Modelos de Integración y Arquitecturas Distribuidas

Por

Daniel Horacio Canepa

Director: *Dr. Claudio Righetti*

Tesis de Grado para la Facultad de Ciencias Exactas y Naturales de la Universidad de Buenos Aires para cumplir con los requisitos necesarios para obtener el título de Licenciatura en Ciencias de la Computación

ABSTRACT

La evolución del software generó y está generando una gran diversidad de plataformas de sistemas operativos, tipos de base de datos, redes de comunicación, arquitecturas de hardware y de aplicaciones basadas en estándares de facto o comerciales resultando así una baja interoperabilidad entre los componentes de los sistemas en una organización.

Dentro de las organizaciones existen gran variedad de bases de datos, sistemas operativos, redes de comunicación, etc y al no estar ordenado y coordinado desde su construcción cada uno de estos elementos crea confusión y frustración para los desarrolladores con el objetivo de integrar aplicaciones e información.

Los beneficios de la integración de aplicaciones están relacionados con la interacción que existe entre los datos, procesos y herramientas para soportar un trabajo coordinado y unificado. Existen varias técnicas, arquitecturas y modelos para llevar a cabo la integración.

En este trabajo, se van a identificar conceptos, establecer objetivos, beneficios y estrategias para la integración de aplicaciones.

Indice

Introducción

La sistemas de información ofrecen soluciones a varios dominios. La naturaleza y herramientas de los sistemas de información, envuelven tanto el desarrollo tecnológico como en las necesidades de servicios de información para las organizaciones.

Los sistemas de información iniciales fueron desarrollados para ser operados por los profesionales de la información, pero esto fue extendiéndose para ser usadas por personas que tienen poca o carecen de experiencia con la utilización de computadoras. Esto llevo a la categorización en varios niveles de las aplicaciones como las herramientas, dependiendo del grado de resolución necesitado.

Dificultades y Motivación

Si un individuo o departamento de un organización es responsable de todo o parte de la tecnología de información (***IT***), puede identificar las computadoras de la empresa, la configuración de la red, control sobre los sistemas que ejecutan funcionalidades del negocio, formatos de almacenamiento de los datos, las aplicaciones en mainframe (***legacy systems – sistemas legados***) almacenando datos en formatos propietarios y las dificultades para trabajar con sistemas operativos incompatibles, hardware de red y protocolos.

Dado estas dificultades surgen los siguientes problemas:

- ✓ La evolución del software ha generado gran diversidad de plataformas de sistemas operativos y hardware
- ✓ Prácticas de mala programación
- ✓ Las arquitecturas de aplicación evolucionaron basándose en estándares de facto o comerciales, resultando en reducción en interoperabilidad entre los componentes de las mismas
- ✓ Decisiones gerenciales (no técnicas) han contribuido a la generación de grandes proyectos monolíticos con planificaciones ridículamente optimistas y recursos inadecuados.
- ✓ Aislamiento de sistemas con necesidad de ínter operación

Objetivo

El objetivo es permitir a todas las aplicaciones de una organización una robusta interoperabilidad y portabilidad para la integración de sus sistemas garantizando:

- ✓ Simple terminología

- ✓ Un estructura (framework) común abstracta
- ✓ Un protocolo e interfaces comunes

Los ***componentes de software*** hacen posible la interoperabilidad e integración. Las aplicaciones están en constante cambio; los clientes no aceptan más aplicaciones que hacen de todo y están buscando componentes más pequeños que pueden combinar de una manera flexible y dinámica con el fin de crear soluciones focalizadas en sus necesidades particulares de negocio.

Los componentes trabajan juntos sólo si han sido diseñados y construidos sobre una interfaz estándar.

Para asegurar la interoperabilidad de todas las aplicaciones las interfaces de cada componente debe ser independientes de plataforma, sistema operativo, lenguaje de programación y aún de protocolo de red.

Estas características pueden ser presentadas en una estructura (**framework**) conceptual o **modelo corporativo**, que refleja la arquitectura de todas las aplicaciones de la organización, describiendo los sistemas operativos, bases de datos, archivos de datos comunes, formatos, interacción de cada sistema, etc.

Capítulo 1: Conceptos

En este capítulo se describe conceptos básicos sobre arquitectura de aplicaciones para posteriormente presentar las definiciones de componentes de software y la importancia de contar con herramientas que permitan la integración de diferentes sistemas en una organización.

1.1. Sistemas monolíticos y MainFrames

En los inicios (también en la actualidad), estaban los mainframes [PET94], acompañados con sistemas de base de datos jerárquicas [ELM94] y terminales sin capacidad de procesamiento (terminales bobas), también conocidas como pantallas verdes que usualmente tenían o tienen un ***gran costo de mantenimiento***, pero tienen la capacidad de servir a un gran número de usuarios y tienen la ventaja (o desventaja, dependiendo del punto de vista), de servir a una administración centralizada.

Los sistemas escritos para mainframes son monolíticos, la interfase del usuario, la lógica del negocio y los accesos a los datos están contenidos un una sola aplicación, programa o procedimiento. Como las terminales no realizaban ningún procesamiento, la aplicación completa se ejecutaba en el mainframe, por lo tanto hace que sea una arquitectura única o de un solo componente.

1.2. Arquitectura Cliente-Servidor

La Arquitectura Cliente-Servidor [ORF97] permite dar un grado de libertad para arreglar y unir componentes, en cualquier nivel.

El advenimiento de las PCS hace posible un cambio profundo en el paradigma de la arquitectura monolítica de aplicaciones basados en mainframes. Las aplicaciones basadas en Cliente-Servidor [TAM91] permiten que determinados procesos sean cargados en las PCS o en cada uno de los usuarios (desktops), logrando así parte de procesamiento, por un lado en el ***Servidor*** y otro en el ***Cliente***.

Las aplicaciones Cliente-Servidor típicamente distribuye sus componentes en uno o varios servidores y en los clientes. Generalmente al Cliente se lo identifica como el componente que tiene la interfaz de usuario.

A continuación se presenta los conceptos y características de este tipo de arquitectura.

