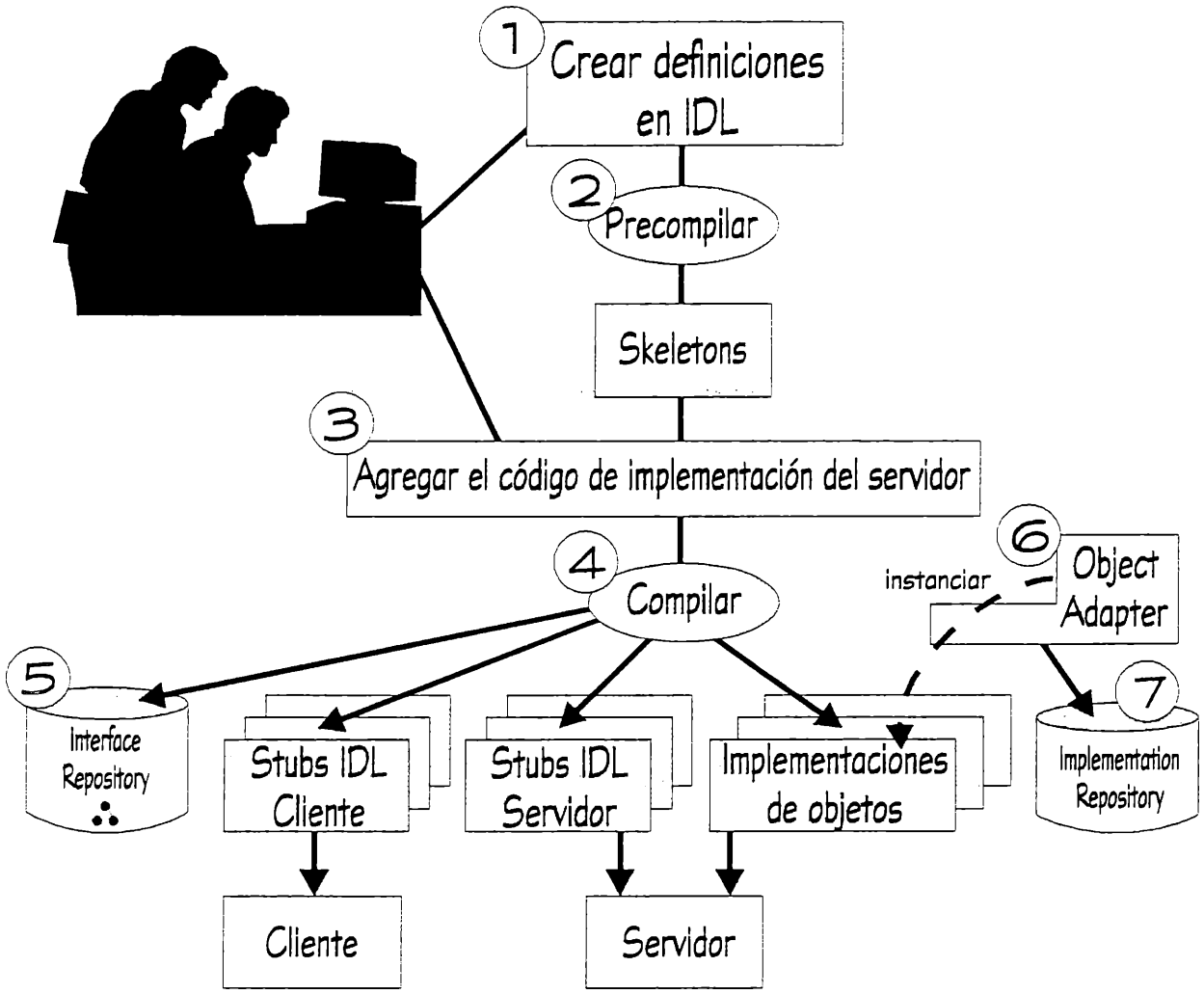


Figura 2-4. Definición de servicios: de IDL hasta los stubs

1.- ***Definir las clases usando IDL.*** Usando IDL, CORBA permite que los objetos le digan a sus potenciales clientes qué servicios ofrecen y cómo deben invocarlos. El lenguaje IDL define la clase de los objetos, sus atributos y los nombre de los métodos públicos junto con los parámetros necesarios. IDL es un lenguaje puramente declarativo.

2.- ***Procesar el código IDL con un precompilador IDL.*** El precompilador analiza el código y genera los skeletons para las clases que se usarán en la implementación.

3.- ***Completar los skeletons con el código específico de la implementación.*** En este paso se agrega el código en la implementación de los métodos. Es decir, definimos las clases de los objetos.

4.- ***Compilar el código.*** Un compilador CORBA-compliant es capaz de generar al menos cuatro tipos de salidas: 1) *archivos de importación* para describir las clases de los objetos en el Interface Repository; 2) los *stubs del cliente* para los métodos definidos con IDL–estos stubs serán invocados por los clientes que necesiten acceder estáticamente a servicios remotos con un ORB; 3) *stubs para el server* para invocar los métodos del server (se denominan también *interfaces de invocación*); y 4) el código que implementa las clases del servidor. La generación automática de los stubs libera a los programadores de tener que escribirlos ellos mismos.
5.- ***Registrar las definiciones de las clases en el Interface Repository.*** Normalmente esto se hace con algún utilitario que compile en el *Interface Repository* la información IDL, para que sea consultada por otros componentes.
6.- ***Instanciar los objetos servidores.*** El adaptador de objetos se encarga de esta tarea cuando llegan pedidos que deben ser atendidos. Estos objetos son instancias de las clases definidas previamente. El adaptador dispone de diferentes estrategias para instanciar y administrar los objetos servidores.
7.- ***Registrar las instancias en el Implementation Repository.*** De esto también se encarga el adaptador, a medida que necesita instanciar las clases. El ORB utiliza toda esta información para localizar los objetos activos o activarlos cuando es necesario.

Invocación dinámica de métodos: paso por paso.

Las APIs para la invocación dinámica permiten que un cliente arme y envíe pedidos dinámicos de servicios a otros objetos. El cliente debe especificar el objeto destino, el método y los parámetros. Normalmente esta información se recupera del Interface Repository. La invocación dinámica ofrece la máxima flexibilidad para construir aplicaciones en entornos distribuidos.

Para la invocación dinámica el cliente debe cumplir la siguiente secuencia de pasos:

1.- ***Obtener la descripción del método buscando en el Interface Repository:*** CORBA especifica cerca de diez llamadas distintas para localizar objetos en el repository. Luego de recuperar el objeto, el mensaje *describe* obtiene su descripción completa en IDL
2.- ***Armar la lista de argumentos:*** hay una estructura definida llamada *NamedValueList.* Esta lista es creada con el mensaje *create_list* y luego usar *add_arg* para cargarle los argumentos.
3.- ***Armar el mensaje:*** debe especificarse la referencia del object, nombre del método y lista de argumentos.
4.- ***Enviar el mensaje con el pedido.*** Hay tres formas de hacerlo: 1) Con *invoke* se envía el mensaje y se esperan los resultados (como RPC); 2) Enviar *send* y luego enviar *get_response* para obtener los resultados y 3) Enviar sólo *send.* En este caso no se espera un resultado.

CORBA: el lado de los servidores

"El objetivo inicial con CORBA fue soportar muchas implementaciones distintas. La especificación de CORBA no elimina o descarta los distintos enfoques usados para implementar un ORB."

- Geoff Lewis, Chairperson
OMG Object Services

¿Cuáles son los requerimientos de un ORB para implementar objetos servidores? Se requiere una infraestructura que registre las clases, instancie nuevos objetos y les asigne nuevos IDs, publicite su existencia, invoque los métodos cuando los clientes envíen mensajes y maneje la concurrencia. Si quisiéramos mayor sofisticación, deberíamos agregar a la lista el balanceo de la carga de ejecución y la seguridad. Estamos hablando de toda la funcionalidad que ofrecen los *monitores de procesamiento de transacciones* (MPT). ¿Quién hace todo este trabajo? El *object adapter*.

¿Qué es un Object Adapter?

Es un mecanismo que permite a los objetos acceder a los servicios del ORB. Suministra un entorno de ejecución completo para correr aplicaciones que actúen como servidores. Algunos de los servicios ofertados por el object adapter son:

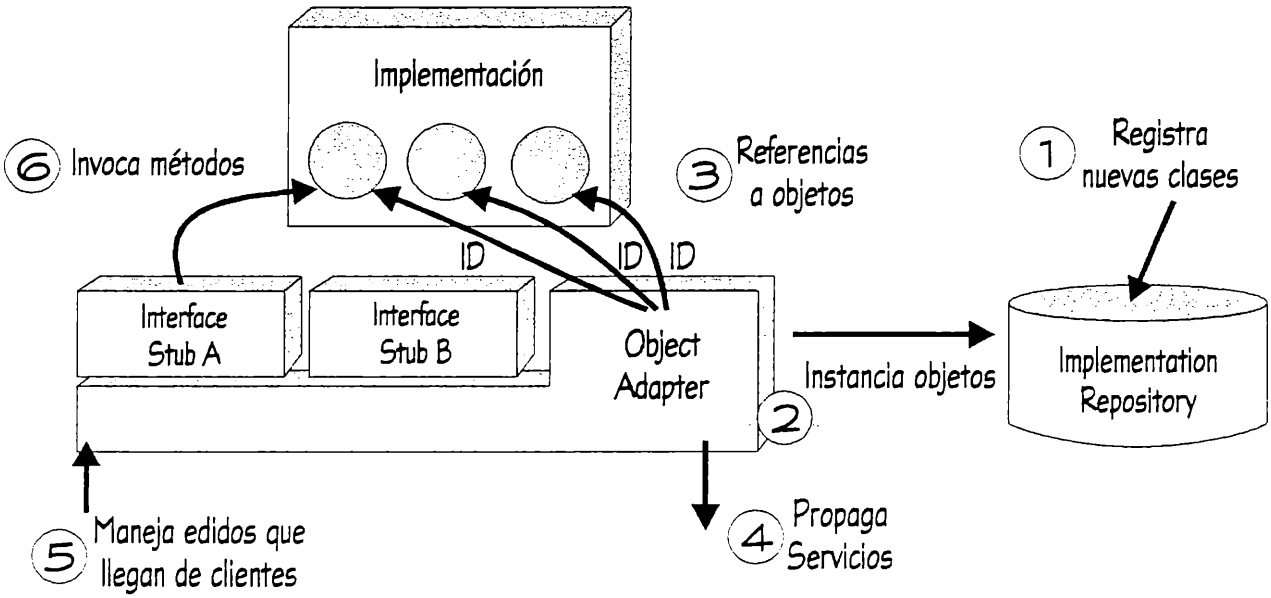


Figura 2-5. Estructura de un Object Adapter típico

1.- ***Registrar las clases en el Implementation Repository.*** Podemos ver al Implementation Repository como un medio de almacenamiento controlado y administrado por el Object Adapter.

2.- ***Instancia nuevos objetos.*** Se encarga de crear nuevas instancias de las clases que implementan a los objetos servidores. El número de instancias creadas depende del tráfico de mensajes entrantes. El adapter se encarga de balancear la carga de ejecución.
3.- ***Generar y administrar las referencias a objetos.*** El object Adapter asigna referencias (IDs únicos) a los nuvos objetos que instancia. Es responsable del mapeo entre las instancias (implementación) y la representación específica del ORB de las referencias.
4.- ***Divulgar la presencia de los objetos server:*** es decir, debe difundir los servicios disponibles en los objetos del ORB y responder las consultas que se hagan.
5.- ***Maneja las llamadas entrantes de los clientes:*** interactúa con la capa superior del ORB, recuperando los pedidos y pasándolos al stub de la interface correspondiente. Éste a su vez, debe acomodarlos para que se pueda hacer la invocación del método sobre el objeto.
6.- ***Rutea o dirige las llamadas hacia el método correcto:*** implícitamente el object adapter se encarga de invocar los métodos descriptos en los stubs (o skeletons) y así activa la implementación (de los métodos).

Inicialización de CORBA ¿O cómo se encuentra su ORB un componente?

Hay un conjunto de APIs qie permiten que un objeto se inicialice en un entorno ORB. Estas APIs permiten que los objetos descubran su ORB, el Interface Repository y otros servicios. Por ejemplo: una llamada al API "ORB_int" informa al ORB de la presencia del nuevo objeto y permite que éste obtenga una referencia a él (al ORB).

El ORB intergaláctico

CORBA 1.1 se ocupaba sólo de poder crear aplicaciones portables, dejando de lado la interoperabilidad. CORBA 2.0 incluyó la interoperabilidad mediante la especificación de un protocolo para la comunicación entre ORBs. Esta especificación se conoce como *Internet Inter-ORB Protocol* (IIOP). El IIPO es básicamente TCP/IP con algunos esquemas de mensajes específicos para intercambio de información. Todo ORB que se considere CORBA-compliant debe implementar el IIOP en forma nativa o al menos ofrecer un puente de comunicación hacia él.. Luego, cualquier ORB propietario puede conectarse con el universo de ORBs traduciendo los pedidos desde y hacia IIOP.

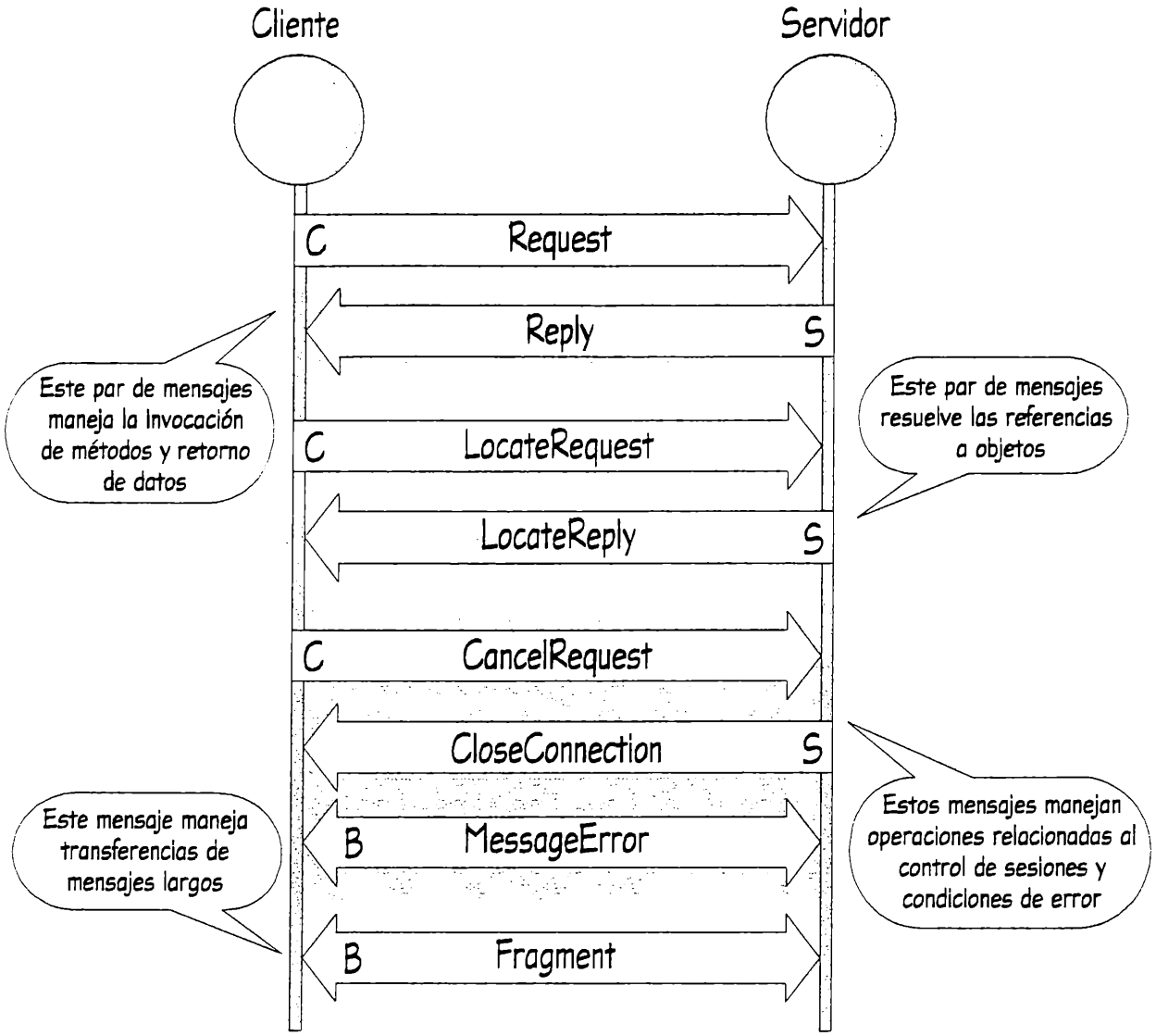
> "La elección de soluciones que interoperen a través ORBs no produce ningún impacto sobre el desarrollo de software de aplicaciones. La interoperabilidad es una cuestión a resolver entre los fabricantes de ORBs, no entre los fabricantes y los usuarios. Pocos, si algunos, desarrolladores tendrán que involucrarse alguna vez programando con GIOPs y ESIOPs."
>
> - Tom Mowbray et al., Autor
> The Essential CORBA
> (Wiley, 1995)

GIOP

El ***Protocolo General de ORBs (General Inter-ORB Protocol, GIOP)*** especifica un conjunto de formatos de mensajes y representaciones comunes de datos para la comunicación entre ORBs. El GIOP fue diseñado específicamente para la interacción entre ORBs. Se definió para funcionar directamente sobre cualquier protocolo de comunicación orientado a conexión. Los objetivos perseguidos con GIOP fueron la simplicidad, escalabilidad y generalidad. La simplicidad asegura facilidad de uso e implementación para los fabricantes de ORBs. La escalabilidad se refiere a soportar ORBs y redes de ORBs del tamaño de Internet. El diseño del formato de mensajes de GIOP para usar protocolos orientados a conexión cumple con el requisito de generalidad. Finalmente, usando TCP/IP como mecanismo de transporte de red, GIOP logra el mayor alcance posible.

Para el garantizar el intercambio de información entre diferentes sistemas, GIOP especifica un método para codificar los tipos de datos de IDL en una representación que todas las partes son capaces de entender. La representación de datos comunes (***CDR, Common Data Representation***) define la representación de los tipos primitivos (integer, long, doubles, etc) y compuestos (structure, union, array, string). CDR también codifica pseudo-objetos que no son ni básicos ni compuestos, como los contextos de información para la invocación de mensajes y excepciones. Las referencias a objetos son pseudo-objetos particularmente importantes, pues representan los objetos sobre los que se hacen las invocaciones de métodos. Dentro del mismo ORB, una referencia a un objetos podría ser simplemente un puntero a un objeto. Pero cuando esa referencia debe ser enviada a otro ORB, se codifica como una referencia de objeto interoperable (***Interoperable Object Reference, IOR***), que es una estructura de datos que registra la referencia al objeto (object key) y cierta información dependiente del protocolo de transporte que indica cómo contactar al objeto.

Para mantener la simplicidad, GIOP define sólo ocho mensajes. En la siguiente figura aparecen estos mensajes y las entidades que los originan. Hay tres que son originados por los clientes (**C**) y tres que son enviados por los servidores (S). Los últimos dos mensajes, ***MessageError*** y ***Fragment***, son bidireccionales (**B**). En general, los mesajes de GIOP tienen tres partes: una cabecera GIOP, una cabecera específica del mensaje y un body o segmento de datos.



Los mensajes del GIOP

- ***Request*** es el mensaje básico de invocación de métodos en CORBA. Incluye información concerniente al objeto destino, el método que se invoca (y sus parámetros), si el cliente espera o no una respuesta y un identificador de pedido (ID). Este ID es generado por el cliente y permite que ambas partes identifiquen unívocamente cada pedido.
- ***Reply*** es la respuesta del servidor a un request. Tiene el ID del request, el status del request y el body de éste. Si el status indica que no se produjo ninguna excepción, el body tiene los valores de retorno asociados al método invocado. En cambio, si se produjo una excepción, el body contiene la codificación de la excepción. Un tercer status, "*Location Forward*" indica que el objeto sobre el que se debe hacer la invoación se ha movido a otro ORB. En este caso, el body guarda la nueva localización. El ORB

cliente deberá retransmitir transparentemente el pedido inicial hacia esa nueva localización.

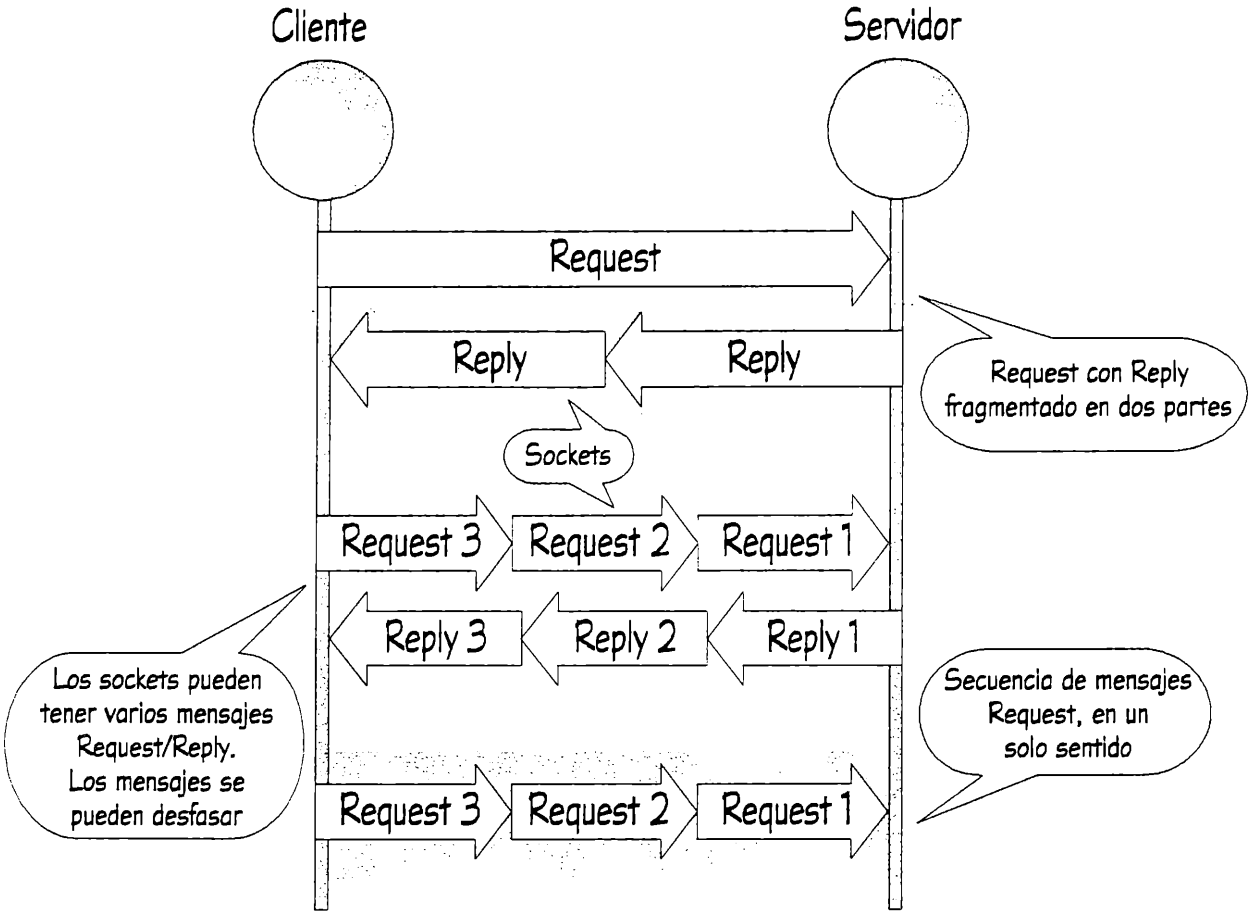
- ***LocateRequest*** permite a los cliente determinar si una dada referencia a un objeto es conocida o no, si el servidor en cuestión puede procesar un pedido a ese objeto y si no, a cuál server deben enviarse los pedidos de ese objeto. La información obtenida con ***LocateRequest*** también es suministrada por ***Request***, pero en el caso de una situación de *"Location Forward"*, permite a los clientes evitar una transmisión potencialmente extensa de parámetros asociada a un pedido particular.
- ***LocateReply*** es la respuesta del servidor a ***LocateRequest***. Contiene el ID del ***LocateRequest*** correspondiente y un status de localización que indica si el objeto en cuestión es conocido o no por el servidor, o está localizado en otro lugar. En última instancia, ***LocateReply*** también incluye la nueva localización del objeto.
- ***CancelRequest*** tiene como único parámetro un ID de un pedido (***Request***). Los clientes utilizan este mensaje para notificar a los servidores que dejan de esperar una respuesta (***Reply***) de un pedido pendiente o ***LocateRequest***.
- ***CloseConnection*** es enviado por los servidores para notificar a los clientes que la conexión se está cerrando. Los pedidos pendientes se pierden y deben ser reenviados usando otra conexión. Los clientes, en cambio, podrían cerrar la conexión sin previo aviso y los servidores deberían saber manejar esta situación. ***CloseConnection*** permite a los ORBs reclamar y reusar las conexiones muertas.
- ***MessageError*** es enviado en respuesta a cualquier mensaje desconocido o que esté formateado de manera inapropiada.
- ***Fragment*** permite a los ORBs partir los mensajes en secciones y enviar a cada una como un mensaje GIOP separado. Si la cabecera de un mensaje GIOP de reply o request indica que siguen más fragmentos, éstos serán enviados como mensajes de ***Fragment***.

IIOP

El ***Internet Inter-ORB Protocol (IIOP)*** especifica cómo los mensajes del GIOP se envían sobre una red TCP/IP. Permite usar Internet como puente para comunicar los ORBs. Cualquier ORB compatible con CORBA debe soportar GIOP sobre TCP/IP.
Mientras GIOP define la forma y contenido de los mensajes, IIOP codifica la información necesaria para invocar métodos sobre objetos usando IORs. Estos IORs se componen de un número de versión, el host y port del ORB a donde deben enviarse los mensajes, una clave para identificar al objeto y una serie de componentes con información que se usan cuando se invoca el método (por ejemplo: los parámetros de seguridad del ORB que origina el mensaje, etc). La distinción estricta entre servidores y clientes es uno de los aspectos de la manera en que IIOP usa la capa de transporte subyacente. En la figura mostramos las diferentes formas de comunicación entre clientes y servidores.

En la primera secuencia, un request de un cliente es respondido con un reply del servidor, en este caso usando dos fragmentos. Notar que cada stream representa un conexión de socket (bidireccional). Como los request están identificados unívocamente, es posible que varios mensajes compartan la misma conexión e incluso es posible un desfasaje de requests y replies asincrónicos, como en la segunda secuencia. Esto permite compartir los recursos de

manera más eficiente. Algunos ORBs (especialmente el de Expersoft) permiten elegir entre conexiones dedicadas o multiplexadas, de tal manera que los mensajes largos puedan enviarse por un socket dedicado para evitar congestionar la transmisión de los mensajes más cortos. Por último, en la tercera secuencia, el cliente puede enviar una serie de requests en un solo sentido, sin esperar respuesta del servidor. Por ejemplo: un cliente podría usar este esquema para hacer periódicamente un *ping* al servidor para notificarle que sigue con vida.



Flujo de mensajes usando IIOP

ESIOPs

Finalmente están los ***protocolos especializados*** (***Enviroment-Specific Inter-ORB Protocols, ESIOPs***), que se usan para lograr una interoperación no estándar sobre redes específicas de transporte. CORBA 2.0 especifica a DCE como uno de los protocolos ESIOPs opcionales. La combinación DCE-ESIOP ofrece una excelente alternativa para ORBs que deban cumplir con requerimientos altos de rendimiento y confiabilidad. Se incluyen características avanzadas como seguridad con Kerberos, directorios globales, manejo distribuido del tiempo, RPC autenticado, etc. DCE permite transmitir grandes volúmenes de datos en forma eficiente.

METAINFORMACION EN CORBA: IDL Y EL INTERFACE REPOSITORY

"La metainformación es información autodescriptiva que sirve para describir servicios e información. Con la metainformación se pueden agregar y descubrir nuevos servicios en un sistema durante la ejecución."

- Tom Mowbray et al., Autor
The Essential CORBA
(Wiley, 1995)

La metainformación es el ingrediente que nos permite crear sistemas cliente/servidor ágiles. Un sistema ágil es un sistema autodescriptivo, dinámico y reconfigurable. Un sistema así ayuda a sus componentes a decubrirse mutuamente durante la ejecución y les suminitra la información que necesitan para interoperar. Un sistema así permite desarrollar software sin necesidad de cablear las llamadas e invocaciones a servidores particulares. Finalmente, suministra información para que las herramientas de desarrollo puedan crear y manejar componentes.

Un sistema ágil se diferencia de los sistemas tradicionales por su uso intensivo de la metainformación, que describe en forma consistente todos los servicios, componentes y datos disponibles. La metainformación permite que componentes desarrollados independientemente se descubran entre si y sean capaces de colaborar.

CORBA es un sistema ágil. Para ser CORBA-compliant, cualquier componente debe ser autodescriptivo. Cualquier servicio u objeto que sobreviva en un bus CORBA debe ser autodescriptivo. El mismo bus CORBA es autodescriptivo. Luego, CORBA es un sistema autodescriptivo. Ya vimos que el lenguaje para describir esta información es el IDL (Interface Definition Languaje). El depósito donde se registra toda esta información se denomina Interface Repository, que no es otra cosa que un base de datos dinámica que guarda la especificación de la interface–escrita en IDL–de los componentes que maneja un ORB.

Un vistazo al IDL de CORBA

"Separar las interfaces y las implementaciones es una cuestión crítica para lograr la interoperabilidad sobre los lenguajes de programación, versiones de objetos, ejecutables, librerías dinámicas, espacios de direcciones y plataformas."

- Richard Adler, Coordinated Computing
(Marzo de 1995)

Ya vimos que IDL define interfaces que son contratos que especifican los servicios disponibles para los potenciales clientes. IDL es un lenguaje neutral y completamente declarativo. No especifica detalles relativos a la implementación de los objetos–componentes.
Podemos usar IDL para especificar los atributos de un componente, las clases de las que hereda, la excepciones que levanta, eventos y métodos–incluyendo el tipo de los parámetros de entrada y resultados. La gramática del IDL es un subconjunto de C++ con sentencias específicas para soportar conceptos de distribución.

La estructura del IDL

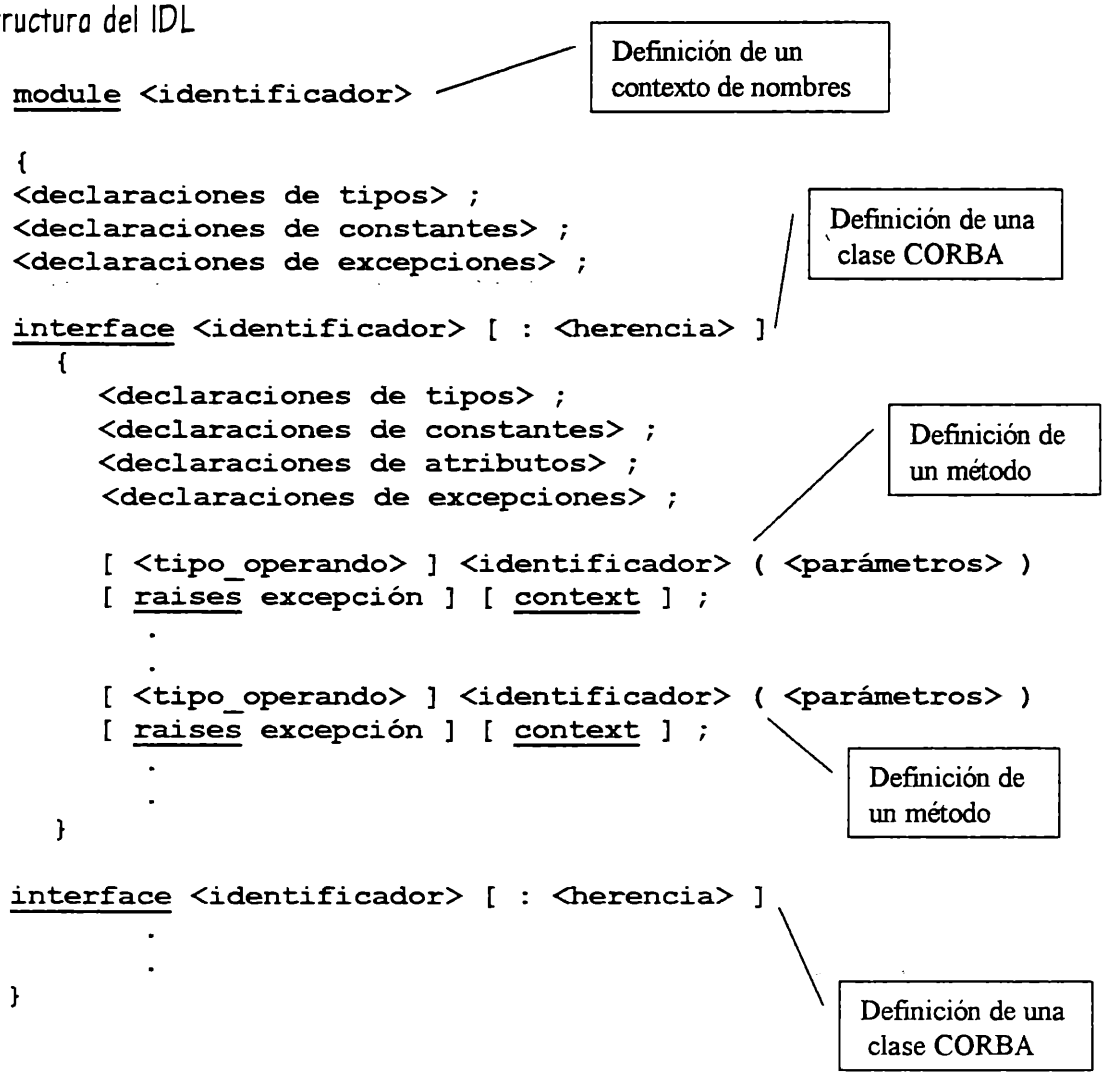


Figura 2-6. Estructura de un archivo con definiciones IDL

Los principales elementos del IDL CORBA son:

- ***Módulos***: ofrecen espacios de nombres (namespaces) para describir grupos de clases (o interfaces, según la terminología del OMG). Los módulos se identifican por la cláusula *module*. Cada módulo tiene un nombre formado por uno de más identificadores

separados por "::". El propósito de los módulos es agregar un nivel extra en la jerarquía del espacio de nombres de IDL.

- ***Interfaces***: son las definiciones de clases. Especifican las operaciones–métodos–que pueden ser invocados por los clientes. También definen los atributos de los objetos. Los atributos son valores para los que el compilador genera automáticamente *accesors* (métodos para consultar y asignar el valor del atributo). Cada interface puede declarar una o más excepciones para indicar que los métodos no se han ejecutado correctamente. Una interface puede ser derivada a partir de una o más interfaces (superclases); con esto estamos diciendo que IDL soporta herencia múltiple.
- ***Operaciones:*** son el equivalente en CORBA a los métodos. Denotan los servicios del objeto que están disponibles para los clientes. IDL define cada método por un nombre, los parámetros y el tipo del resultado que retorna. Cada parámetro tiene un *modo*, que indica el sentido del pasaje del valor del parámetro. Es decir, si el valor va del cliente al servidor (*in*), del servidor al cliente (*out*) o en ambos sentidos (*inout*). Cada parámetro tiene un *tipo* que restringe los valores posibles. Finalmente, es posible declarar en el método las excepciones que podrían levantarse en caso de que ocurran errores (es útil para interceptar las excepciones con un manejador propio).
- ***Tipo de datos:*** se usan para describir los valores aceptados en los parámetros CORBA, atributos, excepciones y resultados de los métodos. Estos tipos de datos son objetos CORBA independientes de los lenguajes, sistemas operativos y ORBs. CORBA soporta dos categorías de tipos: *básicos* y *compuestos*. Los *tipos básicos* disponibles son: *short*, *long*, *unsigned short*, *float*, *double*, *boolean*, *char* y *byte*. Los *tipos compuestos* incluyen: enum, string, struct, array, union, sequence y any. El tipo *struct* es similar a las estructuras de C++. El tipo *sequence* permite manejar secuencias (colecciones) de objetos; son parecidos al *array* pero sin las restriciones de tener que declarar la cantidad máxima de elementos. El tipo *any* es útil en situaciones dinámicas porque puede representar cualquier objetoque tenga una interface IDL. En la implementación de las clases, estos tipos se mapean a tipos nativos disponibles en el lenguaje de programación elegido.

Ejemplo con IDL

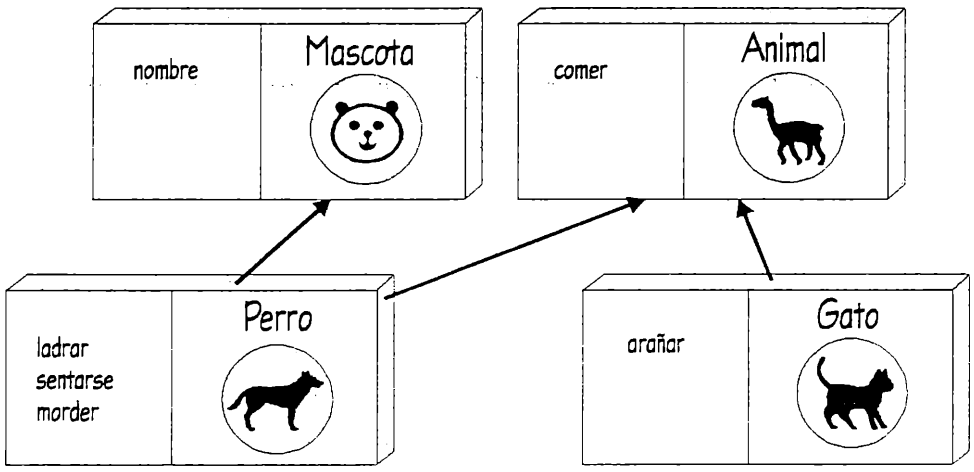


Figura 2-7. Jerarquía de clases: Perro y Gato

En el ejemplo de más abajo mostramos la definición IDL para crear dos clases, ***Perro*** y ***Gato***, en el módulo ***Animales***. En la figura superior aparecen la ejeraquía de clases que pretendemos representar.
De la definición IDL se desprende que las interfaces ***Perro*** y ***Gato*** se derivan de las clases ***Mascota*** y ***Animal***. La clase ***Perro*** define el atributo edad, para el cual el compilador IDL automáticamente generará los métodos de acceso que permitan asignar y consultar el valor.
Esta clase tiene tres métodos (u operaciones): ***ladrar***, ***sentarse*** y ***morder***. Levanta una excepción cuando el perro no está interesado en cumplir con la orden (ladrar, sentarse o morder). Por otro lado, la clase ***Gato*** deriva solamente de ***Animal*** y define un solo método: ***arañar*** (por supuesto que los gatos también son mascotas, aunque hayamos decidido no representar este hecho en la jerarquía).

Type Codes: datos autodescriptivos de CORBA

CORBA define *type codes* que representan cada uno de los tipos de datos IDL. Estos type codes se usan crear información autodescriptiva que se pueda pasar a través de los sistemas operativos, ORBs e Interface Repositories. Cada type code tiene globalmente un ID único.
Ejemplo del uso de los type codes:

- ***Las Interfaces de Invocación Dinámicas*** los utilizan para indicar el tipo de datos de los argumentos.
- ***Los protocolos inter-ORB–incluyendo IIOP***–los usan para especificar los tipos de datos de los argumentos de un mensaje enviado de un ORB a otro. Sirven como una representación canónica de datos.
- ***Los Interface Repositories*** los usan para crear descripciones IDL neutrales.
- ***El tipo de datos any:*** para suministrar un parámetro genérico autodescriptivo.

Listado IDL:

```
module Animales
{

/* Definición de la clase Perro */

interface Perro : Mascota, Animal
   {
      attribute integer edad;
      exception NoInteresado {string explicacion};

      void ladrar ( in short cuantoTiempo);
         raises ( NoInteresado )

      void sentarse ( in string donde );
         raises ( NoInteresado )
```

Además de las clases anteriores, que representan las estructuras posibles en IDL, CORBA define la clase ***Repository*** que es la superclase de todas las anteriores. En la figura anterior aparece la jerarquía de composición de los distintos objetos–instancias de las clases anteriores–dentro del repository. Por ejemplo, las instancias de ***ModuleDef*** contienen otros objetos y además están contenidas dentro del repository.

Interface Repository: la jerarquía de clases

Los arquitectos del Interface Repository de CORBA notaron la jeraquía de composición anterior y definieron tres clases abstractas para describirla: ***IRObject***, ***Contained*** y ***Container***. La siguiente figura muestra la jeraquía de herencia, en contraposición a la jeraquía de composición que vimos antes.
La clase ***IRObject*** suministra comportamiento para determinar el tipo de cualquier objeto registrado en el repository y para destruirlo–eliminarlo del repository. Las subclases de ***Container*** heredan el comportamiento necesario para navegar en el repository. La clase ***Contained*** define el comportamiento de los objetos que están contenidos en otros objetos. Luego, cualquier objeto guardado en el repository se deriva de alguna de esas clases o de ambas (herencia múltiple).

Para acceder a la metainformación guardada en el Interface Repository invocamos los mismos métodos en los diferentes objetos allí registrados–gracias al polimorfismo. Con esto podemos navegar por el repository y encontrar la información buscada. Estos métodos son heredados de ***Container*** y ***Contained***:

- ***Describe***: cuando se manda este mensaje a un objeto del repository, retorna una estructura que describe, en IDL, su interface.

```
        void morder (in string quien );
           raises ( NoInteresado )
    }

/* Definición de la clase Gato */

interface Gato : Animal
    {
        void arañar ();
    }
} /* Fin de la definición de Animales */
```

Figura 2-8. Definición IDL del módulo Animales

Interface Repository

El ***Interface Repository*** de CORBA es una base de datos on-line que guarda la definición de los objetos. Estas definiciones se pueden derivar directamente de un compilador IDL o bien con las funciones de escritura que ofrece el Repository.
La especificación de CORBA detalla cómo se debe organizar y recuperar la información del repository. Para esto define un conjunto de clases cuyas instancias representan la información registrada. La jerarquía de clases refleja especificación del lenguaje IDL. Así, la información almacenada se organiza con instancias que representan una versión compilada del archivo escrito en IDL.

¿Para qué necesitamos un Interface Repository?

El ORB usa la información registrada para:

- ***Hacer el chequeo de tipos*** en los parámetros de los métodos invocados.
- ***Ayudar a la interconexión de ORBs heterogéneos***. Se necesita contar con esta información cuando un objeto debe viajar a través de ORBs heterogeneos interconectados.
- ***Suministrar "metainformación" para clientes y herramientas de desarrollo***. Los clientes usan esa información para crear invocaciones de métodos *"on the fly"*. Las herramientas–browsers de clases, generadores de aplicaciones, compiladores–la usan para saber cuáles son las jerarquías de herencia y definiciones de las clases.
- ***Permitir que los objetos puedan autodescribirse:*** el método *get_interface* está disponible para todos los objetos y retorna información sobre la interface de esos objetos (primero es necesario registrar en el repository la interface IDL del objeto).

Los Interface Repositories pueden ser administrados localmente o ser mantenidos como recursos departamentales o corporativos. Son fuentes valiosas de información sobre las estructuras de clases e interfaces de los componentes. Finalmente, un mismo ORB podría acceder a varios repositories si estuviera interconectado con otros ORBs–esto permite definir una *federación de repositories*, en la terminología del OMG.

Las clases del Interface Repository: una jerarquía de composición

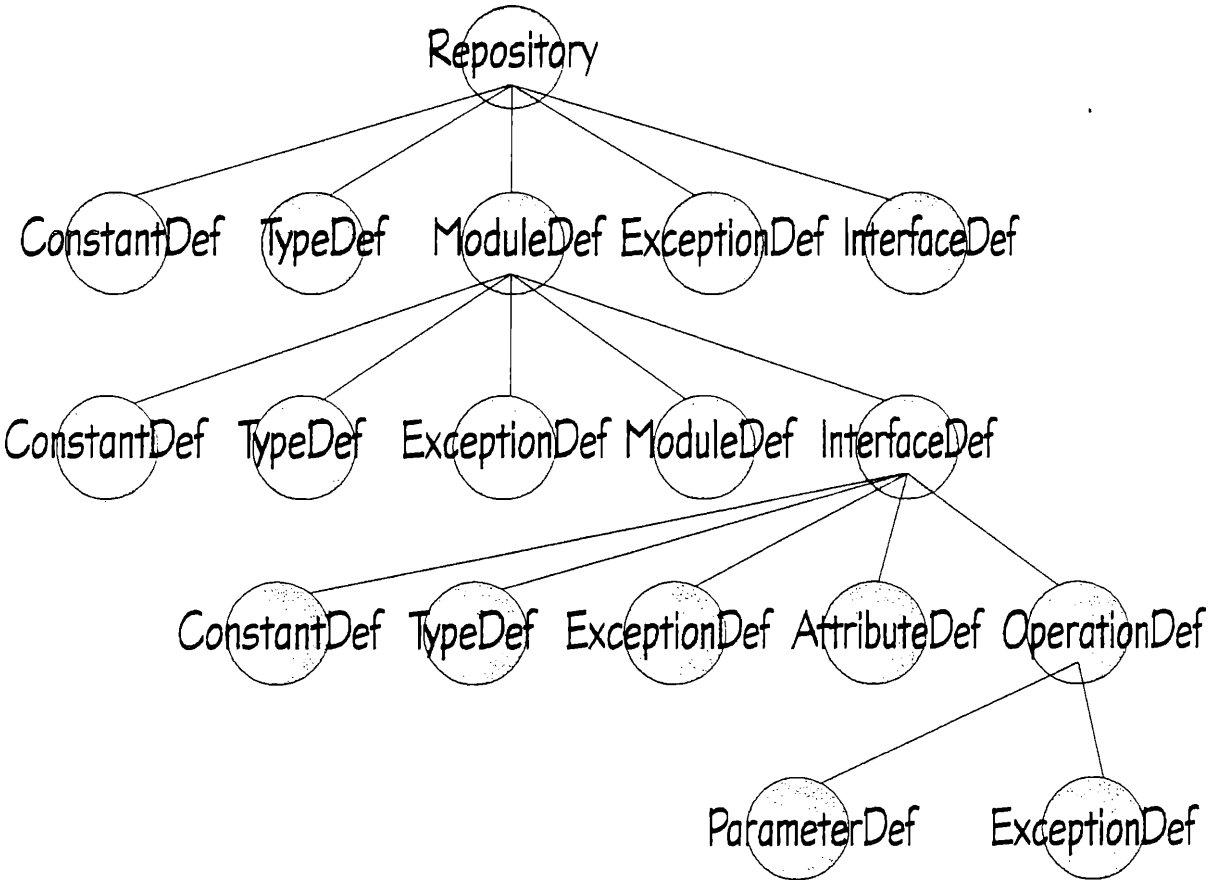


Figura 2-9. Jerarquía de composición de las clases del Interface Repository

El Interface Repository está implementado como un conjunto de objetos que representan la información que almacena. Estos objetos pueden ser persistentes, es decir se pueden guardar en medios no volátiles. CORBA agrupa la metainformación en módulos que definen espacios de nombres (namespaces). Dentro del repository, los nombres de los objetos cada módulo son únicos. Hay una interface–clase–para cada una de las ocho estructuras IDL:

- ***ModuleDef*** define un agrupamiento lógico para los objetos. Como ocurre en un archivo IDL, los módulos sirven para agrupar las interfaces y navegar a través de los grupos.
- ***InterfaceDef*** para definir la interface de un objeto; contiene listas de constantes, excepciones y definiciones de otras interfaces.
- ***OperationDef*** para definir la declaración de un método en la declaración de una interface; contiene listas de parámetros, sus tipos, las excepciones.
- ***ParameterDef*** para definir un argumento de un método.
- ***AttributeDef*** para definir un atributo en una interface
- ***ConstantDef*** define la declaración de una constante
- ***ExceptionDef*** define la declaración de una excepción.
- ***TypeDef*** para declarar tipos en las definiciones IDL.

Además de las clases anteriores, que representan las estructuras posibles en IDL, CORBA define la clase ***Repository*** que es la superclase de todas las anteriores. En la figura anterior aparece la jerarquía de composición de los distintos objetos–instancias de las clases anteriores–dentro del repository. Por ejemplo, las instancias de ***ModuleDef*** contienen otros objetos y además están contenidas dentro del repository.

Interface Repository: la jerarquía de clases

Los arquitectos del Interface Repository de CORBA notaron la jeraquía de composición anterior y definieron tres clases abstractas para describirla: ***IRObject***, ***Contained*** y ***Container***. La siguiente figura muestra la jeraquía de herencia, en contraposición a la jeraquía de composición que vimos antes.
La clase ***IRObject*** suministra comportamiento para determinar el tipo de cualquier objeto registrado en el repository y para destruirlo–eliminarlo del repository. Las subclases de ***Container*** heredan el comportamiento necesario para navegar en el repository. La clase ***Contained*** define el comportamiento de los objetos que están contenidos en otros objetos. Luego, cualquier objeto guardado en el repository se deriva de alguna de esas clases o de ambas (herencia múltiple).

Para acceder a la metainformación guardada en el Interface Repository invocamos los mismos métodos en los diferentes objetos allí registrados–gracias al polimorfismo. Con esto podemos navegar por el repository y encontrar la información buscada. Estos métodos son heredados de ***Container*** y ***Contained***:

- ***Describe***: cuando se manda este mensaje a un objeto del repository, retorna una estructura que describe, en IDL, su interface.

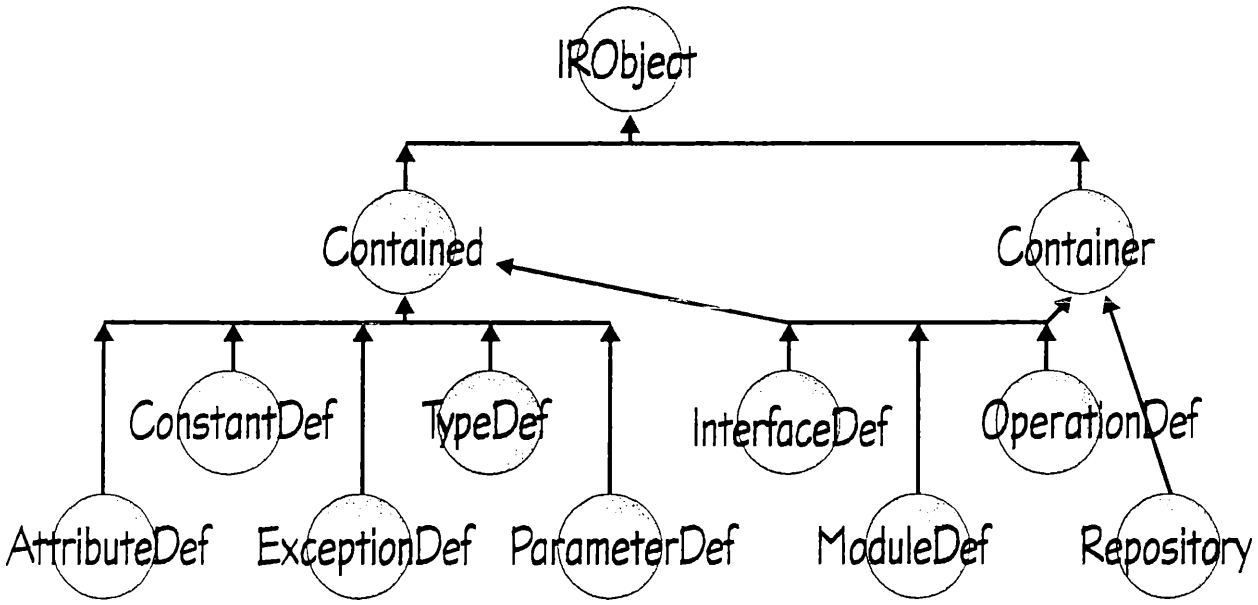


Figura 2-10. Jerarquía de herencia de las clases del Interface Repository

- ***Lookup***: enviando este mensaje a un objeto contenedor de otros, retorna una sequencia con los objetos que contiene.
- ***Lookup_name***: para buscar un objeto por su nombre, dentro de otro.
- ***Contents***: enviando este mensaje a un objeto contenedor, retorna una lista con los objetos que contiene o heredan de él (en la jerarquía de composición). Se usa para navegar por el contenido del repository; por ejemplo enviando este mensaje a la instancia de ***Repository*** obtenemos todo su contenido y así sucesivamente.
- ***Describe_Contents***: retorna una secuencia con descripciones de los objetos que contiene.
- ***Is_a***: enviado a una instancia de ***InterfaceDef*** retorna TRUE si la interface es idéntica o hereda de la interface pasada como parámetro en el mensaje.
- ***Lookup_id***: este mensaje enviado al repository retorna el objeto cuyo id es el que se pasó como parámetro.

¿Cómo encontramos la interface de un objeto dado?

Hay tres maneras de lograrlo:

1.- Enviando el mensaje ***Object::get_interface*** al ORB y pasando el objeto en cuestión como parámetro. El resultado será una instancia de ***InterfaceDef*** que describa la interface del objeto.

2.- Navegando por el Interface Repository. Por ejemplo, si sabemos el nombre del módulo donde está declarada la interface del objeto, comenzamos por ahí.

3.- Conociendo el ID del objeto dentro del repository. De esta manera accedemos directamente, enviando el mensaje ***lookup_id*** al repository.

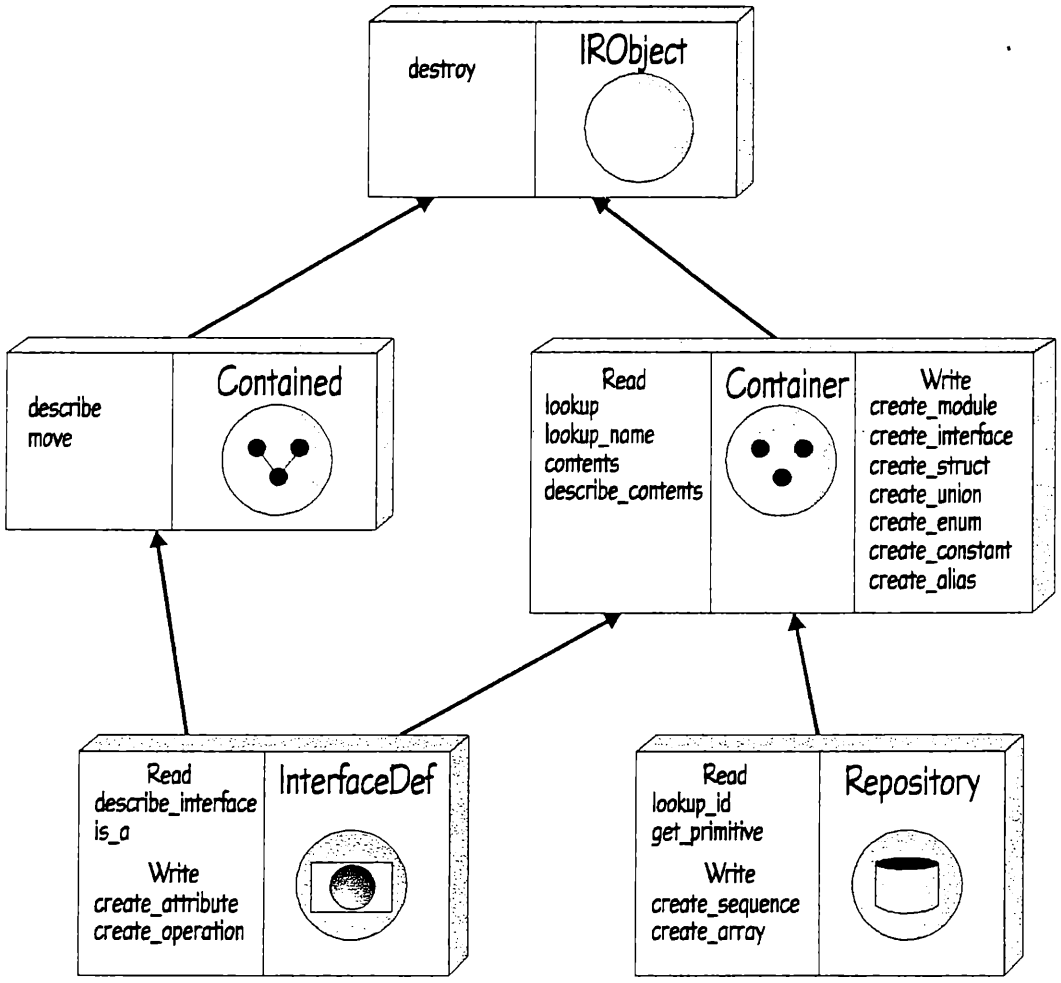


Figura 2-11. Interface Repository: Jerarquía de clases e interfaces

Federaciones de Interface Repositories

Es posible crear una federación de repositories interconectando los Interface Repositories de varios ORBs. Para evitar las colisiones de nombres, cada repository asigna nombres únicos–***Repository IDs***–a las interfaces y métodos manejados globalmente. Dicho con otras palabras, significa que se preserva la identidad única de cada interface a través de los ORBs. Los Repository IDs identifican globalmente a los módulos, interfaces (clases), constantes, excepciones, atributos, operaciones (métodos) y parámetros. Se usan para sincronizar las definiciones a través de los ORBs.

SERVICIOS DE CORBA

Un ORB por si solo no tiene todo lo necesario para que los objetos interoperen a nivel de sistema. UN ORB es algo así como una linea telefónica–suministra los mecanismos básicos para transmitir los mensajes de los objetos. Todos los otros servicios son ofrecidos mediante objetos que tienen definida una interface IDL y que residen por encima del ORB. El IDL junto con el ORB suministran la funcionalidad del *bus de software*; los servicios extienen este bus. Los componentes que funcionan al nivel del usuario utilizan a ambos–bus y servicios.

Los servicios de CORBA fueron diseñados para complementarse. Cada uno cubre un aspecto bien específico y es tan complicado como se requiere que sea.
Hemos preferido analizar los servicios agrupándolos por la funcionalidad que ofrecen, en lugar de seguir el orden cronológico en que fueron definidos.

SERVICIOS DE CORBA: NAMING, EVENTS, LIFE CYCLE Y TRADER

El Naming Service es como una guía telefónica para encontrar objetos, por nombre. El Trader Service es como la una guía de páginas amarillas: publica los servicios de los objetos y permite hacer búsquedas. El Life Cycle Service se encarga de cuestiones existenciales–incluyendo la vida, muerte y reubicación de los objetos; permite crear, mover, copiar y destruir objetos. Finalmente el Events Service se encarga de manejear interacciones asincrónicas entre objetos que ni siquiera se conocen entre si (objetos anónimos) y notifica los eventos interesantes que ocurren en el ORB.

NAMING SERVICE

Es el principal mecanismo para que los objetos del ORB puedan localizar a otros objetos. Los *nombres* son valores con significado humano asignados a los objetos para identificarlos. El Naming Service se encarga administrar el mapeo entre esos nombres y las referencias a los objetos correspondientes. Decimos que un *name binding* (respetando la terminología del OMG) es una asociación entre entre un nombre y un objeto. Un *naming context* es un espacio de nombres dentro del cual cada nombre es único. Antes dijimos que todo objeto tiene un ID (reference ID) que es único para ese objeto. Podemos asociar uno o más nombres a cada referencia, en relación a un naming context dado.

Este servicio nos permite definir jerarquías de nombres. Los clientes pueden navegar a través de los árboles de contextos de nombres buscando objetos. Luego, es posible unir contextos

de nombres de diferentes dominios y crear una federación de naming services. CORBA no requiere que exista una *raiz universal* en esa jerarquía.

¿Qué es un nombre para un objeto?

Es posible referenciar los objetos usando una secuencia de nombres que forman un árbol jerarquíco de nombres. El la siguiente figura, los nodos oscuros son contextos de nombres (naming contexts). Así, el nombre de un objeto consiste en una secuencia de nombres (o *componentes*) que forman un *nombre compuesto*. Cada componente–excepto el último– nombra un contexto. El último componente es el *nombre simple* del objeto.
Resolver un nombre significa encontrar los objetos asociados con ese nombre en un dado contexto.

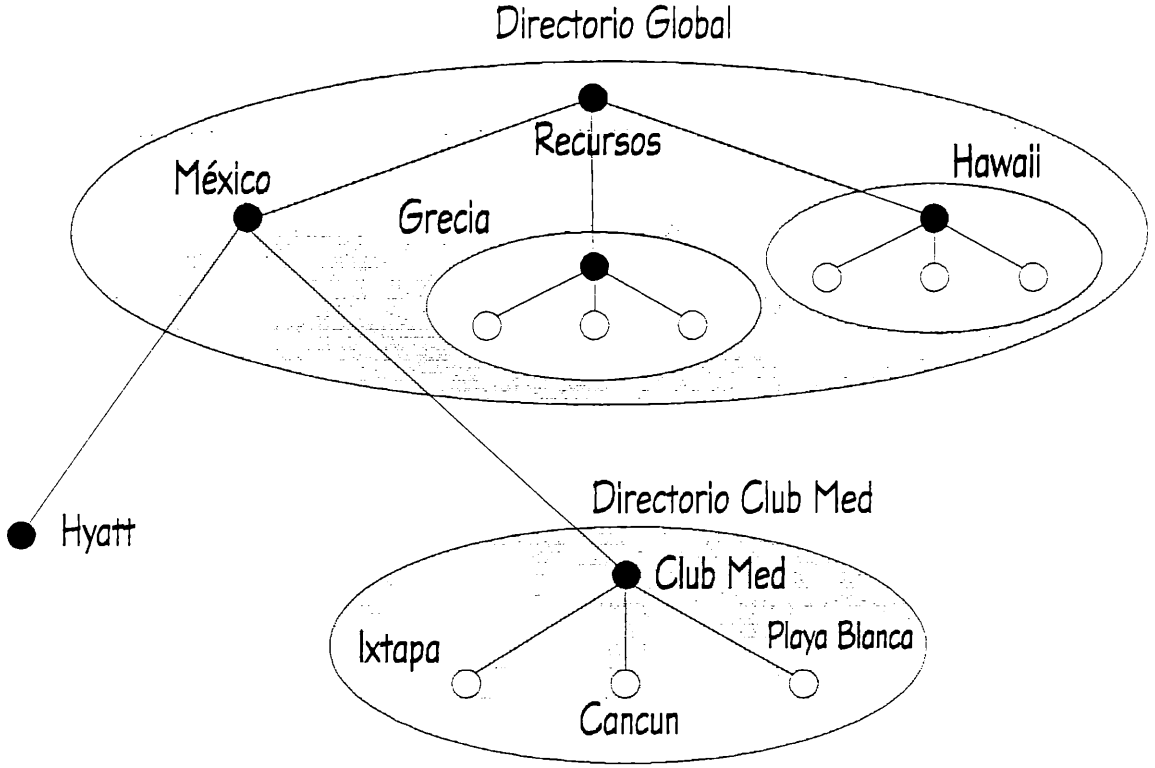


Figura 2-12. Soporte para nombres jerárquicos de CORBA

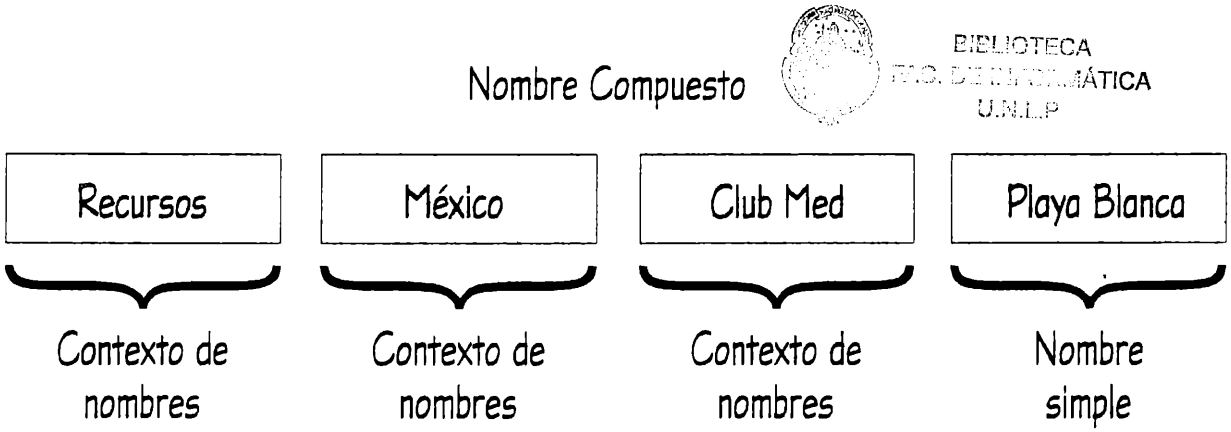


Figura 2-13. Nombre compuesto

Este servicio se deriva de dos clases (interfaces): ***NamingContext*** y ***BindingIterator***. Las instancias de ***NamingContext*** administran un conjunto de asociaciones de nombres a objetos, en las que cada nombre es único. Estos objetos también pueden estar relacionados con otros contextos para armar una jerarquía de nombres.

El método *bind* de ***NamingContext*** sirve para asociar un nombre a un dado objeto dentro de un contexto–con esto creamos las jerarquía de nombres. Invocando a *unbind* eliminamos una asociación nombre-objeto de un contexto. Por otra parte el método *bind_new_context* crea un nuevo contexto y le asigna el nombre que se pase como parámetro. El método *destroy* es para eliminar un contexto completo.
Usando el método *resolve* es posible buscar objetos; *resolve* retorna un objeto a partir de un nombre, en un dado contexto. El método *list* permite iterar a través de un conjunto de nombres (retorna una instancia de ***BindingIterator***). Con el iteraror se deben usar los métodos *next_one* y *next_n*. Para eliminar el iteraror se usa el mensaje *destroy*.

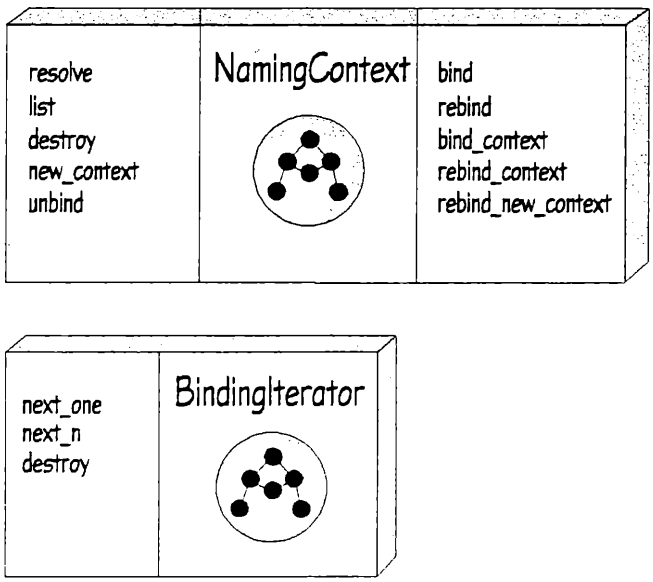


Figura 2-14. Interfaces Object Naming Servicer

TRADER SERVICE

La idea es registrar los servicios ofrecidos, asociando un trader (comerciante) a cada objeto que los ofrezca. Los traders se encargan de administrar toda la información relevante, incluyendo:

- ***Una referencia al objeto***. Es usada por los clientes para acceder al objeto qu ofrece los servicios requeridos.
- ***Una descripción del tipo de servicio ofrecido***. Incluye los nombres de las operaciones (o métodos) junto con los parámetros y el tipo de los resultados. También incluye algunos atributos adicionales de los servicios y el contexto en que se obtienen.

La función de los traders es almacenar esta información en forma persistente. Los clientes se ponen en contacto con los traders para conocer cuáles servicios son ofrecidos o pedir un determinado servicio de determinadas características–como en la guía telefónica de páginas amarillas. El trader se encarga de buscar lo más apropiado a los requerimientos del cliente entre las ofertas de los objetos proveedores que estén suscriptos.

Nuevamente es posible construir federaciones de traders interconectando diferentes dominios. Esto facilita que los sistemas publiquen sus servicios en un pool global, mientras mantienen sus propias políticas–locales–de control.

LIFE CYCLE SERVICE

Este servicio ofrece operaciones para crear, copiar, mover y eliminar objetos. Este servicio se extendió recientemente para que todas operaciones puedan aplicarse a grupos de objetos relacionados. Por ejemplo: relaciones de contención, relaciones de referencias entre objetos, relaciones de integridad, etc.

Ejemplo de un ciclo de vida compuesto

Veamos cómo se aplican las operaciones del ciclo de vida a un objeto que tiene referencias a otros objetos. En el ejemplo tenemos un documento que tiene una o varias páginas, que a su vez almacenan objetos de texto e información en multimedia. El documento está almacenado en una carpeta y mantiene referencias a un catálogo (el catálogo a su vez tiene una entrada para el documento). El servicio del ciclo de vida se encarga de admisnitrar un grafo para todas las relaciones anteriores.

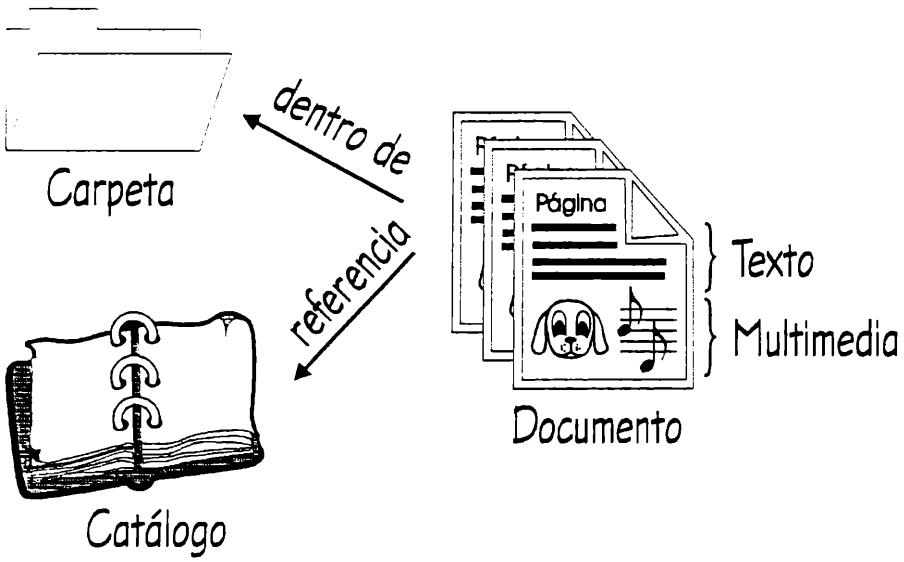


Figura 2-15. El Life Cycle Service debe tratar con objetos relacionados

Una operación de *deep move* aplicada al documento provoca éste sea movido junto con todos sus objetos dependientes (las páginas y sus contenidos), se actualice la referencia en el catálogo, el documento se saque de la carpeta origen y sea insertado en la carpeta destino. Si el documento fuera eliminado, sus páginas y objetos gráficos deben eliminarse automáticamente; también se eliminan las referencias desde el catálogo y la carpeta.

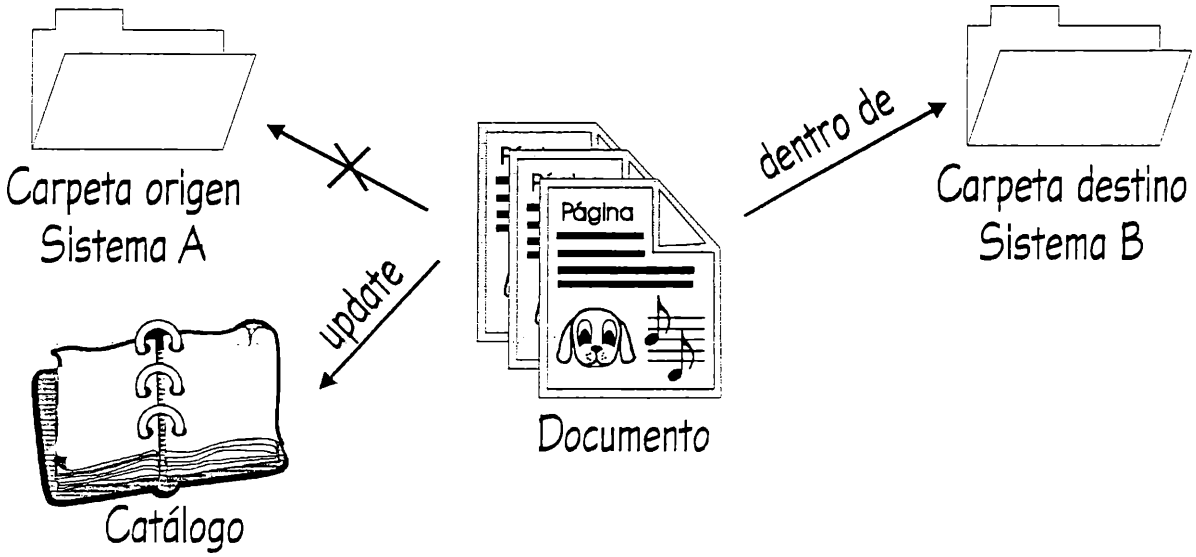


Figura 2-16. Una operación de "deep move" se encarga de las relaciones entre objetos

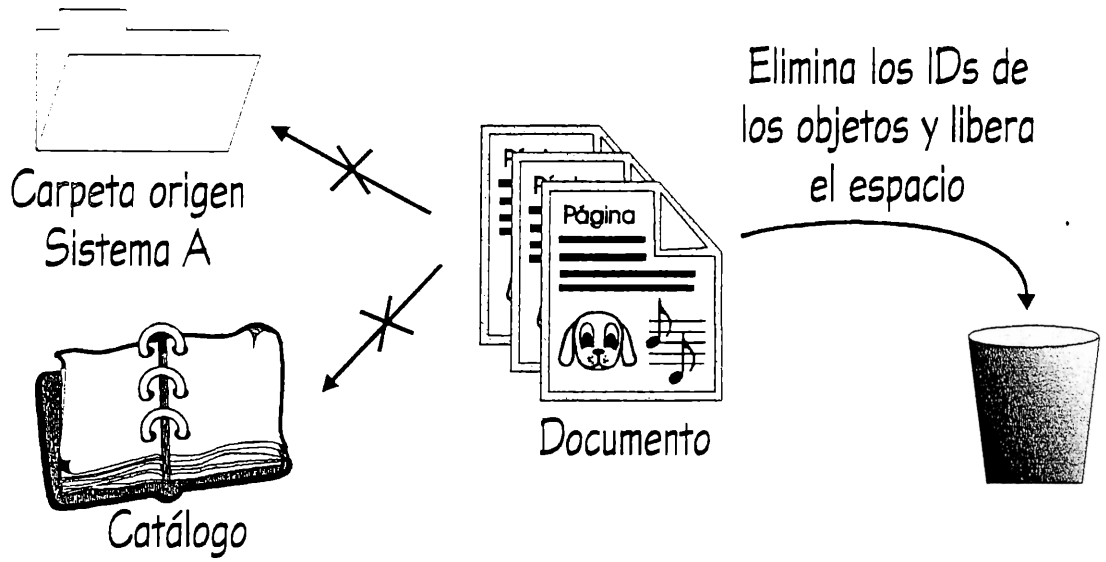


Figura 2-17. Una operación de "delete" elimina los objetos y las dependencias

Las interfaces del Life Cycle Service

Los clientes que usan el servicio tienen una visión realmente simple: solamente invocan las operaciones *move* y *copy* en el documento; todos los objetos involucrados son manejados transparentemente.

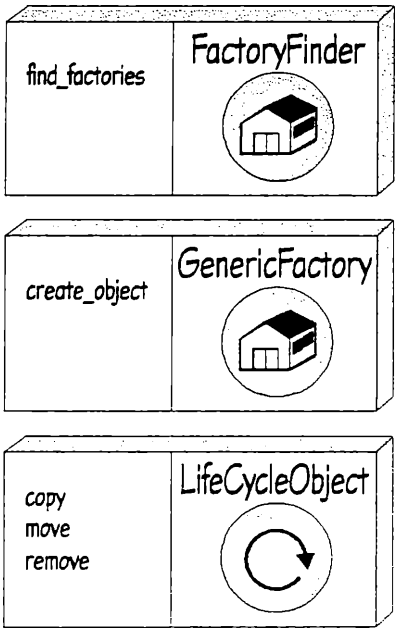


Figura 2-18. Interfaces del Life Cycle Service

Para crear un nuevo objeto, el cliente primero debe encontrar un *factory* (un objeto capaz de instanciar un objeto de la clase que corresponda), enviarle el mensaje *create* y obtener la referencia al nuevo objeto. También es posible crear un objeto nuevo clonando otro ya existente, con la operación *copy*. Los factories se encargan de reservar los recursos necesarios para instanciar el objeto, manejar las referencias y registrar el nuevo objeto en adapter y en el Implementation Repository. Si quisiéramos crear un objeto remoto a través de una red se invocaría al factory correspondiente en el nodo destino involucrado.

En general, las interfaces que participan en este servicio definen tres operaciones básicas: *copy*, *move* y *remove*. *Copy* obtiene una copia de un objeto y retorna la referencia; *move* provoca que un objeto navegue hacia una nueva localización y *remove* elimina el objeto en cuestión.
La clase ***FactoryFinder*** define la interface para buscar factories. Como cada objeto requiere distintos recursos para ser creado, es imposible definir una interface que sirva para todos. La clase ***GenericFactory*** define una operación general *create_object*.
Las interfaces del ciclo de vida compuesto
Estas clases se agregaron recientemente al Life Cycle Service. Permiten manejar operaciones de *deep copies*, *move* y *delete* sobre los objetos. La clase ***OperationsFactory*** crea instancias de la clase ***Operations***. Notar que estas instancias también soportan *copy*, *move* y *remove* y sólo agregan *destroy*.

Las clases ***Node***, ***Role*** y ***Relationship*** se derivan de las correspondientes clases de otro de los servicios–Relationship Service. Estas clases permiten crear relaciones entre objetos que no se conocen entre si. Para definir relaciones se asocia a cada objeto un rol y luego se relacionan los roles entre si. Luego, las mismas operaciones de *copy*, *move*, *remove* y *destroy* son aplicables a la relación.

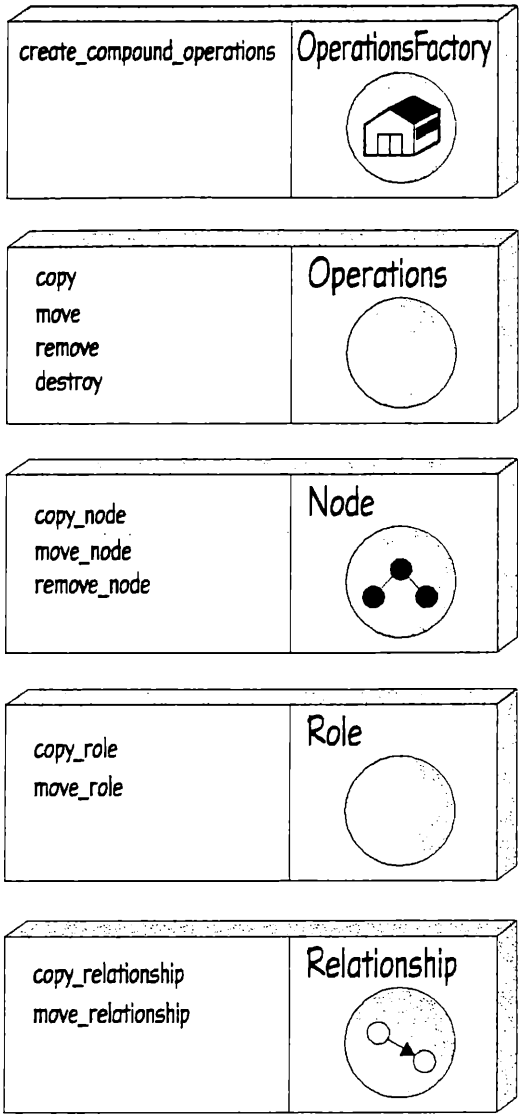


Figura 2-19. Interfaces del Life Cycle Service Compuesto

EVENTS SERVICE

Este servicio permite que los objetos registren (y remuevan) su interés en eventos específicos. Un *evento* es la ocurrencia de algo dentro de un objeto, que es de interés para otros objetos. Una *notificación* es un mensaje que envía un objeto a las partes interesadas informándoles que un evento particular ha ocurrido. Normalmente, el objeto que genera el evento no conoce a las partes interesadas (otros objetos). Todo esto es administrado por el servicio de eventos, que crea un canal de comunicación entre los objetos involucrados (el que genera el evento y los interesados en ser notificados). Así se logra que los objetos no necesiten conocerse, pues el servicio desacopla la interacción.

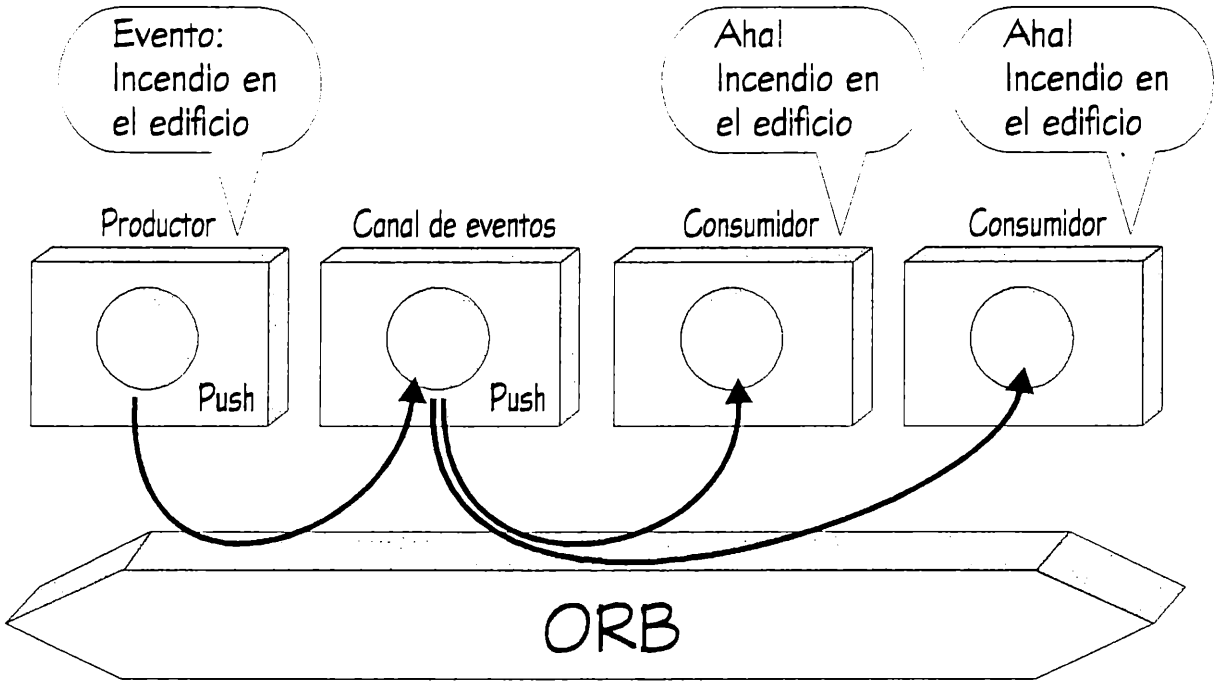
Productores y consumidores de eventos

Dijimos que el servicio de eventos desacopla la comunicación entre los objetos participantes. Para esto, servicio identifica dos roles: *productores* y *consumidores*. Los *productores* producen eventos; los *consumidores* los procesan a través de manejadores de eventos (handlers). La comunicación de los eventos entre los productores y los consumidores respeta el protocolo estándar de envío de mensajes definido en CORBA.

Por otro lado tenemos dos modelos para la gestión de los eventos: *push* y *pull*.
En el ***modelo push*** (empujar), el productor del evento toma la iniciativa y envía el evento al consumidor. En el ***modelo pull*** (tirar), es el consumidor el que toma la iniciativa y pide los eventos ocurridos al productor.
Hay un tercer objeto involucrado que se denomina *canal de eventos*. El canal de eventos es al mismo tiempo productor y consumidor. Permite que varios productores se comuniquen con varios consumidores asincrónicamente sin que se conozcan. Los canales de eventos son objetos CORBA que residen en el ORB y desacoplan la comunicación entre los productores y los consumidores.
El canal de eventos soporta los dos modelos de notificación:

- Con el ***modelo push***, el productor envía un mensaje *push* a un canal de eventos; el canal por su parte envía un *push* a los consumidores. Un consumidor puede dejar de recibir notificaciones de eventos enviando el mensaje *disconnect_push_consumer* al canal. En cambio, para comenzar a recibirlos debe registrar su interés enviando el mensaje *add_push_consumer*.
- Con el ***modelo pull***, el consumidor envía un mensaje *pull* al canal; y éste a su vez envía *pull* al productor. Con *try_pull*, el consumidor puede verificar en forma periódica si hay eventos pendientes.Un productor puede dejar de recibir los mensajes de pull enviando *disconnect_pull_supplier* al canal. En cambio, debe enviar *add_pull_supplier* para registrarse, pasando una referencia de si mismo y los servicios que ofrece.

A) Modelo Push



B) Modelo Pull

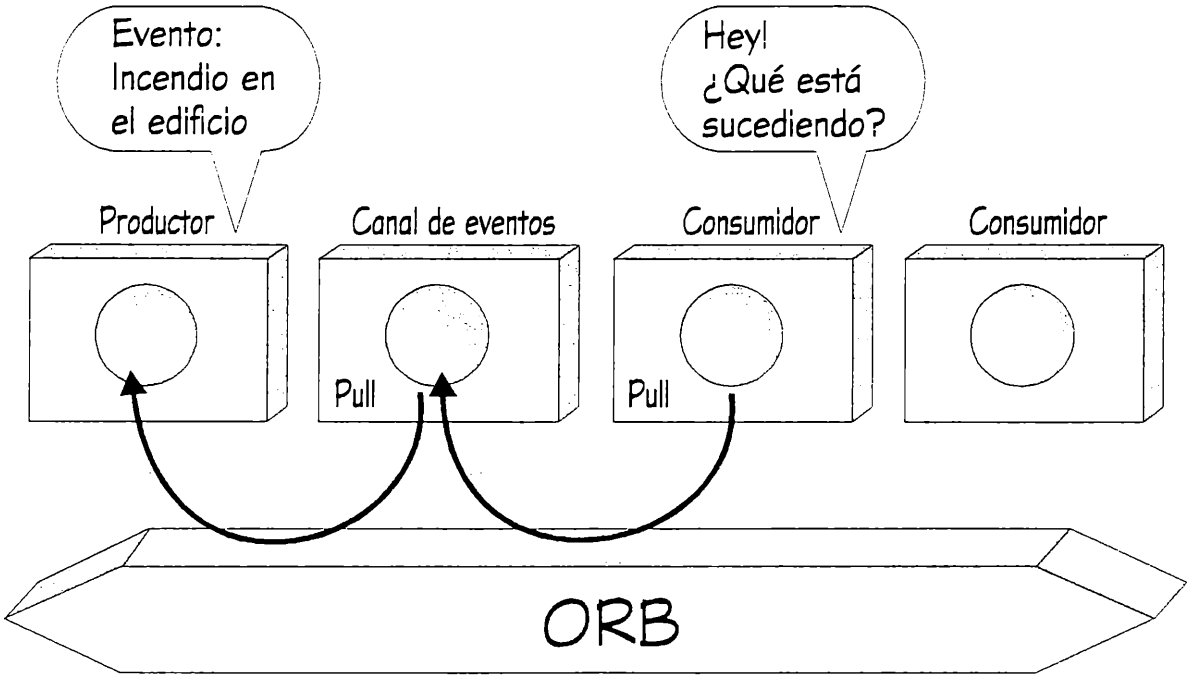


Figura 2-20. Modelos de eventos: Push y Pull

El canal de eventos

En general, los canales de eventos no entienden el contenido de la información referida a los eventos que están notificando. Más bien son los productores y consumidores los que deben acordar una semántica común para los eventos. Sin embargo, el Event Service también soporta un modelo de eventos tipados que permite que las aplicaciones describan el tipo de sus eventos con IDL. Esto permite definir eventos especiales–por ejemplo, para manejar documentos o a nivel de sistema. Luego, los consumidores pueden suscribirse a un tipo de evento en particular. El tipado de los eventos es poderoso para filtrar la información de los eventos notificados. Permite obtener notificación sólo de los eventos que nos interesan. Finalmente, la administración de los eventos permite administrar diferentes niveles de confiabilidad e incluso almacenar los eventos en forma persistente para guardarlos durante el tiempo que fuera necesario (por ejemplo: guardar ciertos eventos importantes durante una semana y luego activarlos).

SERVICIOS DE CORBA: TRANSACTIONS Y CONCURRENCY

"Si pudiéramos combinar el poder de los objetos con la confiabilidad de las transacciones, podríamos lanzar la computación comercial a una nueva Era."

- John Tibbets and Barbara Bernstein

Las transacciones son esenciales para la construcción de aplicaciones distribuidas confiables. El Servicio de Transacciones (Object Transaction Service, OTS) define interfaces IDL que permite que los objetos distribuidos en distintos ORBs participen en transacciones atómicas–aún ante la presencia de fallas catastróficas. OTS soporta opcionalmente también transacciones anidadas.

El Servicio de Control de Concurrencia (Concurrency Control Service) permite que los objetos coordinen su acceso a recursos compartidos utilizando locks. Para acceder a un recurso compartido los objetos deben obtener el *lock* apropiado–del servicio. Cada lock está asociado a un único recurso y pertenece a un único cliente. El servicio define distintos modos de bloqueo que corresponden a las distintas categorías de acceso y a diversos niveles de granularidad. Podemos decir que este servicio complementa al OTS, permitiendo obtener y liberar locks dentro de las transacciones.

OBJECT TRANSACTION SERVICE (OTS)

Es posiblemente la pieza más importante del middleware para objetos intergalácticos.
Con OTS, los ORBS sumnistran un entorno transparente para ejecutar aplicaciones críticas de componentes. En ese sentido, podemos decir que los ORBs representan la siguiente generación de monitores de procesamiento de transacciones.

¿Qué es una trannsacción?

"La idea de sistemas distribuidos sin control de transacciones es como una sociedad sin leyes. Uno no quiere las leyes, pero realmente las necesita para resolver los conflictos."

- Jim Gray

Las transacciones son bastante más que eventos o acontecimiento: se han vuelto una filosofía para diseñar aplicaciones que garantiza la robustez de los sistemas distribuidos. En un entorno ORB cada transacción debe ser monitoreada desde el origen (cliente) y a través de todos los servidores involucrados. Cuando una transacción termina, todas las partes

involucradas deciden si la transacción falló o tuvo éxito. En cierta forma una transacción es como un contrato que vincula al cliente con uno o más servidores. Y es así como las transacciones se convierten en la unidad fundamental de recuperación, consistencia y concurrencia en un sistema de objetos distribuidos.

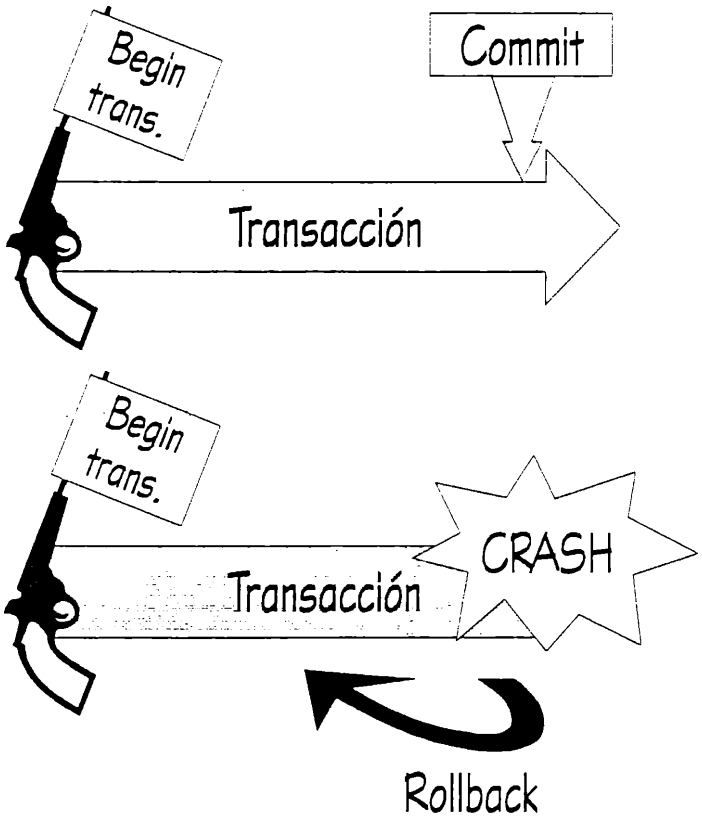


Figura 2-21. Transacción plana: todo o nada

Por supuesto que todos los objetos participantes deben adherir a esta disciplina de transacciones; caso contrario, un simple objeto podría corromper un sistema completo. En un mundo ideal, todas las interacciones entre los objetos distribuidos se basarían en transacciones.

Los modelos de transacciones definen cuándo comienza una transacción, cuándo termina y cuáles son los mecanismos de recuperación en caso de una falla. El modelo de transacciones planas (flat model) corresponde a la generación actual de monitores de procesamiento de transacciones (MPT). Se denominan *planas* porque toda la actividad hecha en la transacción está al mismo nivel. Cada transacción comienza con *begin_transaction* y termina con *commit_transaction* o *abort_trasaction*. Es una cuestión de todo o nada–no hay forma de confirmar o abortar partes de una transacción plana. Sin embargo, los modelos nuevos–por ejemplo el modelo de transacciones anidadas–ofrecen un control mucho más fino de la granularidad de las partes que forman la transacción. Estos modelos son sumamente atractivos porque tienen el potencial suficiente para reflejar al mundo real.

¿Qué es una transacción anidada?

La mayoría de las alternativas para las transacciones planas se basan en mecanismos que extienden el flujo de control más allá de la unidad lineal de trabajo. Dos de las maneras más obvias de extender el flujo de control son encadenando unidades de trabajo en secuencias lineales de *mini* transacciones (transacción encadenada) o creando una jerarquía anidada de unidades de trabajo (transacciones anidadas).