1.2.1. *¿ Que es Cliente-Servidor ?*

No existe una definición única que describa arquitectura Cliente-Servidor, pero se presenta a continuación características que describen este concepto. Como el

nombre lo indica, Cliente y Servidor son ***entidades lógicamente separadas***, y juntas trabajan sobre una red para llevar a cabo una tarea. A continuación se describen las propiedades de un sistema Cliente-Servidor [ORF97]:

- ✓ **Servicio**

  Cliente-Servidor es primeramente una relación entre procesos que corren en máquinas independientes. El proceso servidor es un ***proveedor de servicios*** (tareas). El Cliente es un consumidor de servicios. En esencia , Cliente-Servidor provee una clara separación de funciones basadas en una ***idea de servicios***.

- ✓ **Recursos Compartidos**

  Un Servidor puede servir a muchos Clientes al mismo tiempo y regular los accesos para compártir los recursos.

- ✓ **Protocolos Asincrónicos**

  Hay una *relación muchos a uno* entre Clientes y Servidor. El Cliente siempre inicia un diálogo mediante un pedido de servicio. Los Servidores están pasivamente a la espera de requerimientos desde los Clientes.

- ✓ **Transparencia de Localización**

  El Servidor es un proceso que puede residir en la misma máquina del Cliente o es una máquina diferente a través de la red. Los software de Cliente-Servidor usualmente ***enmascaran*** la localización de los Servidores desde los Clientes, direccionando las llamadas de los servicios cuando los necesitan. Un programa puede ser un Cliente, un Servidor o ambos.

- ✓ **Intercambios Basados en Mensajes**

  Clientes y Servidores son sistemas débilmente acoplados que interactúan a través de un mecanismo de pasaje de mensajes.

- ✓ **Encapsulamiento de Servicios**

  El Servidor es un ***especialista***. Un mensaje es recibido por el Servidor por una solicitud de servicio; luego es una decisión del Servidor determinar como va a llevar a cabo el trabajo. Los Servidores pueden ser actualizados sin afectar a los Clientes, hasta tanto la publicación la interfaz del mensaje no haya sido cambiada.

- ✓ **Escalabilidad**

  Los sistemas Cliente-Servidor pueden ser escalados horizontalmente o verticalmente. La escalabilidad ***Horizontal*** significa agregar o remover

estaciones de trabajo clientes. La escalabilidad ***Vertical*** significa migrar a un servidor con mayor capacidad de procesamiento y/o almacenamiento.

✓ **Integridad**

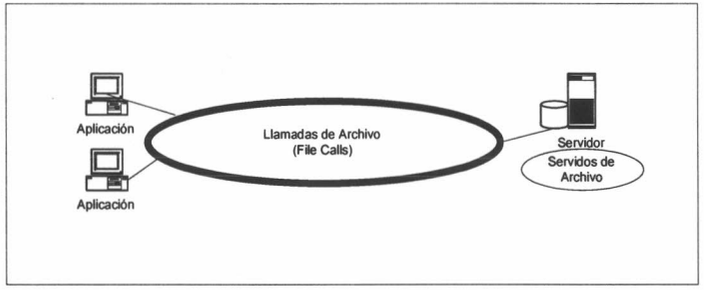
El código del Servidor y datos del Servidor son almacenados centralmente, cuyo resultado es más barato para el mantenimiento y salvaguardar la integridad de los datos compartidos. Los componentes Clientes permanecen en forma independiente.

1.2.2. *Clases de arquitecturas Cliente-Servidor*

Varios sistemas con diferentes arquitecturas han sido llamados "Cliente-Servidor". Vendedores de software usualmente utilizan el término Cliente-Servidor solamente para ser aplicados a sus paquetes específicos. A continuación se presenta diferentes soluciones de arquitecturas Cliente-Servidor.

✓ **Servidores de Archivos**

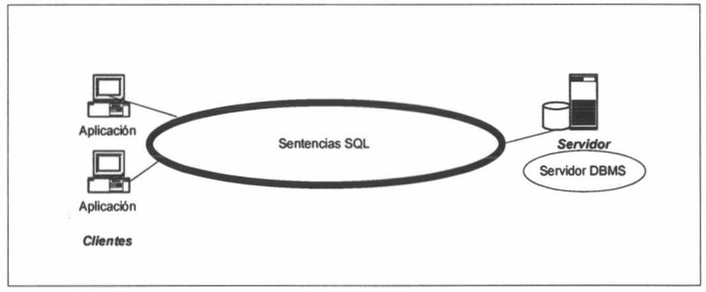
Los *Servidores de Archivos* son usualmente utilizados para compartir archivos a través de la red. Son repositorios, que permiten interactuar de una forma centralizada de almacenamiento para grupos de usuarios de un red.



Cliente-Servidor con Servidor de Archivo

✓ **Servidores de Base de Datos**

El Cliente pasa el pedido de SQL [ELM94] como mensajes hacia el Servidor. El resultado de cada comando SQL es devuelto a través de la red y fue realizado por el Servidor. El código que procesa el pedido de SQL es el Cliente y el dato reside en el Servidor.

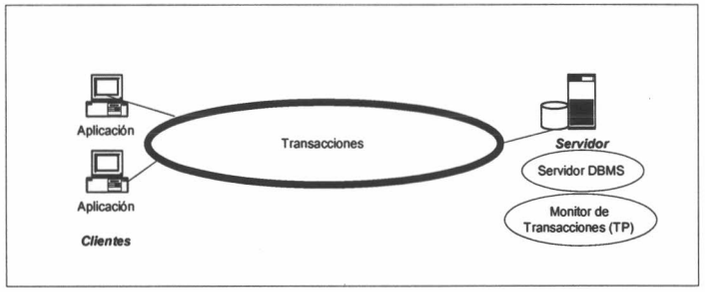


Cliente-Servidor con Servidor de Base de Datos

✓ **Servidores de Transacciones**

El Cliente invoca ***Procedimientos Remotos***, que residen en un Servidor. Estos procedimientos remotos son ejecutados en el Servidor. El intercambio en la red consiste en simples pedidos / mensajes de respuestas. Estos mensaje / pedidos son llamados ***Transacciones***.

Con un *Servidor de Transacciones*, se puede crear aplicaciones Cliente-Servidor escribiendo códigos tanto para el Cliente como el Servidor. La componente Cliente usualmente incluye una ***Interfase de Usuario Gráfica***. La componente Servidor usualmente consiste de transacciones de solicitud de datos.

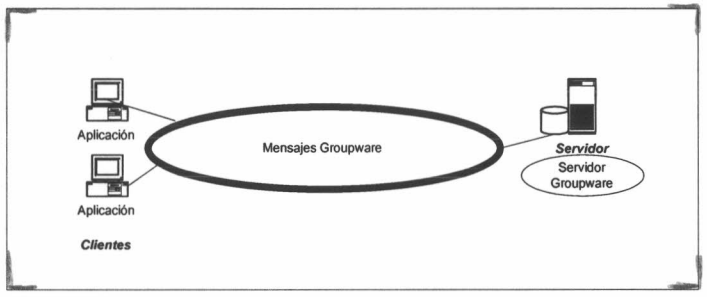


Cliente-Servidor con Servidor de Transacciones

✓ **Servidores Groupware**

El *Groupware* permite la administración de información semi-estructurada como texto, imagen, correo, boletines y flujo de trabajo. Estos sistemas Cliente-Servidor posicionan a las personas en contacto directo con otras personas. **Lotus Notes** es

un ejemplo de estos Servidores. Estas aplicaciones son creadas usando un lenguaje de scripts e interfaces basados en documentos provistos por el vendedor. La comunicación entre el Cliente y el Servidor es de un vendedor específico.

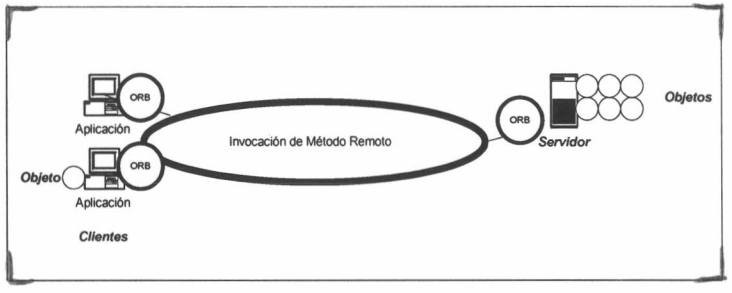


Cliente-Servidor con Servidor de Groupware

✓ **Servidores de Objetos**

La aplicación Cliente-Servidor es escrita como un ***conjunto de comunicaciones de objetos***. Los objetos Cliente se comunican con *Servidores de Objetos* usando un ***Object Request Broker*** (ORB). El Cliente invoca un método sobre un objeto remoto.

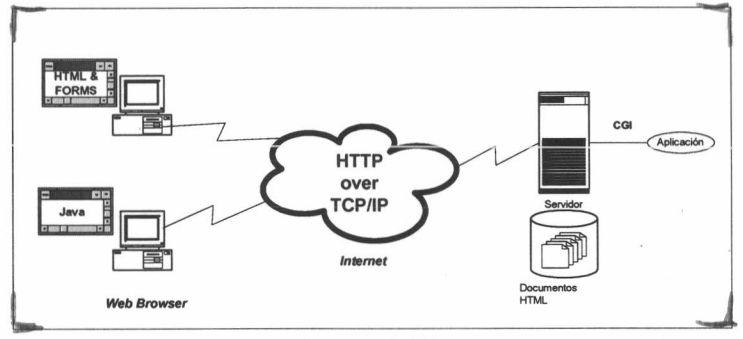
El ORB localiza una instancia de una clase del *Servidor de Objeto,* invocando el método requerido, y devuelve el resultado al objeto Cliente. El objeto Servidor debe proveer soporte para la concurrencia y el compartimiento. El ORB trae todo junto.



Cliente-Servidor con Objetos Distribuidos

✓ **Servidores Web**

Este nuevo modelo de Cliente-Servidor consiste en un cliente ***delgado***, ***portable*** y ***universal*** que habla con uno o varios Servidores. Un Servidor Web devuelve documentos cuando el Cliente pide por ellos mediante el nombre. Los Clientes y Servidores se comunican usando un protocolo como RPC [TAN91] llamado HTTP [TAN96]. Este protocolo define un conjunto simple de comandos; parámetros son pasados como cadenas de caracteres, con ninguna provisión de datos tipados. El Web está siendo extendido para proveer formularios más interactivos para sistemas Cliente-Servidor. Además, el Web y los objetos distribuidos están empezando a venir juntos. *Java* es la primer manifestación de este nuevo ***Objeto Web***.



Cliente Servidor con Servidores Web

1.2.3. *Middleware*

Middleware es un ***programa*** que provee mecanismos estándar de diálogo entre aplicaciones a través de una red. Permite la interacción entre *Clientes* y *Servidores*.

Se puede dividir al Middleware en dos clases:

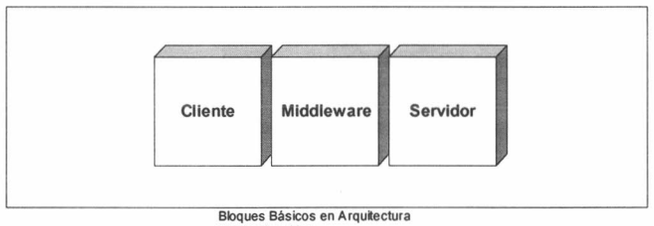
✓ **Middleware de propósito general**

Es la base de la interacción de aplicaciones Cliente-Servidor. Esto incluye pilas de comunicación, directorios distribuidos, servicios de autentificación, tiempos de red, llamadas a procedimientos remotos y servicios de colas. Esta categoría también incluye extensiones de sistemas operativos de red como archivos distribuidos y servicios de impresoras.

✓ **Middleware específico de servicios**

Es necesario para llevar a cabo un tipo de servicio Cliente-Servidor en particular, esto incluye, middleware para base de datos, OLTP (On Line

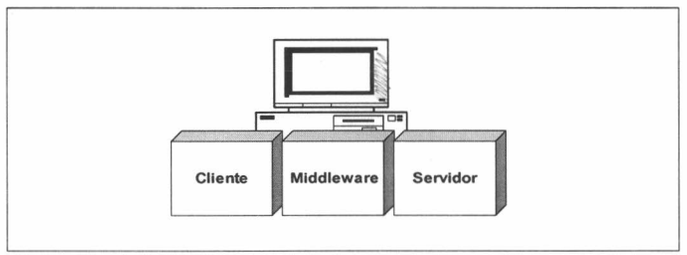
Transaction Processing), de objetos, de Intenet y de administración de sistemas.