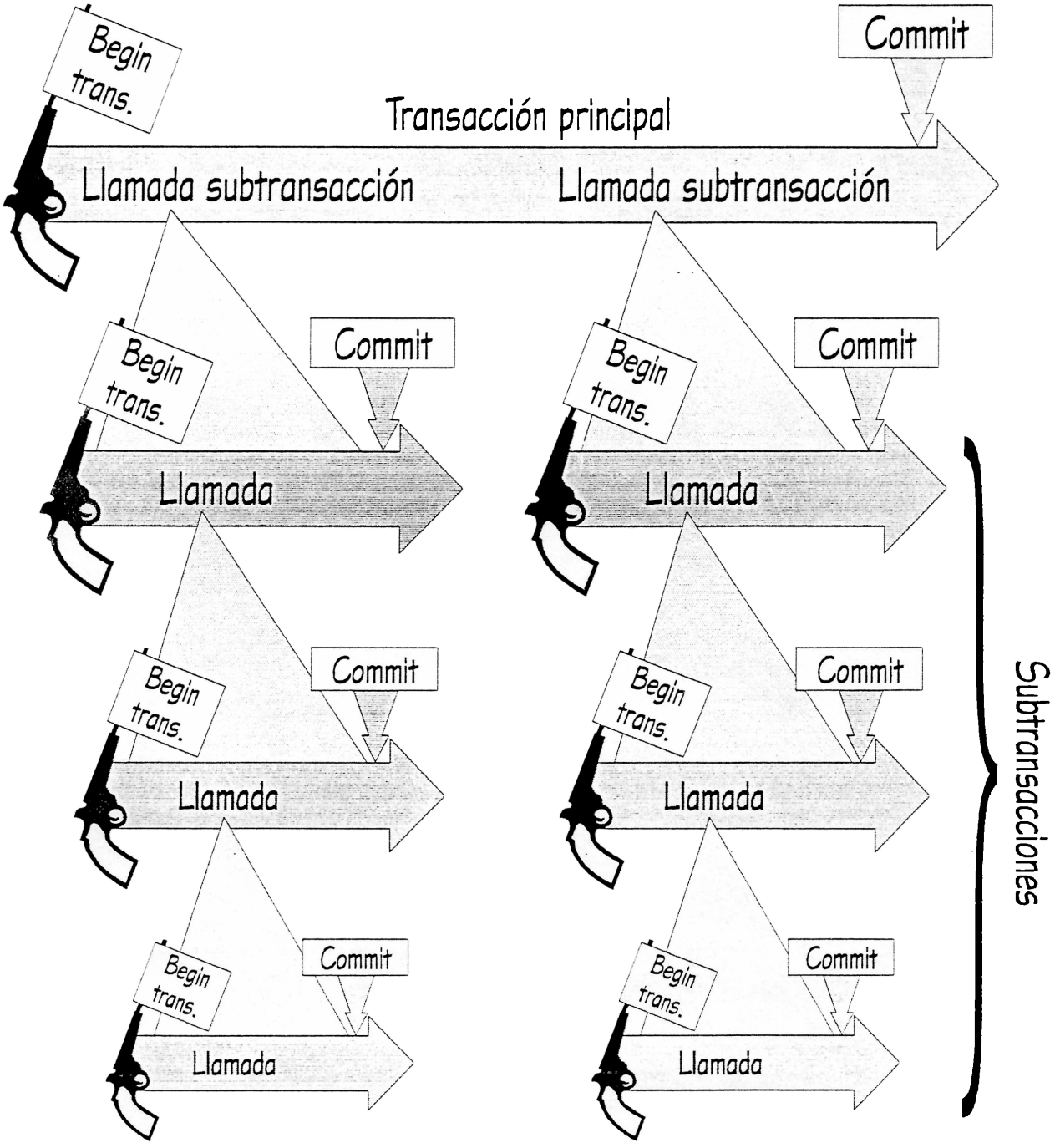


Figura 2-22. Transacciones anidadas

Las *transacciones anidadas* permiten definir transacciones dentro de otras transacciones. Para lograrlo, crean jeraquías de subtransacciones, de la misma forma que un programa procedural está compuesto de procedimientos. La transacción principal inicia a las subtransacciones, que se comportan como transacciones dependientes de ésta. A su vez, una subtransacción puede iniciar sus propias subtransacciones y así sucesivamente–esto conduce una estructura recursiva. Los efectos de una subtransacción se vuelven permanentes cuando ésta y sus ancestros hacen la operación de *commit*. Si la transacción padre aborta, todas las substransacciones automáticamente abortan–sin importar si ejecutaron o no *commit*.
El principal beneficio del anidamiento es que la falla de una subtransacción puede ser interceptada y reintentada usando un método alternativo, permtiendo que la transacción principal tenga éxito.

Características del servicio de transacciones

El OTS de CORBA tiene las siguientes características:

- ***Soporte para transacciones anidadas y planas.*** Todas las implementaciones del servicio deben soportar el modelo plano de transacciones; el soporte para el modelo de transacciones anidadas es opcional
- ***Permite involucrar en la misma transacción aplicaciones ORB y aplicaciones no ORB.*** OTS permite mezclar transacciones de objetos con transacciones procedurales que adhieran al estándar X/OPEN DTP.
- ***Soporte de transacciones que se expandan a través de varios ORBs heterogéneos.*** En una misma transacción es posible involucrar objetos de distintos ORBs. Además es posible que el mismo ORB soporte diferentes servicios de transacciones.
- ***Soporte para las interfaces IDL existentes.*** Una misma interface soporta implementaciones basadas en transacciones y también las que no están basadas en transacciones. Para que un objeto soporte transacciones alcanza con heredarlo de una clase abstracta especialmente definida. Este enfoque evita la combinación explosiva de variantes de IDL que difieren sólo en las características de las transacciones.

Elementos involucrados en el OTS

Los objetos involucrados en las transacciones pueden asumir uno de tres roles:

- **Cliente transaccional:** ejecuta un conjunto de invocaciones de métodos que ocurren dentro de ciclo begin/end. Los mensajes enviados dentro del ciclo pueden ser tanto para objetos transaccionales como para los no transaccionales. El ORB intercepta la llamada al *begin* y la redirige al Transaction Service, que asocia un contexto para la transacción al thread del cliente. Luego, el cliente se encarga de enviar los mensajes a los objetos remotos. Implícitamente, el ORB se encarga de insertar el contexto y propagarlo en todas las subsecuentes comunicaciones entre los objetos participantes de la transacción. El ORB también se ve involucrado cuando el cliente dispara una operación de *commit* o *rollback* y notifica al Transaction Service. El cliente no se percata de todo ese escenario; solamente comienza la transacción, envía los mensajes y confirma o deshace la transacción.

- **Servidor transaccional:** es una colección de uno o más objetos cuyo comportamiento es afectado por la transacción, pero que no tienen estados de recuperación o recursos propios. Implícitamente, el ORB propaga el contexto de la transacción cuando estos objetos envían mensajes a un recurso recuperable. Un servidor de transacciones no participa en la terminación de la transacción, pero puede hacer que la transacción se deshaga.

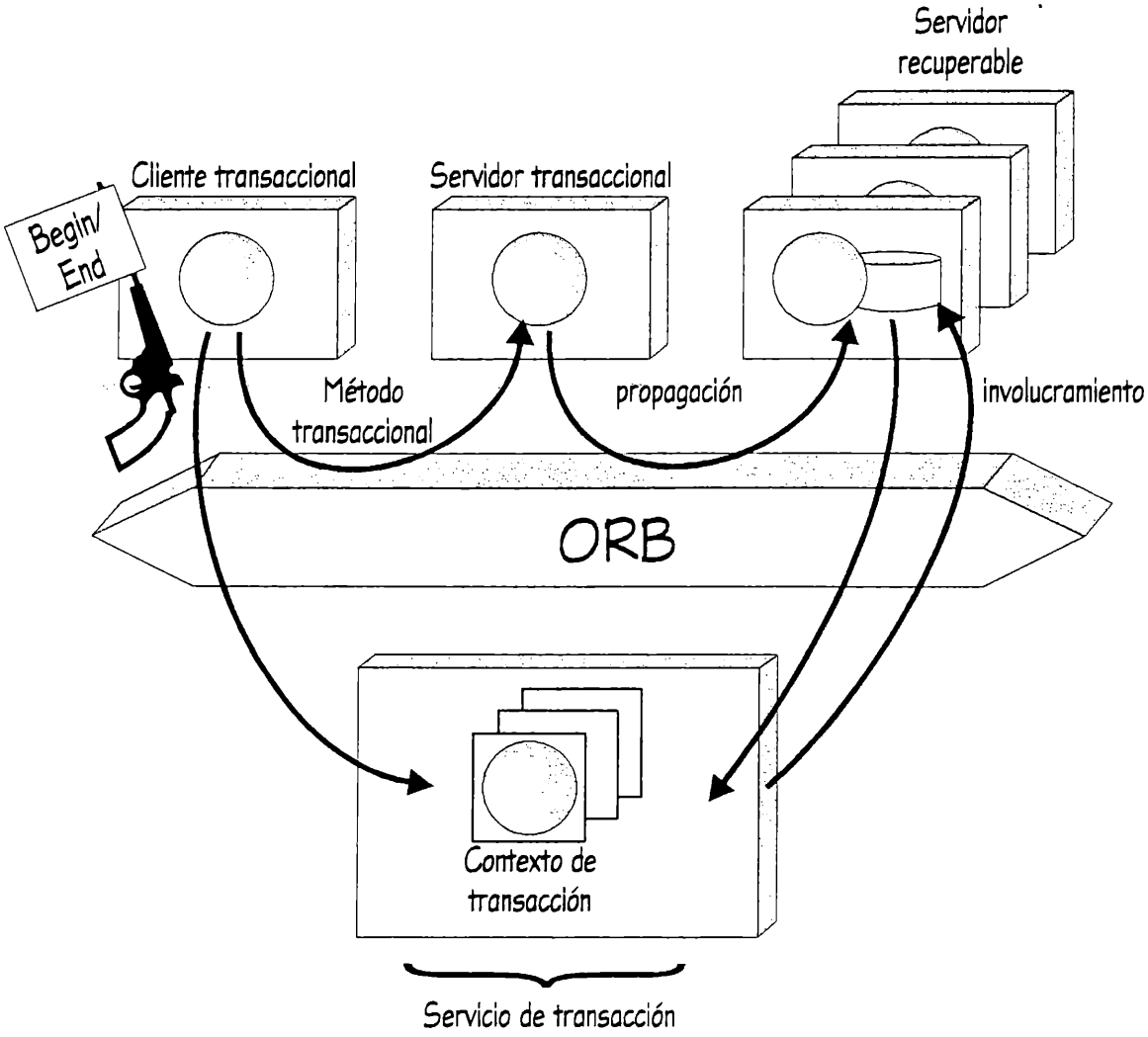


Figura 2-23. Object Transaction Service

- **Un servidor recuperable:** es una colección de uno o más objetos cuyos datos (o estado) son afectados cuando la transacción se confirma o deshace. Los *objetos recuperables* son *objetos transaccionales* que administran recursos que deben protegerse. Algunos ejemplos de recursos recuperables son: archivos, colas y bases de datos. Los objetos recuperables envían el mensaje *register_resource* al Transaction Service para indicarle que el recurso en cuestión ha sido involucrado en una transacción cuyo contexto fue propagado con la invocación de un cliente. Por otro lado, los objetos recuperables suministran métodos que se usan por un coordinador de transacciones (el

coordinador es el servicio de transacciones) para ejecutar el protocolo de dos fases entre varios ORBs.

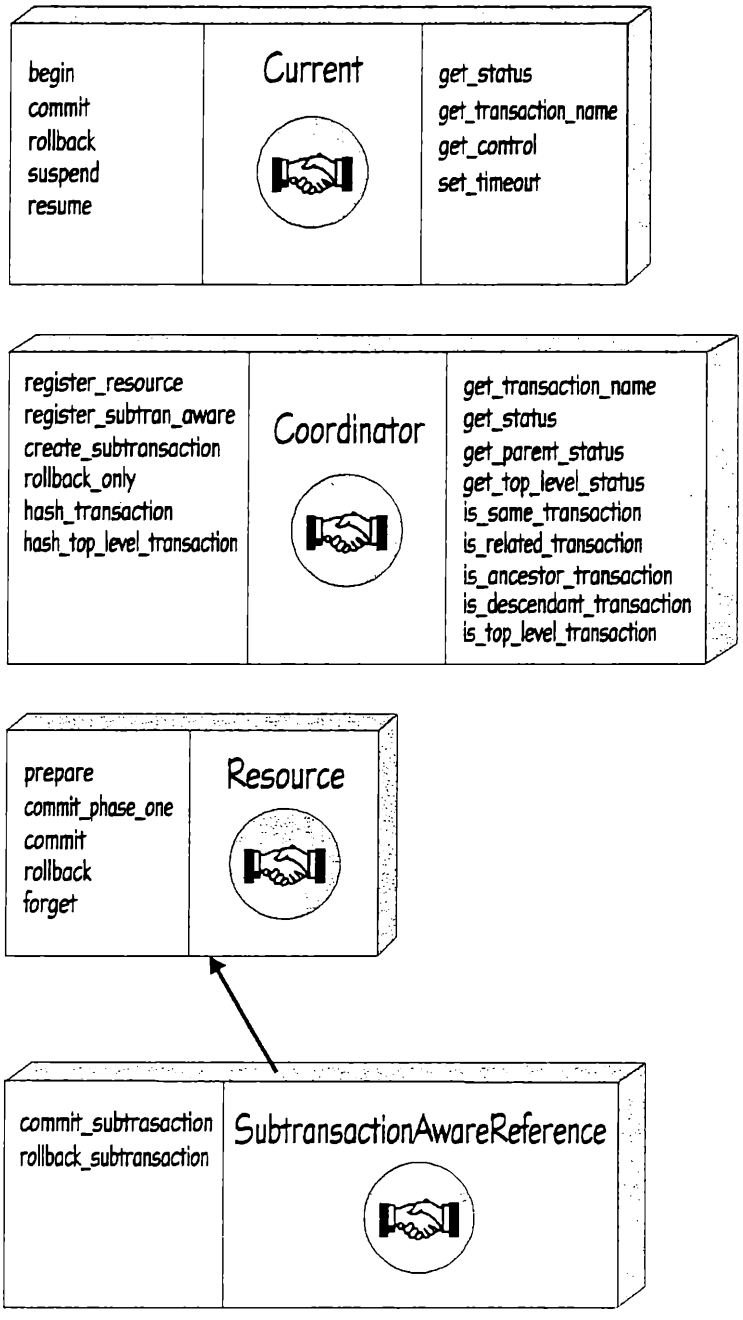


Figura 2-24. Principales interfaces del OTS

Conclusión: OTS se vale del ORB para propagar en forma automática el contexto de las transacciones. Notar que el alcance de una transacción está definido entonces por un contexto para esa transacción que es compartido por todos los objetos participantes. El contexto es nulo en los threads de ejecución que no involucran transacciones. El contexto es propagado por el servicio mediante el ORB.

Interfaces OTS

Current: define un pseudo-objeto CORBA que facilita el acceso de los clientes al OTS. Los clientes invocan *begin* y *commit* para comenzar y terminar sus transacciones.
El ORB se encarga de propagar el contexto del pseudo-objeto al OTS y a todos los objetos que participen en la transacción. El contexto tiene un ID que identifica a la transacción unívocamente, junto con cierta información de estado. El cliente invoca *rollback* para abortar la transacción; *suspend* para suspender la transacción, para que no se propague el contexto con cada mensaje enviado; puede invocar *resume* para reanudar la propagación del contexto. También hay un mensaje *set_timeout* para definir el máximo lapso de tiempo que tienen las subtransacciones para completarse antes de ser abortadas.

Coordinator: está implementada por el OTS. Es usado por los objetos recuperables para coordinar su participación con el OTS. Los servidores invocan el método *register_resource* para participar en una transacción. Invocando *create_subtransaction*, pueden crear una transacción anidada que dependa de la transacción actual. Con *rollback_only* el coordinador puede abortar la transacción completa. También hay operaciones de la forma *hash_* que retornan handlers a la transacción actual; esto es útil para que los servidores puedan verificar si un recurso dado ya está involucrado en una transacción, antes de usar *register_resource*.

Resource: está implementada por el servidor de objetos recuperables para participar en el protocolo de dos fases del commit. OTS se vale de un protocolo de dos fases para coordinar el *commit* o *rollback* de las transacciones que son ejecutadas sobre distintos server objects. Es decir, OTS centraliza la decisión de hacer el commit, pero le da a cada participante el derecho a veto.
En la primera fase del protocolo, el OTS envía el mensaje prepare a todos los participantes. Cada participante–un resource–retorna un parámetro que representa su voto, cuyo valores posibles son: *vote_commit*, *vote_rollback* o *vote_readonly*. Basándose los resultados de esta votación, el servicio dispara el commit o el rollback.

SubtransactionAwareResource: implementada por un server objetos recuperables que maneja transacciones anidadas. Esta interface se deriva de la interface ***Resource***. Agrega dos métodos nuevos métodos: *commit_subtransaction* y *rollback_transaction*. Estos métodos se invocan cuando se completan las subtransacciones. Las instancias de esta clase deben primero registrarse con el coordinator enviando el mensaje *register_subtran_aware*.

TransactionalObject: es una clase abstracta (y no define operaciones), pero agrupa a todos lo objetos transaccionales. Para agregar el manejo de transacciones a un objeto alcanza con

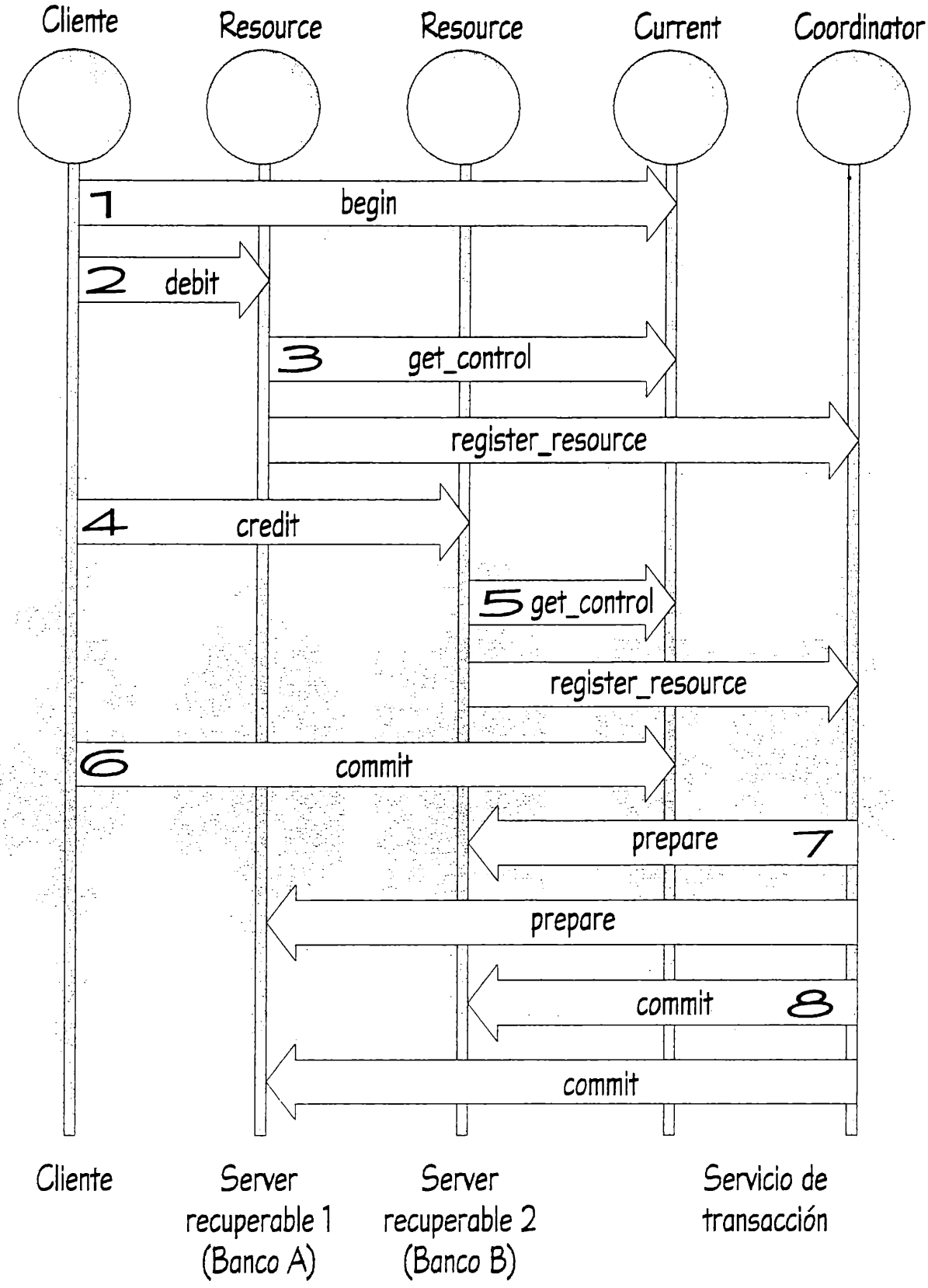


Figura 2-25. *Object Transaction Service: un escenario*

heredarlo de esta clase. El ORB automáticamente propaga el contexto de transacciones asociado a los clientes que invoquen métodos de los objetos cuyas clases sean subclases de ***TransactionalObject***.

Un escenario de transacciones

En la figura anterior se muestra el escenario de un cliente efectuando un débito en un servidor y un crédito en otro. El objetivo es "transferir $100 de la cuenta A a la cuenta B". No queremos que esto falle, ¿o si?. Los objetos que representan las cuentas A y B heredan de las clases ***Resource*** y ***TransactionalObject*** (abstracta). También tenemos 2 objetos para el servicio de transacciones que son instancias de ***Current*** y ***Coordinator***. El cliente inicia la transacción involucrando a los dos servidores y ejecutando commit cuando termina. Con suerte el dinero estará en la cuenta B.
La explicación de lo que ocurre en la figura:

1.- *El cliente inicia la transacción.* Invoca begin en el objeto ***Current***. El ORB pasa la información al servicio de transacciones que mantiene una instancia de ***Current*** para cada transacción activa.
2.- El cliente hace el débito a la cuenta bancaria A. Para esto envía el mensaje *debit* al objeto cuenta. Este objeto implementa la lógica de la transacción bancaria y además hereda de ***Resource*** y ***TransactionalObject***. Se trata de un recurso recuperable.
3.- La cuenta A se registra su recurso. El servidor recuperable primero invoca *get_control* en la instancia de ***Current*** (la notación abreviada para expresarlo es: *Current::get_control*) para obtener una referencia al objeto coordinador. Luego envía el mensaje *Coordinator::register_resource* para registrarse como participante en la transacción. El coordinador mantiene el rastro de todos los participantes.
4.- El cliente hace la operación de crédito en B. Envía el mensaje *credit* en la cuenta B. Este objeto implementa en ese método la lógica de la transacción bancaria y también hereda de ***Resource*** y ***TransactionalObject***. Nuevamente se trata de un recurso recuperable.
5.- La cuenta B registra su recurso. Es similar a la explicación de la cuenta A.
6.- El cliente dispara el commit. El cliente envía el mensaje *Current::commit*. El ORB informa al servicio de transacciones que la transacción ha terminado.
7.- El coordinador ejecuta la primera fase del commit. Envía a cada participante el mensaje *prepare* para recibir el voto de cada uno. Supongamos que queremos darle un final feliz a esto y que cada uno retorna *vote_commit*.
8.- El coordinador ejecuta la segunda fase del commit. El coordinador le indica a todos los participantes que ejecuten *commit*. Ahora tenemos los $100 en la cuenta B (y podemos invertir).

CONCURRENCY CONTROL SERVICE (CCS)

Este servicio suministra interfaces para adquirir y liberar bloqueos (locks) para permitir que distintos clientes coordinen el acceso sobre recursos que son compartidos.
Un cliente del CCS tiene 2 formas de obtener locks:

- ***Desde una transacción:*** en este caso el TS se encarga de liberar los bloqueos cuando la transacción comete o aborta. Normalmente una transacción retiene los locks hasta que termina.
- ***Como un cliente no transaccional:*** en este caso la responsabilidad de borrar los bloqueos cae sobre el cliente.

De esta forma, tenemos un único mecanismo que puede ser usado por tanto por clientes que estén dentro de transacciones como por clientes que no lo estén.

Locks

Un *lock* es una marca–token–que permite a los clientes acceder un recurso particular. La función de CS es evitar que distintos clientes posean simultáneamente locks sobre mismo recurso, si esto provoca conflictos. El servicio define distintos modos de bloqueo, que corresponden a distintas formas de acceder lo recursos:

- ***Read***
- ***Write***
- ***Intention read***
- ***Intention write***
- ***Upgrade:*** es un lock como ***Read*** pero que genera conflictos consigo mismo y se usa para evitar deadlock.

Locksets

El Servicio de Control de Concurrencia no define el nivel de granularidad de los recursos que pueden bloquearse. Sin embargo define el concepto de *locksets*. Un lockset es justamente una *colección de locks* que se asocian al mismo recurso. Si un objeto es un recurso, internamente deberá crear un lockset. Los locks se obtienen sobre los locksets. Podemos manejar locksets relacionados como un grupo.
El servicio define un *coordinador de bloqueos* (locks coordinator) que administra la liberación de locksets relacionados. Por ejemplo, el coordinador puede liberar todos los locks asociados a una transacción particular cuando la transacción efectúa el commit. Además, la coordinación puede administrar el release de un grupo de transacciones relacionadas–por ejemplo, transacciones anidadas.

Transacciones anidadas y bloqueos

La regla general es que las transacciones no deben sufrir efectos parciales de otras transacciones que luego podrían abortar. Sin embargo, el CCS debilita esta regla para las transacciones anidadas y tolera un cierto nivel de conflictos entre los locks asignados a la misma familia de transacciones anidadas. Esto es posible porque el anidamiento crea dependencias para abortar entre transacciones padres e hijas: si la transaccion padre aborta, sus hijos también, por lo cual no interesa si el hijo sufre algunos efectos de la transacción padre.
Cuando una transacción anidada pide un recurso que está bloqueado por su transacción padre, la transacción anidada pasa a ser la nueva dueña del recurso. Cuando una transacción anidada efectúa el commit–o cancela, el servicio automáticamente transfiere la propiedad de los recursos a la transacción padre. Así, una transacción hija puede adquirir un lock sobre un recurso bloquedo por su padre; y puede liberarlo sin provocar que su padre los pierda.

Interfaces para el control de la concurrencia

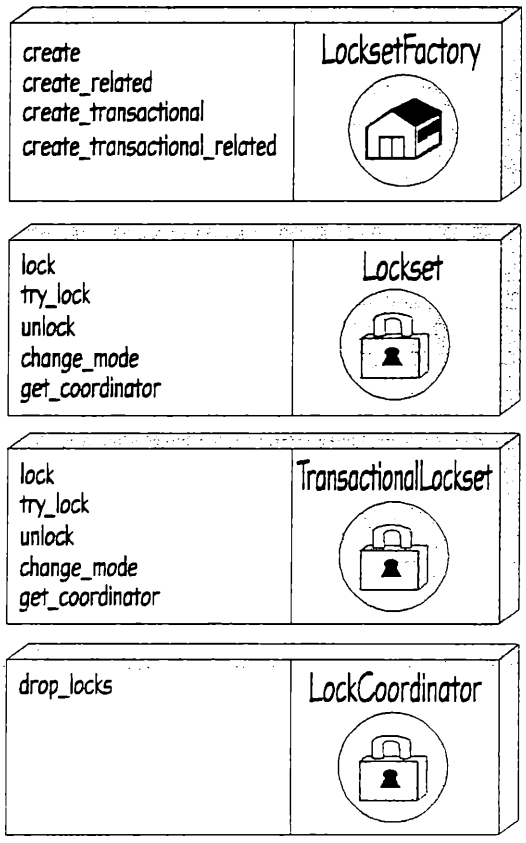


Figura 2-26. Interfaces del Concurrency Control Service

- ***LocksetFactory:*** permite la creación de los locksets. Con *create* podemos crear un lockset común y con *create_transactional* creamos un lockset transaccional. Para crear nuevos locksets relacionados con otros ya existentes podemos usar: *create_related* y

create_transactional_related. Cuando los lockset están relacionados liberan sus recursos en forma conjunta.

- ***Lockset.*** Permite obtener y liberar locks. Con *lock*, el cliente obtiene el lock o bien queda trabado hasta lograrlo. Si no queremos trabarnos, usamos *try_lock*–el control retorna al cliente si el lock no está disponible. Finalmente con ***unlock*** liberamos el lock. Con *change_mode* se cambia el modo de bloqueo.
- ***TransactionalLockSet:*** es igual a ***Lockset***, pero requiere el ID de la transacción a la que pertenece el cliente en forma explícita. En cambio, ***Lockset*** usa el contexto de la transacción asociada al thread del cliente en forma implícita. Así, ***Lockset*** soporta clientes transaccionales y clientes no transaccionales–el contexto de transacción es nulo.
- ***LockCoordinator:*** OTS lo usa para liberar todos los locks de una transacción cuando ésta comete o cancela–aborta.

"La unión de la tecnología de los ORBs y los Monitores de Procesamiento de Transacciones creará una nueva infraestructura capaz de explotar las virtudes de los monitores y suministrar un ambiente de desarrollo de aplicaciones basadas en objetos reusables. Realmente es una alternativa cliente/servidor superior."

- Edward E. Cobb, IBM
Senior Technical Staff Member
Object Magazine (Febrero de 1995)

SERVICIOS DE CORBA: PERSISTENCE Y OBJECT DATABASES

"La siguiente generación de software consistirá en componentes vinculados a una gran variedad de productos para bases de datos. Por primera vez, el Persistent Object Service (POS) suministra una interface a los clientes para almacenar objetos, más allá de que los objetos se guarden en sistemas de archivos, contenedores Bento, bases de datos relacionales o bases de datos de objetos."

- Roger Sessions, Author
Object Persistence
(Prentice Hall, 1996)

A diferencia de los objetos tradicionales (C++, Smalltalk), los objetos distribuidos son persistentes. Esto significa que deben mantener su estado interno después que el programa o aplicación que los crea termina. Para ser persistente, el estado del objeto debe ser almacenado en un medio no volatil–por ejemplo bases de datos relacionales o sistemas basados en archivos. ¿Quien controla la persistencia de cada objeto?. Por un lado, los vendedores de ODBMSs creen que la persistencia debe ser completamente transparente. Un OBDMS resuelve el movimiento de los objetos entre el disco y la memoria, según sea necesario. Todo lo que vemos es un almacenamiento de un solo nivel para los objetos.
Pero en el otro extremo del espectro están aquellos que hablan de los terabytes de datos ya almacenados sobre otros medios (léase bases relacionales).
Asi pues, necesitamos una manera de lograr transformar la informacion almacenada en los medios actuales en objetos (y viceversa). Para esto, necesitamos involucrarnos con los mecanismos que permitan acceder a esos datos y asociarlos al estado de los objetos. En el medio del espectro están aquellos que usan archivos planos para salvar los datos. En ese caso se requiere un servicio de streaming que exporte e importe los datos de los objetos a y desde archivos.
A continuación veremos en que consiste el Persistent Objet Service (POS). Este servicio permite que cada objeto persista más allá del tiempo de vida de la aplicación que lo creo o los clientes que lo usen. El tiempo de vida de un objeto es independiente de eso y puede ser muy corto o indefinido. El POS permite que el estado de los objetos sea salvado sobre un medio persistente para ser restaurado cuando sea necesario.

Almacenamientos de un nivel vs. almacenamientos de dos niveles

El POS ofrece una interface cliente para almacenar objetos más allá del medio o mecanismo usado para el almacenamiento. El POS puede tratar tanto con almacenamientos de un nivel (single-level stores)–por ejemplo los ODBMSs–como con almacenamientos de dos niveles (two-level stores)–por ejemplo: las bases de datos SQL, sistemas basados en archivos, etc.

En un sistema de un solo nivel, el cliente es incapaz de determinar si el objeto está en memoria o disco. Es un almacenamiento virtual gigantesco–la memoria y el medio persistente son la misma cosa. Y aunque los ODBMSs usan caches y otras técnicas para el manejo de transacciones, todo ocurre en forma transparente.
En contraposición, los mecanismos de almacenamiento de dos niveles separan y diferencian la memoria del medio persistente. El objeto debe ser cargado explícitamente desde la base de datos (o archivo) a memoria, y viceversa.

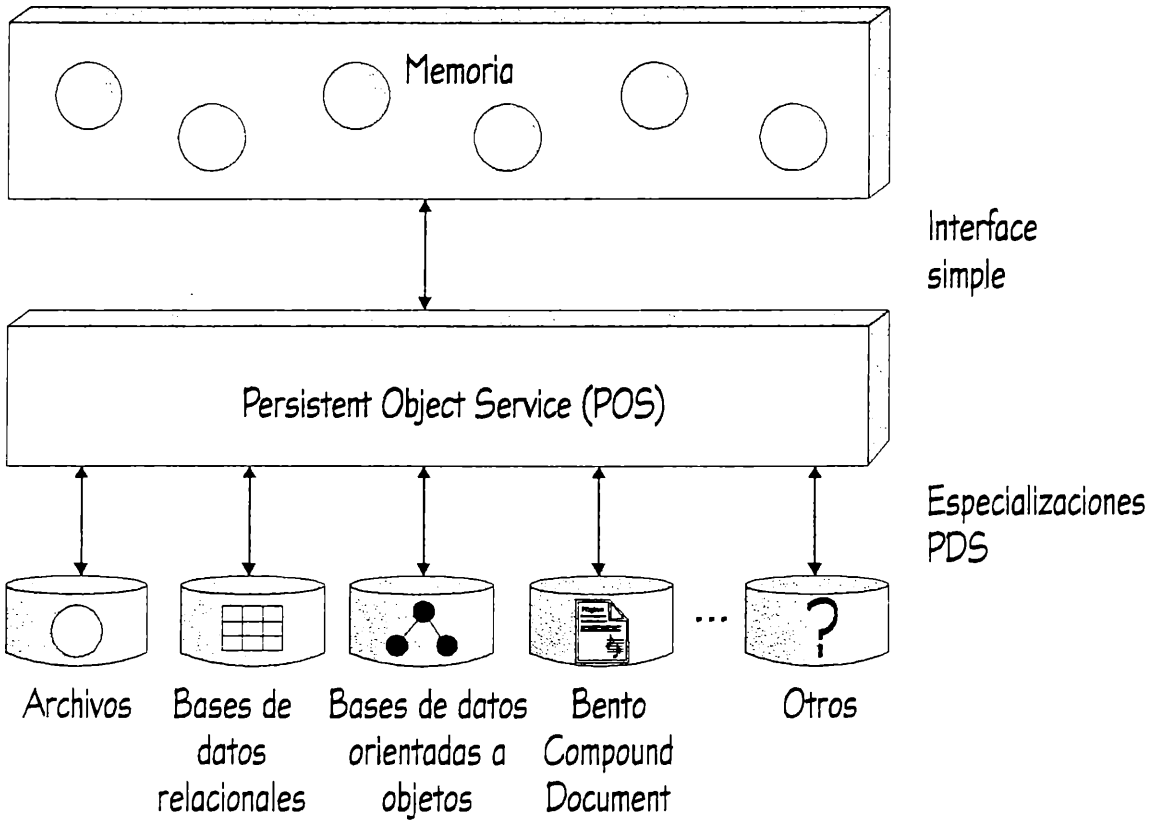


Figura 2-27. Objetos persistentes: una interface para diferentes medios de almacenamiento

La mayoria de la información ya disponible en el mundo está almacenada con mecanismos de dos niveles. En un mundo ideal, todos los objetos seguramente se almacenarían en ODBMSs, que suministran un solo nivel. La ventaja de esto es que no se introduce un nivel extra de indirección entre los objetos y su medio de almacenamiento.

POS: la visión del cliente

Algunas veces, los clientes de un objeto necesitan controlar o intervenir en el manejo de la persistencia. POS permite que los clientes puedan involucrarse a diferentes niveles.
En uno de los extermos, el servicio puede ser completamente transparente a las aplicaciones que lo usan. Esto significa que el cliente ignora por completo los mecanismos de

persistencia; los objetos aparecen mágicamente bajo demanda, sin importar el estado en que se hallen. En el otro extremo, las aplicaciones pueden usar protocolos específicos para controlar todos los detalles del mecanismo de persistencia subyacente. Nuevamente, la idea es ofrecer soluciones a las demandas de los clientes. Algunos clientes necesitan un control fino y preciso del medio de almacenamiento, mientras que para otros la ignorancia de esos detalles es indiferente.

Los cliente eligen que nivel de control quieren tener. POS suministra nueve operaciones definidas en tres clases para que los clientes puedan controlar la persistencia de los objetos. Estas interfaces no violan los principios de encapsulamiento de los objetos, pero nos dan cierta visibilidad de ellos. Más específicamente, permiten que el cliente elija cuándo los objetos se almacenan y recuperan.

POS: La visión de los objetos persistentes

En última instancia, son los objetos persistentes (PO) quienes están a cargo de sus persistencia; ellos deciden cuál protocolo usar y cuánta visibilidad permiten a los clientes. Un PO también puede delegar el manejo de su información persistente a algún servicio externo o bien mantener un control fino y preciso sobre todo el proceso de interacción con el sistema de almacenamiento. Y Finalmente, un PO puede heredar toda la funcionalidad relacionada con la persistencia.
Un PO debe colaborar con los sistemas de almacenamiento para permitir que el servicio de persistencia puede representar su estado interno en un formato apropiado.

Los elementos que componen el POS

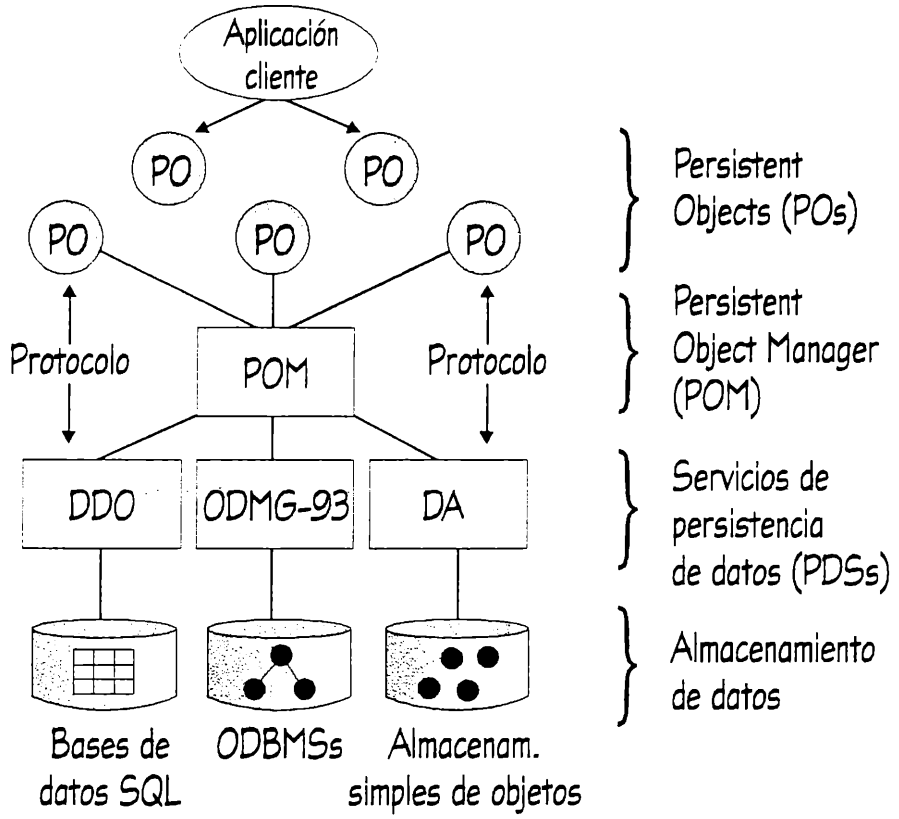


Figura 2-28. Componentes del Persistent Object Service

- ***Persistent Objects (POs):*** Son los objetos cuyo estado nos interesa hacer persistente. Un objeto puede ser persistente heredando la funcionalidad (vía IDL) de la clase PO. También debe heredar (o suministrar) un mecanismo para exportar su estado cuando le sea requerido desde algún mecanismo de almacenamiento (via algún protocolo). Todos los objetos persistentes tienen un identificador persistente (PID) que describe la localización del objeto en un sistema de almacenamiento, con un string. Normalmente, los clientes interactúan con la interface de PO para controlar la persistencia de los objetos.
- ***Persistent Object Manager (POM):*** Se trata de una interface independiente de la implementación que reúne varias operaciones relacionadas con la persistencia. Aisla los POs de un Persistent Data Service particular. El POM se encarga de redirigir la llamada hacia un Persistent Data Service particular analizando la información codificada en el PID. Hay un POM para cada Persistent Data Service disponible. Con esto se logra presentar una visión uniforme de la persistencia por encima de todos los servicios que la implementen.
- ***Persistent Data Services (PDSs).*** Son las interfaces de las implementaciones concretas. La función de los PDSs es mover los datos desde los objetos hacia los medios de

almacenamiento. Todos los PDSs deben implementar esta interface definida en IDL, que es general. Sin embargo, algunos pueden agregar soporte para su propio protocolo, dependiente de la implementación. Este protocolo es un mecanismo para sacar e ingresar datos desde y hacia los objetos. OMG se refiere a estos protocolos como *conspiraciones* (conspiracies) entre los objetos y los PDSs. Estas conspiraciones son directas entre los objetos y los ODBMSs, pero pueden ser bastante complejas cuando se debe hacer el mapeo entre objetos y modelos de almacenamiento de dos niveles (ejemplo: usando bases de datos SQL). POS define–por ahora–tres protocolos para administrar estas conspiraciones: *Direct Attribute* (DA), *Object Database Management Group* (ODMG-93) y *Dynamic Data Object* (DDO).

- ***Almacenamientos de datos:*** Guardan el estado de los objetos en forma persistente, independientemente del espacio de direcciones donde viva el objeto. Algunos ejemplo de alamacenamientos son los ODBMSs, los archivos, Bento y las bases de datos SQL.

Como conclusión podemos decir que POS define tres niveles diferentes de abstracción para ocultar la implementación del almacenamiento de los objetos. Para la mayoría de las aplicaciones clientes, el mecanismo de persistencia será totalmente transparente. Si se requiere algún control sobre la persistencia, se usan las interfaces de PO. Por supuesto, cada objeto es responsable de almacenar y restaurar su estado. Sólo podemos indicarle cuándo hacerlo, con la interface de PO.

Los protocolos de POS: la conspiración entre los objetos y PDS

POM suministra interfaces genéricas que permiten que distintos PDSs puedan añadirse en forma transparente. Los medios de almcenamiento ofrecen una única interface al POM denominada PDS. Pero además agregan protocolos específicos para cada implementación. Estos protocolos actúan como *conspiraciones* que permiten que un objeto exponga su estado a los mecanismos de almacenamiento. Podemos verlo como un acuerdo entre los desarrollores de objetos y los implementadores de PDS respecto a la manera de sacar y meter datos de y en los objetos.

Algunos investigadores opinan que si todos los objetos usaran el protocolo de streams (que es una parte del CORBA Externalization Service), entonces los PDSs de dos niveles solamente necesitarían dar soporte para una sola conspiración. Pero además de los streams, POS define tres protocolos específicos para mover datos entre un objeto y distintos PDSs.

- ***Direct Access (DA):*** este protocolo se basa en la propuesta hecha por SunSoft y permite el acceso directo a los datos usando un lenguaje parecido al IDL.
- ***ODMG-93:*** este protocolo suministra acceso directo desde C++ usando el *Data Definition Language* específico de ODMG. Lo veremos en detalle más adelante.
- ***Dynamic Data Object (DDO):*** este protocolo permite una representación neutral e independiente del medio de almacenamiento de los datos persistentes del objeto; define una estructura que contiene todo el estado persistente del objeto. Es posible usar este protocolo para decribir un objeto dinámicamente sin utilizar IDL. También sirve para formular consultas dinámicos. Básicamente resuelve el acceso a las bases de datos SQL.

Podemos concluir que los protocolos solamente se encargan del mapeo entre los objetos y los medios de almacenamiento. Los ODBMSs usan un superconjunto del IDL para describir los objetos; los representan directamente, sin requerir conversiones intermedias. Por esta razón prácticamente no existen conspiraciones entre los objetos y los ODBMSs.
En cambio, este no es el caso de los bases de datos SQL y otros medios de almacenamiento: se requieren conversiones intermedias. Como consecuencia, necesitamos conspiraciones para lograr un intercambio en ambos sentidos. Pero realmente es un precio bajo si a cambio logramos traer al dominio de los objetos toda la información ya almacenada y disponible en el mundo.

Las interfaces del POS

Los clientes sólo necesitan conocer la definición de las fábricas, PO y PID. En la figura vemos que PO, POM y PDS suministran las mismas operaciones, pero a diferentes niveles de abstracción.

- ***PIDFactory:*** permite crear un PID de tres maneras. Todo objeto requiere tener un PID para poder registrar su estado en forma permanente. Este PID describe dónde viven los datos que conforman el estado del objeto.
- ***POFactory:*** suministra un única operación, *create_PO* para crear una instancia de un objeto persistente.
- ***PID:*** define una sola operación, *get_PIDString* que retorna un string que representa al PID.
- ***PO:*** suministra cinco operaciones para que los clintes puedan controlar externamente la relacion entre un objeto y sus datos persistentes. Con *store* el estado del objeto es guardado en un medio de almacenamiento, en la posición codificada en el PID. Con *restore* obtenemos el efecto opuesto. Todos los objetos persistentes se derivan de esta interface.
- ***POM*** suministra cinco operaciones para que los objetos persistentes puedan comunicarse con el medio de almacenamiento subyacente; son las mismas operaciones que los objetos persistentes exponen a sus clientes.
- ***PDS:*** ofrece las cinco operaciones que un ***POM*** usa para comunicarse con los medios de almacenamiento subyacentes; se trata de las mismas operaciones que los objetos persistentes exponen a sus clientes. ***PDS*** es la interface de nivel más bajo que maneja el movimiento de datos entre un objeto y el medio de almacenamiento. Para incluir un nuevo medio de almacenamiento en el Persistent Object Service, debemos especializar las clases ***PDS*** y ***PID*** (y definir los protocolos).

Las interfaces ***PS CLI*** encapsulan los estándares X/Open CLI, IDAPI, y las funciones ODBC. Estos tres estándares permiten el acceso a bases de datos relacionales, sistemas de archivos XBase e incluso bases de datos jerárquicas.

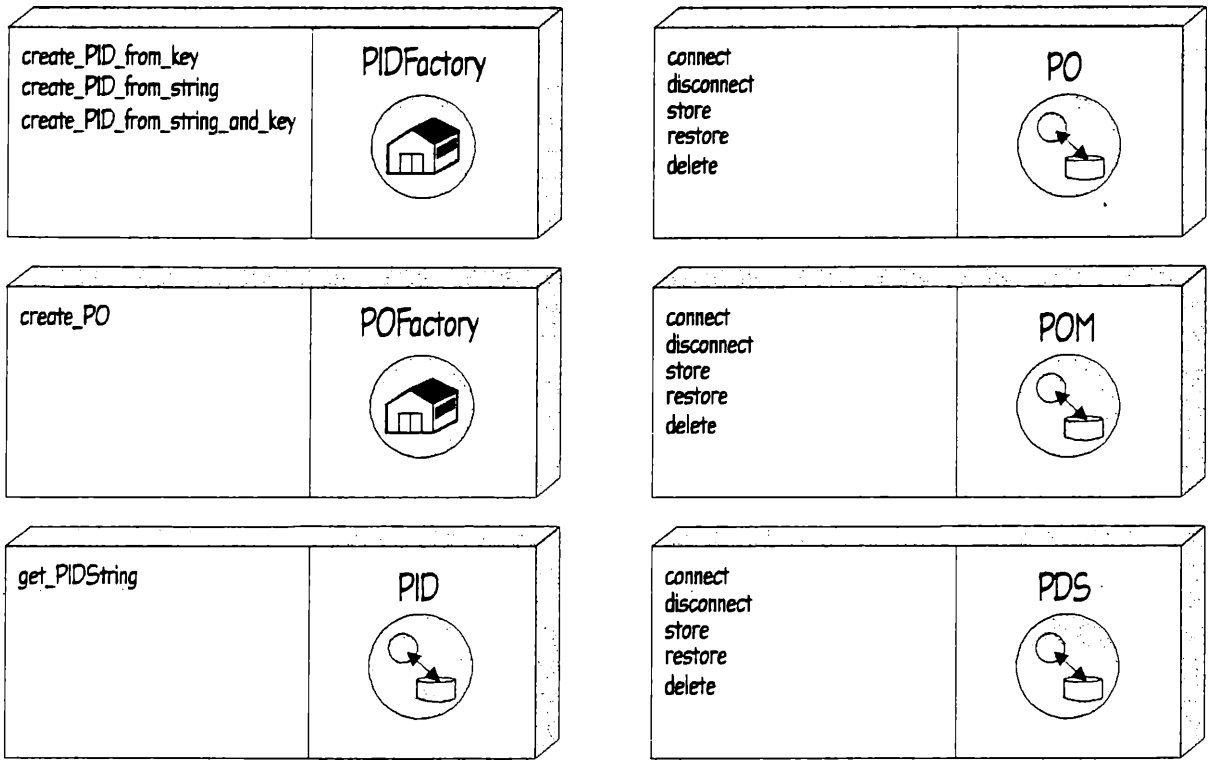


Figura 2-29. Interfaces del Persistent Object Service

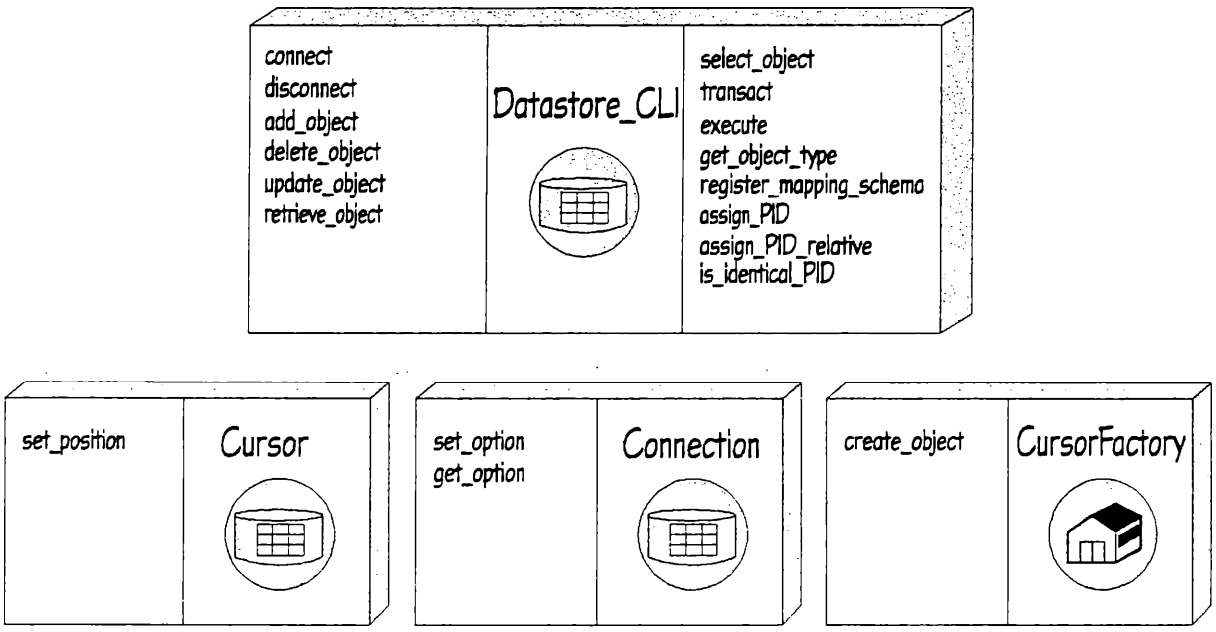


Figura 2-30. Las pricipales interfaces POS CLI

ODBMS: SISTEMAS DE BASES DE DATOS DE OBJETOS

"El mayor problema con las bases de datos relacionales es que trabajan con un nivel intermedio de abstracción de datos. Los usurios esperan navegar por el mundo real y registrar las transacciones ... esto requiere agregar un nuevo esquema, una interface de programación de aplicaciones de mayor nivel. La manera en que estamos resolviendo esto [en Oracle8] es descomponiendo las datos de los objetos y almacenándolos en tablas bidimensionales."

- Larry Ellison, CEO de Oracle
(Mayo de 1994)

Los ODBMS suministran una arquitectura cliente/servidor significativamente distinta a la que ofrecen los sistemas de bases de datos relacionales (RDBMSs). Los ODBMSs toman un enfoque que es completamente revolucionario para compartir los datos y se centra en el manejo de objetos persistentes. Hay un poco de esta tecnología en servicio de persistencia de objetos del OMG.
Hoy en día, la mayoría de los desarrolladores de ODBMSs ofrecen acceso eficiente a los objetos, en todos los niveles de granularidad.
Hacia fines de 1993, los principales desarrolladores de ODBMS liberaron el estándar *ODMG-93*. La intención con el estándar es que se convierta en el SQL de los ODBMS. Por supuesto, con un estándar común, el mercado de los ODBMSs podría despegar y competir con los sistemas SQL.

¿Qué es un ODBMS?

"Definimos a un ODBMS como un DBMS que integra todas las capacidades de las bases de datos y los lenguajes de programación orientada a objetos. Un ODBMS permite que los objetos almacenados en una basses de datos se presenten como objetos comunes, disponibles en los lenguajes de programación."

- Rick Cattel, Chairman
ODMG-93 Committee

Un ODBMS ofrece un medio persistente para almacenar objetos en un entorno cliente/servidor multiusuario. El ODBMS se encarga de manejar el acceso concurrente a los objetos, suministra locks, protección en las transacciones y además realiza automáticamente tareas comunes y tradicionales como backup y restauración después de una caída.
Lo que establece las diferencias de un ODBMS cuando se lo compara los sistemas relacionales es que los primeros almacenan objetos, en lugar de filas de datos sobre tablas. Los objetos se referencian entre si con un PID, que identifica en forma unívoca a cada objeto. Estos PID permiten crear restricciones de integridad y relaciones de contención entre los objetos. Los ODBMS también respetan el encapsulamiento y soportan herencia. Podemos decir que los ODBMS combinan todas las propiedades de los objetos con la

funcionalidad tradicional de los DBMSs–como por ejemplo: locks, protección, transacciones, consultas, versiones, concurencia y persistencia.

En lugar de usar un lenguaje separado como SQL para definir, recuperar y manipular los datos, los ODBMS definen clases y se valen de constructores disponibles en los lenguajes de programación orientados a objetos (C++ y Smalltalk) para definir y acceder al información. Un sistema ODBMS es simplemente una extensión multiusuaria y persistente de las estructuras manejadas en memoria por los lenguajes de programación. En otras palabras, el cliente es un programa escrito en algún lenguaje y el servidor es el ODBMS–no hay intermediarios visibles como SQL y RPC. Los ODBMS integran laas capacidades de las bases de datos directamente en los lenguajes.

¿Para qué sirve un ODBMS?

"En lugar de tratar de separar los datos para acomodarlos en tablas relacionales es mejor almacenarlos en su forma natural."

- Jonathan Cassell, IS Manager
Granite Construction
(Julio de 1995)

Durante mucho tiempo, los ODBMS eran un área de investigación e interés para los ambientes académicos. Los primeros sistemas comerciales aparecieron recién hacia fines de los años '80. Estos sistemas estaban diseñados para aplicaciones que trataban con tipos complejos de datos y transacciones–incluyendo diseño asistido por computadora, CASE, etc. Con el surgimiento de la multimedia, groupware y los objetos distribuidos, los sistemas Los ODBMSs están comenzando a ser considerados para las aplicaciones cliente/servidor. Podemos decir que la tecnología de los ODBMSs llena el gap existente entre las bases de datos relacionales y su principal debilidad–los tipos complejos de datos, versiones, transacciones de larga duración, transacciones anidadas, almacenamiento de objetos persistentes, herencia y tipos de datos definidos por el usuario. Las principales ventajas son:

- ***Libertad para crear nuevos tipos de datos:*** Los ODBMS nos permiten crear y almacenar nuevos tipos de datos usando mecanismos estándares para crear objetos: herencia y definición de clases.
- ***Acceso rápido.*** Los ODBMS permiten identificar a los objetos a través de su PID. Así, es posible navegar entre los objetos sin necesidad de operaciones de comparación basadas en claves externas u otras técnicas asociativas (modelo relacional).
- ***Visión flexible de las estructuras de composición:*** Los ODBMS permiten que objetos individuales participen en cualquier relación de contención, creando diferentes vistas y maneras de acceder al mismo objeto. Los objetos pueden manejar referencias a otros objetos de manera recursiva.
- ***Integración con lenguajes de programación orientada a objetos.*** Los ODBMSs se presentan a si mismos como extensiones a los lenguajes de programación orientados a objetos. Con esto se logra minimizar la impedancia entre los programas y los datos

persistentes, al mismo tiempo que se preservan todas las características de herencia, encapsulamiento, etc. Los ODBMSs permiten un acceso rápido y ágil a los objetos almacenados. En contraposición, los RDBMSs requieren varias transformaciones para representar los objetos tal como están en la memoria en una forma tabular. Los sistemas relacionales pueden almacenar objetos, pero primero deben desinflarlos y descomponerlos en estructuras más simples que se puedan guardar en las tablas..

- ***Soporte para estructuras de información personalizables usando herencia múltiple.:*** con un ODBMS los tipos de datos se definen con clases. Es decir que podemos especializar las clases ya existentes para obtener nuevas representaciones de la información. Los ODBMSs extienden hacia la base de datos el concepto del reuso de objetos a través de la herencia.
- ***Depósitos para almacenar objetos distribuidos.***
- ***Soporte para la administración del ciclo de vida de objetos compuestos:*** los ODBMS también han perfeccionado el arte de manejar objetos compuestos como una unidad. Por ejemplo, podemos ensamblar, desensamblar, almacenar, copiar, mover, eliminar y restaurar objetos compuestos. El ODBMS se encarga de mantener automáticamente las relaciones entre las partes que componen un objeto y trata al todo como una unidad. Esto es el resultado de la maduración por la que pasaron cuando se utilizaban en CAD.

ODMG-93: la lengua franca para los ODBMSs

El estándar *ODMG-93* es la respuesta ODBMS a SQL. Este estándar es el resultado del trabajo hecho por el Object Database Management Group (ODMG)–un grupo de empresas que incluye a los principales desarrolladores de ODBMSs. El ODMG es un subgrupo del OMG y tiene el objetivo de enviar el estándar a ISO y ANSI. En teoría, la adopción de ODMG-93 debería permitir que las aplicaciones funcionen con ODBMS de cualquier fabricante que adhiera al estándar. Sin embargo, todas las implementaciones cumplen sólo con una parte–aunque grande–de toda la especificación.

ODMG-93 y CORBA

ODMG-93 es una extensión del Persistent Object Service de CORBA y define cómo implementar un protocolo que suministre un PDS (Persistent Data Service) eficiente para la administración de objetos de granularidad fina. El estándar se basa en el modelo de objetos del OMG. El rol de un ODBMS en un ambiente ORB es ofrecer acceso concurrente a medios de almacenamiento capaces de administrar millones de objetos. Para lograr su cometido, el OMG hace referencia a un protocolo PDS especial denominado OMG-93. Este protocolo reemplaza a las invocaciones RPC definidas con IDL por llamadas a APIs directas del medio de almacenamiento de objetos. Esto agiliza el acceso a los datos.

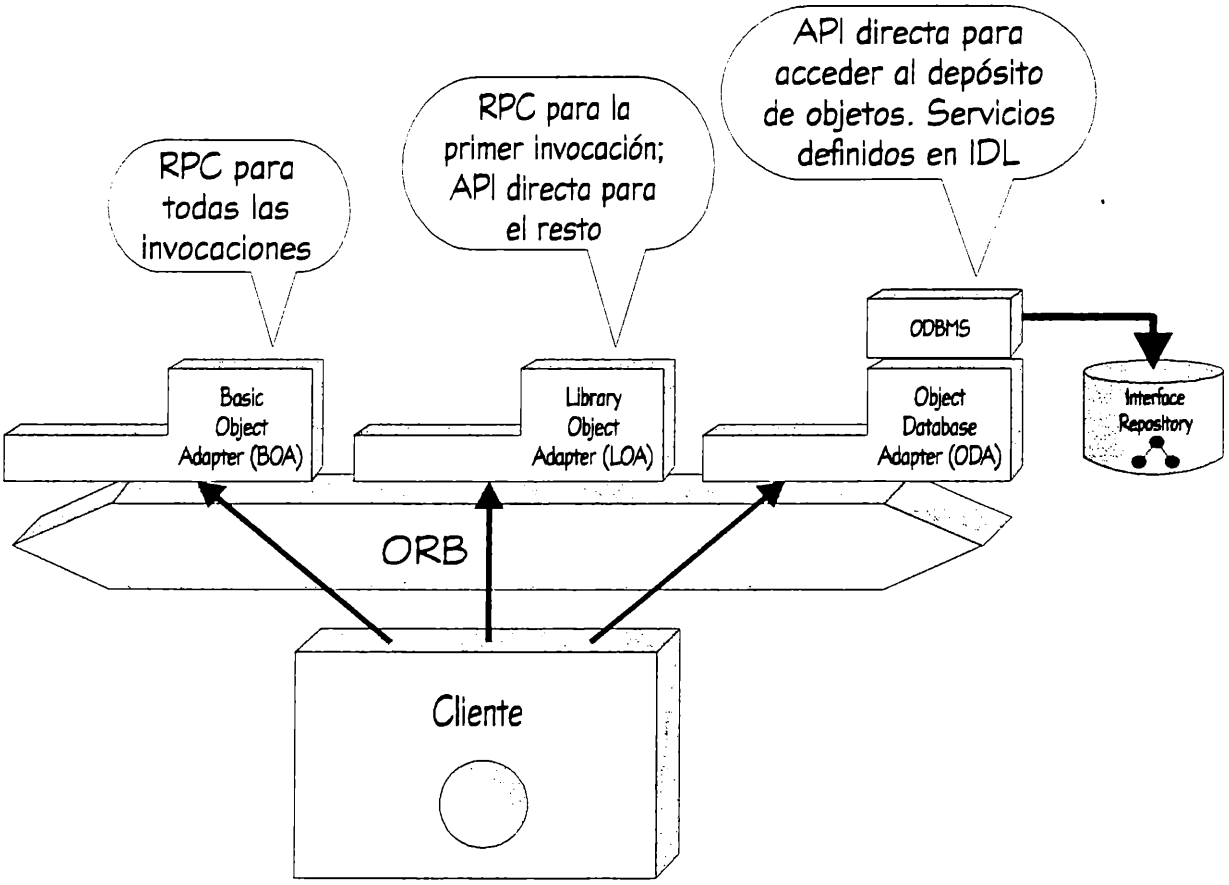


Figura 2-31. ODMG-93 en un ORB

Los desarrolladores de ODBMS están promoviendo activamente en el comité que trabaja con el ORB de CORBA un adapter específico–Library Object Adapter (LOA) que ofrezca acceso directo a las APIs especializadas a través del ORB. Más aún, en la definición del ODMG-93 se afirma que CORBA debería asimilar ese adapter como propio e incorporarlo como el Object Database Adapter (ODA). En la figura vemos las diferencias entre el BOA, LOA y ODA.

El ODA debería ofrecer la máxima flexibilidad posible para registrar subespacios de identificadores de objetos en el ORB, en lugar registrar los millones de objetos almacenados en el ODBMS. Del lado del cliente, estos objetos registrados en pequeños subespacios se presentan de manera similar a cualquier objeto accesible por el ORB.
El ODA facilita la administración y permite un acceso más rápido–como con el LOA–para mejorar el rendimiento de las aplicaciones que usan ODBMSs.
Sintetizando, podemos decir que los fabricantes de ODBMSs están empujando a CORBA a que sea más flexible en lo referido al tratamiento de aplicaciones que manejan millones de objetos de granularidad fina. La especificación del POS indica que el OMG entendió el mensaje, pero resta saber si extenderán la flexibilidad para que sea suministrada por el ORB mismo.

Los elementos del ODMG-93

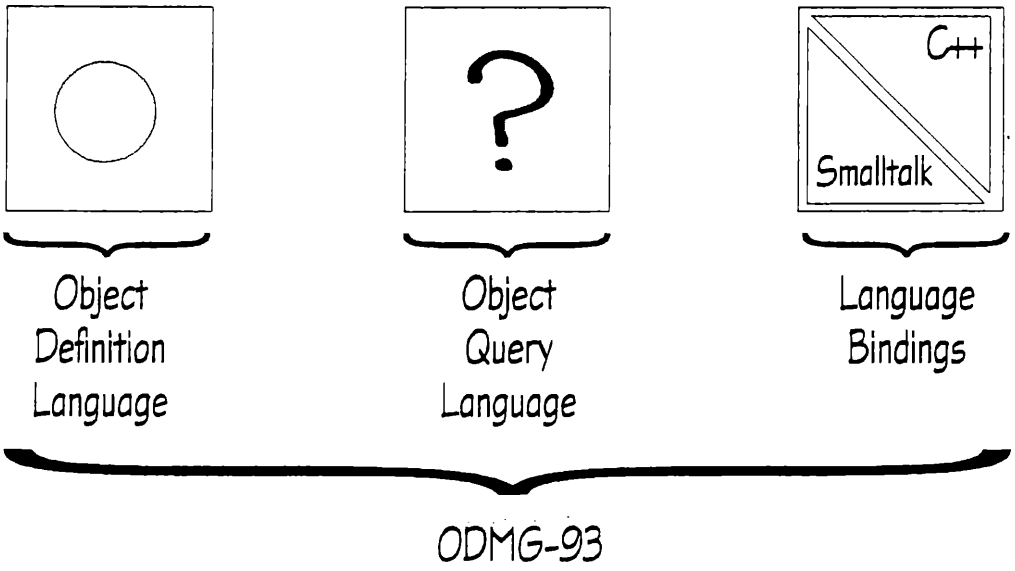


Figura 2-32. Componentes de ODMG-93

- ***Object Definition Language (ODL):*** ODMG-93 utiliza al IDL como lenguaje para la definición de los datos. ODL es un super-conjunto del IDL en el sentido que define elementos que no pertenecen a éste, como las clases de las colecciones y relaciones basadas en referencias entre objetos. El ODL permite describir la metainformación independientemente del lenguaje de programación y se procesa con un precompilador que genera los stubs que permiten enlazar el ODBMS y el lenguaje de programación (C++ o Smalltalk). ODL suministra portabilidad a través de los lenguajes y ODBMS para la definición de datos e interfaces
- ***Object Query Language (OQL):*** OMGD-93 define un lenguaje declarativo–al estilo de SQL–para formular consultas y actualizar los objetos. Soporta todas las formas comunes de la sentencia *SELECT*, incluyendo joins; pero no soporta *UPDATE*, *INSERT* o *DELETE* (para esto se usan las extensiones hechas a C++ y Smalltalk). ODMG-93 intencionalmente no se basó en la semántica de SQL3 para los objetos por *"las limitaciones en el modelo de datos y la pesada carga histórica"*. Sin embargo, veremos que OQL y SQL3 convergen hacia la misma dirección. OQL ofrece primitivas de alto nivel para consultar sobre colecciones de objetos–incluyendo *sets*, que son colecciones desordenadas sin elementos repetidos; *bags*, que son colecciones desordenadas que admiten elementos repetidos; y *lists* que son colecciones ordenadas de elementos.

- ***Bindings a los lenguajes C++ y Smalltalk:*** ODMG-93 define cómo escribir código portable en C++ y Smalltalk para manipular objetos persistentes. El estándar define las extesiones en C++ para el Object Manipulation Language (OML). El OML para C++ incluye extensiones para OQL, iteradores para navegar a través de los containers de objetos y soporte para transacciones. ODMG no cree en un lenguaje *"universal"* para la manipulación de los datos (como SQL). En cambio propone un *"modelo unificado de objetos para compartir datos a través de los diferentes lenguajes de programación ya disponibles, a la par de un lenguaje común de consulta"*. El objetivo es *"lograr que OML respete la sintaxis del lenguaje base sobre el que es insertado. Esto permite a los programadores sentir que están trabajando con un único lenguaje de programación que integra el soporte para la persistencia"*. En teoría, debería ser posible leer y escribir información al mismo ODBMS desde C++ y Smalltalk, siempre y cuando los programadores se manejen con los mismos subconjuntos de tipos de datos y definiciones soportadas.

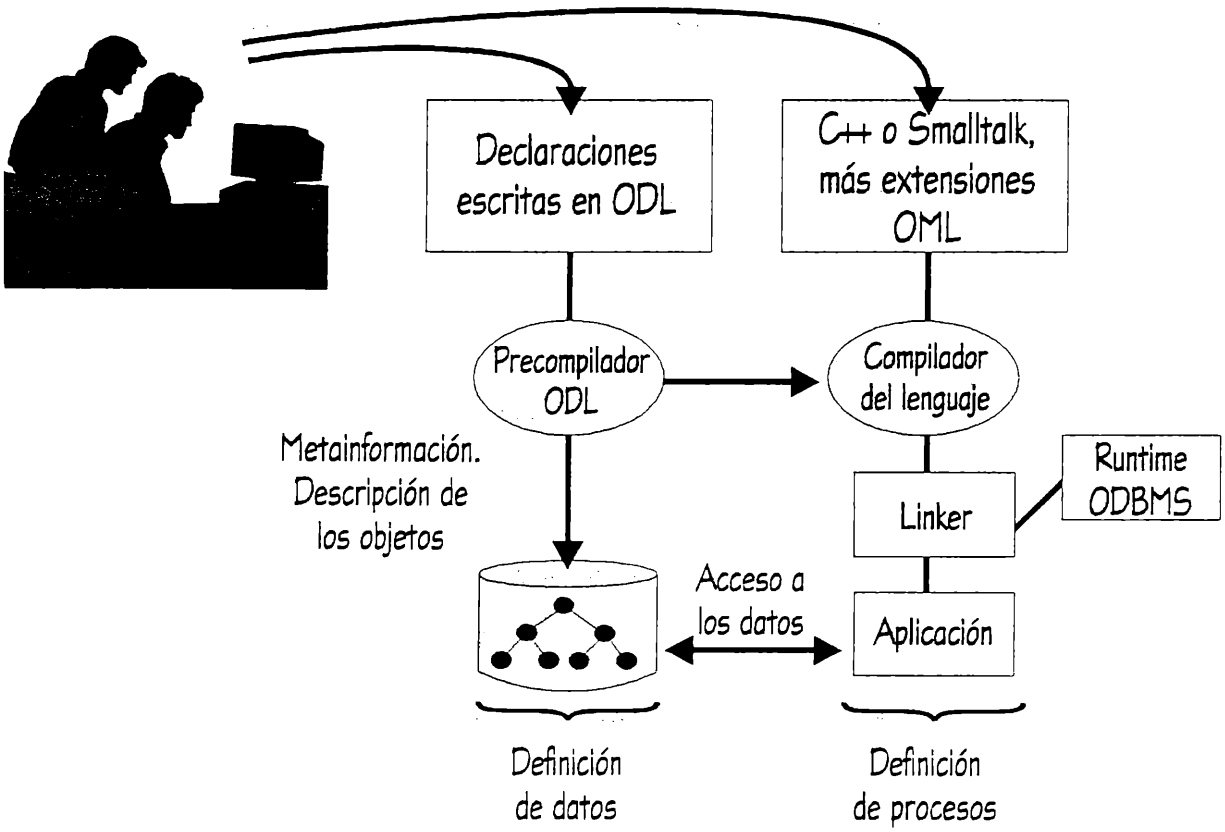


Figura 2-33. Procesos del ODMG-93

En la figura mostramos la secuencia de pasos necesarios para usar un ODBMS compatible con ODMG. El proceso es similar al uso de IDL, excepto que los bindings a los stubs son para el ODBMS y la aplicación escrita en un lenguaje de programación orientada a objetos.

SERVICIOS DE CORBA: QUERY Y RELATIONSHIP

"Los objetos distribuidos no flotan en el espacio, sino que están conectados unos con otros."

- CORBA COSS Specification
(Marzo de 1995)

QUERY SERVICE

El Servicio de Consulta de Objetos (Object Query Service) permite encontrar objetos cuyos atributos concuerdan a los criterios de búsqueda especificados en una consulta. Debemos comenzar señalando que estas consultas nunca tienen acceso al estado interno de los objetos. Esto significa que no violan el encapsulamiento del estado interno.
Es posible formular las consultas usando varios lenguajes: OQL de ODMG-93, SQL (con extensiones a objetos) o bien alguna variante que represente un subconjunto de estos dos lenguajes.

Federaciones de Queries

"Diseñamos el Object Query Service para poder consultar y manipular cualquier objeto CORBA-volátil o persistente, concreto o meta, local o remoto, individual o sobre una colección. Si hay un servicio que pueda utilizar y unificar todos los servicios disponibles para los objetos con un ORB, se trata del Object Query Service."

- Dan Chang,
Object Architect IBM
(Junio de 1995)

El servicio de consultas puede ejecutar las consultas por si mismo o bien delegar el trabajo a un *query evaluator*. Por ejemplo, el servicio podría usar las facilidades nativas disponibles para la formulación de consultas o bien delegar el trabajo en una base de datos orientada a objetos para ejecutar un query anidado.
El servicio de consultas combina los resultados obtenidos de todos los evaluadores involucrados y retorna el resultado final al objeto que lo invocó. Por esta razón, es posible usarlo para coordinar el funcionamiento de federaciones de administradores de queries nativos. La gran ventaja de esto es que cada base de datos puede usar su propio motor de búsqueda optimizado mientras participa en una búsqueda global.

Colecciones para manipular los resultados de las consultas

Cuando se ejecuta una consulta, el servicio retorna una *colección* con los objetos que satisfacen el criterio de búsqueda especificado en una operación de la forma *select*. Notemos que CORBA usa el término *query* en una connotación más general. Un *query CORBA* hace bastante más que encontrar objetos; también permite manipular una colección de objetos. La colección es el resultado de la ejecución de la consulta. El servicio trata a las colecciones como objetos: define operaciones que permiten manipularlas y navegar. También permite agregar y eliminar miembros de una colección.

Query Service: las interfaces de las colecciones

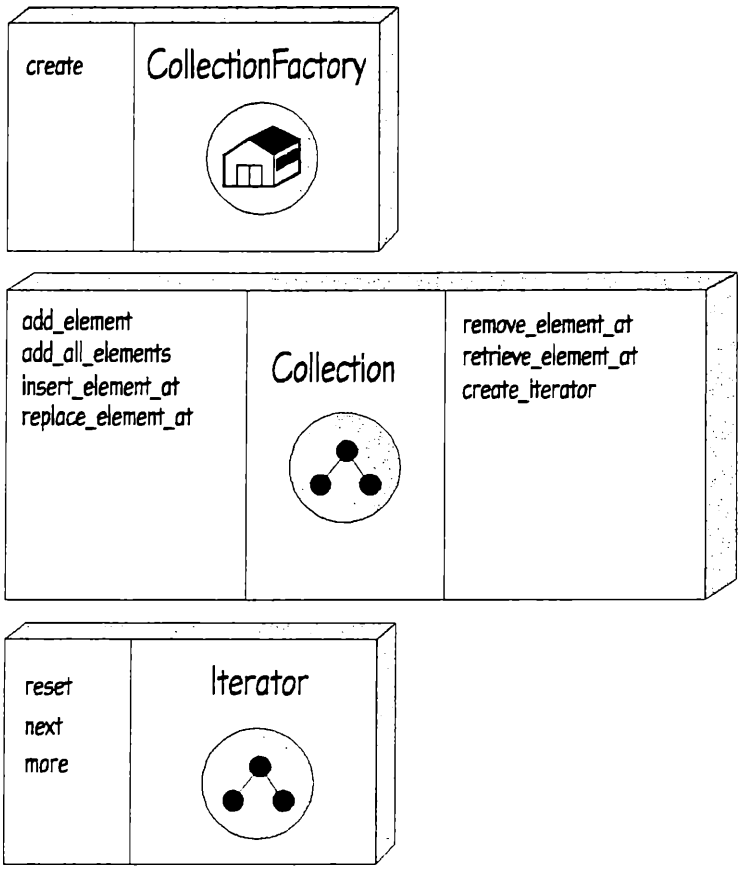


Figura 2-34. Query Service: interfaces de colecciones

El Query Service ofrece un servicio mínimo para el manejo de colecciones, necesario para manipular cómodamente los resultados de las consultas. Hay tres interfaces:

- ***CollectionFactory:*** define una única operación, *create*, para crear nuevas instancias de colecciones vacías.
- ***Collection:*** define protocolo para agregar, reemplazar, recuperar y eliminar miembros de una colección. Con *add_all_elements* podemos agregar todos los elementos de otra

colección. *Insert_element* sirve para agregar un elemento en una posición particular. *Create_iterator* retorna un puntero móvil para navegar por la colección.

- ***Iterator:*** define tres operaciones para recorrer la colección. *Reset* mueve el puntero al comienzo de la colección; *next* incremnta el puntero a la próxima posición; *more* sirve para determinar si aún quedan elementos para recorrer en la colección–este método retorna true si el puntero no está al final de la colección y false en caso contrario.

Query Service: las interfaces de consulta

El servicio de consultas suministra un framework formado por cinco clases que se encargan de preparar y ejecutar las consultas. Veamos qué hacen cada una de ellas:

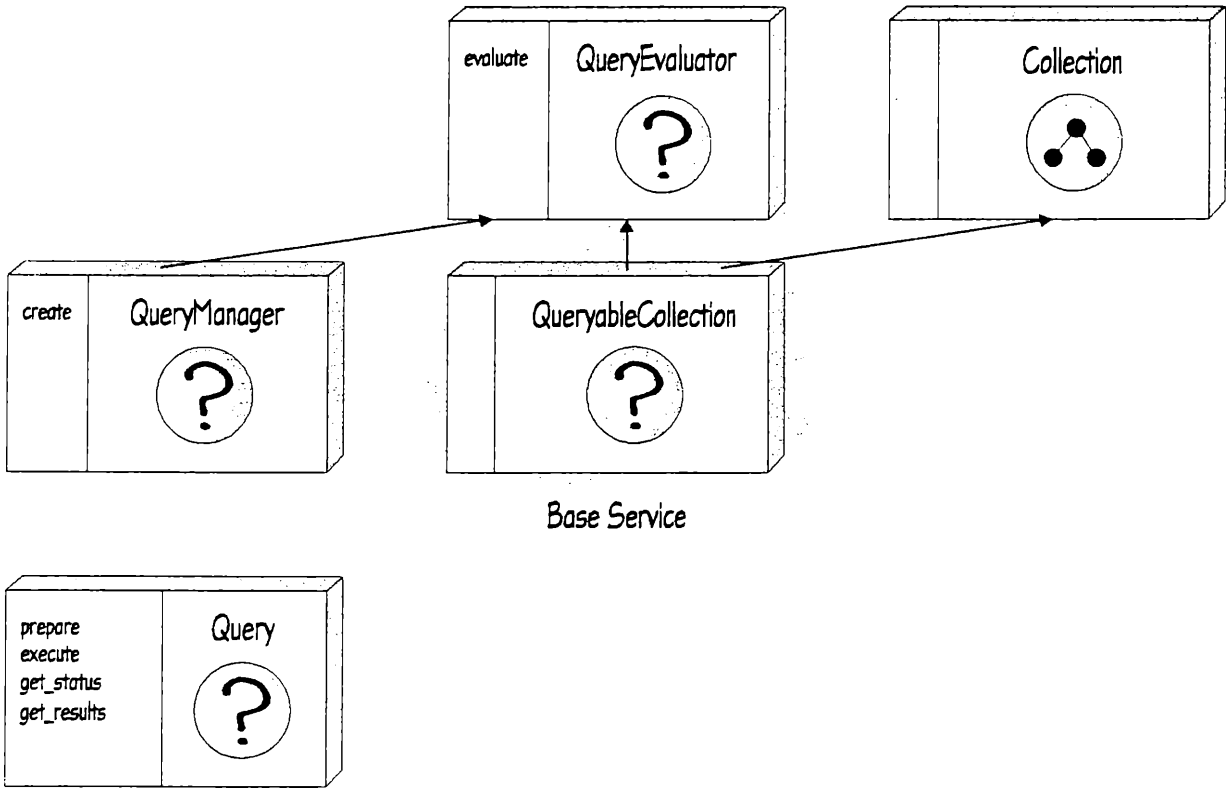


Figura 2-35. Query Service: interfaces de consultas

- ***QueryEvaluator:*** define la operación *evaluate* para evaluar una consulta, usando el lenguaje de consulta especificado o bien el que esté disponible por omisión. Un ejemplo de un objeto ***QueryEvaluator*** es un sistema de base de datos.
- ***QueryManager*** es una variante más poderosa de ***QueryEvaluator***. También permite generar instancias de ***Query***.
- ***Query*** define cuatro operaciones que se ejecutan e cualquier instancia de una consulta–cada consulta se representa con una instacia de esta clase. *Prepare* compila el query y lo prepara para la ejecución. *Execute* corre la ejecución de un query compilado. *Get_status*

sirve para determinar el estado de preparación y/o ejecución de un query dado. Y finalmente, *get_results* es para obtener el resultado de la ejecución del query.

- ***QueryableCollection*** no introduce nuevas operaciones. En cambio, hereda su funcionalidad de ***QueryEvaluator*** y ***Collection***. Las instancias de esta clase evalúan un query sobre una colección dada de objetos. Notemos que cualquier colección puede a su vez ser miembro de una colección; esto significa que podemos tener un número infinito de subqueries anidados.

Podemos extender estas cuatro interfaces usando herencia para obtener una funcionalidad más especializada. Por ejemplo: podemos extender a ***Query*** para desarrollar un browser de propósito general que muestre y registre los sucesivos resultados de una consulta.
El Query Service básico se compone sólo de dos clases: ***QueryEvaluator*** y ***QueryableCollection.***

Un ejemplo simple

Comenzaremos analizando el uso de un objeto que implementa la interface ***QueryableCollection***. Recordamos que se trata simplemente de un objeto qe sabe cómo ejecutar un query sobre una colección de otros objetos.

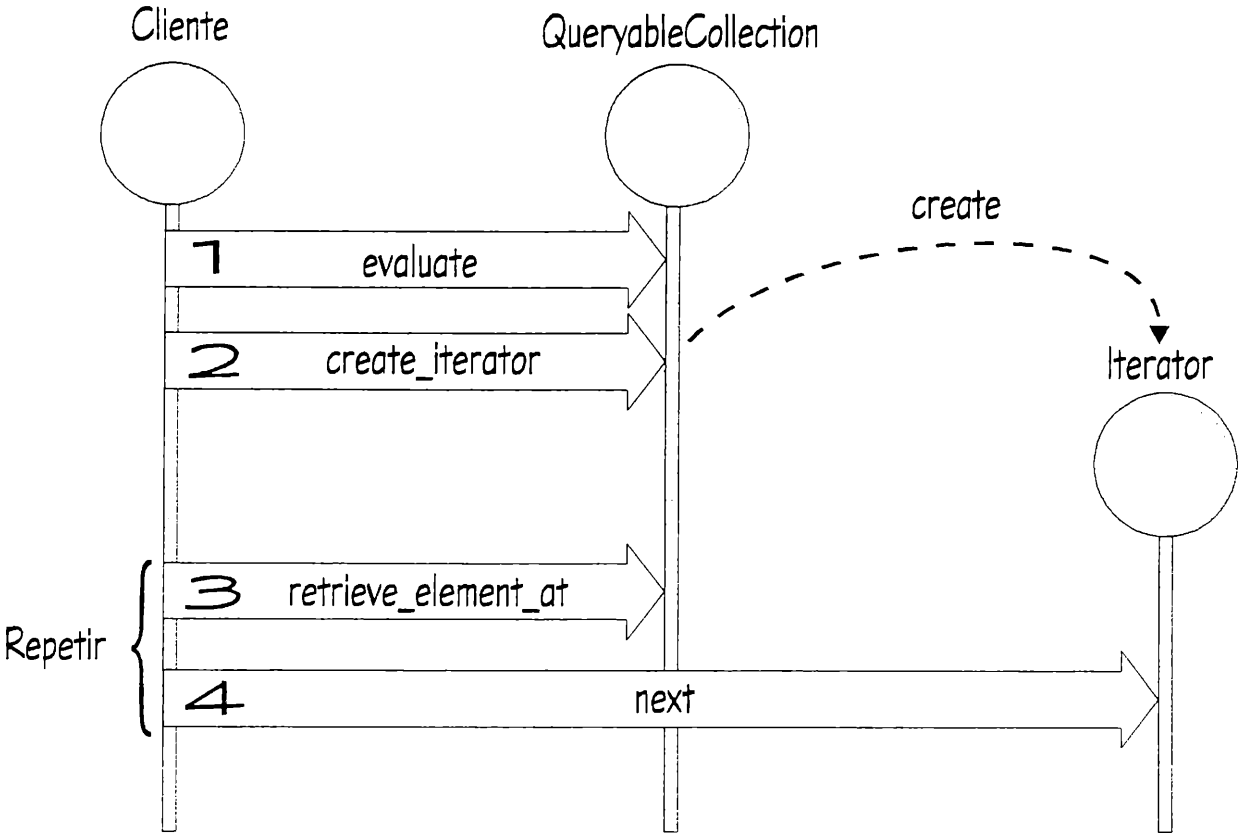


Figura 2-36. Un escenario simple

1.- ***Preparar y enviar la consulta.*** Se envía *evaluate* a la instancia de ***QueryableCollection*** para que ejecute la consulta. Se pasan parámetros que incluyen la sentencia de la consulta y el lenguaje usado para expresarla (OQL o SQL). El objeto ejecuta la consulta y retorna el resultado en una colección–que él controla.
2.- ***Crear un puntero para navegar por la colección del resultado.*** Con el mensaje *create_iterator* se obtiene un iterador inicializado al inicio de la colección de objetos.
3.- ***Leer el primer elemento de la colección.*** Con *retrieve_element_at*, recuperamos el objeto de la posición (o una row, expreada con el tipo *any* de CORBA) de la posición a la que apunta el iterador.
4.- ***Avanzar el puntero a la siguiente posición.*** Con el mensaje *next* el puntero se corre al siguiente objeto o row de la colección.

Ciclando entre los dos últimos pasos–**3)** y **4)**–se recuperan todos los objetos de la colección del resultado de la consulta.

Un ejemplo más complejo

Teniendo en cuenta que las consultas son complejas y demandan recursos, a veces es deseable un control más preciso sobre su procesamiento. En particular quisiéramos: 1) usar alguna herramienta gráfica para armarlas; 2) precompilar las consultas y guardarlas para ejecutarlas más tarde; 3) ejecutar la consulta en forma asincrónica–esto nos permite hacer alguna otra tarea mientras se ejecuta la consulta y luego obtener los resultados; y 4) poder chequear el estado de la ejecución de las consultas complejas y eventualmente decidir cancelarlas.

Para obtener estos niveles de control, usamos una instancia de ***QueryManager*** y una instancia asociada de ***Query***. El ***QueryManager*** controla un conjunto de colecciones sobre las que se formula la consulta. Se encarga de asignar el query (y la colección contra la que se ejecuta) a una instancia de ***Query***. Es posible interactuar con ese objeto para controlar la ejecución. El siguiente escenario describe la secuencia de pasos.

1.- ***Crear un objeto Query.*** Con el mensaje *create* enviado a un objeto que soporte la interface de ***QueryManager***, creamos una instanciaa de ***Query***. El ***QueryManager*** se encarga de crear el nuevo ***Query*** y retorna una instancia a él (para interactuar directamente).
2.- ***Precompilar la consulta.*** Enviando el mensaje *prepare* al objeto ***Query***, éste se precompila y guarda para una ejecución posterior. Los parámetros que se pasan en el mensaje incluyen la expresión del query y el lenguaje (OQL o SQL) utilizado para expresarla.
3.- ***Ejecutar la consulta.*** Se puede ejecutar una consulta precompilada tantas veces como se quiera (se pueden cambiar los parámetros de búsqueda si fuera necesario, en cada ejecución). Para esto se envía el mensaje *execute* al query (este objeto retiene todo el contexto necesario para correr la consulta preparada previamente).
En el escenario que mostramos, el query retorna el resultado de la ejecución en una instancia de ***QueryableCollection***. Normalmente, el ***QueryManager*** retiene todos los objetos de la colección.

4.- Recuperar el resultado del query. Enviando el mensaje *get_results* tenemos la certeza de que el query se ejecutó satisfactoriamente. Normalmente el resultado es una referencia a una colección con los objetos encontrados.
5.- Iterar entre los objetos del resultado. La última etapa es similar al escenario anterior y consiste en iterar por la colección de objetos (o rows de una tabla).

Con esto concluimos la explicación del segundo escenario. Está claro que hay diferentes niveles de complejidad en el mismo servicio y que las colecciones son fundamentales para procesar el resultado de una consulta.

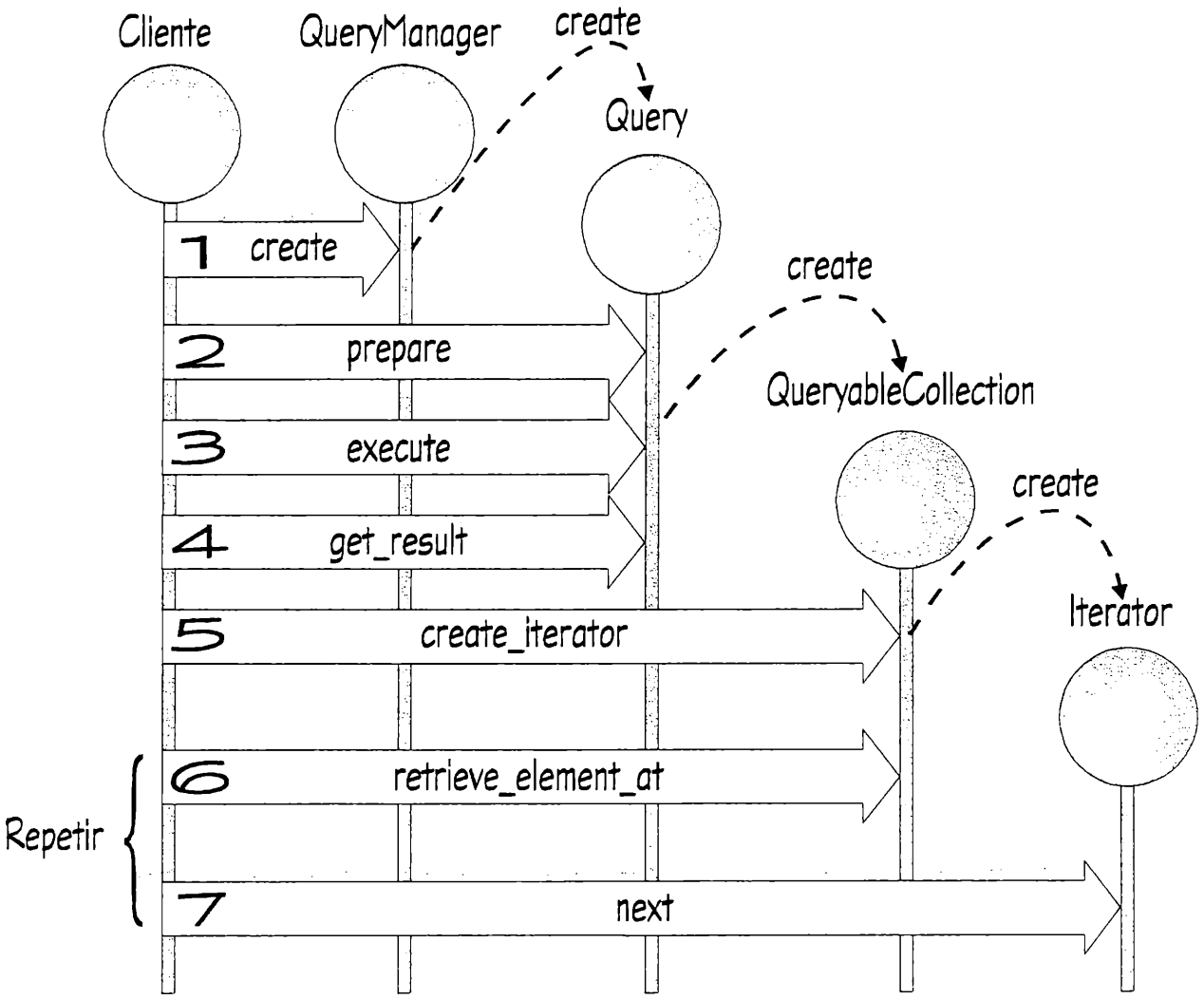


Figura 2-37. Un escenario más complejo

COLLECTION SERVICE

Las colecciones permite manejar muchos objetos en forma grupal. Algunos ejemplos de colecciones son: *queues*, *stacks*, *lists*, *arrays*, *sets*, *trees* y *bags*. Cada una de estas colecciones exhibe un comportamiento diferente. Por ejemplo, en los *sets* están disponibles las siguientes operaciones: agregar un nuevo elemento, verificar la pertenencia de un elemento, verificar si está vacío, union, intersección, etc. Algunas colecciones son ordenadas–establecen un orden entre los elementos–otras usan claves para el acceso a los elementos. Finalmente, algunas colecciones sirven para mantener referencias a objetos; otras guardan datos (es el caso de una tabla que contiene rows de datos).

El objetivo del Collection Service de CORBA es ofrecer una manera uniforme de crear y manipular colecciones. Una vez que CORBA defina el IDL para estas clases generales de colecciones, se espera contar con diferentes implementaciones que sean intercambiables. Así, por ejemplo podremos contar la version CORBA IDL-izada de las clases de colecciones en C++ de Taligent, el Standard Template Library (STL) del ANSI C++, etc. Si esto ocurre, entonces no será necesario que reinventemos la rueda para manejar colas, pilas, conjuntos, etc.
¿Cómo se relacionan las colecciones definidas en este servicio con las clases de colecciones del Query Service? El Query Service define sólo un servicio mínimo de manejo de colecciones, con clases generales de muy alto nivel. El Collection Service define subclases que especializan esas características para colecciones específicas.

RELATIONSHIP SERVICE

"Las aplicaciones se contruyen a partir de objetos existentes que se conectan entre si."

- CORBA COSS Specification
(Marzo de 1995)

En el mundo real, los objetos nunca existen aisladamente. Más bien podemos decir que forman redes de relaciones con otros objetos. Estas relaciones van y vienen. Pueden ser estáticas, espontáneas, dinámicas, ... Los objetos distribuidos (y los componentes) deben ser capaces se modelar a sus contrapartes del mundo real. Así que deberíamos poder establecer y mantener relaciones entre objetos, aunque éstos no las soporten en forma nativa. Quisiéramos hacer esto sin tener que recompilar o cambiar la definición de los objetos involucrados en las relaciones.

El Relationship Service de CORBA permite que los componentes y los objetos que no se conocen entre si estén relacionados. Y esto es posible sin tener que modificar sus interfaces. En otras palabras, es posible crear relaciones dinámicas entre objetos inmutables

¿Por qué un servicio de relaciones?

Sin un servicio, los objetos deberían encargarse de administrar sus propias relaciones usando punteros. Por ejemplo, un objeto podría manejar una colección con todas la referencias hacia todos los objetos con los que se relaciona. Además debería manejar información sobre el tipo de relación y sus atributos.

Esta solución ad-hoc–basada en referencias entre objetos–no es demasiado convincente ni útil. Las referencias entre objetos son unidireccionales y es muy difícil navegar libremente sobre una relación entre objetos administrada con punteros. No es fácil exportar la información de los punteros hacia otros objetos que necesitan conocer y entender la relación para navegar sobre ella. Por ejemplo: podríamos necesitar un grafo que describa la relación a un servicio de copia de relaciones para que determine cuáles objetos debe copiar o mover (Life Cycle Service). Si la relación se mantiene con referencias, es muy difícil generalizar el servicio.

Pero más importante aún es que el servicio de relaciones permite construir relaciones dinámicas sobre los objetos, "*on the fly*". Las relaciones cableadas entre objetos no permiten llegar muy lejos en ambientes de componentes dinámicos. Así pues, se necesita un Relationship Service.

¿Qué es exactamente una relación?

Los objetos del mundo real siempre están involucrados en relaciones. Por ejemplo, consideremos un libro, con sus capítulos, personas que lo compran y los autores.

- ***Relación de pertenencia entre las personas y los libros.*** Cada persona posee un libro; cada libro es poseído o pertenece a una o más personas. Esto describe una *relación de pertenencia*.
- ***Relación de contención entre los capítulos del libro.*** Un libro contiene uno o más capítulos; cada capítulo está contenido en un libro. Es una *relación de contención*. En el ejemplo, el libro contiene los capítulos de CORBA, OLE y OpenDoc.
- ***Relaciones de referencias entre libros***. Un libro hace referencia a otros libros; un libro es referenciado en uno o más libros. En el ejemplo, el libro *hace referencia* al libro "CORBA Design Patterns"
- ***Relaciones de autoría entre un libro y sus autores.*** Un libro es escrito por uno o más autores; un autor escribe uno o más libros. En el ejemplo, el libro en cuestión es *autoría* de Bob, Dan y Jeri.
- ***Relaciones de empleo entre las empresas y las personas.*** Una empresa emplea a una o más personas; una persona es empleada por una o más empresas. En la relación del ejemplo, Jeri trabaja en Tandem Computers y Bob y Dan trabajan en IBM.

Una relación se define por un conjunto de *roles* que dos o más objetos juegan (los roles se resaltan con círculos negros). Por ejemplo, en una relación de empleo, la empresa juega el rol de empleador y la persona juega en rol de empleado. Es posible que una misma persona

juegue diferentes roles en varias relaciones. Por ejemplo, Jeri es al mismo tiempo empleada y autora, pero en relaciones diferentes. El *grado* de una relación se refiere al número de roles necesarios para establecer la relación. En el ejemplo, todas las relaciones son *binarias*–de grado dos. La *cardinalidad* se refiere a la máxima cantidad de relaciones en que está involucrado un rol.

Por ejemplo, un libro mantiene una relación de contención *muchos-a-uno* con sus capítulos (cardinalidad n); pero los capítulos están contenidos en un solo libro (cardinalidad uno).