Bloques Básicos en Arquitectura
Cliente-Servidor

A continuación se presentan distintas formas en el que puede participar un middleware en una arquitectura ***Cliente-Servidor***.

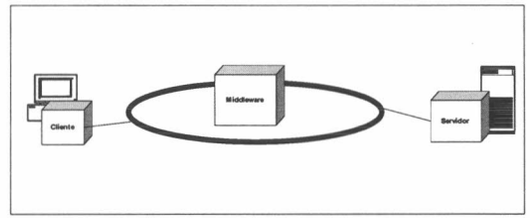
Modelo Básico en una Estación de Trabajo



Arquitectura Cliente-Servidor Integrada

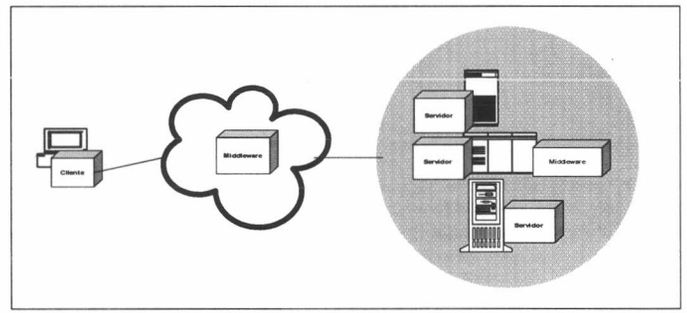
Los bloques del Cliente, Middleware y Servicios de negocio (Servidor) corren en la misma máquina.

Modelo Departamental



Cliente-Servidor para pequeños Negocios y Departamentos

Modelo a Gran Escala



Cliente-Servidor a Gran Escala

1.2.4. *Anatomía de un programa Servidor*

El rol de un programa Servidor es **servir** a múltiples Clientes. A continuación se detalla lo que realiza un programa Servidor.

- ✓ **Espera por un Pedido Iniciado por el Cliente**

  El programa Servidor pasa la mayor parte del tiempo esperando pedidos de los Clientes, en forma de mensajes, que llega sobre la sesión de comunicación. El Servidor siempre tiene que dar una respuesta al Cliente y estar preparado para los *tráficos picos* , cuando muchos Clientes requieran el mismo servicio al mismo tiempo.

- ✓ **Ejecución de Varios Pedidos al Mismo Tiempo**

  El programa Servidor debe hacer el trabajo requerido por el Cliente rápidamente. Un programa de Servidor que no provea multitarea, correrá el riesgo de tener Clientes a la espera de recursos del sistema. El Servidor debe

proveer concurrencia de servicios a múltiples Clientes, mientras que a la vez proporciona integridad en los recursos compartidos.

✓ **Prioridades de Pedidos al Servidor**

Un programa Servidor debe proveer diferentes niveles de prioridades de servicios a los Clientes. Por ejemplo, un Servidor puede responder a un pedido para un reporte o un trabajo por lote (proceso batch) en baja prioridad mientras mantiene una respuesta tipo OLTP (On Line Transaction Processing) para Clientes de alta prioridad.

✓ **Inicio y Ejecución Tareas de Entorno**

Un programa de Servidor debe poder correr tareas de entorno gatilladas (triggers), para ejecutar procesos que no están relacionados con el servicio principal solicitado. Por ejemplo, un pedido para bajar registros de una base de datos fuera de hora.

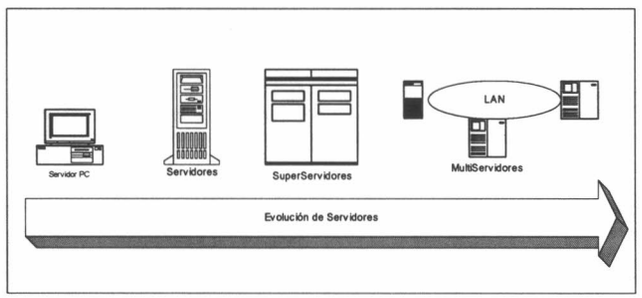
✓ **Mantenerse en Línea**

Un programa de Servidor es una aplicación típicamente de misión crítica. Si el Servidor se cae, impacta en todos los Clientes que dependen de ese servicio. El programa del Servidor y el ambiente en el cual corren debe ser robusto.

✓ **Crecimiento más Grande y Pesado**

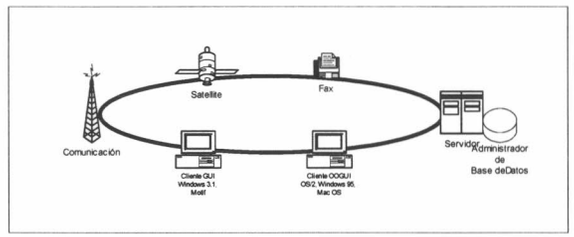
Un programa de Servidor parece tener un apetito insaciable por memoria y poder de procesamiento. En ambiente del Servidor debe ser escalable y modular.

El límite de los Servidores dependen del tipo de servicio requerido por los Clientes. Por tal razón el diseño de la arquitectura de los Servidores, siempre tienen que tener en cuenta la actualización de sus recursos.



1.2.5. *Anatomía de un programa Cliente*

Del lado del Cliente se visualiza los servicios que el sistema provee. Todas las aplicaciones Clientes tienen una tarea en común: **realizan pedidos al Servidor**. Lo que hace diferente a las aplicaciones Cliente es como disparan los pedidos y como es la interfaz con el usuario. Se pueden clasificar a las Clientes en tres categorías: *Clientes No-GUI, Clientes GUI* y *Cliente OOUI (Interfaces de Usuario* Orientadas a Objetos).



Tres Tipos de Clientes

1.3. Tecnología de Objetos

A partir de los problemas de análisis, diseño de la solución, uso y mantenimiento de sistemas, existe la necesidad de integrar cada uno de los componentes de un paso a otro en forma independientemente. La tecnología de objetos captura esta necesidad [WIR90].