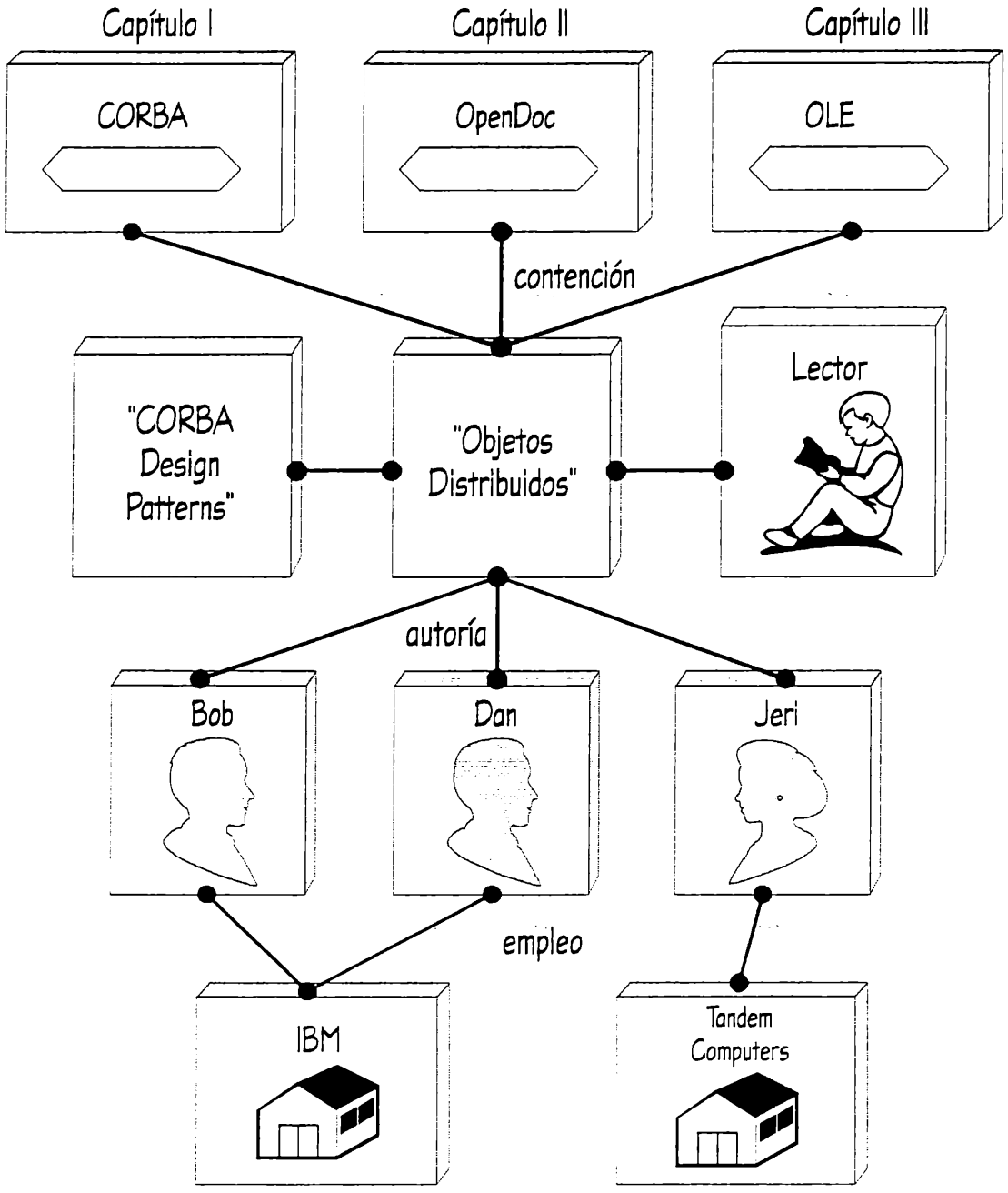


Figura 2-38. Un grafo de objetos relacionados

Una relación puede tener uno o más atributos. Por ejemplo podemos agregar un atributo "título" a las relaciones de empleo. Una relación puede también soportar la invocación de métodos que retornan información. Notar que los atributos y métodos de una relación son totalmente independientes de los objetos que represntan. Podemos pensar que las relaciones introducen sus propias semánticas indenpendientes. Un conjunto de objetos relacioneados forma un grafo. Los objetos son nodos del grafo; las relaciones son las aristas del grafo.

Niveles en el servicio de relaciones

El servicio de relaciones define interfaces genéricas que permiten asociar roles y relaciones con objetos (CORBA) ya existentes. Es posible navegar por las relaciones (en cualquier dirección) usando sofisticadas interfaces de grafos.
El servicio se hace cargo de mantener las relaciones entre los objetos. También define dos relaciones específicas entre objetos: *contención* y *referencia*. El servicio permite crear relaciones de cardinalidad y grado arbitrarios; se encarga de forzar las restricciones de grado y cardinalidad especificadas y detectar eventuales violaciones.
Por otro lado, trata a las relaciones, roles y grafos como objetos CORBA de primera clase, por lo que es posible derivar especializaciones con subclases.

La principal característica de este servicio es permitir establecer relaciones entre objetos que se desconocen entre si y sin afectarlos. Las interfaces permiten crear y navegar por las relaciones. Estas interfaces están agrupadas en tres categorías: *basicas*, *grafos* y *especificas*.

Relationship Service: interfaces básicas

Estas interfaces definen operaciones para crear instancias de ***Role*** y ***Relationship*** y navegar por las relaciones en las que participan los roles. Heredan de ***IdentifiableObject***, lo que permite chequear la igualdad de dos referencias a objetos CORBA. Estas interfaces están agrupadas en dos módulos: ***CosObjectIdentity*** y ***CosRelationships***.

- ***IdentifiableObject*** suministra un única operación, *is_identical*, que retorna TRUE si dos objetos CORBA son idénticos. El servicio de relaciones necesita esta operación para administrar los objetos que define.
- ***RelationshipFactory*** define una operación *create* para instanciar relaciones. Se debe pasar por parámetro la secuencia de roles que representan los objetos que se pretende relacionar. La fábrica, informa a cada rol–instancia de ***Role***–de la relación en la que se lo involucra con el mensaje *link*.
- ***RoleFactory*** define la operación *create_role* para asociar un rol a un objeto CORBA pasado por parámetro.
- ***Relationship*** define un única operación, *destroy*, para eliminar una relación. Los roles involucrados son descoplados antes de destruir la relación.
- ***Role*** define operaciones para navegar por las relaciones en las que participa un rol dado y además permite enlazar un nuevo rol a una relación ya existente. Con *get_other_role* se obtiene el rol relacionado por la relación; con *get_other_related_object* se obtiene el

objeto asociado al rol del otro extremo de la relación. Con *get_relationships* se obtienen todas las relaciones en las que está involucrado un rol. Con *destroy_relationships* se libera al rol de las relaciones que lo involucran. Con *destroy* de elimina un rol que no participa en ninguna relación. *Check_minimun_cardinality* retorna TRUE si el rol satisface las restricciones de cardinalidad mínima. *Link* se usa por las fábricas para enlazarun rol en una relación. *Unlink* se usa en la destrucción–operación *destroy*–para eliminar un rol de una relación.

- ***RelationshipIterator*** nos permite iterar a través de relaciones adicionales en las que participa un rol. Los iteradores se retornan enviándo el mensaje *Role::get_relationships*.

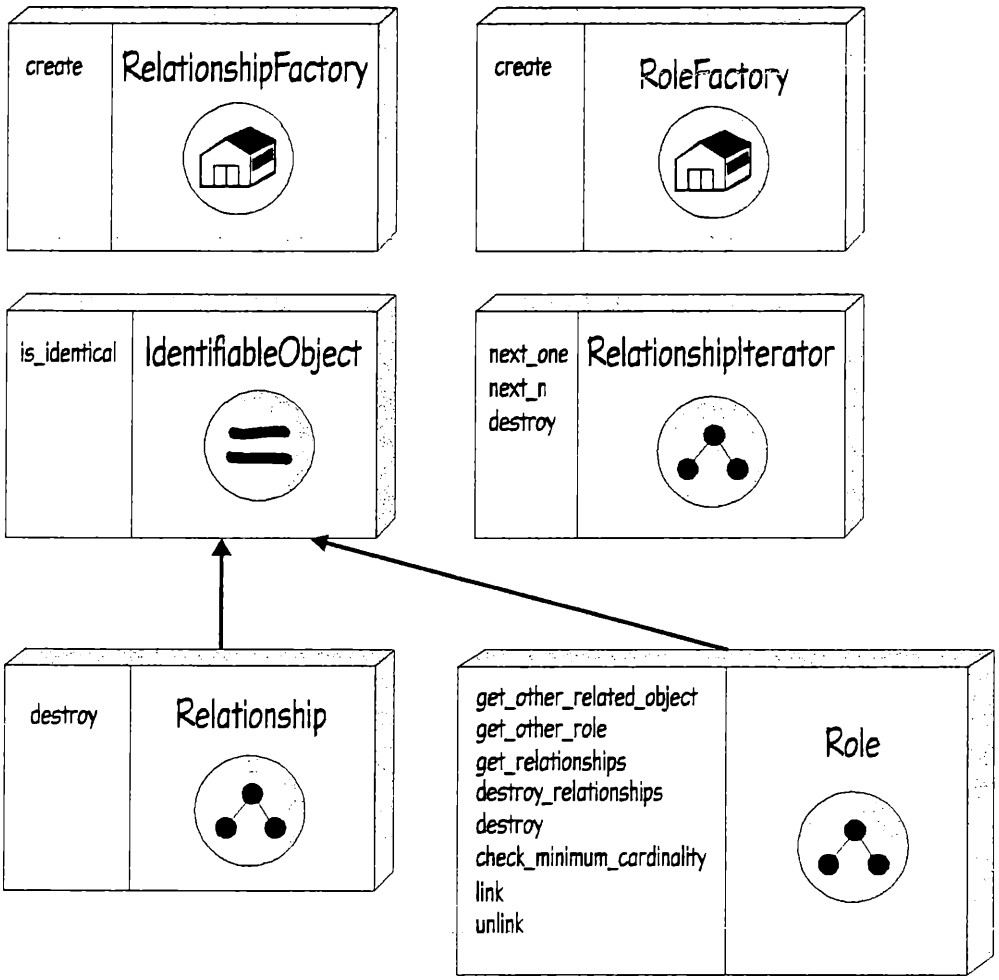


Figura 2-39. Relationship Service: interfaces básicas

Relationship Service: grafos de objetos relacionados

Como dijimos más arriba, un grafo es un conjunto de *nodos* y *aristas*. Los nodos son los objetos relacionados; las aristas son las relaciones. Un nodo puede soportar uno o más roles. Las interfaces de grafos permiten describir y navegar por los grafos de relaciones.

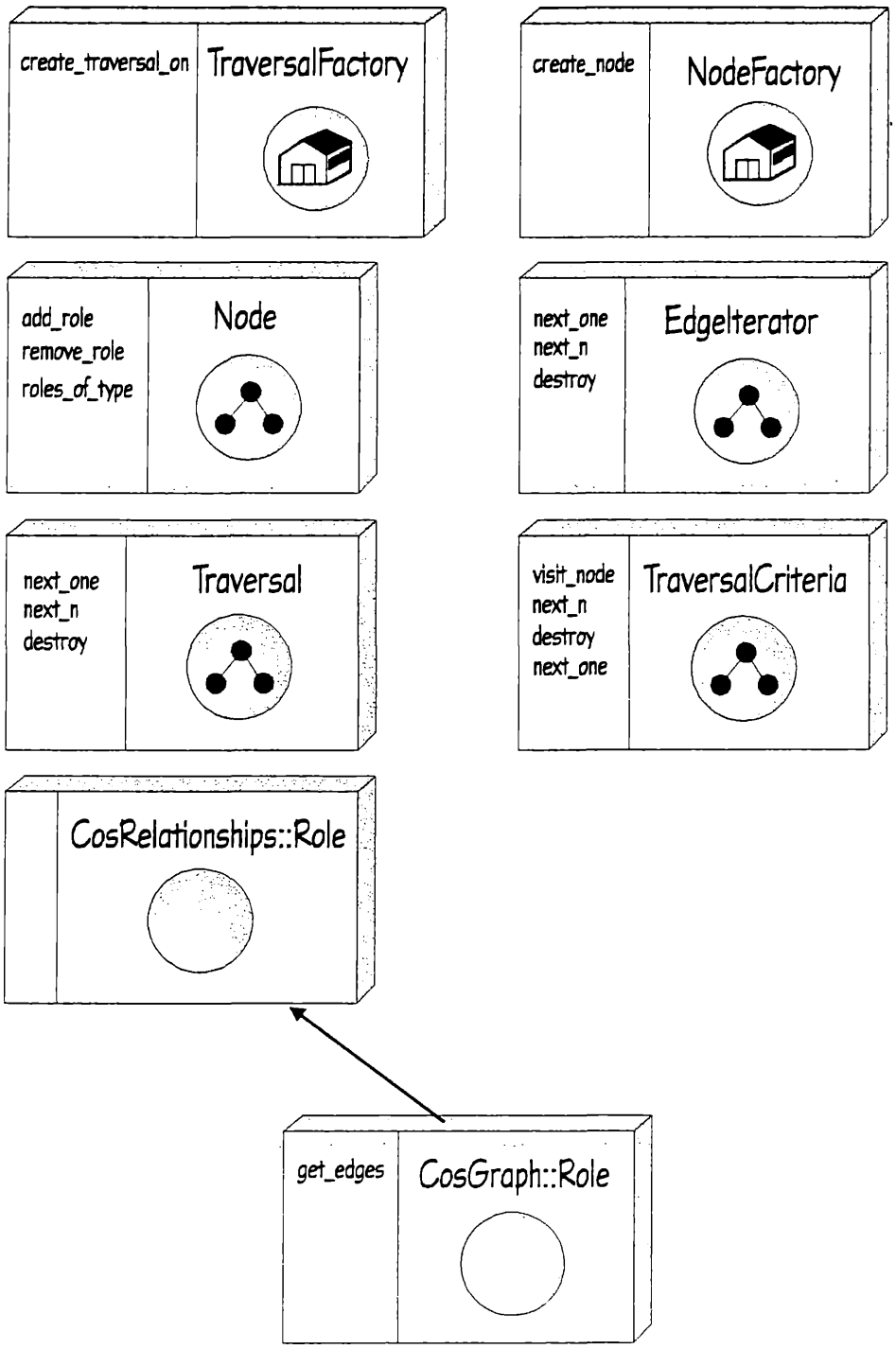


Figura 2-40. Relationship Service: interfaces de grafos

- *Node* asocia un objeto con sus roles. Suministra operaciones para agregar y quitar roles. La operación *roles_of_type* nos permite consultar los roles de tipo en particular.
- ***Traversal*** es para navegar a través de un grafo de objetos relacionados, comenzando en un nodo especificado.
- ***TraversalCriteria*** permite asociar reglas de la forma de funciones que se invocan para determinar cuál es el próximo rol o relación que debe visitarse. Las operaciones definidas en estos objetos deben ser invocadas por ***Traversal***.
- ***Role*** se deriva de ***CosRelationship::Role***. Suministra una nueva operación: *get_edges*. Esta operación retorna las relaciones de un rol, en la forma de estructura o como un iterador (***EdgeIterator***).
- ***EdgeIterator*** suministra dos operaciones básicas para iterar por las relaciones asociadas a un rol. También suministra una operación de autodestrucción, *destroy*. Para crear una instancia de ***EdgeIterator*** se debe enviar el mensaje *Role::get_edges*.

Como vemos, estas interfaces pueden ser sumamente útiles. Por ejemplo, podemos usarlas para describir todos los objetos (y relaciones) mostrados en la figura anterior.
Luego, podemos enviar un grafo que describa estos objetos (junto con los objetos mismos) a un destino remoto, para que sean recreados y recorridos. Lo importante es que logramos esto sin afectar a los objetos involucrados o sus atributos. Los objetos ni siquiera tienen noción de que están participando en grafo. ¡Magia!
Las interfaces de grafos son usadas por el Life Cycle Service para implementar *deep copies* y *moves* de objetos. También se usan en el Externalization Service para exportar grupos de objetos relacionados.

Relationships Service: las relaciones de contención y referencias

Las relaciones de contención y referencias entre objetos son relaciones muy comunes. En consecuencia, el servicio nos ofrece un conjunto estandarizado de interfaces para ambas relaciones. Podemos usar estas interfaces para modelar cómo crear nuestras propias relaciones (por ejemplo: referencia de integridad, documentos compuestos, familia, amigos, etc). Notamos en la figura que estas dos relaciones no introducen nuevas operaciones. Simplemente son derivadas de las interfaces ya definidas en el servicio. Estas interfaces derivadas definen tipos IDL en CORBA que son específicos para las relaciones de contención y referencia. Los atributos derivados permiten definir el grado de cada relación. La relación de contención define dos interfaces: ***ConstainsRole*** y ***ContainedInRole***. Por su parte, la relación de referencia define a ***RefererecesRole*** y ***ReferencedInRole***. Ambas relaciones son binarias (de grado 2). Las fábricas fuerzan estas restricciones cuando se crean nuevas instancias de estas relaciones.

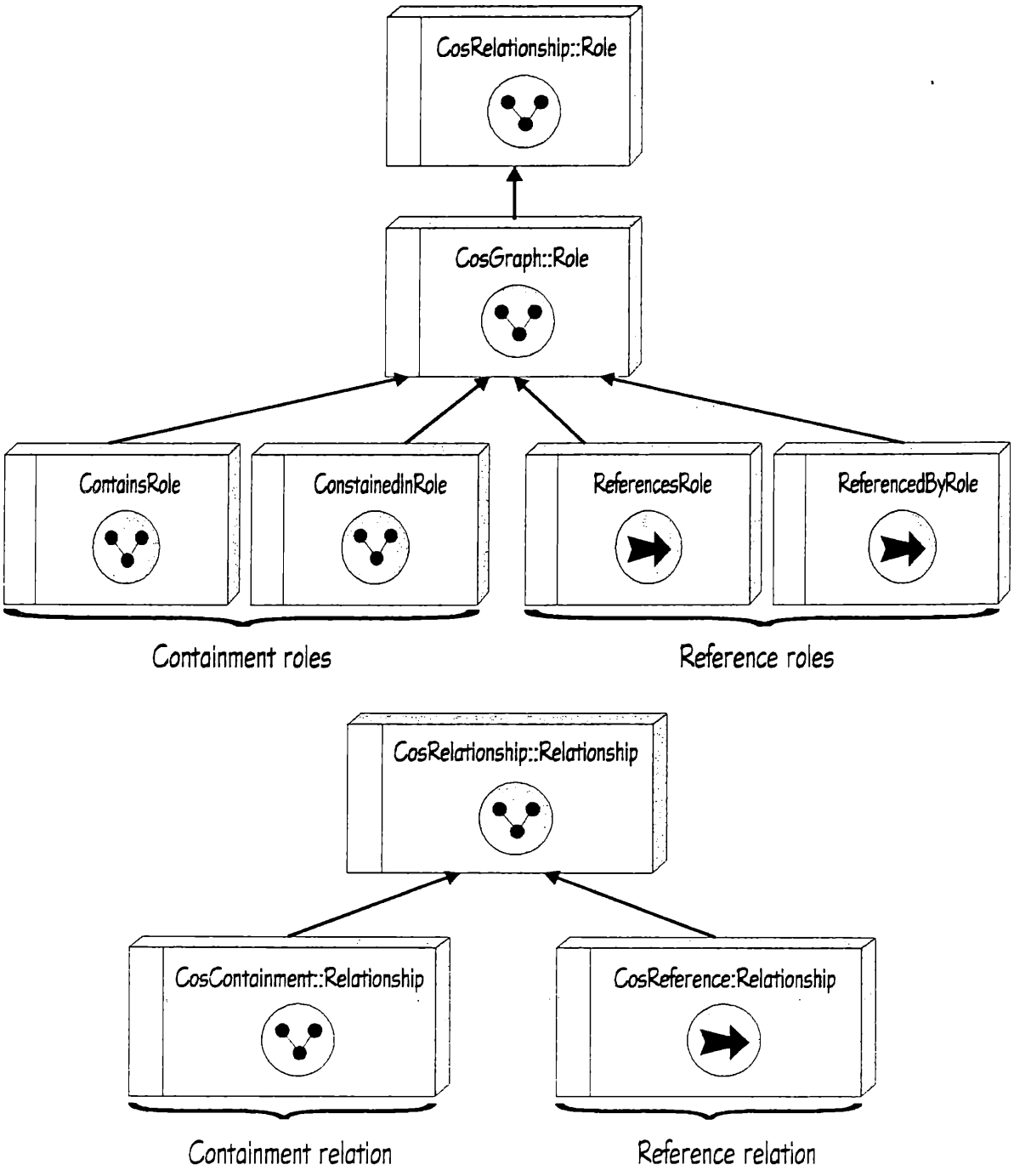


Figura 2-41. Relationship Service: ralaciones de contención y referencias

SERVICIOS DE CORBA: SYSTEM MANAGEMENT AND SECURITY

Esta sección cubre los siguientes servicios: Security, Licensing, Time, Change Managemente, Properties y Externalization

EXTERNALIZATION SERVICE

> "Los streams son realmente fascinantes porque se pueden usar para muchos propósitos. Sería virtualmente imposible desarrollar seriamente un objeto que no soporte una interface de manejo de streams."
>
> - Rogers Sessions, Author
> Object Persistence
> (Prenctice Hall, 1996)

El servicio de exportación–originalmente desarrollado por Taligent–define interfaces para exportar un objeto a un stream y viceversa. La mayoría de los programadores están familiarizados con el poder y simplicidad de los streams.
Un stream es un area de datos con un cursor asociado. El cursor es un puntero móvil que avanza y retrocede a medida que se leen y escriben datos desde y al stream. El area de datos puede estar en memoria, un archivo en disco o bien en algún lugar remoto accesible a través de una red. ¡No es posible establecer la diferencia! Usamos el servicio de exportación a streams para transportar un objeto a otro proceso, máquina u ORB. Importamos el objeto cuando cuando es necesario traerlo de vuelta a la vida, en su nuevo destino.

El poder de los streams

En algún sentido, la exportación y posterior importación de objetos es similar a copiarlos usando el *Life Cycle Service*. La diferencia es que la exportación e importación dividen el proceso de copia en dos pasos: 1) Se copia el estado del objeto al stream; 2) Se copia el estado guardado en el stream al objeto destino. Este proceso de dos fases permite exportar los objetos fuera del entorno del ORB.
En contraposición al Life Cycle Service, el Externalization Service, divide la operación de *copy* o *move* en dos pasos y nos da la oportunidad de hacer algún procesamiento extra con los resultados intermedios.

Los streams permiten copiar y mover objetos. También sirven para pasar objetos por valor en parámetros de mensajes CORBA (actualmente CORBA solamente soporta el pasaje de parámetros por referencia en el envío de mensajes).

Servicio de exportación: las interfaces básicas

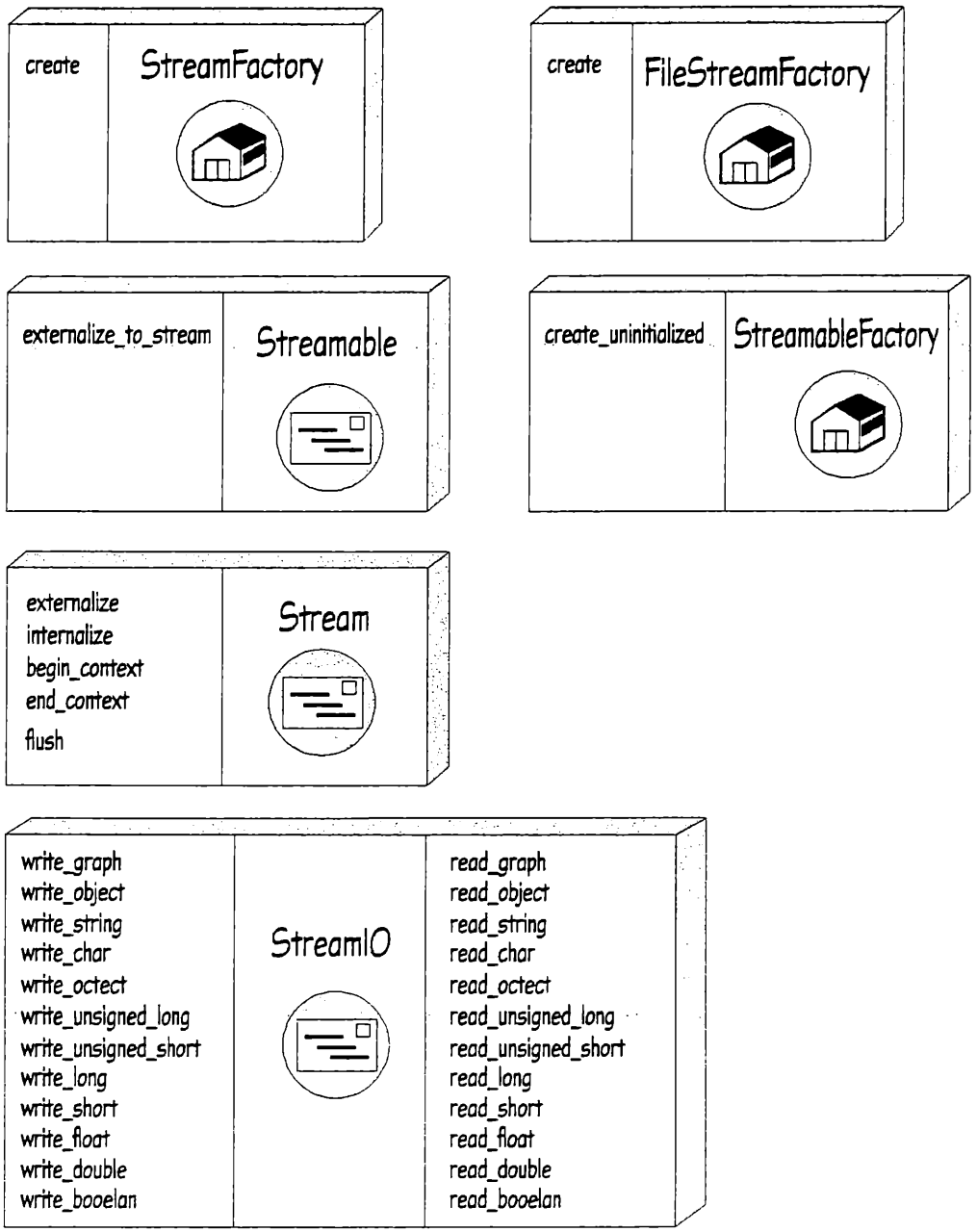


Figura 2-42. Externalization Service: interfaces básicas

La visión que tiene el cliente sobre la exportación de objetos es realmente simple. El cliente crea (o localiza) una instancia de ***Stream*** y pasa como parámetros el o los objetos que deben exportarse. Para crear un stream se envía un mensaje a ***StreamFactory***. Es posible crear streams de archivos usando la clase ***FileStreamFactory***. Con el mensaje *externalize* se pide a cada objeto que exporte su estado al stream. También es posible exportar varios objetos al mismo stream de la siguiente forma: 1) enviar *begin_context*; 2) enviar el mensaje *externalize* por cada objeto que deba exportarse; y 3) mensaje *end_context*. Se usa el mismo mensaje *externalize* para exportar un solo objeto o un grafo completo de objetos relacionados.

Cuando un stream recibe un mensaje *externalize*, se da vuelta y envía el mensaje *externalize_to_stream* al objeto destino (éste debe soportar la interface ***Streamable***).
Todos los objetos exportables deben implementar esta interface (normalmente como un mixin usando herencia múltiple). Los objetos exportables usan las operaciones de ***StreamIO*** para leer o escribir su estado del o hacia el stream. Esta interface sumministra operaciones de lectura y escritura–sobre stream–para todos los tipos IDL de datos. También permite leer y escribir grupos de objetos relacionados usando *write_graph* y *read_graph*.
Los objetos ***Stream*** implementan la interface de ***StreamIO*** o bien pasan la referencia a un objeto que la soporte, cuando hacen la invocación al objeto que debe exportarse.

Para importar un objeto a partir de un stream, el cliente invoca *internalize* en un objeto ***Stream***. Este objeto debe localizar (o crear) la instancia de ***Streamable*** que pueda importar su estado del stream; para crear una nueva instancia usa la interface ***StreamableFactory***. Luego, el stream envía el mensaje *internalize_from_stream* al objeto y éste utiliza las operaciones disponibles en la interface ***StreamIO*** para leer el contenido del stream.
¿Suena complicado? Realmente no lo es. Ejemplificamos con un escenario.

Un escenario con streams

En la figura representamos un escenario que muestra cómo se exportan e importan objetos. La secuencia de pasos involucrados es:

1.- ***El cliente obtiene un objeto stream.*** El cliente envía el mensaje *StreamFactory::create* para obtener una instancia de ***Stream***.
2.- ***El cliente indica al stream que exporte el objeto.*** El cliente envía el mensaje *Stream::externalize* y pasa la referencia al objeto como parámetro (este objeto debe soportar la interface ***Streamable***).
3.- ***El stream le indica al objeto que se exporte a si mismo.*** El stream envía el mensaje *Streamble::externalize_to_stream* para que el objeto exporte su estado interno al stream.
4.- ***El objeto escribe su estado en el stream.*** Usando las operaciones *write_<type>* de ***StreamIO***, el objeto escribe en el stream el contenido de sus datos. Para exportar objetos anidados el objeto usa *write_object*. También es posible que un objeto forma parte de un grafo de objetos relacionados. Es decir que podría estar conectado con otros objetos por el Relationship Service. Cuando queremos exportar el grafo de la relación se envía el mensaje

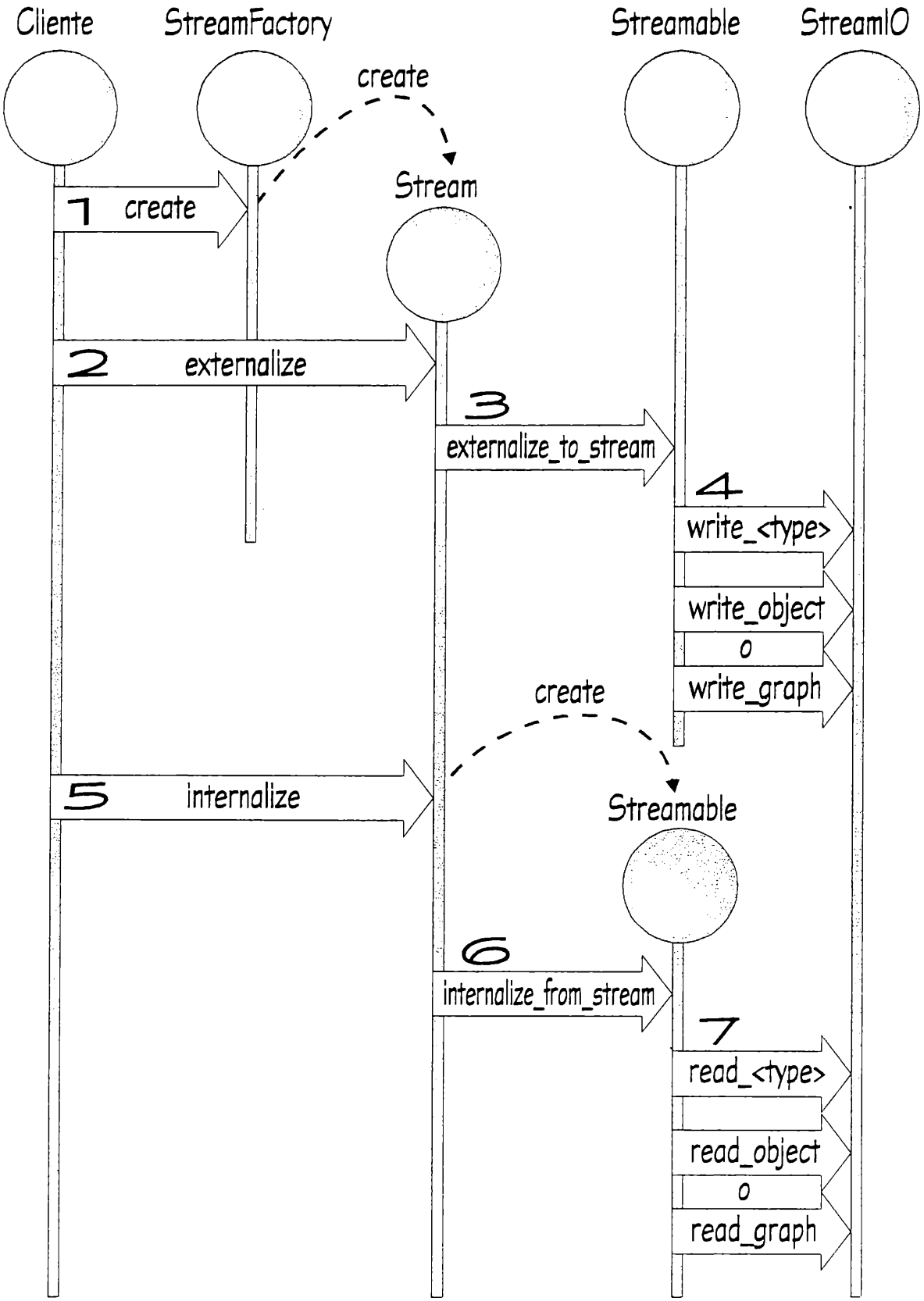


Figura 2-43. Escenario Exportación/importación usando streams

write_graph para que el servicio de stream coordine la exportación del grafo de objetos relacionados con la ayuda del Relationships Service.
5.- ***Más tarde, el cliente desea importar el estado guardado en el stream.*** El stream que guarda la exportación de un objeto puede guardarse todo el tiempo que se quiera. Incluso puede llevarse fuera del entorno del ORB. Cuando se quiere importar, se invoca *Stream::internalize*. El objeto stream busca dentro del stream la clave que sirve para localizar la fábrica que crea el objeto con la implementación apropiada del objeto exportado.
6.- ***El stream le indica al objeto que se importe a si mismo.*** El stream invoca *Streamable::internalize_from_stream* para indicar al objeto que importe su estado desde el stream (el objeto debe soportar la interface ***Streamble***).
7.- ***El objeto lee su estado desde el stream.*** Usando operaciones de la forma *read_<type>* de la interface ***StreamIO***, el objeto lee el contenido de sus datos. Para leer objetos anidados se usa *read_object*. Si el objeto es un nodo dentro de un grafo de objetos relacionados, invoca *read_graph* para permitir que el servicio de stream coordine la importación de los objetos relacionados con la ayuda del Relationship Service.

El Externalization Service también define una *Formato Estándar de Datos para Streams* (Standard Stream Data Format) para el intercambio de streams a través de redes, plataformas de sistemas operativos e implementaciones de medios de almacenamiento. Esta representación estándar de los stream usa formatos autodescriptivos de tipos IDL de datos y encabezados que describen el tipo de objetos contenidos dentro de un stream. Un objeto ***StreamIO*** es responsable de codificar los datos dentro de un stream usando una representación canónica y además se encarga de restaurar el contenido del stream.

OBJECT LICENSING SERVICE

"A medida que crece el avance de las redes sobre las aplicaciones de escritorio, se vuelve más difícil controlar el uso de los componentes. Los componentes serán desarrollados por fuentes externas e invadirán las organizaciones por el uso de las redes."

Gartner Group
(Febrero de 1995)

Para que el mercado de los componentes evolucione, las tecnologías de administración de licencias de software deberán contemplar las licencias sobre componentes. Y los componentes deberán desarrollarse para registrarse automáticamente con los administradores de licencias. El servicio de administración de licencias de CORBA cumple con todos estos requisitos. Básicamente, permite medir el uso de los componentes y facturar apropiadamente. Los mecanismos de licenciamiento tradicionales–por ejemplo, licencias sobre un site completo o sobre nodos–no sirven para el mundo de los componentes distribuidos con ORBs. Se requieren mecanismos más flexibles para medir el uso de cada componente.

¿Qué hace el Licensing Service?

Permite manejar un abanico muy amplio de opciones, apropiadas para cualquier tipo de negocio. Por ejemplo: permite ofrecer un "período de gracia" a los potenciales usuarios para que prueben los componentes; permite garantizar que las licencias siempre estén disponibles a los consumidores de prioridad alta; también es posible usar la misma licencia sobre un grupo de componentes, etc. El Licensing Service separa el requisito *"Quiero ser controlado"* del requisito *"Cómo debo ser controlado"*. El componente notifica al servicio cuándo desea ser controlado sin entrar en los detalles de cómo se gestiona el control.

El servicio permite recolectar métricas sobre el uso de los componentes. Esto es útil para determinar cuáles son los componentes más utilizados. Todas las licencias deben tener un momento de inicio y una duración o momento de finalización. Las licencias se otorgan a usuarios específicos, a grupos de usuarios o bien a organizaciones completas.
Y finalmente, el Licensing Service debe ser un recurso seguro en el servidor. No queremos un sistema impostor que libere las licencias de manera indiscriminada.

Las interfaces del Licensing Service

> "El negocio de los componentes debe protegerse a si mismo de la copia y distribución de componentes sin retribución."
>
> - Dr. Ivar Jacobson, Author
> Object-Oriented Software Engineering
> (Addison-Wesley, 1993)

El servicio se descompone en dos interfaces que suministran todas las operaciones que necesita un componente para protegerse a si mismo con licencias. Los componentes usan estas interfaces para que el servicio monitoreador sepa cuándo están siendo usados (incluyendo quién los usa y durante cuánto tiempo).

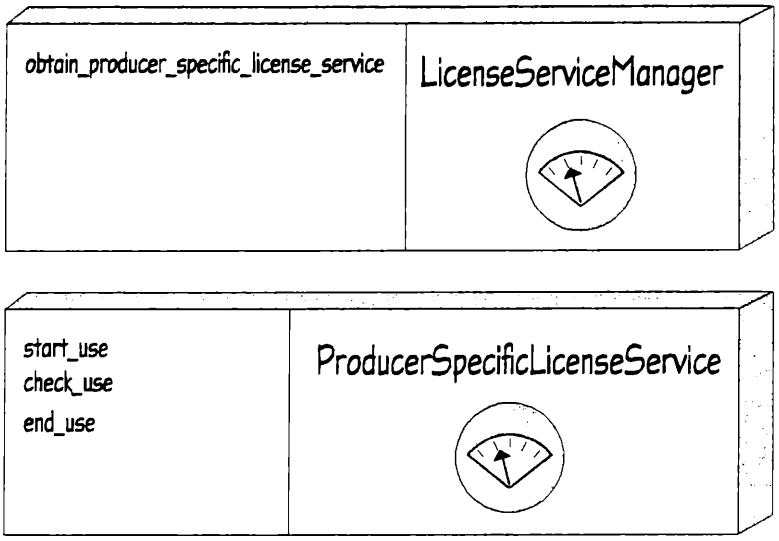


Figura 2-44. Interfaces del Licensing Service

- ***LicenseServiceManager*** es como un mini-agente que localiza servicios de licencias que implementan determinadas características. Nos permite ponernos en contacto con un servicio apropiado a cada componente. La interface define una única operación, con un nombre muy largo: *obtain_producer_specific_license_service*. Retorna una referencia a un servicio particular. El *"producer"* es el componente que se licencia. Es la propiedad intelectual producida por la que reclamamos una recompensa.
- ***ProducerSpecificLicenseService*** suministra tres operaciones para hacer todo el trabajo. El componente licenciado invoca *start_use* la primera vez que es utilizado. Se debe pasar por parámetro toda la información asociada al usuario que usa el componente. Con *stop_use* el componente indica que ha terminado de ser usado. Por otro lado, invoca *check_use* periódicamente, durante el uso para que el servicio sepa que la conexión con el usuario aún está viva. El servicio se encarga de verificar el momento de expiración de la licencia y retorna un mensaje apropiado. ¿Cómo sabe el componente cuándo debe invocar *check_use*? Lo sabe consultando al servidor a un intervalo de tiempo preestablecido o bien, asicrónicamente, el servidor le envía una notificación–evento–para que haga la invocación.

Un escenario con manejo de licencias

En la siguiente figura representamos un escenario que muestra la interacción entre un componente con licencia y el Licensing Service. Veamos la secuencia de pasos:

1.- ***Obtener una referencia a un servicio de licencias.***
Todo componente protegido con una licencia debe obtener una referencia a un servicio de licencias que implemente una política apropiada al componente. Esto se hace invocando *obtain_producer_specific_license_service* en la interface ***LicenseServiceManager.***

2.- ***Notificar al servicio cuando el cliente comienza a usar el componente.*** Debemos determinar qué significa usar un componente y cómo queremos cobrar la recompensa. En uno de los extremos, podríamos cobrar por la invocación de cada método. En el otro, podríamos cobrar solamente la conexión de acceso a una colección de objetos. En cualquier caso, debemos invocar *start_use* para indicar al servicio que el componente ha comenzado a ser usado. Se debe pasar el nombre del componente, su versión, y una referencia para una posterior notificación de eventos. También se debe pasar el contexto del usuario, que el servicio utiliza para determinar cómo tratar con el usuario del componente–por ejemplo, podríamos querer formular las siguientes preguntas: ¿Es la liecencia del usuario válida? ¿Cómo debería ser recargado por el uso del componente?. El servicio envía instrucciones sobre cómo proceder en el parámetro retornado al invocar *action_to_be_taken*.

3.- ***El servicio envía una notificación de evento.*** El servicio indica al componente que debe hacer *check_use*. Lo hace con el Event Service de CORBA. Alternativamente, el componente podría disparar *check_use* a intervalos de tiempo especificados por el server.

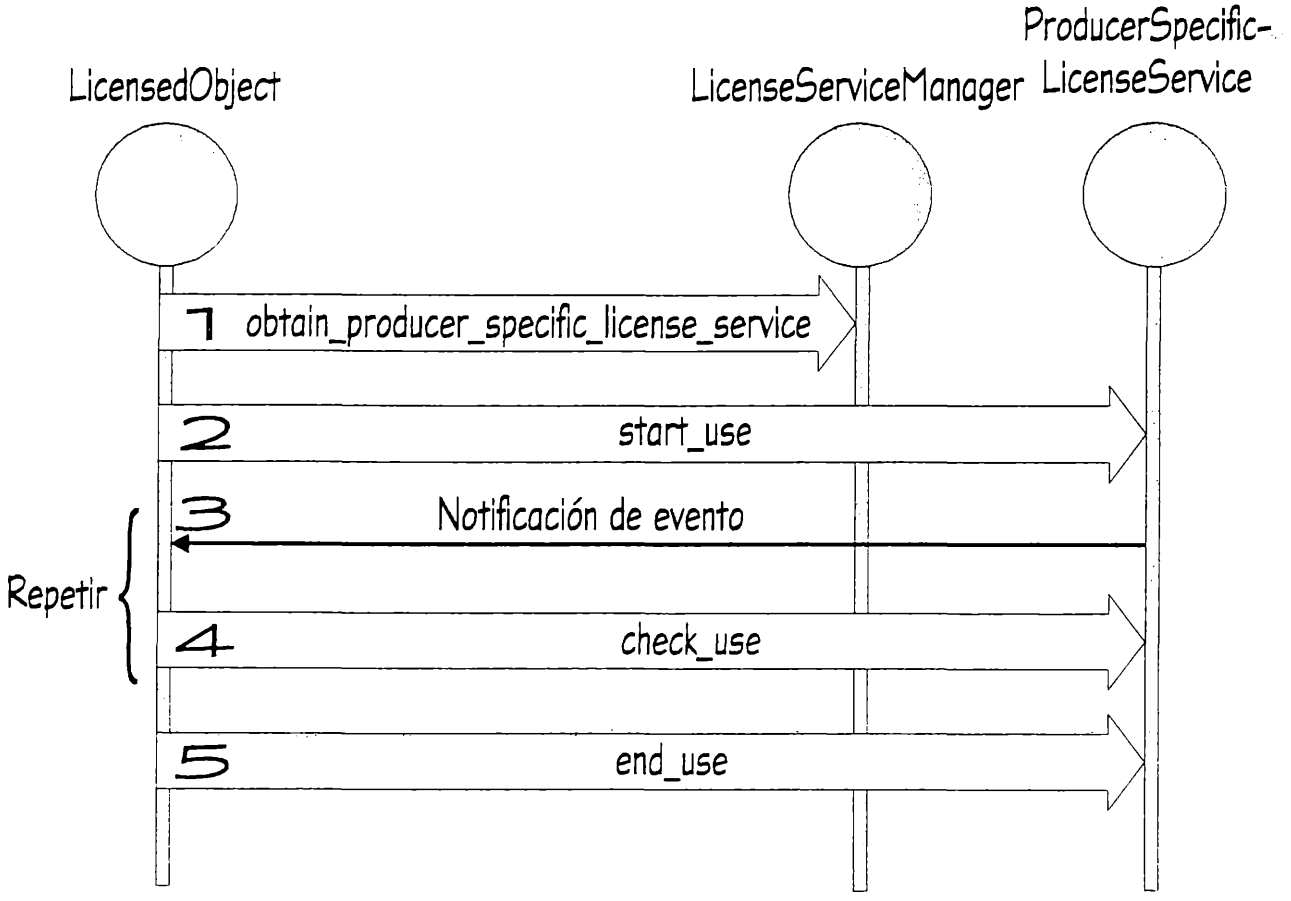


Figura 2-45. Escenario: cómo un componenet protegido por una licencia interactúa con el servicio

4.- ***Enviar el mensaje check_use.*** Invocando *check_use* para indicar al servicio de licencias que el componente está siendo usado. La invocación retorna instrucciones del servicio de licencias. Esto permite que el servidor de licencias y el componente estén continuamente al tanto de la existencia mutua cuando el componente está en uso.
5.- ***Informar al servicio cuando el componente deja ser usado.*** Invocando *stop_use*, se indica al usuario que el suario (o cliente) deja de usar el componente.

En suma, podemos decir que se debe notificar al servicio cuando el componente se usa por primera vez, mientras está siendo usado y cuando se termina de usar.

OBJECT PROPERTY SERVICE

Este servicio permite asociar dinámicamente nuevos atributos a los componentes. Es posible definir estos atributos dinámicos (o propiedades) en tiempo de ejecución sin usar IDL. Así que podemos asociarlos a objetos ya existentes. Una vez definida una propiedad, se le fija un nombre, se asigna y consulta su valor, se definen los modos de acceso y eventualemnte se la elimina. En contrapartida, con los atributos definidos por IDL, sólo es posible asignar y consultar el valor. No se puede crear al vuelo un atributo con IDL, definir el modo de acceso o borrarlo.

Técnicamente, las propiedades son valores tipados que tienen un nombre y que se asocian dinámicamente a un componente, por afuera del esquema de tipos de IDL. Por ejemplo, podríamos agregar un propiedad de *archivo* a un documento ya disponible y marcar el documento como listo para ser *archivado*. La información de *archivo*–la propiedad–está asociada al documento, pero no es parte de su estructura interna.

Interfaces del servicio de propiedades

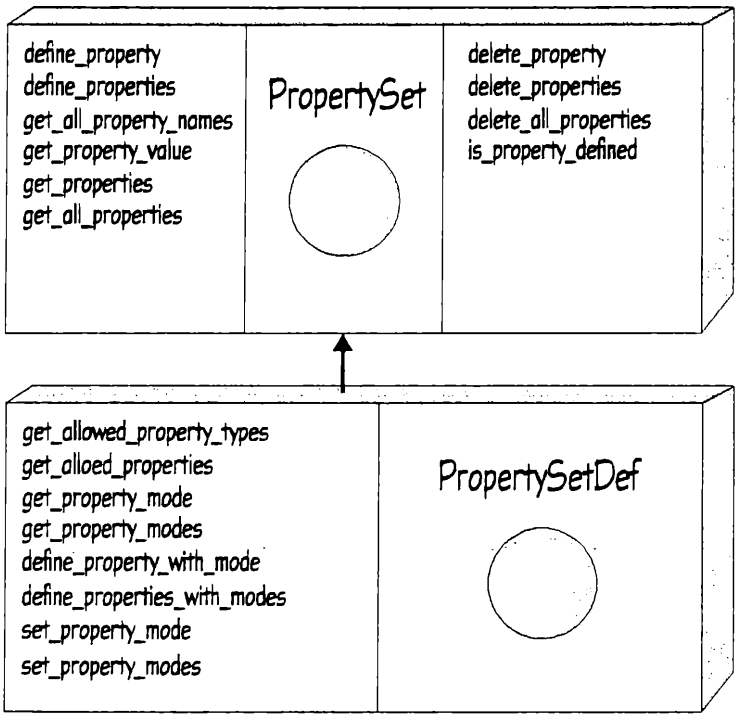


Figura 2-46. Las principales interfaces del Properties Service

Este servicio está conformado por seis interfaces; cuatro de las cuales son fábricas e iteradores. Las dos interfaces principales son ***PropertySet*** y ***PropertySetDef***. ***PropertySetDef*** es subclase de ***PropertySet*** y suministra operaciones adicionales para manipular y controlar el modo de la propiedad.