La tecnología de objetos significa implementar un modelo del mundo real. Es decir, un objeto en un sistema se corresponde con un objeto real (crea, usa y manipula). Se definen procesos de negocios, enfocados en los datos y no dentro de procedimientos de sistemas (existe una separación entre datos y procesos).

Los objetivos relevantes en la tecnología de objetos son [WAY94]:

- ✓ **Modelar el Mundo**
- ✓ **Reusabilidad** [YOU99]
- ✓ **Mantenibilidad**
- ✓ **Método de Software Unificado** [JOH90]

1.3.1. *Complejidad*

La experiencia en resolver problemas ha demostrado que la mejor forma de manejar la complejidad es dividir el problema en un número de partes pequeñas manejables (subproblemas), con el cuidado en la descomposición. Definiendo así, una relación uno a uno de los subproblemas con los componentes del modelo de objetos.

Los componentes son fácil de entender para los clientes, analistas, programadores y usuarios, y las interacciones entre los objetos aparecerán lógicamente y bien fundamentadas.

La ***orientación a objetos*** además de dar un modelo del mundo real, facilita la solución de los tres mayores problemas de programación [GAM95]:

- ✓ **Actualización**
- ✓ **Desarrollo**
- ✓ **Mantenimiento correctivo, perfectivo y evolutivo**

1.3.2. *¿ Que es un objeto ?*

La ***orientación a objeto*** permite aproximar el manejo de la complejidad de los problemas del mundo real realizando una abstracción del conocimiento y encapsularlo en un objeto o conjuntos de objetos relacionados [WIR90].

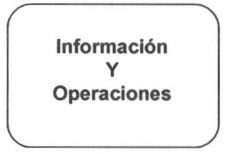
Usualmente la información se divide en dos tipos: ***funciones*** y ***datos***.

Las operaciones asociadas con los objetos caracteriza su comportamiento. Los objetos con el mismo comportamiento son agrupados en ***Clases*** y son conocidos como instancias de sus clases.

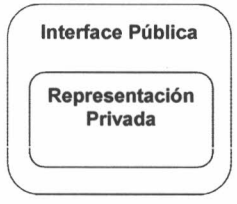
Al trabajar con objetos hay un interés en el ***¿ QUE HACE ?*** y no ***¿ COMO LO HACE ?*** [FOO88].

Las características que distinguen a los objetos son [WIF90]:

- ✓ **Encapsulamiento**

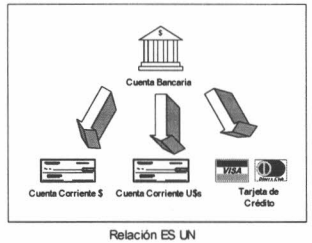


- ✓ **Information Hiding**



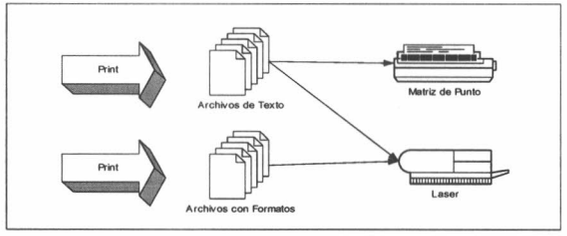
El ***encapsulamiento*** e ***information-hiding*** trabajan juntos para aislar una parte del sistema de otras partes, permitiendo que el código pueda ser modificado y extendido sin correr el riesgo de introducir innecesarios y malentendidos efectos colaterales.

- ✓ **Herencia**



Relación ES UN

- ✓ **Polimorfismo**



1.3.3. *Conclusiones*

- ✓ Los objetos son componentes de software, que contiene datos y pueden ser manipulados. Un objeto puede mandar mensajes a otros objetos y recibir una respuesta. Son diseñados para reusabilidad.
- ✓ Si tenemos más de una implementación de un objeto con la misma interfaz (sin importar el lenguaje de programación, o el tipo de almacenamiento interno), se puede sustituir uno por otro en el sistema y el cliente no se daría cuenta de esto, porque la respuesta al mensaje no cambiaría. Esto es sustancialmente la clave de una ambiente de ***componentes de software***.
- ✓ El encapsulamiento permite una transparente localización , porque no hay que reconfigurar los objetos movidos a través de la empresa.
- ✓ La herencia permite definir nuevos objetos a partir de objetos existentes con el mismo comportamiento.

1.4. Computación Distribuida Heterogénea

Una ***aplicación distribuida*** es una aplicación cuyos componentes de sistemas residen en una o más computadoras sobre una red, con dicha red típicamente compuesta con diversas computadoras y sistemas operativos.

La diversidad de hardware y software es un hecho hoy en día y los ambientes de red son más diversos. También se conoce que desde una gran lista de computadoras asignadas a diferentes aplicaciones son usadas por una variedad de sistemas operativos y ~~realizadas~~ ejecutan por diferentes lenguajes de programación.

Clasificación de problemas en redes heterogéneas:

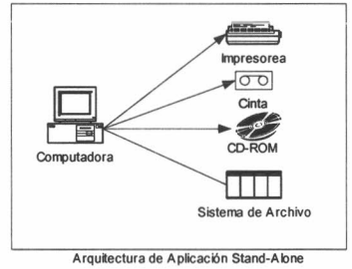
- ✓ Dificultad de obtener hardware (computadoras y redes) que trabajen juntos en forma transparente

- ✓ Complejidad de sistemas heterogéneos que trabajen juntos (formatos, almacenamiento de datos, sistema operativo, etc.)
- ✓ Consumo de Recursos

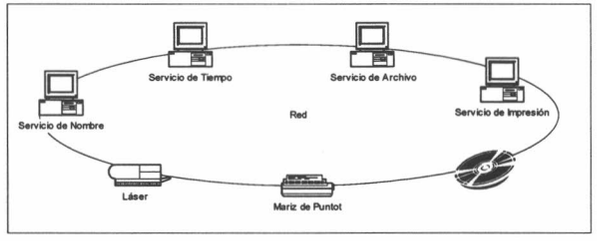
Existe la necesidad de definir componentes de software, para lograr la combinación de flexibilidad y dinamismo, creando herramientas orientadas a la necesidad de un objetivo en particular. Los componentes trabajarán juntos si solo si han sido diseñados y construidos en una interfaz estándar [T1199] para su ínter operación.

Los objetivos de una arquitectura distribuida son:

- ✓ Simplificar los desarrollos de las aplicaciones
- ✓ Proveer una base flexible para los servicios a nivel de usuario



Arquitectura de Aplicación Stand-Alone



Arquitectura de una Aplicación Distribuida

1.4.1. *Definición de un Sistema Distribuido*

Un sistema distribuido es un conjunto de procesadores débilmente acoplados (no comparte memoria ni reloj), interconectados por una red de comunicaciones. Desde

el punto de vista de un procesador específico en un sistema distribuido, los demás procesadores y sus recursos respectivos son ***remotos***, mientras que sus propios recursos son ***locales***.

1.4.2. *Administración de Recursos*

Si varias instalaciones (con capacidades diferentes) están conectadas entre sí, entonces el usuario de una instalación puede utilizar los recursos disponibles en otra. Por ejemplo, un usuario de la instalación A puede usar una impresora láser que sólo está disponible en la instalación B; mientras tanto, un usuario en B puede acceder a un archivo que reside en A. En general, la administración de recursos en un sistema distribuido [PET94] proporciona mecanismos para compartir archivos en instalaciones remotas, procesar información en una base de datos distribuidas, imprimir archivos en instalaciones remotas, utilizar dispositivos remotos especializados (procesador de arreglos de alta velocidad) y otras operaciones.

1.4.3. *Mejora de Cálculos*

Si un cálculo puede dividirse en varios subcálculos que pueden ejecutarse concurrentemente, entonces la disponibilidad de un sistema distribuido permite distribuir los cálculos entre varias instalaciones y ejecutarlos en forma concurrente.

1.4.4. *Confiabilidad*

Si en un sistema distribuido, hay un error en una instalación, los restantes, potencialmente, pueden continuar trabajando. Si existe suficiente redundancia en el sistema (tanto de hardware como de datos), generalmente puede continuar con su trabajo, incluso si han fallado algunas de las instalaciones. El sistema debe detectar la falla de una instalación y tomar las medidas necesarias para recuperarse.

1.4.5. *Complejidad*

El desarrollo de aplicaciones distribuidas muestra complejidades;

- ✓ **Inherentes**
  - ➢ Dirección del impacto del retardo
  - ➢ Detección y recuperación de las fallas parciales de las redes y los servidores
  - ➢ Carga del balance y particionamiento del servicio
  - ➢ Consistencia en el ordenamiento de los eventos distribuidos.
- ✓ **Accidental**

  - ➢ Falta de seguridad, portabilidad, re entrada, interfaces de llamadas de sistemas extensibles y librerías de componentes
  - ➢ Inadecuado soporte de debugging

1.4.6. *Tipos de Sistemas Operativos*

Un sistema operativo distribuido proporciona a los usuarios acceso a los distintos recursos que ofrece el sistema, entendiendo por ***recursos*** tanto el hardware como el software. El acceso a estos recursos lo controla el sistema operativo; existen dos esquemas básicos complementarios para proporcionar este servicio [ORF97] [PET94]:

- ✓ **Sistemas Operativos de Red**
- ✓ **Sistemas Operativos Distribuidos**

1.4.7. *Nominación y Transparencia*

La ***nominación*** es una correspondencia entre objetos lógicos y físicos [PET94]. Por ejemplo, los usuarios tratan con objetos de datos lógicos representados por nombres de archivos, mientras que el sistema manipula bloques de datos físicos almacenados en las pistas de discos. Generalmente, un usuario se refiere a un archivo utilizando un nombre, el cual se transforma en un identificador numérico de bajo nivel, que a su vez se corresponde con bloques en disco. Esta correspondencia multinivel ofrece a los usuarios la abstracción de un archivo, que oculta los detalles de cómo y dónde se almacena el archivo de disco.

La ***transparencia*** significa esconder la red y sus servidores de los usuarios y aún de las aplicaciones programadas. Hay varios tipos de transparencia que permiten el acto de desaparición de la red [PET94]: trabajar en un nivel abstracto ?

- ✓ **de localización**
- ✓ **de Nombres**
- ✓ **de Entrada a la Red (logon)**
- ✓ **de Replicación**
- ✓ **de Acceso Local / remoto**
- ✓ **de Tiempos**
- ✓ **de Fallas**
- ✓ **de Administración de Red**

1.5. Objetos distribuidos

Los objetos distribuidos son usualmente vistos como configuraciones ***Cliente-Servidor***. Los objetos responden mensajes de requerimiento, proveen un recurso o servicio a un solicitante (Objeto Cliente), que residen en distintas computadoras a

través de la red. Las características que tienen los objetos con respecto al encapsulamiento, la propiedad de transparencia es inherente en los sistemas distribuidos.

Los objetos del lado del ***Servidor*** ofrecen servicios y recursos. Los Objetos del lado del ***Cliente*** solicitan servicios y recursos, sin importar en dónde residen. Ellos ínter operan intercambiando mensajes entre los objetos, cumpliendo cada el rol de solicitante (Cliente) y proveedor (Servidor).

1.5.1. *Modelo de Objetos Distribuidos*

Las aplicaciones de objetos distribuidos permite:

- ✓ **Reusabilidad de funcionalidades existentes**

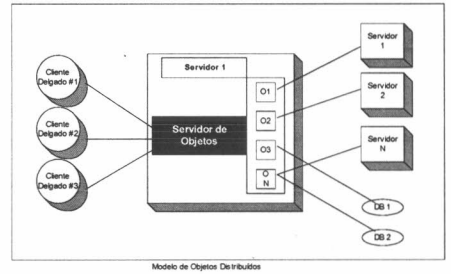
  El mayor beneficio en el desarrollo de objetos distribuidos es la habilidad de reusabilidad de código, permitiendo de esta manera el desarrollo rápido de aplicaciones (RAD-Rapid Application Development), usando componentes ***plug and play*** (instalar y listo para usar), sin importar su ubicación, sólo conocer su invocación y disponibilidad en la red.

- ✓ **Desarrollo Aislado**

  Para que la ínter operación ocurra, los módulos deben comunicarse utilizando un protocolo o interfaz común. Esta característica facilita a los proyectos de gran escala definir múltiples equipos para trabajar en forma consistente con distintos módulos de la aplicación y a posterior construyendo los módulos a través de la interfaz convenida.