El servicio de propiedades define cuatro modos mutuamente excluyentes: 1) *normal* significa que no hay restricciones sobre la propiedad; 2) *read-only* significa que el cliente puede leer y borrar la propiedad pero no modificarla; 3) *fixed-normal* significa que la propiedad puede ser modificada pero no eliminada y 4) *fixed-readonly* significa que la propiedad solamente puede leerse. Podemos pensar que estos modos son metainformación que define las restricciones sobre la propiedad. Todo objeto que soporte propiedades debe implementar alguna de las interfaces ***PropertySet*** o ***PropertySetDef***.

- ***PropertySet*** define diez operaciones para definir, eliminar, enumerar y chequear la existencia de propiedades. Suministra operaciones de lectura de los valores y metainformación. También hay operaciones para manejar conjuntos de propiedades como un todo. Estas son operaciones cuyo nombre termina con *"s"*. Notar que cada propiedad está definida por un *nombre* (string), un *valor* (de cualquier tipo) y un *modo*.
- ***PropertySetDef*** define ocho operaciones para controlar y modificar los modos de las propiedades. Es posible manipular estos modos individualmente o en grupos.

Es fácil asociar una propiedad a un objeto y luego manipularla individual o grupalmente (esto es especialmente útil en ambientes distribuidos).

OBJECT TIME SERVICE

Mantener un noción global del tiempo es sumamente importante para ordenar eventos que ocurren en sistemas de objetos distribuidos. ¿Cómo logra un ORB sincronizar los relojes de máquinas distintas? ¿Cómo se compensan los desfasajes entre las sincronizaciones? ¿Cómo se recrea la ilusión de que todas los relojes de todas máquinas avanzan simultaneamente?.
La respuesta obvia es el Object Time Service.

El servicio resuelve el problema del tiempo distribuido usando las siguientes técnicas complementarias:

- Periódicamente se sincronizan los relojes de todas las máquinas de la red. El servicio podría definir un agente en cada máquina–DCE se refiere a estos agentes como *Time Clerks*–que pregunte a los servidores globales de tiempo (Time Servers) por la hora correcta y hagan los ajustes correspondientes a la hora local. El agente puede consultar a más de un Time Server y luego calcular la hora correcta más probable y el error basándose en las respuesta recibidas. Por otro lado, el agente puede actualizar la hora local gradualmente o en forma abrupta.
- Podrían introducirse factores de corrección para compensar los desfasajes ocurridos en la sincronización. Se pueden configuar los agentes para que conozcan las limitaciones y retrasos de funcionamiento del clock local (hardware). Luego, los mismos agentes se pueden encargar de calcular los factores de corrección y preguntar la hora local cuando estiman que el factor de corrección sobrepasa cierto límite.
- Usar objetos Time Servers que respondan consultas sobre el tiempo. Siguiendo el modelo DCE, un ORB puede suministrar al menos tres time servers; uno de ellos debe estar conectado a una fuente externa que sumnistre el tiempo actual. Los servidores se consultan mutuamente para ajustar la hora de sus relojes. La fuente externa puede ser un dispositivo de hardware que ajusta la hora por una señal de radio o teléfono (eventualmente, el adiministrador del sistema podría suministrar la hora manualmente). El formato más comunmente usado es el UTC (Coordinated Universal Time) que registra el tiempo transcurrido desde el inicio del calendario Gregoriano–15 de octubre de 1582. Ese tiempo se ajusta teniendo en cuenta las diferentes zonas horarias (por ejemplo, -5 hs. en New York).

OBJECT SECURITY SERVICE

Los objetos distribuidos tienen todos los problemas de seguridad de los sistemas cliente/servidor tradicionales–y más. El ambiente cliente/servidor introduce nuevas amenazas a la seguridad, más allá de las que encontramos en los sistemas tradicionales de tiempo compartido. En los sistemas cliente/servidor, no se puede confiar en ninguno de los sistemas operativos clientes de la red para que protejan del acceso no autorizado los recursos del servidor. Y aún cuando la máquina del cliente fuera completamente segura, la red misma es altamente accesible. Nunca se puede confiar en la información que está en tránsito. Los dispositivos sniffer pueden ratrear fácilmente el tráfico entre las máquinas e introducir caballos de Troya al sistema. Esto significa que los servidores deben buscar todas las maneras de protegerse a si mismos sin obstaculizar el uso normal de los usuarios. Además de todas estas amenazas, los objetos distribuidos presentan las siguientes complicaciones:

- ***Los objetos distribuidos pueden jugar al mismo tiempo los roles de clientes y servidores.*** En una arquitectura cliente/servidor tradicional, está bien claro quién es un cliente y quién es un servidor. Normalmente, se puede confiar en los servidores, pero no en los clientes. Por ejemplo, un cliente confía en un servidor de base de datos, pero la recíproca no vale. En los sistemas de objetos distribuidos, no podemos distinguir claramente entre clientes y servidores. Estos son roles meramente alternativos que el mismo objeto puede tomar.
- ***Los objetos distribuidos evolucionan continuamente.*** Cuando se interactúa con un objeto, sólo vemos la punta del iceberg. Tal vez el objeto en cuestión delegue parte de su implementación sobre otros objetos–no visibles–que se cambian en tiempo de ejecución, dinámicamente. También, por la subclasificación, las implementaciones de un objeto pueden variar con el tiempo, sin afectar el funcionamiento previsto originalmente.
- ***Las interacciones entre objetos distribuidos no son bien entendidas.*** Debido al encapsulamiento, no es posible saber con certeza todas las interaccciones que tienen lugar entre los objetos. Hay mucha actividad que ocurre *"detrás de escena"*.
- ***Las interacciones entre los objetos distribuidos son menos predecibles.*** Como los objetos distribuidos son más flexibles y granulares que otras formas de sistemas cliente/servidor, pueden interactuar de muchas otras maneras. Esta es una gran ventaja de los objetos distribuidos, pero también un gran riesgo para la seguridad.
- ***Los objetos distribuidos son polimórficos.*** Los objetos son flexibles; es fácil reemplazar uno por otro que adhiera a la misma interface. Esta situación es tentadora para los "caballos de Troya"; pueden personificar objetos legítimos y luego hacer estragos.
- ***Los objetos distribuidos no tienen límites de escalabilidad.*** Como cualquier objeto puede ser un servidor, el ORB tendría que administrar el acceso a millones de objetos.
- ***Un entorno distribuido es inherentemente anarquista.*** Los objetos van y vienen. Se crean dinámicamente y se autodestruyen cuando no son necesarios. Este dinamismo, por supuesto, otra gran ventaja de los objetos, es una pesadilla para la seguridad.

Para mantener la ilusión de un sistema único, cualquier usuario confiable (un objeto) debe tener acceso transparente a todos los otros objetos. ¿Cómo se logra esto si cada PC

conectada representa una amenaza potencial a la seguridad de la red? ¿Es necesario que los administradores deban otorgar permisos de acceso para cada objeto posible?

Las buenas noticias son que muchos de estos problemas se resuelven moviendo la implementación de la seguridad al ORB mismo. El ORB debe administrar la seguridad para un amplio rango de sistemas, desde dominios simples y confiables (en máquinas con un solo procesador) hasta las situaciones de inter-ORBs intergalácticos. El desarrollo, administración y portabilidad de componentes que no deban ocuparse de las cuestiones de la seguridad es más fácil. Además, llevar la seguridad al ORB reduce el overhead de performance.

Las distintas propuestas recibidas por el OMG para la especificación del servicio de seguridad ofrecen más que seguridad de nivel *C2* para los objetos distribuidos.
C2 es un estándar del gobierno norteamericano para los sistemas operativos. Requiere que los usuarios y las aplicaciones se identifiquen antes de obtener acceso a cualquier recurso disponible. Para obtener una certificación C2 sobre una red, todos los clientes deben suministrar un ID autenticado del usuario, todos los recursos deben protegerse con listas de control de acceso, deben suministrarse esquemas de auditoría y los permisos de acceso no deben transferirse a otros usuarios que usen los mismo items. Veamos qué características debe tener un ORB para cumplir con este estándar.

Autenticación: ¿eres quien dices ser?

En los sistemas de tiempo compartido, el sistema operativo se encarga de la autenticación usando passwords. Los ORBs deben usar algo mejor que eso, pues cualquier hacker o sniffer podría capturar las passwords y reusarlas. ¡Asi que encriptemos las passwords!
¿Pero quién administrará las claves secretas y todo eso? Afortunadamente, los ORBs tienen una respuesta: los sistemas autenticadores ya disponibles, provistos por terceras partes (por ejemplo, Kerberos) que permiten que los objetos distribuidos se prueben mutuamente que son quien dicen ser.
Es algo así como dos espías que se encuentran en una esquina secreta y susurran las frases mágicas que establecen una *"relación confiable"*. Ambas partes deben obtener la frase mágica por separado, de un tercero que sea confiable. El resultado es que el usuario–llamado principal–es autenticado una sola vez por el ORB y se le asigna un rol, permisos de acceso y un número de seguridad (security clearance).

El cliente autenticado puede acceder a cualquier servidor de objetos desde cualquier lugar–incluyendo oficinas, hogar, teléfonos celelares, etc–usando un único identificador. ¿Cómo funciona esto? Esto se logra con el *contexto de seguridad* que se propaga a través del ORB. El usuario sólo hace el login una vez, pasa la fase de autenticación y obtiene un conjunto de tickets de seguridad (llamados tokens) para los objetos que quiere acceder. Toda esta actividad es realizada–implícitamente–por los mecanismos de seguridad del ORB. No se guarda ninguna password en la fase de login.

El ID autenticado que se propaga automáticamente por el ORB–como parte del contexto. también sirve para los siguientes propósitos: 1) contar las acciones de los clientes (o usuarios); 2) permite a un servidor determinar cuáles recursos puede acceder un usuario; 3) identifica unívocamente quién envía un mensaje; 4) ayuda a los que ofrecen un servicio a determinar a quién facturar por el uso del servicio; y 5) sirve como un privilegio de acceso que un objeto puede traspasar a otros objetos.

Autorización: ¿puedes usar este recurso?

Una vez que los clientes han sido autenticados, los objetos servidores son responsables de verificar qué operaciones pueden realizar los clientes. Los servidores usan ACLs (Access Control Lists) y una variante llamada Capability Lists para controlar el acceso de los usuarios. Las ACLs pueden estar asociadas a cualquier recurso disponible en una computadora. Registran la lista de nombres (y nombres de grupos) de los principales y el tipo de operaciones que pueden realizar sobre cada recurso. Es posible disponer de distintas políticas ACL en el mismo ORB. El ORB puede implementar algunas; el servidor puede implementar el resto.

En un sistema de objetos distribuidos, un objeto es la unidad de control de acceso. Sin embargo, también es deseable controlar el acceso a niveles más finos de granularidad, haciendo que cada método sea un recurso controlado. Pero esto se traduce en un overhead bastante importante para la ejecución. Debe buscarse un balance en el nivel de granularidad para no comprometer el rendiemiento del sistema.

Esquemas de auditoría: ¿dónde has estado?

Los servicios de auditoría permiten a los administradores de sistemas monitorear los eventos del ORB, incluyendo intentos de login y saber cuáles son los servidores u objetos utilizados.

Non-repudiation: ¿este mensaje fue alterado?

Un ORB contar con mecanismos para proteger la información, su envío y recepción. El ORB debe suministrar al receptor una prueba de la identidad del emisor del mensaje y a éste una prueba de la identidad del receptor. Para el tratamiento de estas cuestiones, los ORB cuentan al menos con dos mecanismos: 1) *Encriptación:* permite que dos principales mantengan una comunicación segura; y 2) *Checksums criptográficos*–una solución menos extrema–asegura que los datos recibidos no fueron modificados a través de la red.

OBJECT CHANGE MANAGEMENT SERVICE

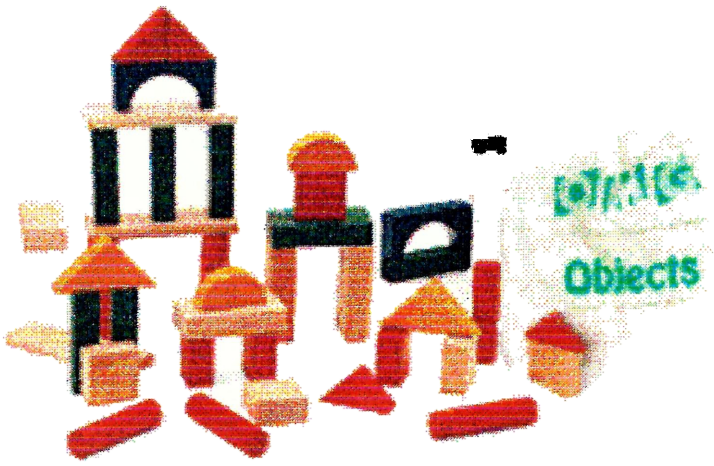
La administración de cambios es otro servicio importante. Permite hacer un seguimiento de las diferentes versiones de los componentes (y sus interfaces) a medida que evolucionan; también registra la historia de su evolución. Esto permite mantener varias versiones del mismo componente, de su interface e implementación.

El Change Management Service asegura que una instancia de un objeto usa una versión consistente de su implementación. Por ejemplo, un versión vieja de una instancia de un objeto usará una versión vieja de la implementación. La versión de un objeto se codifica directamente en la referencia al objeto. Esto significa que podemos invocar a diferentes versiones del mismo objeto simultaneamente (con distintas referencias). Además cada versión tiene asociado un contexto–por ejemplo: *demo*, *test* y *release*. Normalmente la versión por omisión es la más reciente.

Las versiones de las interfaces son capturadas y registradas en el Interface Repository.
Las versiones de objetos se pueden crear en forma explícita–por ejemplo, invocando *create_version* o implícitamente–por ejemplo, cuando termina una transacción de larga duración.

Parte 3

Frameworks para Business Objects y Componentes



FRAMEWORKS DE CORBA PARA BUSINESS OBJECTS Y COMPONENTES

"Desde el inicio, una promesa importante con la tecnología de objetos era hacer que el software se comportara tal como el mundo real se comporta."

- John R. Rymes, Editor
Distributed Computing Monitor
(Enero de 1995)

¿Qué se requiere para crear componentes que se comporten "como el mundo real se comporta"? El bus intergaláctico de objetos y los servicios a nivel de sistema son un buen punto de partida–permiten que cualquier componente dialogue con cualquier otro.
Pero, ¿qué se dicen estos componentes luego de conectarse? ¿Qué reglas de integración siguen para comportarse como "objetos del mundo real"? ¿Qué clase de infraestructura semántica necesitan? Y ¿cómo construimos, desarrollamos y mantenemos este tipo de componentes?

Para responder a todas estas preguntas, introducimos el concepto nebuloso de *frameworks*. Como es nebuloso, puede significar distintas cosas para diferentes personas. En general, un framework suministra un entorno de ejecución organizado. También suministra herramientas para construir componentes que funcionen respetando las reglas de integración del framework. En el bus de objetos cualquier cosa es posible. En cambio, en los frameworks se maneja cierta anarquía: "cualquier cosa es válida siempre que respete las reglas del framework."

¿Qué son los frameworks?

"El diseño es una disciplina dura. Una manera de evitarla es reusando diseños ya existentes."

- Dr. Ralph Johnson, Universidad de Illinois
(Abril de 1995)

En un artículo publicado en 1991 por R. Johnson y Vincent Russo, se afirma que:

"Una clase abstracta es un diseño para un solo objeto. Un framework es el diseño para un conjunto de objetos que colaboran para cumplir con sus responsabilidades. Así, los frameworks son diseños de mayor escala que las clases abstractas. Los frameworks son una manera de reusar diseños de alto nivel."

Veamos también una explicación proveniente de Taligent (1993):

> "Los frameworks no son simplemente colecciones de clases. Sino que vienen con una funcionalidad muy rica e interconexiones fuertes entre las clases de los objetos que suministran una infraestructura para los desarrolladores."

¿Cómo interactúan los componente con un framework?
En primer lugar, se debe indicar los eventos que se quieren personalizar. Luego, debe suministrarse el código que se encargue de manejarlos. El framework se encarga de invocar ese código cuando los eventos ocurran; el código no invoca al framework. Los programas no deben preocuparse del flujo de ejecución o invocaciones a APIs del sistema; el framework se encarga de todo.

Como los frameworks definen la arquitectura de las aplicaciones que los usan, es importante poder trabajar con frameworks que sean flexibles y extensibles. Normalmente, los framework se adaptan usando herencia para sobreescribir algunos métodos.

CORBA: frameworks y business objects

> "Mientras continuamos focalizando nuestra atención en las cuestiones de la infraestructura interna, también estamos incluyendo areas importantes para los usuarios finales y consumidores de tecnología como las Common Facilities, Application Objects-business objects y application frameworks-y dominios verticales como Salud, Finanzas, etc."
>
> - Chris Stone, President of OMG
> (Mayo de 1995)

Después de haber resuelto muchas de las cuestiones de infraestructura interna, el OMG está trabajando en una infraestructura para business objects que se comporten como "tal como el mundo real se comporta". Para lograrlo, el OMG define *frameworks verticales* y *horizontales* que se levantan a partir del bus de objetos y servicios a nivel del sistema. Estos frameworks se conocen con el nombre de *Common Facilities*.

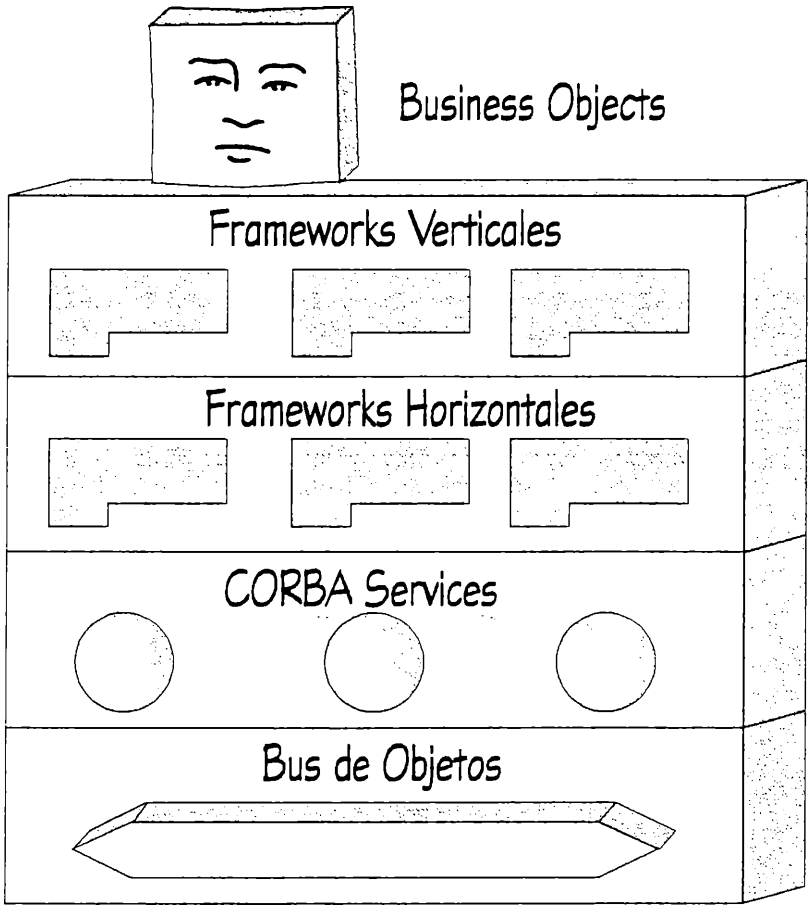


Figura 3-1. Los diferentes niveles de la infraestrcutrura de CORBA para el desarrollo de componentes

¿Qué es un business object?

"Un business object es la representación de una cosa activa en un dominio de negocios, incluyendo al menos un nombre, una definción, atributos, comportamiento, relaciones y restricciones. Un business object puede representar, por ejemplo, una persona, un lugar, un concepto."

- OMG, BOMSIG
(Octubre de 1994)

Los business objects se usarán para diseñar sistemas que modelen procesos de negocios específicos. En el mundo real, los objetos rara vez actúan aisladamente, sino que están relacionados entre si. Para modelar a sus contrapartes del mundo real, los business objects deben ser capaces de comunicarse entre si a un nivel semántico. Estas interacciones se capturan usando metodologías bien conocidas–Uses Cases de Jacobson, Diagramas de Interaccion de Booch, etc.

Un business object debe ser capaz de reconocer los eventos que ocurran en su entorno, modificar sus propios atributos e interactuar con otros business objects. Como cualquier otro objeto CORBA, un business object expone su interface a sus clientes usando IDL y se comunica con otros objetos a través del ORB.

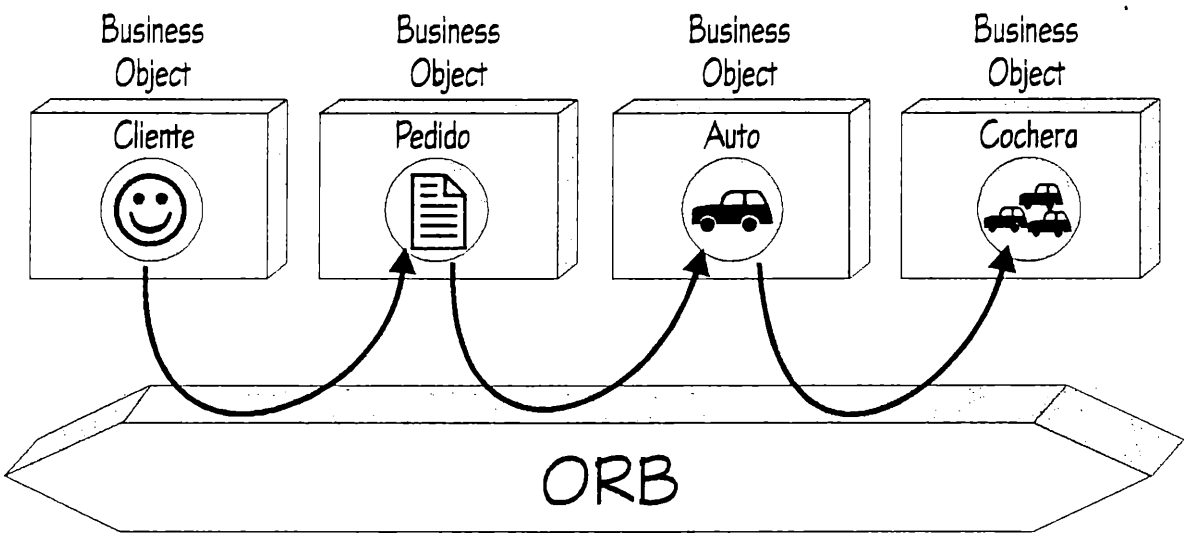


Figura 3-2. Un sistema de reserva de autos con business objects

En la figura mostramos una suite de cuatro business objects que forman parte de un sistema de reservaciones de autos: *cliente*, *pedido*, *auto* y *cochera de autos*. La *cochera de autos* es un business object que contiene a otros business objects, los *autos*.
Está claro que estos business objects tienen cierta semántica común para comunicarse entre si y realizar las transacciones comerciales. Subyacentemente, podrían usar el Transaction Service de CORBA para sincronizar sus acciones.

El principal beneficio de usar esta arquitectura basada en CORBA es que todos los objetos tienen una interface definida en IDL y que funcionan sobre un ORB, por lo cual no importan las cuestiones relativas a la localización fisica, sistemas operativos, redes, etc.

CORBA: COMMON FACILITIES

"En los grandes sistemas, encontramos cientos de abstracciones construidas una encima de la otra. En cualquier nivel de abstracción, encontramos grupos de objetos que colaboran para cumplir con algún comportamiento de mayor nivel."

Gardy Booch, Autor
Object-Oriented Analysis and Design
(Benjamin-Cummnings, 1994)

¿De dónde obtienen los business objects CORBA sus habilidades de colaboración? Estas habilidades provienen de las reglas de integración del framework subyacente, formatos comunes de datos y demás características de la arquitectura.
Estos frameworks definen los puntos estables de diseño para el sistema y la interacción de los componentes. A medida que la tecnología evoluciona, se pueden agregar, modificar y eliminar business objects, mientras el framework se mantiene estables. En otras palabras, se puede decir que el framework trasciende el ciclo de vida de los componentes individuales. Las facilidades comunes de CORBA (Common Facilities) representan frameworks para el desarrollo de componentes de alto nivel y completan la visión CORBA de la interoperabilidad.

¿Qué son las Common Facilities?

Las Common Facilities son colecciones de frameworks definidos en IDL que suministran servicios útiles para los business objects. Se pueden considerar el siguiente escalón de la jerarquía semántica. Las dos categorías de Common Facilities (verticales y horizontales) definen reglas para ensamblar componentes que colaboren entre si. Hay cuatro tipos de facilidades horizontales: *user interface*, *information management*, *system management* y *task management*. Las facilidades verticales suministran frameworks con interface IDL para segmentos especializados del mercado: salud, finanzas, etc.

CORBA HORIZONTAL COMMON FACILITIES

Common Facilities: User Interface

Trata con tecnología de documentos compuestos y servicios de *in-place editing* similares a los que suministran OpenDoc y OLE. La idea es ofrecer un framework para compartir y subdividir una ventana de tal forma que diferentes componentes puedan compartir el espacio visual transparentemente. Obviamente, se requieren ciertas reglas estrictas para lograr esta integración visual. Estos protocolos visuales deben resolver las siguientes cuestiones: administración de la geometría visual, dispatching de ventos de la interface a los componentes y recursos compartidos (por ejemplo menues y barras de herramientas).

User Interface también se ocupa del scripting. Esta tecnología es fundamental para los agentes, transacciones de larga duración, etc. Como los lenguajes de scripting usan mecanismos de late-binding–en contraposición al binding determinado durante la compilación–es posible crear y agregar scripts al vuelo sobre componentes y documentos compuestos. Esta tecnología, denominada *automation,* permite definir colaboraciones dinámicas entre los componentes.

Usando un lenguaje de scripting, sería posible invocar servicios CORBA que tengan definida una iterface IDL. El lenguaje de scripting se vale las interfaces dinámicas para la invocación de los servicios en el ORB. Por ejemplo, con un script podríamos copiar y mover objetos usando el Life Cycle Service de CORBA.

Common Facilities: Information Management

Incluye servicios para almacenar e intercambiar datos de documentos compuestos similares a los ofrecidos por OpenDoc y OLE. También define estándares que los componentes usan para codificar y representar su estado interno y para definir e intercambiar metainformación. Todas estas caraterísticas serán tomadas de otros estándares ya disponibles, como la tecnología de OpenDoc-Bento, de Apple y CI Labs.

Common Facilities: System Management

Incluye interfaces y servicios para administrar, instrumentar, configurar, instalar, operar y reparar componentes de objetos distribuidos. Para que un componente distribuido pueda ser administrado, debe implementar las interfaces de administración definidas en IDL. Este framework define las siguientes interfaces:

- ***Instrumentation:*** permite recolectar información sobre la carga de trabajo de un componente, salida, consumo de recursos, etc.
- ***Data Collection:*** para recolectar información sobre eventos históricos relacionados al componente. Esta interface permite administrar la historia del componente–registrada como un log de eventos.
- ***Security:*** para administrar el sistema de seguridad. El Sercurity Service implementa los mecanismos de seguridad.
- ***Event management:*** para generar, registrar, filtrar y propagar notificaciones de ventos. Se levanta sobre el Event Service de CORBA

Common Facilities: Task Management

Suministra un framework para administrar transacciones de larga duración, agentes, automation, etc. Este framework incluye mensajes semánticos para administrar pedidos orientados a tareas. Una tarea (task) puede estar formada por una o varias operaciones (por ejemplo una secuencia de consultas a una base de datos).

CORBA VERTICAL COMMON FACILITIES

Estas facilidades suministrarán interfaces definidas en IDL y estándares para la interoperabilidad de componentes sobre segmentos verticales, como: salud, finanzas, administración, etc. Algunos ejemplos de frameworks verticales que están actualmente en desarrollo son:

- ***CORBA for Information superhighways:*** este framework utiliza a CORBA como médula de una autopista de información. Usa los servicios disponibles de publicar, medir, monitorear, descubrir, e-commerce, etc.
- ***CORBA for Computer Integrated Manufacturing (CIM)***. Este framework usa CORBA para integrar funciones comunes de manufacturación–incluyendo control de procesos, control de calidad, CAD, manejo de inventarios, etc. La idea es crear lineas de manufacturación ágiles usando la tecnología de objetos distribuidos.
- ***CORBA for Distributed Simulations.*** Este framework se puede usar para simulaciones de control de tráfico areo, video juegos, etc. La idea es configurar una simulación eligiendo componentes CORBA, especificando sus conexiones y suministrándoles datos.
- ***CORBA for Oil and Gas Exploration.*** Este framework controla procesos de búsqueda y recuperación de recursos naturales. Involucra grandes volúmenes de datos, complejos algoritmos y grandes almacenamientos de información.
- ***CORBA for Accounting.*** Este framework suministra un sistema de cuentas distribuido que incluye transacciones bursátiles, intercambios de monedas, ventas, compras, etc.

Esta era sólo una lista parcial de frameworks. Dentro del OMG ya hay grupos trabajando en areas como CORBA for Health (Salud), CORBA for Telecos (Telemunicaciones).

CONCLUSIONES

CORBA es un estándar definido por el OMG con el propósito principal de definir interfaces para software interoperable usando una tecnología de orientación a objetos. La especificación, CORBA, es un estándar avalado con el concenso de la industria. Define una infraestructura de alto nivel para la computación distribuida. En general, la orientación a objetos permite el desarrollo de software modular y reusable y está llevando la tecnología hacia el desarrollo de componentes con características de plug-and-play. El OMG está extendiendo esos beneficios para que estén disponibles a través de sistemas distribuidos heterogéneos.

El ORB es el núcleo que resuelve la comunicación entre los componentes, para que puedan co-existir. Los componentes pueden ser objetos individuales de granularidad media, grupos de objetos que colaboran mutuamente o bien aplicaciones monolíticas ya existentes no orientadas a objetos. Todos se empaquetan con interfaces CORBA. También denominados business objects, estos bloques de construcción emplean mecanismos orientados a objetos como la herencia y el polimorfismo, para obtener una lógica de aplicaciones de mayor nivel.

Un componente CORBA se comporta como un objeto en el rol de servidor para otro objeto, en el rol de cliente, aún cuando ninguno de los dos estén implementados en lenguajes orientados a objetos. Este encapsulamiento de las aplicaciones en componentes se logra escribiendo las interfaces en IDL. IDL es un lenguaje completamente neutral y declarativo, que no especifica ningún detalle sobre la implementación de los objetos. Una especificación en IDL no es un modelo de implementación, sino más bien un modelo de servicios e interfaces.
En particular, el modelo de objetos de CORBA se basa en la separación de la interface de la implementación. Podemos decir que desde el comienzo CORBA estuvo orientado a la integración de subsistemas de componentes o servidores, más que al nivel de objetos individuales, de escasa granularidad.
El IDL de CORBA no es OO puro, porque aunque soporta herencia de interfaces es más bien una herramienta para encapsular entidades que se presenten a los clientes como objetos. Por otro lado, están los bindings para convertir las especificaciones IDL a lenguajes como C++, Smalltalk, C, COBOL, ADA95, etc. Esto permite a las organizaciones describir los objetos de aplicaciones y servicios con una interface IDL para luego entregarlas a los desarrolladores de software, que las implementan usando distintos lenguajes de programación. Así, un componente CORBA (con interface IDL) sirve como especificación de interface o contrato entre los distintos departamentos, secciones, aplicaciones o entidades dentro de otros subsistemas de la organización.

CORBA no sólo es neutral con los lenguajes de programación, sino que también es neutral a la orientación a objetos. Esto significa que el ORB permite la integración de módulos y aplicaciones no orientadas a objetos. Piezas de código no orientadas a objetos pueden interactuar entre si o con subsistemas de objetos puros.

La razón de la neutralidad de CORBA es la necesidad de integrar software y componentes implementados en distintos lenguajes, dentro de la misma organización. Es así como el objetivo con el IDL es contar con un lenguajes descriptivo.

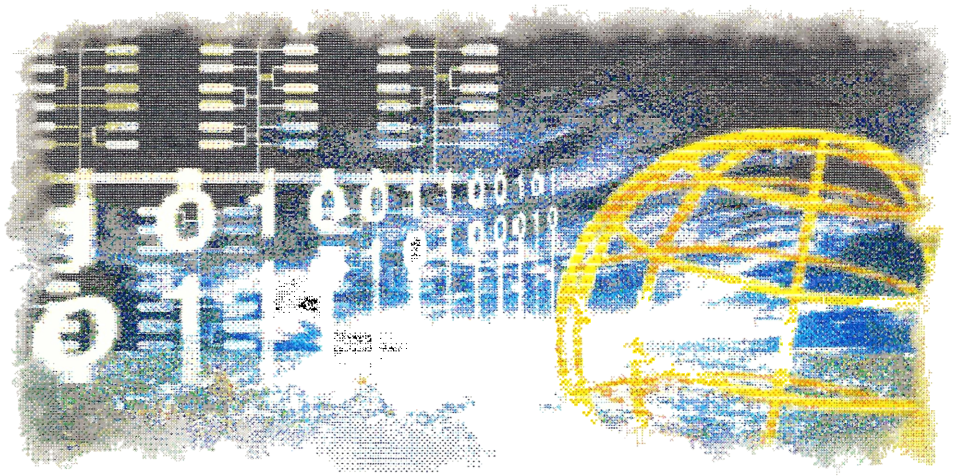
El propósito perseguido con los ORBs en términos de funcionalidad es en algún sentido estrecho: suministrar interfaces de acceso a objetos remotos (componentes) y hacer las veces de router entre los objetos locales y los remotos. CORBA no llega más allá de esta funcionalidad básica por una razón simple: dejar margen para la proliferación de ORBs especializados que cubran los diferentes requerimientos en el espectro de aplicaciones, desde las más simples (escritorio) hasta sistemas e infraestructuras completas en el nivel corporativo.
El denominador común es la interoperabilidad. Sin embargo, tal vez se trate del menor denominador común, pues no permite explotar las ventajas de los lenguajes orientados a objetos nativos, para optimizar la funcionalidad y performance. Por ejemplo: las interfaces IDL no suministran soporte para callbacks de comportamiento asincrónico, envío de mensajes a muchos objetos simultáneamente, etc, pues estas características no son comunes a todos los lenguajes que acceden a un ORB.

De acuerdo al OMG, CORBA se inventó para suministrar una nueva infraestructura que resuelva el problema de interoperabilidad del software. Reemplaza las infraestructuras basadas en RPC y tecnologías precentes, incorporando una interface sobre redes orientada a objetos que simplifica enormente la computación distribuida. Las principales razones por las cuales creemos que CORBA es una tecnología importante para el futuro son:

- ***IDL:*** una notación universal para interfaces de software.
- ***Infraestructura:*** el ORB simplifica la computación distribuida.
- ***Especificación de estándares:*** facilita el desarrollo incorporando el reuso de diseños, servicios de software disponibles comercialmente, interoperabilidad y reuso de código.

Parte 4

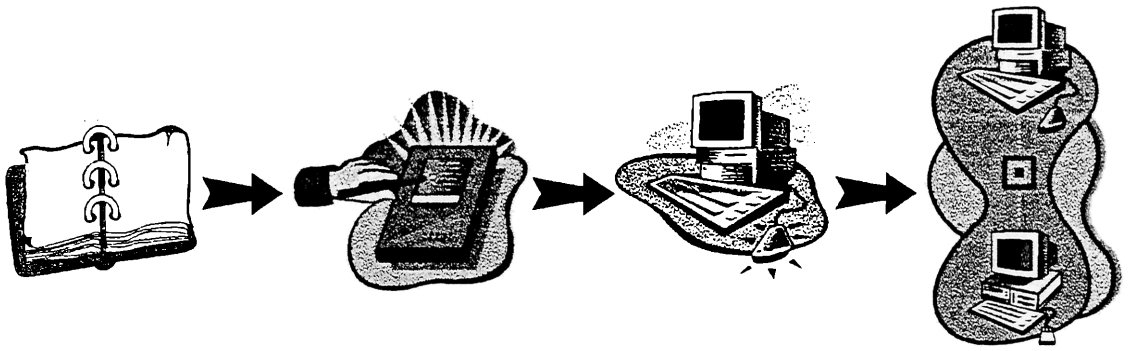
Desarrollo de una Aplicación Distribuida con CORBA



DEFINICION DEL PROYECTO

Odyssey

Las agendas personales han sufrido una gran evolución a lo largo del tiempo. Al principio estaban las agendas de papel para escribir manualmente las actividades, tareas, reuniones y direcciones. Luego aparecieron las agendas electrónicas de bolsillo, para permitir una consulta rápida de la información. La tercera generación de agendas se incoporó recientemente al desktop de las computadoras para brindar administración sofisticada de la información personal, organizar mensajes, citas, contactos, tareas, seguimiento de actividades y mucho más.



Con Odyssey proponemos definir y explorar la siguiente etapa incorporando el concepto de distribución. Las ***agendas personales distribuidas*** están especialmente preparadas para solucionar el problema de contactar y sincronizar actividades entre varias personas que no están en el mismo lugar físico.

El objetivo es crear una verdadera red de agendas que cooperen entre si. Para lograrlo, Odyssey define dos tipos de componentes principales:

- Organizer: ofrece la funcionalidad de una agenda para registrar actividades, reuniones y datos de personas. Cada organizer incorpora características de distribución para interactuar con otros organizers.

- Center: sitio virtual donde se registran las agendas distribuidas (organizers). Cualquier organizer puede acceder a un center e interactuar con los otros organizers que se hayan registrado.

Odyssey Organizer. Los servicios de una agenda

Cada usuario dispone de una agenda personal, ***organizer***, que le permite ordenar su tiempo.
En la agenda se pueden registrar actividades y reuniones. También hay disponible un directorio de contactos. Cada agenda es capaz de interactuar con otras agendas para ofrecer servicios tales como la organización de reuniones con otros usuarios que también tengan agendas.

Actividades y reuniones

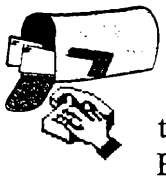
Las actividades representan a las tareas y ocupaciones que un usuario registra en su agenda. Ejemplos: "*cena con clientes hoy a las 21:00 hs.*", "*turno en el dentista entre las 15:30 y 16:15 hs*".
La ocupación de tiempo de las actividades se puede especificar en distintos grados o niveles. Se pueden registrar actividades con definiciones exactas de duración, inicio y finalización pero también es posible especificar los datos parcialmente o de manera imprecisa. En general, podemos decir que las actividades pueden ser ***fijas*** o ***flotantes***.
En las ***actividades fijas,*** la duración es el lapso de tiempo que transcurre entre el inicio y la finalización; por ejemplo, en la actividad: "*clase de chino entre las 14:30 y las 17:30 hs*" la duración es de 3 hs. En las ***flotantes***, la duración es menor al lapso de tiempo comprendido entre el inicio y la finalización: "*tomarme 1 hora entre las 14:00 y las 18:00 hs. para revisar los apuntes de clase*", se refiere a una actividad flotante de una hora de duración que transcurre entre las 14:00 y 18:00 hs.

Las ***reuniones*** son actividades que involucran a muchos participantes. Ejemplos: "*reunión de negocios el próximo jueves a las 17:00 hs. con Olivia y Pedro*". Para organizar una reunión se coordinan las acciones de varias agendas hasta acordar el horario, disponibilidad de tiempo, etc.

Contactos

Una agenda permite registrar los datos de otras personas, empresas u organizaciones usando el concepto de ***contactos.*** Los datos que se registran dependen en gran medida del tipo de contacto. Por ejemplo: cuando se trata de personas tiene sentido guardar el nombre, la fecha de cumpleaños, etc.
A cada contacto se le pueden asociar varios ***medios de contacto*** que permiten llegar hasta ellos para comunicarse. Algunos ejemplos de medios de contacto son: la dirección postal, la dirección de e-mail, teléfono, fax, etc.
Los contactos que disponen de una ***agenda distribuida*** se denominan ***contactos distribuidos.*** Estos contactos ofrecen muchas ventajas para la interacción externa desde otras agendas: concertación de reuniones, consultas sobre la disponibilidad de horarios, etc.

Medios de contacto



Un ***medio de contacto*** especifica una forma de alcanzar o establecer comunicación con otra persona u organización registrada en la agenda (contacto). Algunos ejemplos de medios de contacto son: número de teléfono, domicilio postal, dirección de e-mail, etc.

En el contexto de las agendas distribuidas los medios de contacto permiten el intercambio de información y envío de notificaciones.

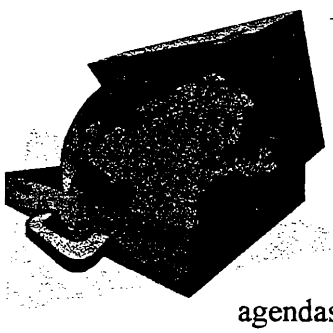
Por ejemplo: para concertar una reunión la agenda puede elegir entre los medios de contacto disponibles para cada participante y enviar las notificaciones de la manera apropiada. En el caso de la dirección de e-mail la agenda enviará un email con el mensaje; en el caso de la dirección postal se imprimirá la carta y el sobre para enviar por correo, etc.

Pero la coordinación de reuniones no es el único uso de los medios de contacto; también sirven para enviar felicitaciones de cumpleaños, mensajes de aviso, etc.

Lo fundamental es que los medios de contacto *canalizan la información hacia los contactos.*

Finalmente, las agendas distribuidas también se consideran medios de contacto, pues permiten enviar notificaciones y concertar reuniones con las personas y empresas registradas en la agenda. Es decir, si una agenda puede interactuar con las agendas de sus contactos, podrá coordinar reuniones, recibir notificaciones, intercambiar información, etc.

Odyssey Centers



Los organizers están preparados especialmente para interactuar entre si y con otros sistemas externos. Esta característica facilita el acceso remoto de los usuarios a sus respectivas agendas y también abre las posibilidad de ofrecer servicios para el trabajo cooperativo entre las agendas. El mejor ejemplo es la sincronización para coordinar reuniones grupales.

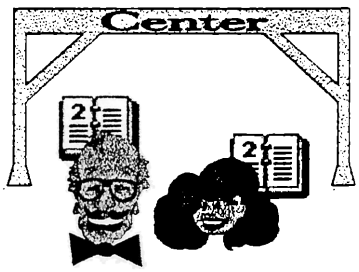
Esta capacidad de interconectar las agendas permite construir una aplicación distribuida sin límites ("red de agendas") escalable a todos los niveles de demanda. Para interconectar las agendas evaluamos las siguientes alternativas:

- ***Opción 1. Solución anarquista.*** El usuario ingresa manualmente las direcciones para conectarse con otras agendas. Es el mecanismo más rudimentario. Una vez que la agenda conoce la dirección de otra agenda, puede guardarla y luego usarla para acceder directamente, sin requerir nuevamente la intervención del usuario. Ventajas: simplicidad. Desventajas: incómodidad en el uso.
- ***Opción 2. Servidores de direcciones.*** Existen sistemas externos especializados que almacenan y publican las direcciones de otras agendas. Estos sistemas pueden verse como *servidores de consulta* para buscar y encontrar las direcciones de otras agendas. La interacción se hace respetando un protocolo de comunicación predeterminado. Los servidores pueden conectarse entre si para facilitar la "navegación" entre las agendas y ofrecen servicios adicionales: estadísticas de uso, mecanismos de seguridad, etc. Ventajas: flexibilidad, comodidad, distribución. Las direcciones de otras agendas se

obtienen *navegando* por los servidores. Deventajas: la complicación de tener que diseñar y programar estos sistemas.

- ***Opción 3. Solución intermedia.*** Crear *carteleras* donde se publican las direcciones de otras agendas. Estas carteleras pueden ser consultadas por otras agendas que también pueden agregar su propia dirección. En ese sentido funcionan como los *servidores de direcciones*, pero son mucho menos sofisticadas, no están interconectadas entre si y no ofrecen los servicios adicionales. Ventajas: solución flexible, distribuida y relativamente fácil de implementar. Más adelante estos sistemas evolucionarán hacia los servidores de direcciones (opción 2).

Los centros de agendas



Los centers son sistemas preparados especialmente para facilitar la interconexión de las agendas distribuidas. Dentro de los esquemas anteriores, los centers se ubican en la tercera categoría y conforman el mecanismo que hemos elegido para fomentar el crecimiento de la "*red de agendas*".

Un center está formado simplemente por un conjunto de organizers suscriptos más algunos servicios adicionales básicos.

Normalmente, cada center tiene un perfil o temática propia que lo diferencia del resto. Por ejemplo: en el center "*Comedy Center*" están registrados los organizers de personas interesadas en hacer reuniones o eventos relacionados al teatro.

Los usuarios pueden entrar a los centros y suscribirse para publicar la dirección de su agenda en las carteleras. También pueden consultar por las agendas suscriptas, obtener sus direcciones, agregarlas a la lista de contactos y luego interactuar con ellas.

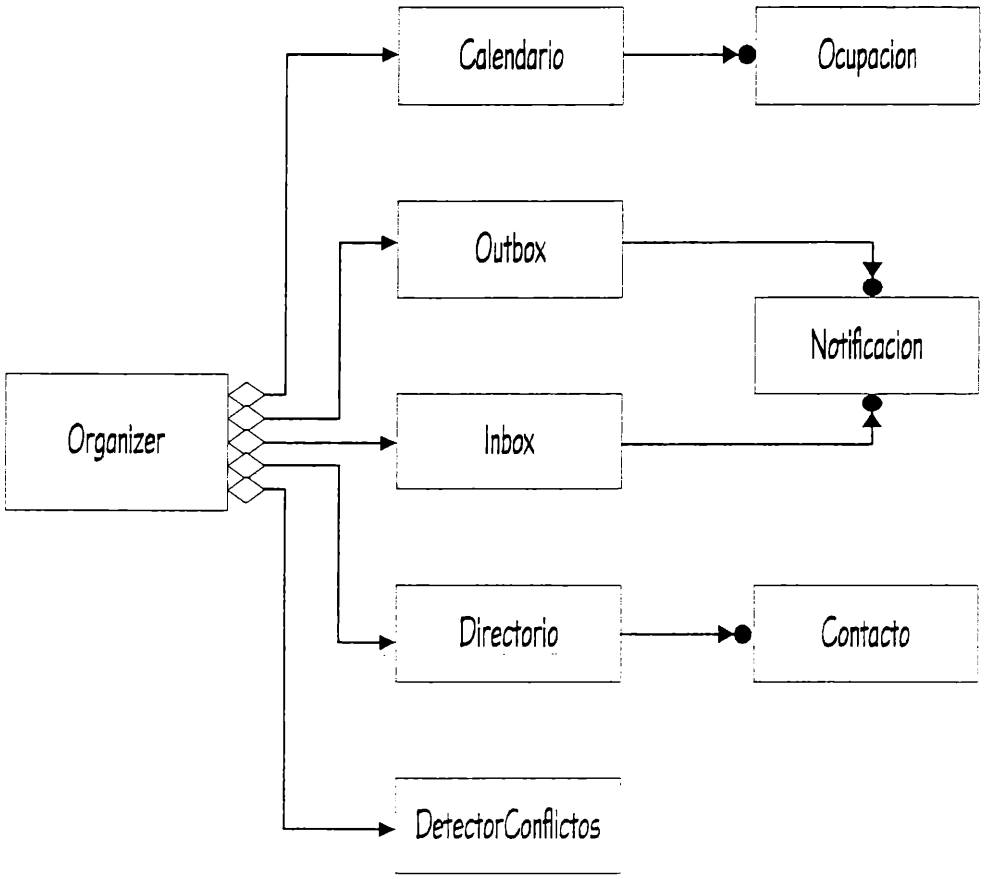
La idea es liberar la creación de los centers a los usuarios para estimular el florecimiento de nuevos servicios y el agrupamiento de las agendas.

DISEÑO

Para describir el diseño usaremos diagramas y una breve explicación informal de las relaciones, colaboraciones y responsabilidades de las clases.