- ✓ **Clientes Delgados**

  La mayoría de los componentes de una aplicación de objetos distribuidos son localizados en el ***Servidor***, las aplicaciones del lado del Cliente deben ser *de baja complejidad*. Esto permite más libertad en los recursos de los sistemas clientes, mientras que el núcleo del procesamiento sea ejecutado en el ***Servidor***.



Modelo de Objetos Distribuidos

1.5.2. *Beneficios de Objetos Distribuidos*

La tecnología de ***Objetos Distribuidos*** se ajusta perfectamente para crear sistemas Cliente-Servidor flexibles porque la ***lógica de los datos y del negocio*** están encapsuladas dentro de objetos, permitiendo ser localizados en cualquier lugar dentro del ***sistema distribuido*** a través de sus interfaces en forma transparente.

La arquitectura de objetos distribuidos permite construir aplicaciones altamente interactivas y escalables en aplicaciones Cliente-Servidor:

- ✓ **Programación aislada**

  Los programadores no se tienen que preocupar sobre los detalles de bajo nivel de la programación de la red. El protocolo de ***Nivel de Aplicación*** [TAM91] es definido como métodos sobre los objetos utilizando un *lenguaje de definición de interfaces (IDL-Interface Definition Language)*. Este protocolo de nivel de aplicación es mapeado a todos los protocolos de cualquier propósito por el sistema de objetos distribuidos.

- ✓ **Concepto de Objetos**

  Los *Sistemas de Objetos Distribuidos* traen los conceptos útiles de la orientación a objetos como separar las interfaces de la implementación, herencia, polimorfismo y reusabilidad de código para sistemas distribuidos.

- ✓ **Reglas de Dominio**

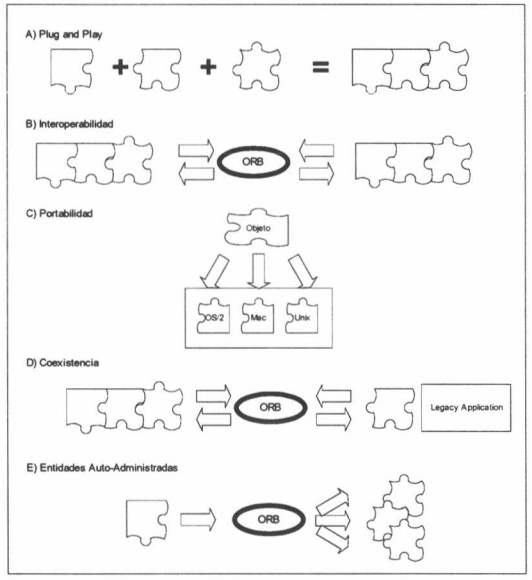
  Separar explícitamente los datos que está resolviendo un proceso o tarea específica de su presentación.

- ✓ **Notificación Asincrónica**

  Modelo de programación que reduce la carga sobre el Servidor. El Servidor está siempre esperando una solicitud para un servicio determinado.

Los objetos son sistemas Cliente-Servidor porque los datos, comportamiento (lógica del negocio) y accesibilidad están encapsulados dentro de los objetos [ORF97], permitiendo la localización en cualquier lugar del sistema distribuido en forma transparente. Los objetos pueden fácilmente enmascararse en elementos de plataformas específicas y hacer que las piezas aparezcan para ínter operar de la misma manera.

Los objetos tienen la particularidad de separar apropiadamente la interfaz de la implementación. Esto significa que se puede usar una interfaz de objeto para envolver las aplicaciones existentes y hacer de ver a ellas, como objetos comunes.



1.5.3. *Componentes*

Los Objetos distribuidos pueden vivir en cualquier lugar de la red. Los objetos son empaquetados como piezas independientes de código que pueden ser accedidos por Clientes remotos vía invocación de mensajes. El Lenguaje y el Compilador usados para crear Servidores de objetos distribuidos son totalmente transparentes para los Clientes.

Al referirse a ***Objetos Distribuidos***, realmente se hace referencia a ***Componentes Independientes de Software***. Estos son piezas de Software que pueden participar en diferentes redes, sistemas operativos, lenguajes de computadoras, o implementaciones. Los objetos son construidos como componentes para ser provistos adecuadamente a las aplicaciones distribuidas (pueden ser usados por cualquier aplicación).

Los objetos distribuidos por definición son ***componentes*** por la manera que fueron empaquetados. La infraestructura de los objetos distribuidos hace más fácil para los componentes ser más ***autónomos***, ***auto-administrables*** y ***colaborativos***.

Los componentes son ***cajas negras*** que reducen la complejidad del desarrollo de procesos.

Las mínimas funciones que debe proveer los componentes son [T1199]:

- ✓ **Entidad Comercial**
- ✓ **No es una Aplicación Completa**
- ✓ **Diferentes Usos**
- ✓ **Correcta Especificación de Interfaz**
- ✓ **Interoperabilidad** [JOU95]

Los ***componentes*** son piezas reusables de software auto contenidas que es independiente a cada aplicación.

Para crear componentes autónomos e ***inteligentes*** deben incluir [ORF97]:

- ✓ **Seguridad**
- ✓ **Licenciamiento**
- ✓ **Versiones**
- ✓ **Administración**
- ✓ **Notificación de Eventos**
- ✓ **Configuración y Administración Independiente**
- ✓ **Auto descriptivo**
- ✓ **Control Transaccional**
- ✓ **Persistencia**
- ✓ **Relaciones**
- ✓ **Fácil de Usar**
- ✓ **Semántica de los Mensajes**

Dado las características anteriores, lo componentes se pueden definir de la siguiente manera:

- ✓ Cualquier subsistema que pueda ser separado y que posea una interfaz reusable y potencialmente normalizada.[T1199]
- ✓ Un elemento de software que puede ser fácilmente invocado y utilizado en diferentes contextos, incluyendo aquellos ambientes no previstos.
- ✓ Un objeto con una interfaz pública para ser utilizado dentro de un ambiente orientado a objetos.
- ✓ Son una combinación de un ***modelo de programación*** y un ***meta modelo de información***.
- ✓ Un subsistema que no está ligado a ninguna aplicación especifica.
- ✓ Un subsistema resultado de un cuidadoso diseño y testeo. Empaquetado para reuso, con una interfaz bien definida y protegida.