El Organizer

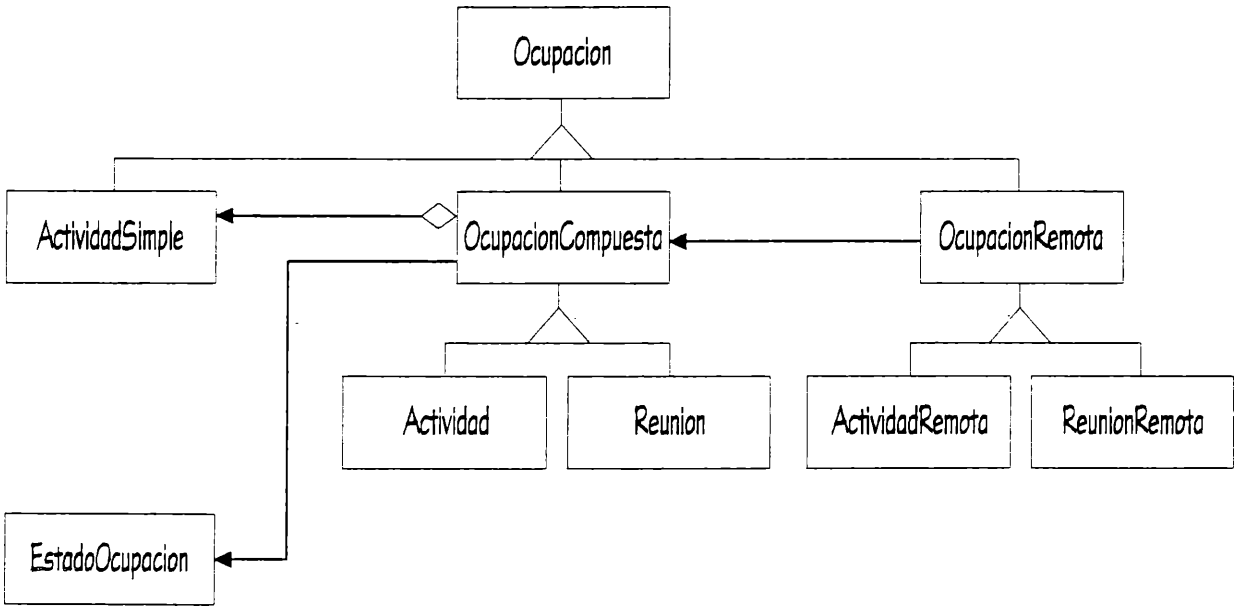
La funcionalidad de cada organizer es suministrada por cinco componentes básicos que colaboran entre si: calendario, outbox, inbox, directorio y detector de conflictos.
El calendario y el detector de conflictos se combinan para ofrecer la funcionalidad de una agenda convencional y registrar tareas y reuniones. El directorio es para guardar toda la información de los contactos (nombre, apellido, medios de contacto, etc). El inbox y el outbox tienen que ver con los aspectos distribuidos de las agendas y registran los mensajes de entrada y salida, respectivamenete, que los usuarios se envían para concertar las reuniones.



La figura anterior muestra el diseño del organizer. La clase de los organizers no es realmente compleja. Se puede decir que todas las responsabilidades quedan del lado de las partes que lo componen (calendario, outbox, etc).

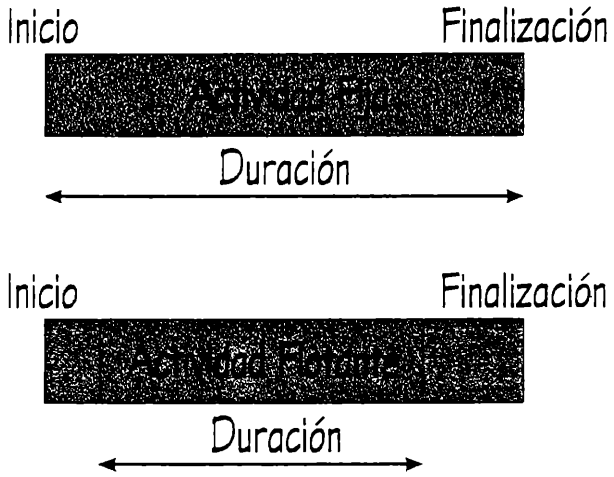
Calendario, actividades y reuniones

El calendario registra las actividades y reuniones del usuario de la agenda. En general, las actividades y reuniones representan ocupaciones. En la figura mostramos la jerarquía de las clases involucradas.



- ***Ocupación:*** es una clase abstracta que agrupa a las actividades y reuniones, locales o remotas.
- ***ActividadSimple:*** una actividad simple tiene una descripción y ciertos parámetros que indican la ocupación de tiempo. Estos parámetros parámetros determinan: 1) *momento de inicio*; 2) *momento de finalización*; 3) la *duración*. Normalmente alcanza con conocer dos de estos parámetros para derivar el tercero. Por ejemplo: dado el inicio y la duración, se calcula el momento de finalización. Los *momentos de inicio y finalización* se especifican con la fecha y hora. La *duración* se puede expresar en distintas unidades: minutos, horas, día, etc. Hay un cuarto parámetro que indica si la actividad es fija o flotante. En una actividad fija, la duración siempre se deriva calculando el lapso de tiempo que transcurre entre el inicio y la finalización. En cambio, en una actividad flotante la duración es menor a ese lapso de tiempo. Las actividades flotantes se usan para describir la ocupación de tiempo de aquellas actividades que tienen una duración determinada pero pueden concretarse en un plazo mayor de tiempo.
- ***OcupaciónCompuesta:*** esta clase agrupa a las clases de las reuniones y actividades locales (registradas en la agenda del usuario). Una ocupación compuesta se compone-salvando la redundancia-de una actividad simple (define la ocupación de tiempo) y un estado. El estado va cambiando durante el ciclo de vida de la ocupación compuesta.

- ***Actividad:*** es una clase concreta. Sus instancias son las actividades que se registran en el calendario.
- ***Reunion:*** también es una clase concreta. Sus instancias son las reuniones que se registran en el calendario. Una reunión se diferencia de una actividad en que tiene muchos participantes, pero ambas se componen de una ocupación de tiempo (ActividadSimple).



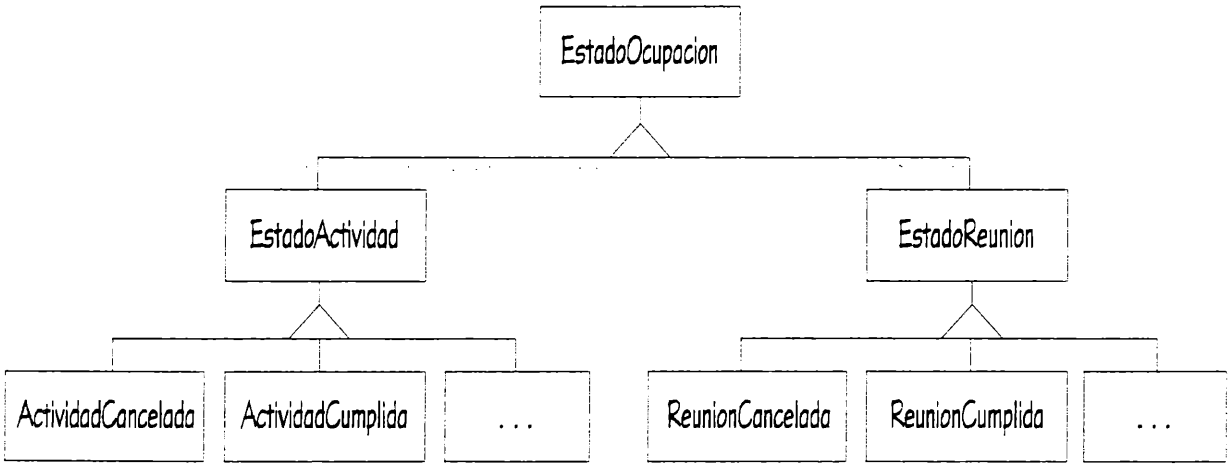
- ***OcupaciónRemota:*** agrupa a las clases de las reuniones y actividades remotas (registradas en la agenda del usuario). Una ocupación remota es un proxy a una ocupación compuesta de otra agenda. Las ocupaciones remotas son capaces de retener localmente ciertos datos de las instancias remotas para optimizar el funcionamiento de las agendas (se comportan como *caches* de los objetos remotos).
- ***ActividadRemota:*** es una clase concreta. Sus instancias son proxies a actividades remotas registradas en otras agendas.
- ***ReunionRemota:*** también es una clase concreta. Sus instancias son proxies a reuniones registradas en otras agendas.

Deteccion de conflictos

Todos los organizers tienen integrado un detector de conflictos. La función de este objeto (***DetectorConflictos***) es controlar si existen superposiciones en las ocupaciones de tiempo de las actividades y reuniones. Cuando se agrega una nueva ocupación al calendario, automáticamente se activa el detector para verificar si existen conflictos con las ocupaciones ya registradas y eventualmente alertar al usuario.
El criterio para controlar los conflictos en la versión actual de Odyssey consiste simplemente en verificar superposiciones usando los momentos de inicio y finalización de las actividades y reuniones. Pero sabemos que sería útil permitir al usuario especificar su propia política de detección de colisiones integrando otros parámetros (por ejemplo si la actividad es o no flotante, los participantes de una reunión, etc.).

Estados de las ocupaciones

Durante su ciclo de vida, las actividades y reuniones pasan por distintos estados. Como cada estado extiende el comportamiento de la actividad o reunión de una manera específica (*State Pattern*) decidimos definir la jerarquía de clases que mostramos en la figura. Esta jerarquía podría ser extendida en el futuro si fuera necesario desdoblar los estados actuales o agregar algunos nuevos. En general, los estados intervienen al agregar actividades o reuniones en el calendario. Por ejemplo: los estados de las reuniones preparan las notificaciones que se envían a los participantes de las reuniones.



Para las actividades identificamos los siguientes estados (cada uno representado por una clase):

- ***EstadoActividadCancelada:*** corresponde a las actividades que se consideran canceladas
- ***EstadoActividadConfirmada:*** corresponde a las actividades que se consideran confirmadas, es decir, están pendientes de ser realizadas.

- ***EstadoActividadCumplida:*** corresponde a las actividades que el usuario marcó como realizadas.
- ***EstadoActividadVencida:*** para las actividades que sobrepasaron el momento de finalización sin ser cumplidas.

Los estados de las reuniones tienen bastante más comportamiento que los estados de las actividades, porque intervienen en la administración de las notificaciones que se envían a los participantes de las reuniones. Hemos identificado los siguientes:

- ***EstadoReunionCancelada:*** corresponde a las reuniones que se consideran canceladas

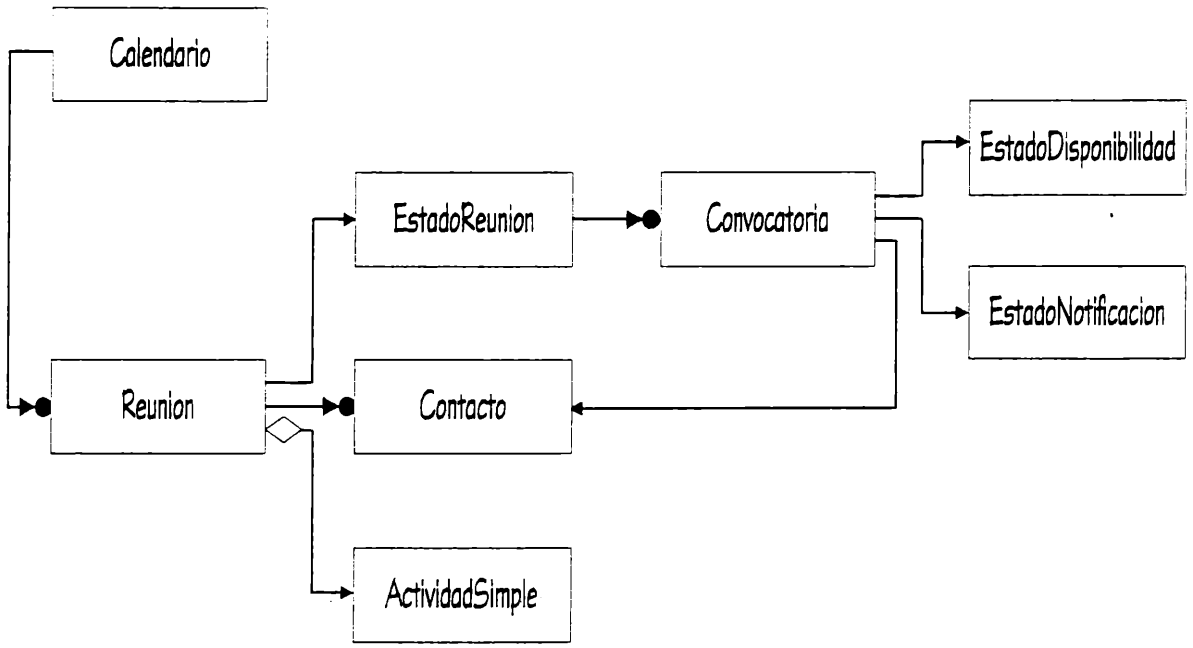
- ***EstadoReunionConfirmada:*** corresponde a las reuniones que se consideran confirmadas, es decir, están pendientes de ser realizadas. Una reunión está confirmada cuando se recibió una respuesta afirmativa de todos los participantes.
- ***EstadoReunionCumplida:*** corresponde a las reuniones que el usuario marcó como realizadas.
- ***EstadoReunionDeliveración:*** corresponde a las reuniones en las que falta confirmar la presencia de todos los participantes. Es decir: resta recibir algunas respuestas o bien algún participantes respondió informando que no asistiría.
- ***EstadoActividadVencida:*** para las reuniones que sobrepasaron el momento de finalización sin ser cumplidas.

Las reuniones

En el siguiente diagrama mostramos la descomposición de las reuniones. Como ya dijimos, las reuniones se registran en el calendario. Cada reunión integra un estado (***EstadoReunion***), una ocupación de tiempo (***ActividadSimple***) y una lista de participantes (***Contacto***). La versión actual de Odyssey exige que los participantes estén registrados como contactos en el directorio del organizer.

El estado de la reunión es encarga de administrar toda la información relacionada con la confirmación de la presencia de los participantes a la reunión. Para esto, se asocia un objeto *convocatoria* para cada participante. Las convocatorias son objetos realmente simples que registran la siguiente información:

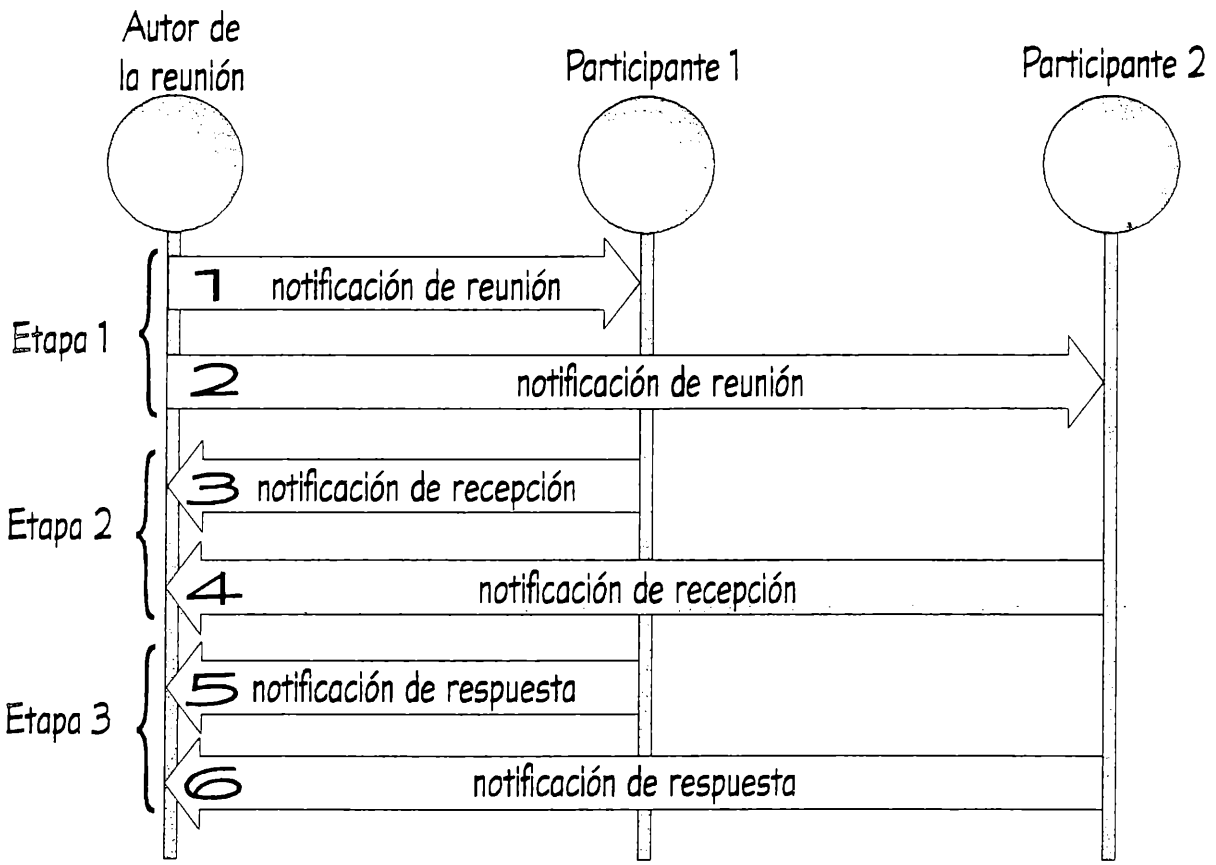
- ***Estado de disponibilidad:*** se refiere a la respuesta que envió el participante. Por ejemplo: 'Estará presente' , 'Ausente', etc.
- ***Estado de notificación:*** indica si el participante recibió la notificación de la reunión (aún cuando no haya enviado una respuesta).



Protocolo para coordinar reuniones

La concertación de las reuniones entre varios participantes que operan desde sus organizers nos llevó a desarrollar un protocolo de comunicación. En la siguiente figura presentamos el protocolo para describir todo el proceso de interaccción para el caso en que el usuario quiera ingresar una nueva reunión en la que los participantes disponen de un organizer. Se descompone en las siguientes etapas:

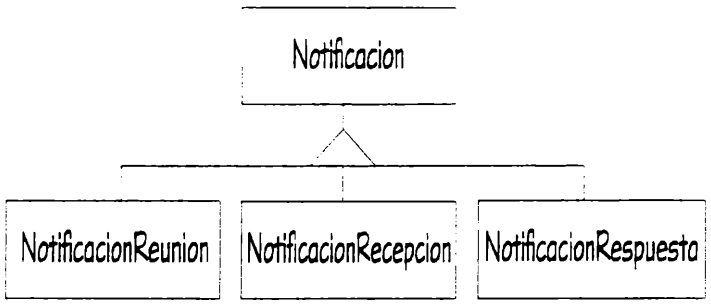
*1.- **Envío de notificaciones:*** el usuario prepara una reunión y decide quiénes son los participantes. Cuando el usuario ingresa la reunión (podría hacerse un chequeo de conflictos antes de esto), la reunión queda registrada en el calendario. El organizer (recibiendo la colaboración del estado de la reunión) envía notificaciones a los organizers remotos. En ese momento, los estados de disponibilidad y notificación de cada participante son "pendientes".
*2.- **Recepción de las notificaciones:*** las agendas de los participantes reciben las notificaciones, que quedan apiladas en el Inbox (bandeja de entrada). Al recibir la notificación de reunión, automáticamente cada organizer despacha una notificación de recepción hacia la agenda donde del usuario que organiza la reunión. Cuando estas notificaciones llegan a esa agenda, se actualiza el correspondiente estado de notificación de los participantes.



*3.- **Envío de respuestas:*** en algún momento, los participantes deciden enviar las respuestas a las notificaciones de la reunión (no se requiere que las envían inmediatamente). Hasta ese momento, el estado de disponibilidad (en el organizer donde está registrada la reunión), está pendiente. A medida que las respuestas llegan a la agenda del usuario que organiza la reunión, se van actualizando los estados de disponibilidad de los participantes.

Las notificaciones

Las notificaciones son los objetos que intervienen en el diálogo entre los organizers. Pueden verse como mensajes de alto nivel. El siguiente diagrama muestra la jerarquía de clases de las notificaciones.



- ***Notificación:*** es una clase que agrupa a todas las notificaciones. En general, todas las notificaciones tienen entre sus atributos a dos medios de contacto: ***1) Medio de contacto origen:*** el medio de contacto por el se espera recibir una respuesta a la notificación (en caso de que deba enviarse una). Este medio de contacto debe estar registrado entre los medios de contacto del dueño de la agenda. ***2)Medio de contacto destino:*** es el medio de contacto hacia donde debe transmitirse la notificación. Este medio de contacto debe estar registrado entre los medios de contacto de la persona hacia la que se envía la notificación.
- ***NotificaciónReunion:*** esta clase define las notificaciones que se envían a los participantes de las reuniones. Estos objetos son capaces de enviar una notificación de recepción a la agenda origen apenas llegan a la agenda destino.
- ***NotificaciónRecepcion:*** estas notificaciones se envían para comunicar la recepción de una notificación de reunión (afectan los estados de notificación asociados a los participantes de las reuniones).
- ***NotificaciónRespuesta:*** estas notificaciones son enviadas por los participantes de las reuniones para confirmar o negar su presencia a una reunión. (afectan los estados de disponibilidad asociados a los participantes de las reuniones).

Inbox y Outbox

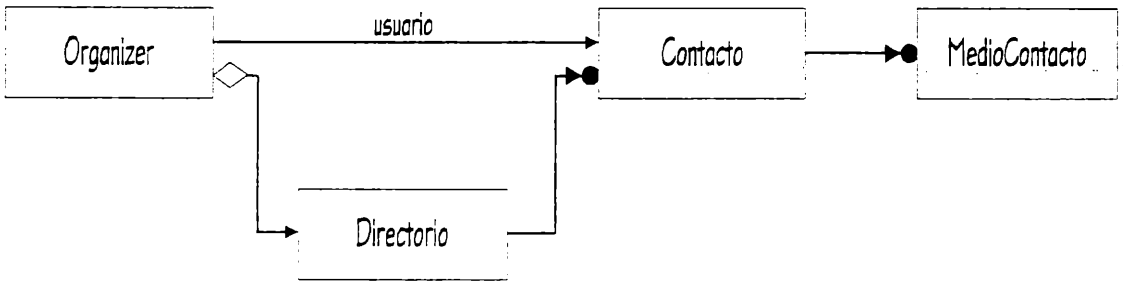
El inbox y outbox representan la bandeja de entrada y salida de notificaciones respectivamente. Las clases que las representan son:

- ***Inbox:*** se comporta como depósito para todas las notificaciones recibidas por el organizer, por ejemplo: las notificaciones recibidas para las reuniones. Las notificaciones recibidas se pueden almacenar todo el tiempo que el usuario desee. Esto es útil por ejemplo para postergar las decisiones sobre notificaciones de reuniones. Hay un solo inbox por cada agenda. El usuario puede vaciar el contenido de inbox en forma selectiva o total.
- ***Outbox:*** se comporta como depósito para todas las notificaciones que deben enviarse a otras agendas. Cuando el usuario organiza una reunión que involucra a contactos que disponen de una agenda distribuida, las notificaciones que deben enviarse se envían al outbox y desde allí son transmitidas. Hay un solo outbox por cada agenda. El usuario puede vaciar el contenido de inbox en forma selectiva o total.

El directorio de contactos

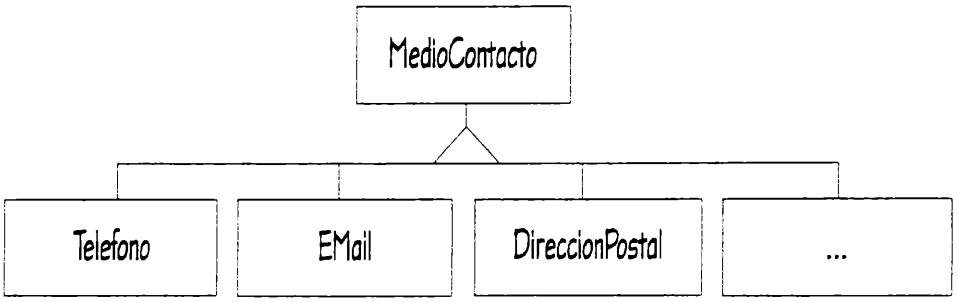
El directorio es básicamente un depósito de contactos. Un contacto es una entidad, física o jurídica, que tiene asociada una lista de medios de contactos. Los medios de contacto permiten enviar notificaciones y establecer comunicación con los contactos.
Es interesante señalar que el usuario de la agenda también es un contacto (aunque no esté registrado en el directorio). Por ser un contacto, tiene asociados varios medios de contacto (dirección postal, dirección email, teléfono, etc). En la siguinte figura mostramos estas relaciones y las clases involucradas.

- ***Directorio:*** se comporta como depósito para todos los contactos registrados en el organizer. Hay un solo directorio por agenda. El usuario puede vaciar el contenido de directorio en forma selectiva o total.



- ***Contacto:*** es una clase abstracta. Las instancias de sus subclases son los objetos que se registran en el directorio. Cada contacto puede tener asociados uno o varios medios de contacto.

- ***MedioContacto:*** es una clase abstracta que agrupa a todas las clases de medios de contacto. Define el protocolo para el envío y recepción de notificaciones. Dada una notificación, cada medio de contacto es capaz de representarla en un formato apropiado para la transmisión, según su naturaleza.



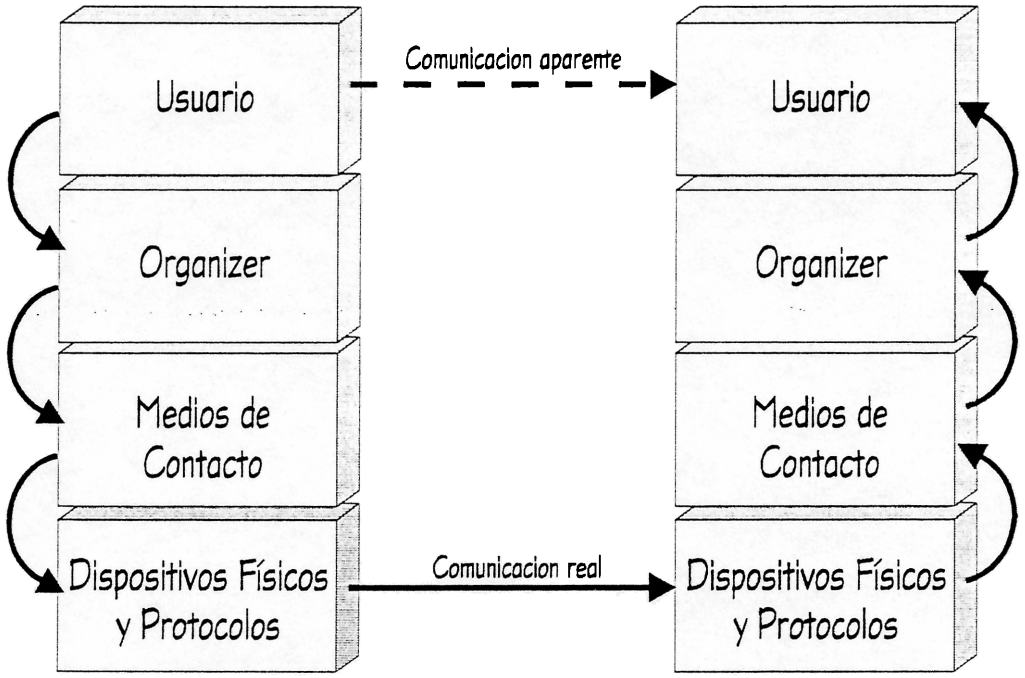
- ***Telefono:*** permite el envío de notificaciones usando una línea telefónica. Es decir, dada una notificación de reunión (***NotificacionReunion***) un medio de contacto telefónico la codifica en una señal de audio (envio) y dada la señal de audio, el medio de contacto obtiene una notificación.
- ***Email:*** permite el envío de notificaciones usando emails. Dada una notificación, un medio de contacto de email la codifica en una mensaje de texto ascii (email) enviable por SMTP y dado el mensaje recibido por POP3, el medio de contacto obtiene la correspondiente notificación (objeto).
- ***DireccionPostal:*** permite el envío de notificaciones usando el correo. Es decir, dada una notificación de reunión (***NotificacionReunion***) un medio de contacto postal genera e imprime el documento y sobre que debe enviarse al participante para comunicarle que están convocado a la reunión.
- ***Fax:*** permite el envío de notificaciones usando fax. Dada una notificación, un medio de contacto de fax genera el documento para enviar por fax.

Capas del protocolo de comunicacion

La concertación de una reunión supone una comunicación entre los contactos que participan. Técnicamente, esta comunicación se produce usando los medios de contacto, pero podemos analizarla en distintos niveles.
Hemos encontrado cierta analogía con las capas del protocolo de cominucación TCP/IP. Allí, cada capa es capaz de transformar la información que le llega en un formato apto para ser manipulada por su capa inmediatamente superior o inferior. En TCP/IP, los niveles superiores son los más abstractos y están más cerca de la comunicación de "alto nivel". En cambio, los niveles inferiores, están más cerca de los aspectos técnicos de la comunicación (red, etc.). En nuestro caso hemos identificado los siguientes cuatro niveles de comunicación:

- *Nivel 1:* los usuarios. Generan reuniones que se registran en la agenda.

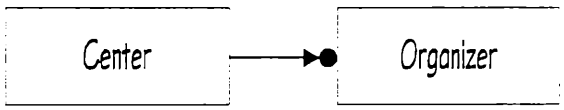
- *Nivel 2:* los organizers que se encargan de registrar las reuniones ingresadas por el usuario y generar las notificaciones que correspondan.
- *Nivel 3:* los medios de contacto. Se encargan de adaptar las notificaciones a un formato apropiado para ser enviadas a otros contactos. Ej: un medio de contacto email sabrá convertir una notificación en un email.
- **Nivel 4:** mecanismos y dispositivos de comunicación: linea de teléfono, TCP/IP, aparato para enviar fax.



Lo importante es que cualquier nivel es capaz de recibir datos y convertirlos a una representación válida para su nivel superior o inferior. Así, la comunicación entre usuarios recorre un circuito que pasa por todos los niveles: organizers, medios de contacto y dispositivos.

Notar que todo este esquema sólo permite enviar/recibir notificaciones de reuniones. La lógica para acordar el horario, confirmar presencia, etc. se construye a partir de estos mensajes.

Odyssey: los centros



Los centers son sistemas preparados especialmente para facilitar la interconexión de los organizers. Un center está formado simplemente por un conjunto de organizers remotos suscriptos.

Los usuarios pueden conectarse a los centers (especificando la dirección IP y el port) para suscribir su organizer. También pueden conectarse para navegar por la lista de los organizers ya suscriptos y agregarlos al directorio de contactos. Una vez que se ha agregado un nuevo contacto al directorio, es posible por ejemplo enviarle notificaciones para concertar reuniones.

IMPLEMENTACION

La implementación de los organizers y los centers se programó íntegramente en lenguaje Smalltalk con extensiones para CORBA. La versión de Smalltalk que utilizada es *VisualWorks 2.5* de ParcPlace. Las extensiones CORBA fueron agregadas instalando *Orbix 1.0 for Smalltalk*, de IONA.

Los centers y organizers funcionan como aplicaciones completamente independientes, que interactúan por TCP/IP. Orbix implementa el protocolo IIOP de forma nativa para la comunicación de los ORBs.

Instalación y puesta en marcha

Para instalar *Odyssey* se deben cumplir los siguientes requisitos:

- Instalación de VisualWorks 2.5 o superior
- Instalación de Orbix for Smalltalk

Recomendamos comenzar la instalación sobre una imagen limpia, para que no queden inconsistencias. Secuencia de pasos a seguir:

1) Copiar a alguna carpeta del disco rígido el contenido de la carpeta *"Odyssey"* del diskette, para trabajar más cómodamente.
2) Copiar al directorio donde esté instalada la imagen (directorio de trabajo) los archivos con la definición de los íconos. Los nombres de los archivos son *"icon01.bmp"* hasta *"icon12.bmp"*.
3) Abrir el ambiente de VisualWorks.
4) Instalar las clases. Para esto abrir un file list y hacer file-in de todos los archivos *"*.st"*. Seguramente se producirán algunos errores al intentar instalar clases que nombran a otras clases que aún no fueron instaladas. Si esto ocurre, seleccionar la opción de *<proceed>*
5) Repetir el procedimiento anterior, para asegurar que todas las clases quedan instaladas correctamente
6) Abrir el Binding Manager de Orbix. Esto se hace desde el menu Orbix del Transcript o evaluando la expresión *<ODLBindingManager open>*
7) Desde el Binding Manager ejecutar la opción de menú *<Restore>*. Esto instala las definiciones de nuevos tipos de datos IDL en el Interface Repository.
8) Desde el Transcript o un Workspace evaluar *"OdysseyRegistry load"*, para cargar en la imagen los bitmaps de los íconos.
9) Salvar la imagen

Con esto concluye la instalación. Para comenzar a utilizar *Odyssey*, las instrucciones están disponibles en el manual del usuario.

MANUAL DEL USUARIO

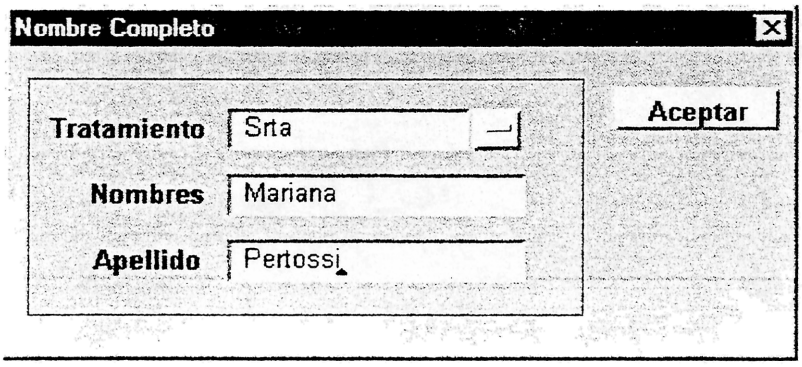
¿Qué es Odyssey Organizer?

***Odyssey Organizer* es una aplicación de administración de información personal.**
Odyssey le provee un directorio donde podrá guardar la información de sus contactos, un calendario donde podrá planificar actividades y concretar reuniones y le provee la posibilidad de subscribirse a *Centers*, desde los cuales podrá tener acceso a todos los *Organizers* subscriptos a cada uno de ellos.

Para abrir un *Organizer* evaluar la expresión: *"AgendaApp new: 'nombre_organizer'.* Para abrir un grupo de *Centers* evaluar: *GrupoDeClubesApp new: 'nombre_grupo'.* Donde *'nombre_organizer' y 'nombre_grupo'* son los archivos donde se guarda en forma persistente la información. La primera vez, estos archivos no existirán y *Odyssey* se encargará de crearlos.

Primera vez que utiliza Odyssey Organizer

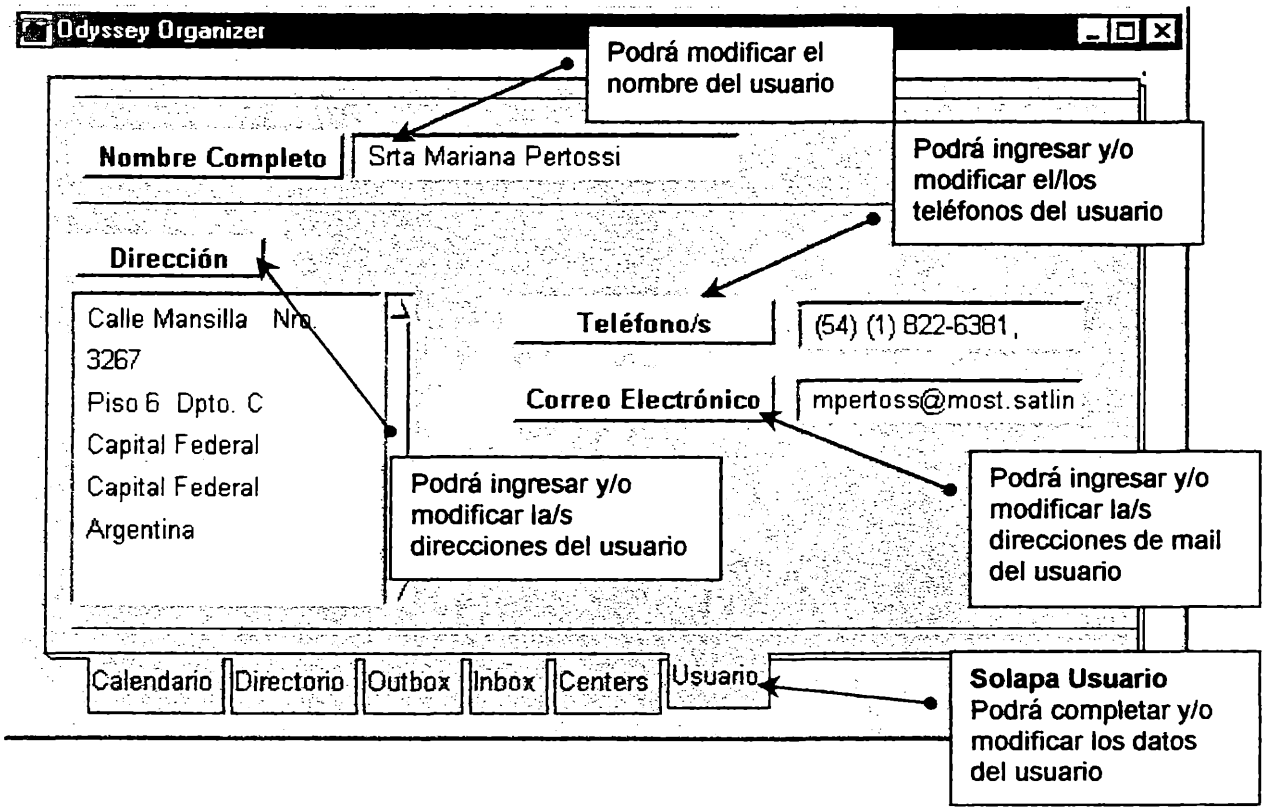
Cuando ejecute la aplicación, si es la primera vez que utiliza *Odyssey Organizer*, deberá ingresar su nombre completo (datos del usuario). Presionando el botón Aceptar se iniciará *Odyssey*.



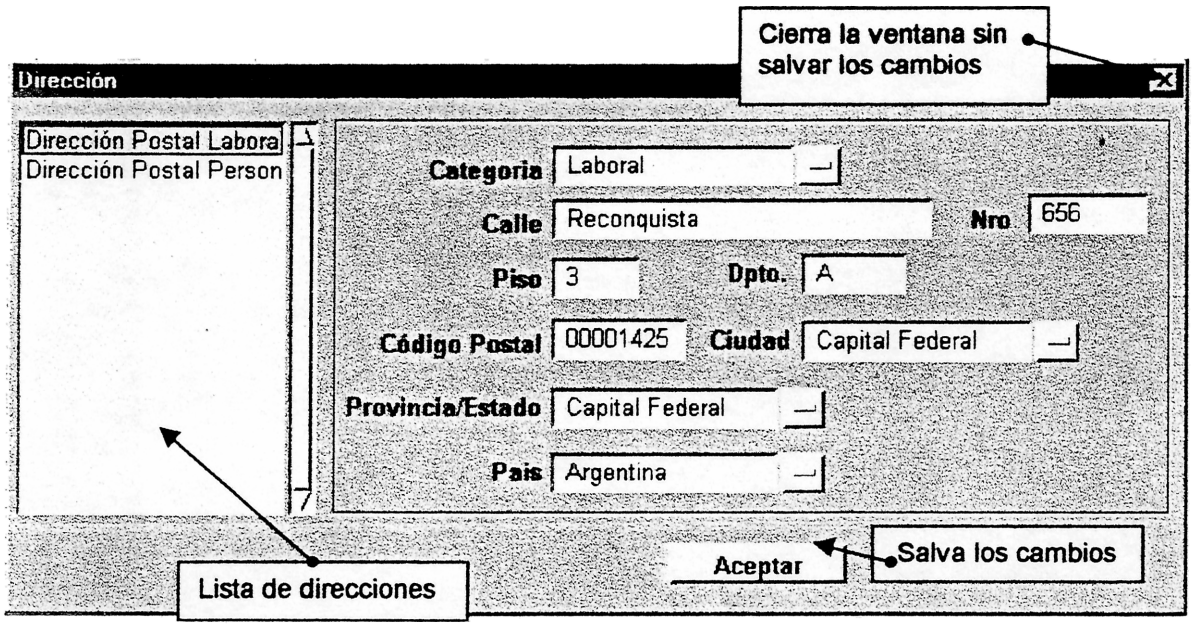
En las próximas ejecuciones directamente se abrirá Odyssey con su información.

Usuario

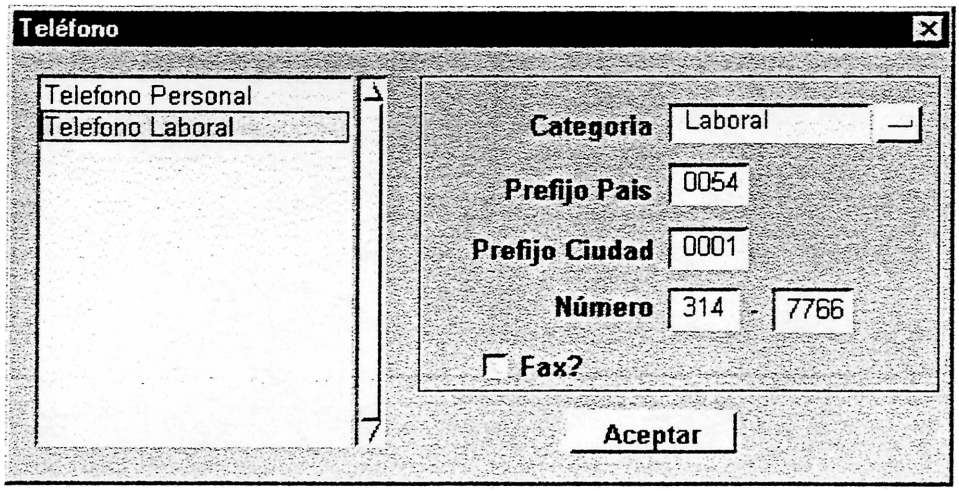
Los datos del usuario podrán completarse en la carpeta del Usuario:



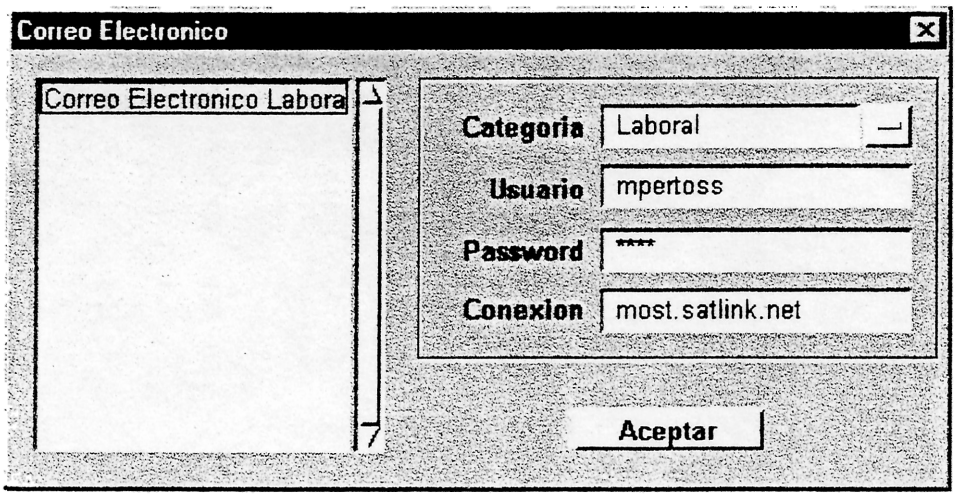
Presionando el botón Dirección podrá ingresar la/s direcciones del usuario.



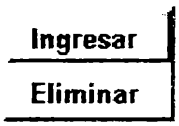
Presionando el botón Teléfono/s podrá ingresar el/los teléfonos del usuario.



Presionando el botón Correo Electrónico podrá ingresar la/s direcciones de mail del usuario.



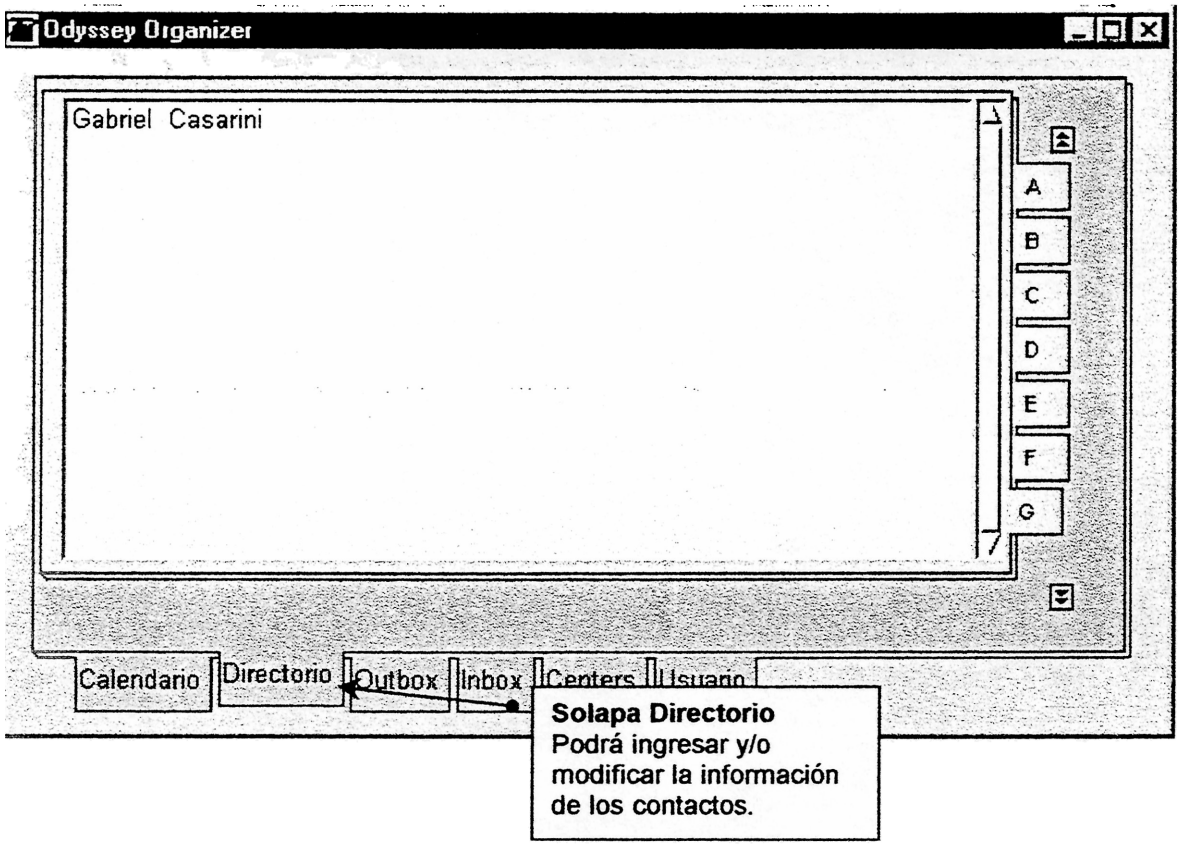
Presionando el botón derecho del mouse sobre cada una de las listas se verá el siguiente menú.



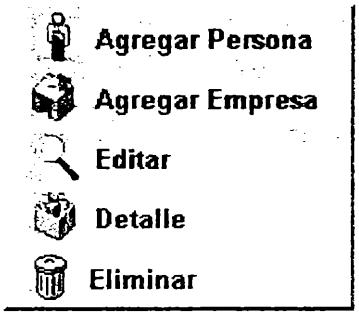
La opción Ingresar permitirá el ingreso de un nuevo medio de contacto.
La opción Eliminar borrará el medio de contacto seleccionado de la lista.

Directorio de Contactos

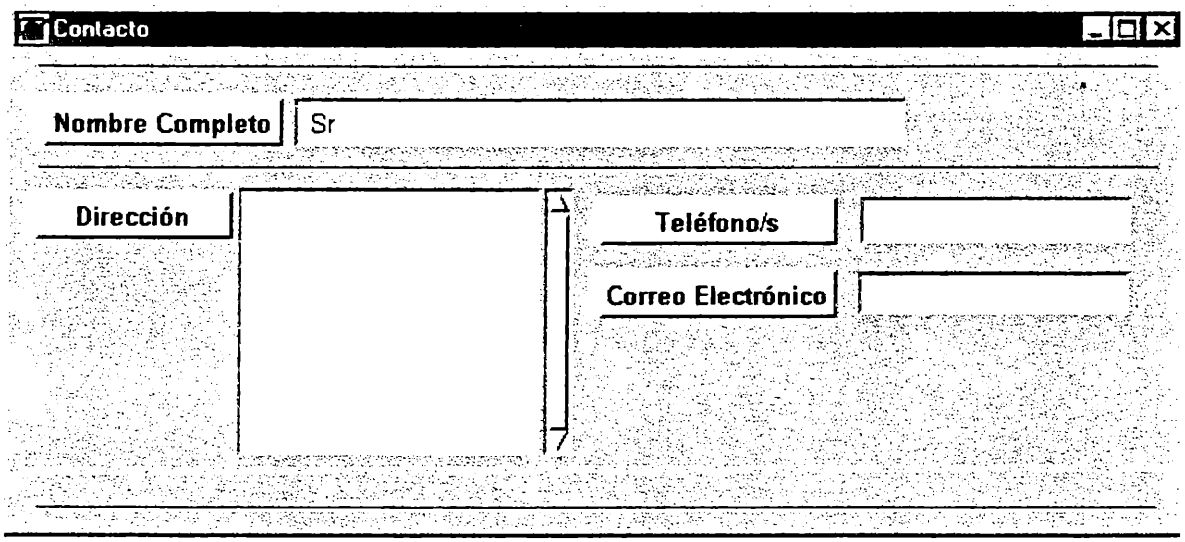
Almacena en orden alfabético la información de los contactos.



Presionando el botón derecho del mouse sobre el directorio se despliega el siguiente menú:



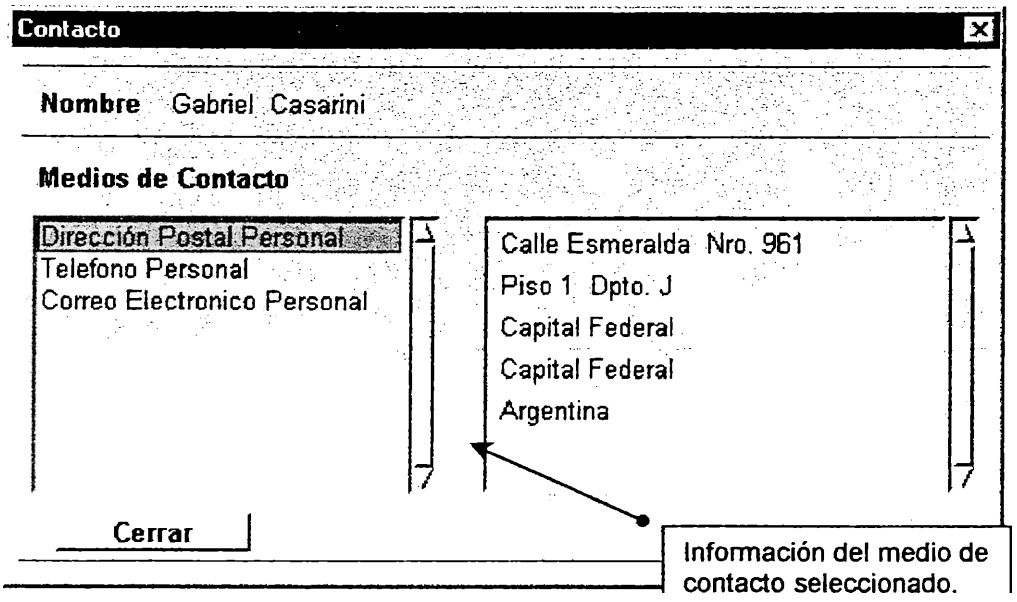
- **Agregar Persona:** Permite ingresar información sobre un contacto persona.
- **Agregar Empresa:** Permite ingresar información sobre un contacto empresa.



La forma de ingresar la información es similar a la descripta para la información del usuario.

- **Editar:** Permite modificar la información del contacto seleccionado. Nota: sólo podrán editarse los datos de contactos **no** remotos.

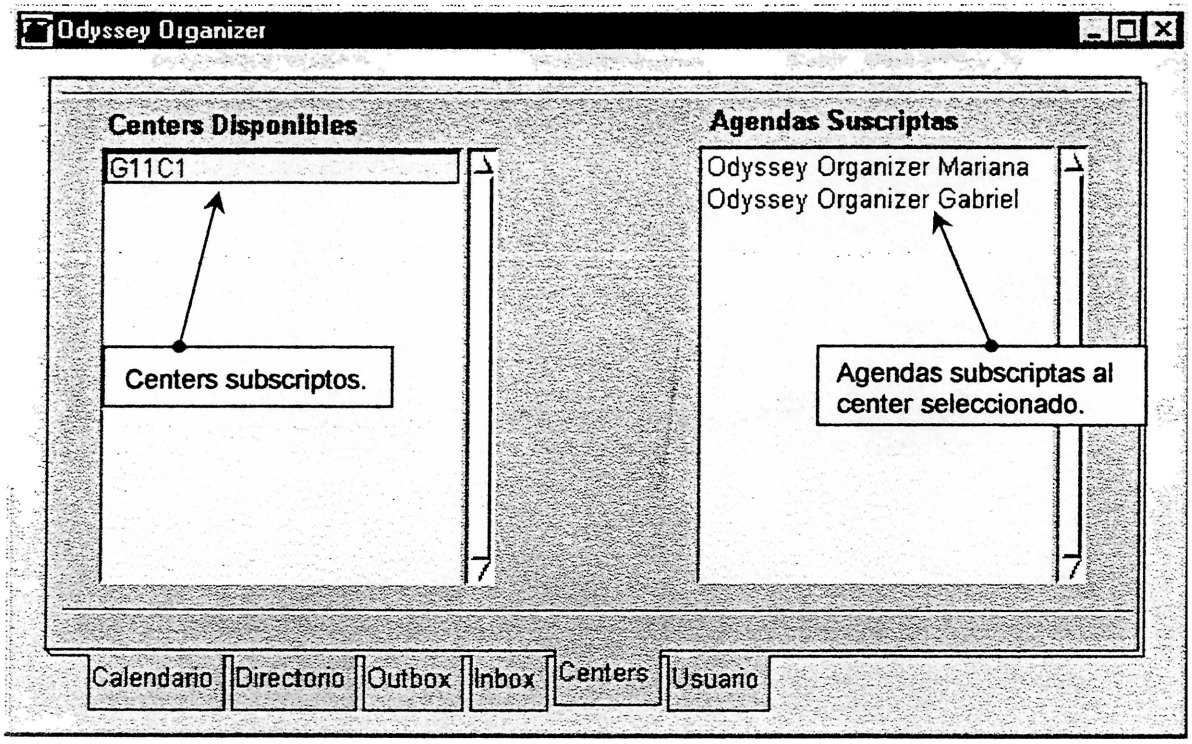
- **Detalle:** Permite ver en una forma abreviada la información del contacto seleccionado.



- **Eliminar:** Permite eliminar el contacto seleccionado. Antes de borrar el contacto se pedirá la confirmación.

Centers

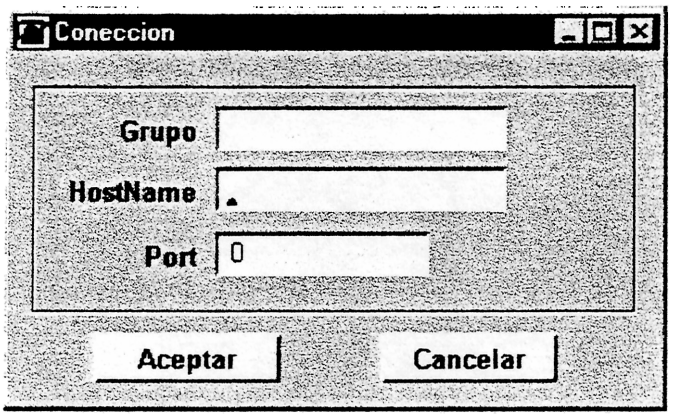
Desde aquí se pueden subscribir nuevos centers, consultar las agendas subscriptas a cada uno de ellos y agregar contactos remotos a sus directorios.



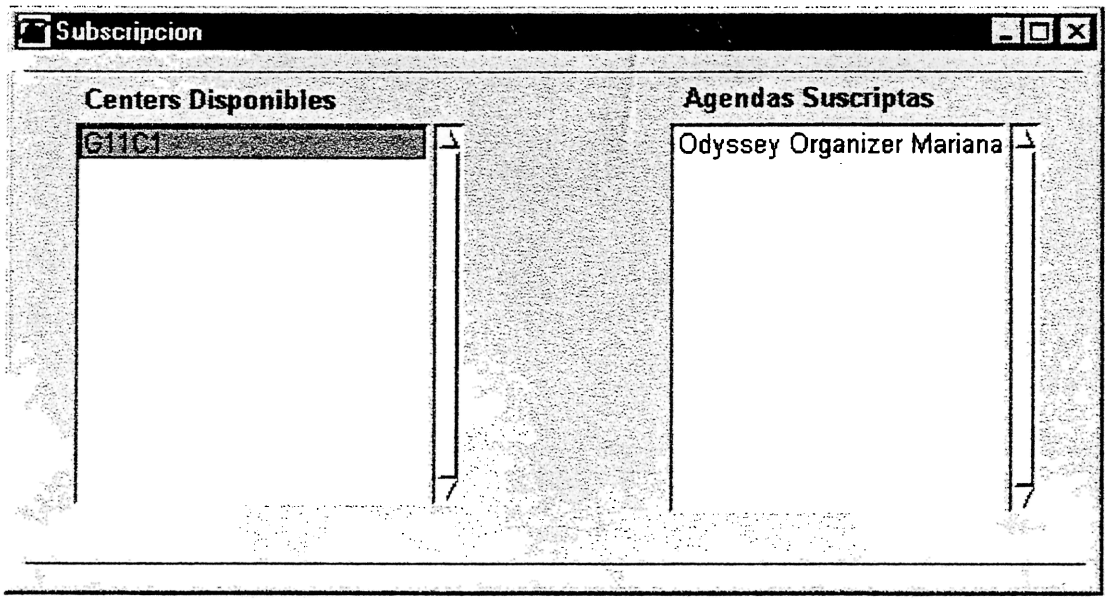
Presionando el botón derecho del mouse sobre la lista de agendas subscriptas se despliega la opción de contactarse con la agenda seleccionada. Es decir, el usuario de la agenda seleccionada pasará a ser un contacto remoto del directorio.

Presionando el botón derecho del mouse sobre la lista de Centers se despliega el siguiente menú.

- **Agregar:** Permite subscribir nuevos Centers. Eligiendo esta opción se despliega la siguiente ventana de conexión.



Se deberá ingresar el nombre del grupo y su dirección IP y port donde están agrupados los centers a los que se quiere acceder.
Si la conexión tuvo éxito se despliega la siguiente ventana donde se pueden consultar los centers agrupados en el grupo ingresado y las agendas subscriptas a cada center.

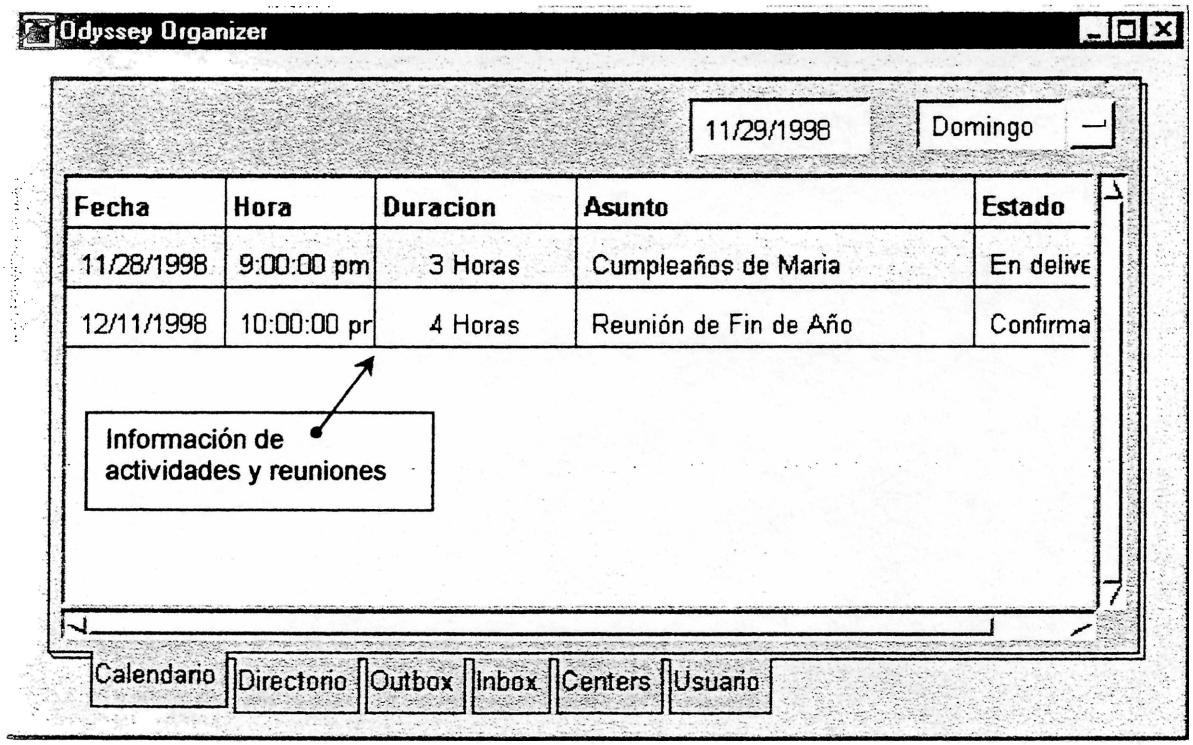


Presionando el botón derecho del mouse sobre la lista de centers disponibles se dará la opción de subscribirse al center seleccionado.

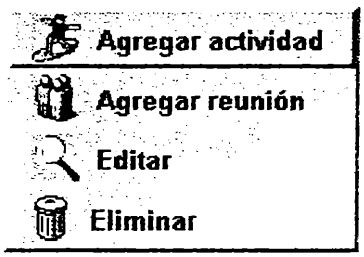
- **Eliminar:** Elimina el center seleccionado.

Calendario

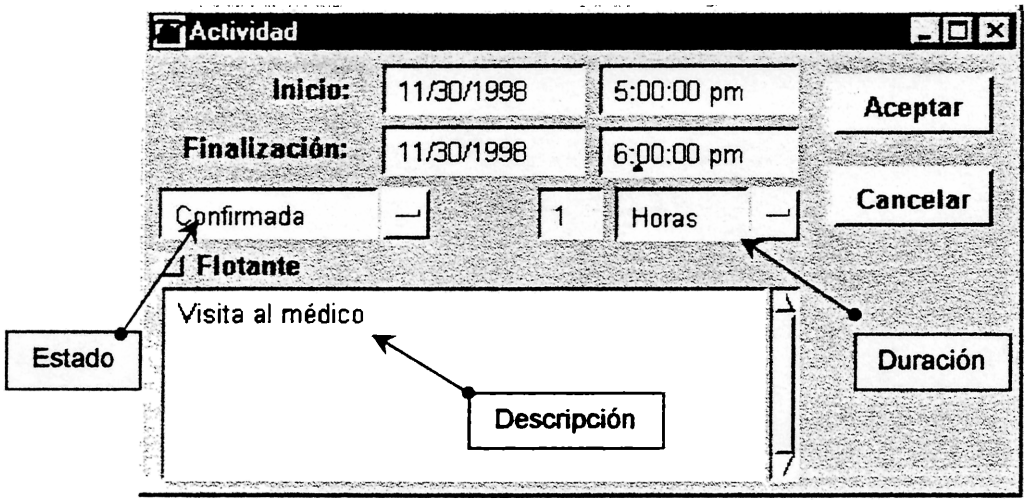
Desde aquí se podrán planificar actividades y concertar reuniones.



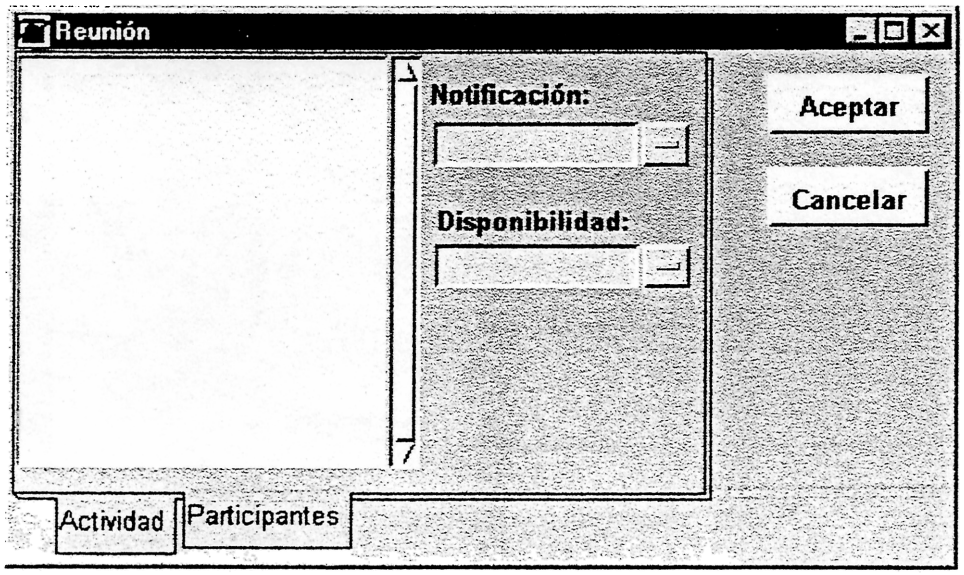
Presionando el botón derecho del mouse se desplegará el siguiente menú:



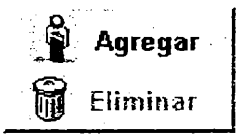
- **Agregar actividad:** Los datos a ingresar para una actividad son los siguientes:



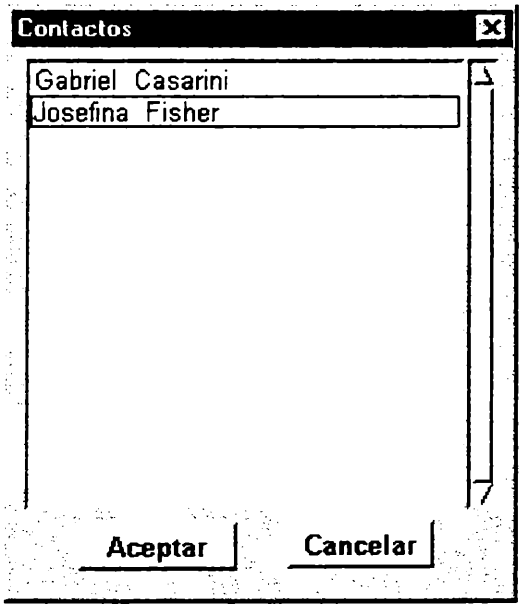
- **Agregar reunión:** Los datos a ingresar para una reunión son similares a los de una actividad, pero además se deberán ingresar los participantes de la misma:



Presionando el botón derecho del mouse sobre la lista se desplegará el siguiente menú:



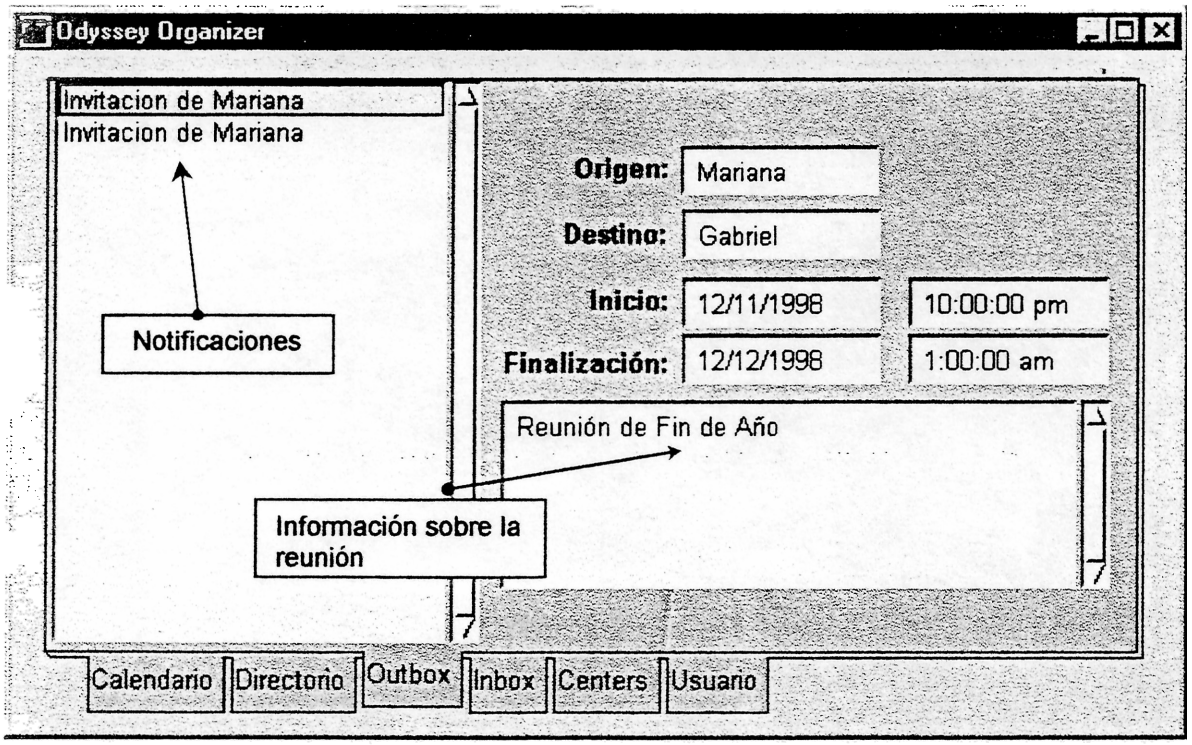
- **Agregar:** Muestra una lista de contactos, de la cual puede seleccionar 1 o varios de ellos para que participen en la reunión.



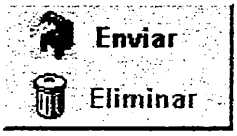
Una vez seleccionados los participantes de la reunión se generan las notificaciones que deberán ser enviadas a cada uno de ellos. Estas notificaciones se guardan en el outbox.

Outbox

En el outbox se registran las notificaciones de salidas.



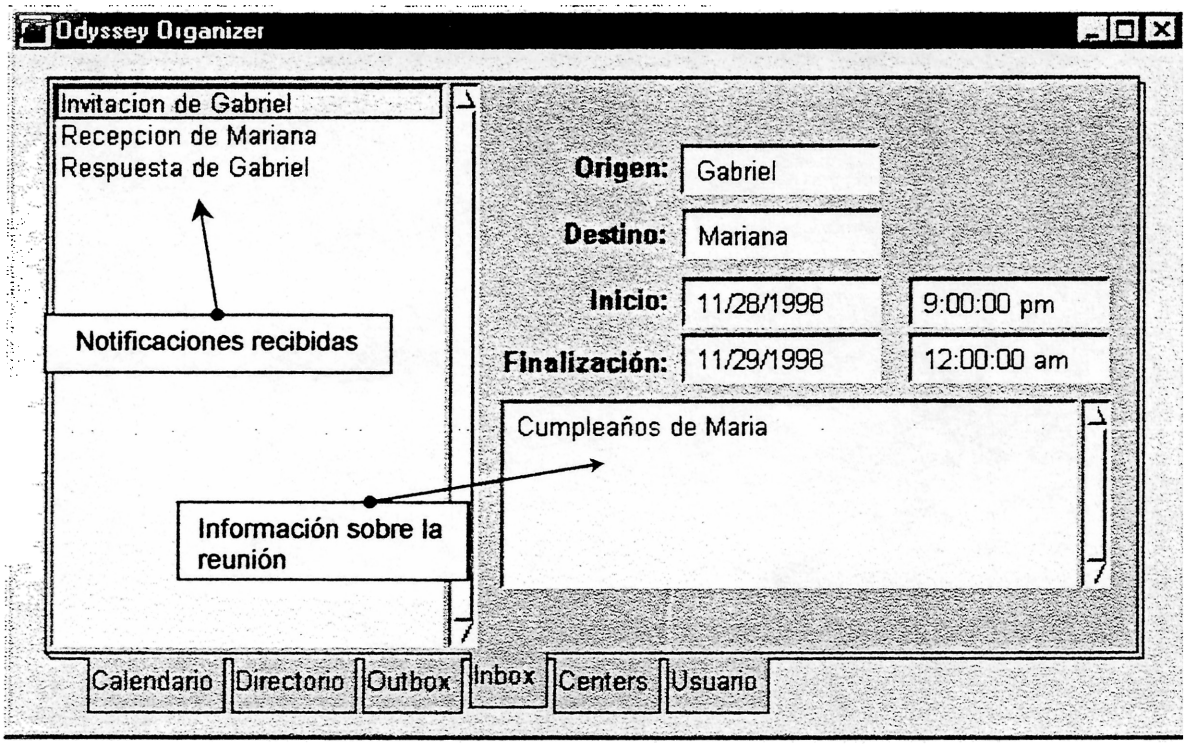
Presionando el botón derecho del mouse sobre la lista de notificaciones se despliega el siguiente menú:



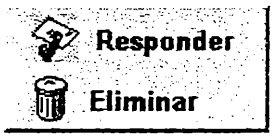
- **Enviar:** Manda las notificaciones de salida a su destino.
- **Eliminar:** Borra la notificación seleccionada.

Inbox

En el inbox se almacenan las notificaciones recibidas.



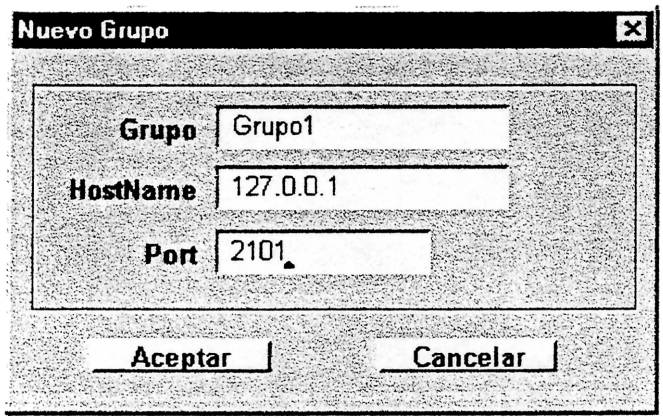
Presionando el botón derecho del mouse sobre la lista de notificaciones recibidas se despliega el siguiente menú:



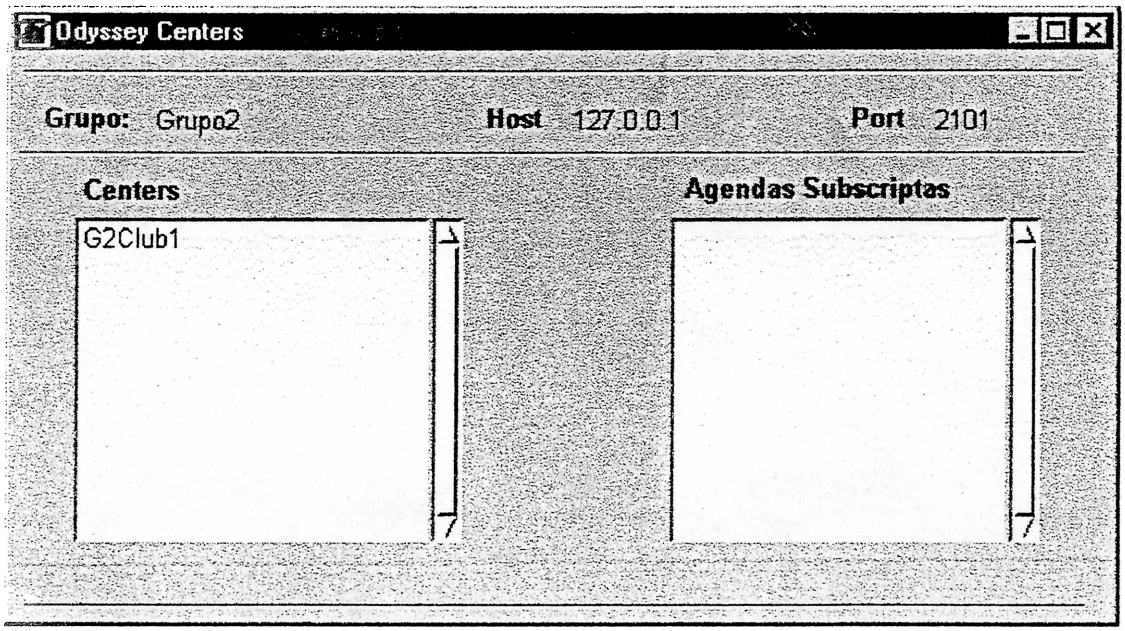
- **Responder:** Genera una notificación de respuesta que se almacena en el outbox para ser posteriormente enviadas.
- **Eliminar:** Elimina la notificación seleccionada.

Grupo de Centers

Los centers se asocian en un grupo. Cada grupo debe estar definido por un nombre, una dirección IP y un port.
Si el grupo no existe se le requiere ingresar los siguientes datos:



Una vez definido el grupo se despliega la siguiente ventana:



Desde esta ventana se podrán ver los centers asociados al grupo, ingresar nuevos y consultar las agendas subscriptas a cada uno de ellos.

Optimizaciones y comentarios

En esta sección presentamos las conclusiones que sacamos después de implementar *Odyssey* usando el modelo de objetos de CORBA para la distribución.

En CORBA los mensajes sólo permiten pasar parámetros por copia de tipos básicos de datos (integer, strings, booleans, etc). En general, todos los otros objetos se pasan por referencia usando un esquema de proxies. Esto es una gran ventaja porque garantiza la consistencia del estado de los objetos y además se realiza en forma completamente transparente.
Sin embargo, en algunos casos es deseable poder pasar copias de los objetos, para instanciarlos en otros espacios de memoria. El mejor ejemplo es el envío de las notificaciones: en lugar de transmitir una referencia a la notificación instanciada localmente, tiene sentido transmitir una copia que quede registrada en el organizer remoto.
El problema se resuelve trabajando con structures (similares a las de C++) que permiten pasar el estado de un objeto e instanciarlo de nuevo en el espacio de memoria del objeto receptor del mensaje (pasaje por copia).
Este esquema se aplicó para pasar por copia también objetos simples, como las fechas y horarios (no son tipos primitivos de CORBA).
Las estructuras (structures) se declaran en IDL entre las declaraciones de alguna interface y se deben compilar para que quedar registradas en el Interface Repository.
Una alternativa a esta solución hubiera sido el uso de los servicios del ciclo de vida (Life Cycle Services) definidos en CORBA. Estos servicios permiten copiar y mover objetos entre ORBs, pero la versión de Orbix que utilizamos no los implementa.

Otro punto interesante son las optimizaciones necesarias para agilizar la interacción entre las agendas. El hecho de manejar referencias a objetos remotos implica comunicación entre las partes involucradas, que normalmente están operando desde distintas máquinas interconectadas por una red. Es deseable entonces disminuir al mínimo necesario el tráfico de información para que los organizers puedan ser utilizados cómodamente.
La solución para esto es el uso de caches para retener copias de ciertos datos mínimos que se acceden continuamente. Esta función es realizada por las instancias de las clases ***ActividadRemota*** y ***ReunionRemota***, que se comportan como proxies de las actividades y reuniones registradas en otras agendas.
El uso de información duplicada, en lugar de referencias directas a los objetos originales, nos conduce a otro problema: la *inconsistencia*. Si el estado del objeto original (remoto) se modifica en alguno de sus aspectos, es necesario actualizar los caches para que no manejen información desactualizada. Esto puede lograrse usando el Events Service de CORBA, de tal manera que el objeto remoto envíe eventos a los objetos que dependen de él (Observer Pattern) y estos actualicen su estado. Aunque Orbix implementa la especificación de Events Service, preferimos no profundizar el tema y en general *Odyssey* maneja referencias directas a los objetos remotos.

EXTENSIONES DEL PROYECTO

A continuación presentamos un breve resumen de todas las extensiones e ideas que podrían aplicarse a *Odyssey*.

En primer lugar, se podría extender la funcionalidad de los organizers. Por ejemplo: registración de tareas (to-do items) y alarmas. Las tareas se diferencian de las actividades en que no tienen momentos de inicio y finalización; sino que son ingresadas como recordatorio de cosas que deben hacerse en algún momento del día o semana (por ejemplo: "Averiguar por las vacaciones en las islas Fiji").

Las alarmas sirven para controlar distintos eventos relacionados con las actividades y el funcionamiento del organizer mismo. Deberían ser muy simples de utilizar: para programarlas sería necesario elegir un evento e indicar las acciones que deben tomarse cuando el evento ocurra. Pueden verse como dispositivos que se programan para que "*cuando ocurra esto, hacer aquello*". Hay dos tipos de alarmas:

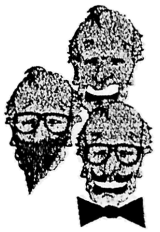
Temporales: controlan el paso del tiempo y se programan para activarse en un momento determinado. Normalmente están vinculadas a las actividades registradas en la agenda y el usuario las utiliza para ser notificado de las reuniones y tareas. Por ejemplo: "*utilizar una alarma para ser notificado 5' antes de una reunión*", otra alarma para "*avisar que se cancela la reunión del 17/6 concertada con las otras agendas*", etc. En el primer ejemplo hay que destacar que el momento de activivación de la alarma es *relativo* al horario de la reunión y que se desplaza automáticamente si éste sufre modificaciones.

Asincrónicas: estas alarmas controlan eventos relacionados al funcionamiento de la agenda. Por ejemplo: "*tener una alarma que despligue un mensaje cada vez que se abre la agenda*", "*tener una alarma que notifica el cambio de domicilio de un contacto*", "*notificar cumpleaños*", etc.

Otra característica interesante es la posibilidad de agrupar las actividades en *categorías*. Las categorías son definidas por el mismo usuario para ordenar el acceso al organizer. Ejemplos de algunas categorías son: "*Objetivos anuales*", "*Tareas semanales*", "*Actividades diarias*", etc. Usando categorías el usuario puede agrupar las actividades de acuerdo a sus necesidades, costumbres de trabajo, normas, planes, etc..

La tecnología de objetos distribuidos es sumamente útil para crear ambientes colaborativos de trabajo. En la versión actual sólo implementamos el proceso de concertación de reuniones. Y aunque ese era el objetivo inicial, se podrían explotar las capacidades de los objetos distribuidos para permitir la interacción de los participantes en tiempo real. Las reuniones se podrían modelar utilizando espacios de trabajo comunes que serían visibles desde los organizers de los participantes. El esquema de comunicación sería muy parecido al chat, pero se podría extender con funcionalidad específica para las reuniones. Por ejemplo: apertura automática de todos los espacios de trabajo cuando llega el momento de la reunión, mecanismos para votar, generación automática de documentación con los temas tratados en la reunión

Otro punto interesante se refiere a la interacción de los organizers. En la implementación actual, el usuario puede suscribir su organizer a un center y permitir que sea accedido desde otros organizers. Podemos extender el concepto y dotar a los organizers de diferentes *personalidades*. La función de una *personalidad* es restringir y proteger el acceso a su agenda en algún aspecto. Los sistemas externos (incluidas otras agendas) no interactúan con las agendas, sino con las personalidades de éstas. Por ejemplo: interactuando con la personalidad $\mathbf{A}_1$ de la agenda **A** sólo es posible hacer consultas del tiempo libre, mientras que accediendo a esa misma agenda a través de su personalidad $\mathbf{A}_2$ es posible registrar nuevas actvidades. Cada usuario puede definir tantas personalidades como quiera para su agenda y especificar para cada una un perfil de acceso distinto.



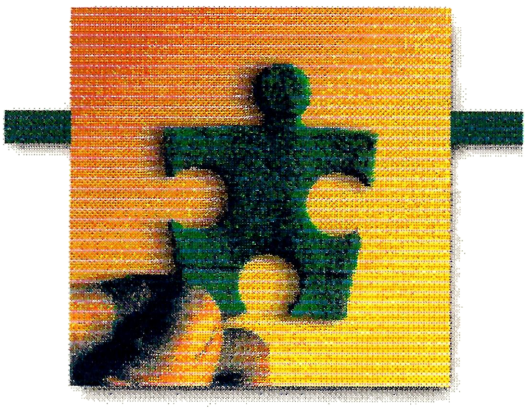
Para controlar y proteger el acceso a las agendas desde otros sistemas o agendas se deben definir mecanismos de seguridad que puedan ser configurados por cada usuario. Esta funcionalidad se logra haciendo que las entidades externas interactúen con las agendas a través de las *personalidades* que éstas presentan. Cuando una agenda se suscribe a un center puede especificar con cuál de sus personalidades se presentará para la interacción. Una característica interesante de este esquema es que permite que la misma agenda se presente con diferentes personalidades en varios centers. Cada personalidad restringe uno o varios niveles de acceso y asi protege a la agenda.

Otra extensión importante para los organizers es el acceso remoto. Podría desarrollarse un nuevo módulo, *Odyssey Mobile*, que actúe como front-end para resolver la interacción remota del usuario con su organizer.

Para permitir el acceso remoto, los organizers primero deberían suscribirse en un center. Luego, usando *Odyssey Mobile* desde alguna máquina se establece conexión con el center, se busca el organizer para interactuar (remotamente). Está claro que para lograr esto debemos garantizar el acceso a los organizers en cualquier momento (deberían residir en un servidor que funcione todo el tiempo).

Parte 5

Apéndices



GLOSARIO DE SIGLAS Y VOCABULARIO

- ***Business object:*** el término business object se refiere a un objeto que realiza un conjunto de tareas asociadas a un proceso o asunto (business) en particular. Puede ser un objeto local o distribuido.
- ***Cliente:*** una entidad o programa que consume recursos o servicios. Los clientes invocan operaciones de otros objetos (servidores).
- ***Common Object Request Broker Architecture (CORBA):*** es una infraestructura emergente y abierta para computación con objetos distribuidos. CORBA está siendo estandarizada por el OMG. CORBA automatiza muchas de las tareas comunes de programación sobre redes, como por ejemplo registración de objetos, localización y activación; manejo de errores, dispatching de operaciones, conversión de parámetros, etc.
- ***Dynamic Invocation Interface:*** Permite la invocación dinámica de métodos; es decir, en lugar de usar una rutina (stub) específica para una operación particular en un objeto particular, el cliente puede especificar el objeto, la operación que se ejecutará y los parámetros con una llamada (o secuencia de llamadas). El cliente debe suministrar entonces la información necesaria para determinar la operación y tipo de parámetros que está pasando (normalmente recupera estos datos del Interface Repository).
- ***Dynamic Skeleton Interface:*** es una interface para el manejo dinámico de las invocaciones. Es decir, en lugar de acceder por un skeleton específico para una operación particular, se llega a la implementación usando una interface dinámica, que permite el acceso suministrando el nombre de la operación y los parámetros (es análogo a la interface de invocación dinámica del lado del cliente). Los skeletons dinámicos se pueden invocar tanto desde stubs como desde interfaces de invocación dinámica; tienen la misma semántica.
- ***Encapsulamiento:*** uno de los beneficios de las arquitecturas orientadas a objetos, se refiere al ocultamiento de los detalles de implementación detrás de una interface pública.
- ***Escalabilidad:*** se refiere al número de clientes y servidores soportados, y es necesaria para el nivel enterprise de desarrollo e integración de software.
- ***Frameworks:*** son bibliotecas preensambladas de clases que se empaquetan para proveer una funcionalidad específica. Los frameworks facilitan el ensamblado de objetos; levantan el nivel de abstracción.
- ***General Inter-ORB Protocol (GIOP)*** especifica un conjunto de formatos de mensajes y representaciones comunes de datos para la comunicación entre ORBs. Se definió para funcionar directamente sobre cualquier protocolo de comunicación orientado a conexión.
- ***Groupware:*** un conjunto de tecnologías que permiten representar procesos complejos que se centran alrededor de actividades humanas colaborativas. Las tecnologías involucradas son: manejo de documentos multimediales, manejo de transacciones complejas y flujo de trabajo, email, conferencias y scheduling. Este esquema de trabajo permite recolectar datos (imágenes, texto, faxes, boletines, etc) y organizarlos en un *documento* que se puede ver, almacenar y duplicar en cualquier lugar de la red.

- ***Interface Definition Language (IDL):*** es un lenguaje declarativo que sirve para definir las interfaces públicas de los objetos, sin detallar la implementación. En términos más generales, se puede afirmar que IDL define las características de un servicio distribuido con tipos comunes de datos para interactuar con ellos en ambientes distribuidos. Los objetivos que se persiguen con IDL son: independencia del lenguaje de implementación; definición de tipos complejos de datos; especificación de servicios distribuidos.
- ***Internet Inter-ORB Protocol (IIOP)*** especifica cómo los mensajes del GIOP se envían sobre una red TCP/IP. Permite usar Internet como puente para comunicar los ORBs. Cualquier ORB compatible con CORBA debe soportarlo.
- ***Implementaciones de objetos:*** normalmente definen las estructuras de datos para las intancias y el código de los métodos. Muchas de las implementaciones usan otros objetos o módulos adicionales de software para implementar el comportamiento de los objetos. Con CORBA se admiten muchas implementaciones de objetos, incluyendo arquitecturas de servidores separados, integración de librerías, aplicaciones encapsuladas, bases de datos orientadas a objetos, etc. Mediante el uso de object adapters es posible soportar virtualmente cualquier tipo de implementación para los objetos.
- ***Implementation Repository:*** registra información que permite al ORB localizar y activar las implementaciones de los objetos. Esta información incluye: políticas de activación de los objetos, seguridad, etc.
- ***Interface de un objeto:*** el protocolo público que el objeto suministra para la interacción externa.
- ***Interface Repository:*** es un servicio que suministra objetos persistentes que representan la información IDL, de una manera dinámica. El Interface Repository es usado por el ORB para hacer las invocaciones. También se puede usar para determinar las operaciones válidas, sus parámetros, etc.
- ***Middleware:*** en este contexto se refiere a la infraestructura necesaria para ofrecer un ambiente organizado de ejecución para los objetos distribuidos, con características tales como transparencia de localización, independencia de la plataforma, etc.
- ***Object Adapter:*** es un mecanismo básico que permite que los objetos accedan a los servicios suministrados por un ORB (generación y administración de referencias a objetos, invocación de métodos, seguridad en las interacciones, activación y desactivación de las implementaciones de objetos, mapping de las referencias de objetos a las correspondientes implementaciones, registración de implementaciones).
- ***Objeto Distribuido:*** es un objeto que puede ser accedido remotamente. Esto significa que un objeto distribuido puede usarse como un objeto tradicional, pero desde cualquier lugar de una red. La localización de un objeto distribuido no es crítica para el objeto que lo accede.
- ***Object Linking and Embedding (OLE):*** tecnología para docuemntos compuestos que suministra un ambiente para la interacción de componentes. OLE compite con la combinación CORBA/OpenDoc. Esta tecnología es propietaria (Microsoft).
- ***Object Management Architecture (OMA):*** es una visión de muy alto nivel de en entorno completamente distribuido. Consiste en cuatro componentes que se clasifican de la siguiente manera: *componentes orientados al sistema* (Object request broker y Object Services) y *componentes orientados a las aplicaciones* (Application Objects y Common Facilities.

- ***Object Management Group (OMG):*** consorcio creado en 1989 con el propósito de promover la teoría y práctica de la tecnología de objetos en sistemas de computación distribuida. En particular, busca reducir la complejidad y costos facilitando la integración de aplicaciones. Originalmente, el OMG estaba constituido por 13 empresas; actualmente lo conforman más de 500. OMG cumple con sus objetivos definiendo estándares que facilitan la portabilidad e interoperabilidad de aplicaciones de objetos distribuidos. No producen software ni lineamientos de implementaciones; sólo especificaciones obtenidas de las ideas propuestas por sus miembros.
- ***Object Request Broker (ORB):*** suministra los mecanismos que permiten la comunicación transparente entre los clientes y los servidores. El ORB simplifica la programación distribuida desacoplando al cliente de los detalles de la invocación de los métodos. De esta manera, los pedidos a servicios remotos son vistos como invocaciones locales. Cuando un cliente invoca una operación, el ORB se encarga de encontrar al objeto que implementa el método, activarlo si fuera necesario, enviarle el pedido y retornar el resultado.
- ***OpenDoc:*** es una arquitectura para componentes de software implementada como un conjunto de clases y servicios para diferentes plataformas. Este modelo de componentes es la mejor expresión del paradigma de documentos compuestos. Los componentes en el mundo OpenDoc se denominan *parts*. Una parte OpenDoc está formada por datos almacenados en documentos compuestos (texto, gráficos, planillas de cálculo, video, etc) y un editor que manipule tales datos.
- ***Referencia a un objeto (object reference):*** se refiere a la información necesaria para especificar un objeto dentro de un entorno ORB. Ambos, clientes e implementaciones de objetos, tienen una noción opaca de las referencias a objetos, pues están aislados de la representación real de éstas (pues dependen del mapping al lenguaje de programación). Internamente, los ORBs pueden diferir en la representación interna que hacen de estas referencias, pero todos deben suministrar el mismo mapeo para cada lenguaje de programación posible. Así, un programa escrito en un dado lenguaje puede acceder a referencias de objetos independientemente del ORB que las administre.
- ***Stubs del cliente:*** en el mapeo de lenguajes no orientados a objetos hay una interface de programación para los stubs de cada tipo de interface. Generalmente, estos stubs permiten el acceso a las operaciones definidas en IDL para un objeto, de una manera previsible (son interfaces estáticas). Los stubs hacen las invocaciones en el ORB. Los lenguajes orientados a objetos, como C++ y Smalltalk, no requieren estos stubs.
- ***Servidor:*** una entidad o programa que suministra recursos o servicios. Los servidores atienden los pedidos de los clientes.
- ***Skeletons de la implementación:*** Para el mapeo de cada lenguaje particular, debe haber una interface (skeleton) para los métodos que implementan los objetos. El ORB invoca las operaciones del objeto (implementación) usando estos skeletons. La existencia de un skeletons no implica la existencia de stubs en los clientes (éstos pueden usar las interfaces dinámicas para hacer las invocaciones).
- ***X/Open:*** es una organización mundial e independiente para sistemas abiertos. Su estrategia es combinar varios estándares en un entorno de sistemas integrales denominado CAE. X/Open apoya sus especificaciones con un conjunto extenso de tests

que cualquier producto debe pasar para obtener la certificación de la marca X/Open (XPG).

BIBLIOGRAFIA Y FUENTES DE INFORMACION

- Thomas J. Mowbray, Raphael C. Malveau, ***"CORBA, Design Patterns"***. John Wiley & Sons, 1997.
- Robert Orfali, Dan Harkey, Jeri Edwards, ***"The Essential Distributed Objects"*** John Wiley & Sons, 1996.
- Timothy W. Ryan, ***"Distributed Object Technology, Concepts and Applications"*** Prentice Hall PTR, 1997.
- David Orchad, ***"Java Component And Distributed Object Technologies"***, Object Magazine, enero de 1998.
- Ted Morin, ***"Migrating Legacy Systems to CORBA"***, Object Magazine, enero de 1998.
- E. Gamma, R. Helm, R. Johnson, J. Vlissides, ***"Design Patterns. Elements of reusable Object-Oriented Software"***. Addison Wesley, 1995.
- Nancy Wilkinson, ***"Using CRC Cards. An Informal Approach to Object-Oriented Development"***. AT&T Bell Laboratories, 1995.
- J. Rumbaugh, M. Blaha, M. Premerlani, W. Lorensen, ***"Object Oriented Modelling and Design"***. Prentice Hall, 1991.
- IONA Technologies, ***"Orbix/Smalltalk Programming Guide"***, release 1.1, 1997
- OMG (Object Management Group). Información publicada en Internet (www.omg.org)
- Douglas C. Schmidt's CORBA page. Internet. (siesta/cs.wustl.edu/~schmidt/corba.htm)