1.5.4. *Características de Objetos Distribuidos*

Los objetos distribuidos aprovechan las características inherentes de los objetos en sí mismo, potenciándose así cuando se habla de una red de aplicaciones residiendo en distintos lugares:

- ✓ **Objetos Distribuidos pueden ser Clientes y Servidores**

  En sistemas tradicionales Cliente-Servidor, está claro quien es el Cliente y el Servidor. Pero en los sistemas de objetos distribuidos no se puede claramente distinguir el Cliente del Servidor.

- ✓ **Evolución Continua**

  Cuando se interactúa con un objeto, solamente vemos la punta del iceberg. Este objeto puede delegar partes de la implementación a otros objetos; que lo realiza en dinámicamente en tiempo de ejecución. A causa de la subclasificación (herencia), la implementación de un objeto puede cambiar a través del tiempo sin que el original programador lo sepa o tenga el cuidado.

- ✓ **Interacción Encapsulada**

  A causa del encapsulamiento, no se puede completamente entender las interacciones que se toman lugar entre los objetos invocados. Hay mucho en la actividad ***detrás en escena***.

- ✓ **Polimorfismo**

  Los objetos son flexibles; fácil de reemplazar por otros objetos simplemente respetando la misma interfaz y sin importar en dónde residen.

- ✓ **Escalabilidad**

  Los objetos puede escalar sin un límite determinado.

- ✓ **Escalabilidad**

  Los objetos van y vienen. Pueden ser creados dinámicamente y auto-destruidos cuando no son más usados.

1.6. Objetos de Negocio

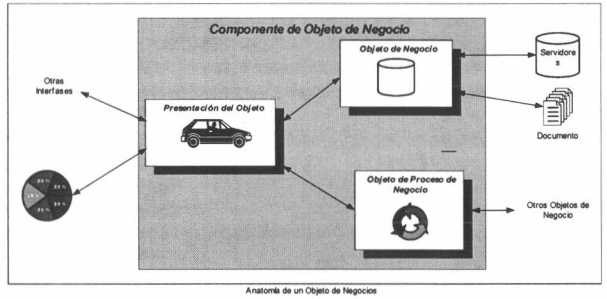
Los negocios y las aplicaciones de software en dónde se desarrollan pueden ser definidos en términos de Objetos de Negocio. El ***objeto de negocio*** es usado en dos aspectos distintos, pero relacionados:

- ✓ En un modelo de negocio que describe una solución para un área específica.
- ✓ En un modelo para un sistema de software o su diseño para el cual refleja los conceptos del negocio.

En el contexto de modelo del negocio o de ingeniería, **un objeto de negocio describe una cosa, concepto, proceso o evento en una operación, administración o planificación**.

El modelo de objetos de negocio especifica atributos, relaciones, y acciones o eventos que son aplicados a los objetos que lo integran y no a las relaciones en la implementación del sistema.

Los objetos de negocios se definen independientemente de las aplicaciones.



Anatomía de un Objeto de Negocios

Las interfaces determinan el comportamiento de los objetos, en particular los objetos de negocio.

En el contexto de software o de una aplicación, **el objeto de negocio representa el concepto de la modelización del negocio y está representado mediante un objeto**.

El ***objeto de negocio*** es la representación abstracta en el sistema del modelo de negocio. Describe un concepto de negocio y la identificación del negocio en un sistema.

Un concepto de negocio provee consistencias y contexto, mediante un proceso y dicho proceso tiene que estar diseñado y modelado dentro del sistema.

Los objetos de negocios pueden ser clasificados en una de las siguientes tres categorías:

- ✓ **Objeto de negocio Entidad**

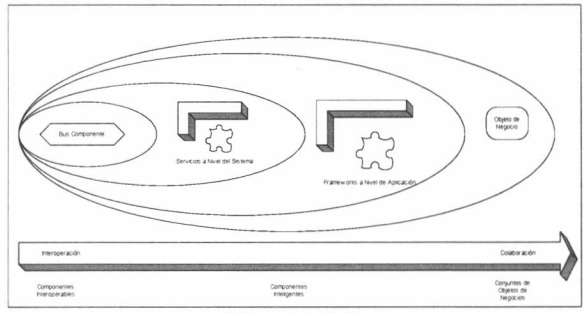
  Representa un nombre de un negocio, como personas lugares, cosas y conceptos; estos son actores y recursos que participan en procesos de negocios.

- ✓ **Objeto de Negocio Proceso**

  Representa acciones de negocio, flujos de trabajo o colecciones de actividades que requiere la participación de múltiples actores; estos son colecciones estructuradas de entidades, interacciones, eventos trabajando juntos en un patrón medido y administrable (Ejemplo: proceso de facturacion de una empresa).

- ✓ **Objeto de Negocio Evento**

  Son causas o resultados de procesos o acciones; ocurrencias, interrupciones, pasajes de tiempo; los eventos ocurren en cada interacción de las entidades de objetos de negocio, a pesar que no todo evento puede ser medido o administrado.



Evolución de los Componentes y Límites de Infraestructura

1.7. Conclusiones

Se presentaron los conceptos básicos de arquitectura de sistemas y definiciones de objetos, objetos distribuidos y objetos de negocio, que son claves a la hora de definir arquitecturas de interacción e integración de aplicaciones, y no son nada nuevos en la industria de desarrollo de sistemas de información.

Las características y propiedades de estas entidades son aprovechadas para la construcción de modelos más generales, logrando un nivel de abstracción en distintos niveles en el análisis, diseño, programación y mantenimiento de aplicaciones, coexistiendo con los procesos y datos en el cuál son utilizados los sistemas de información.

En los capítulos subsiguientes se presentan modelos de arquitecturas de aplicaciones para desarrollar políticas de convivencia de sistemas de información, sin tener en cuenta el sistema operativo en el cual se ejecutan, red de información, protocolo de comunicación, administrador de base de datos, etc.

Capítulo 2: CORBA – Common Object Request Broker Architecture

En este capítulo se describe que es *CORBA*, y como surge como una infraestructura abierta de objetos distribuidos siendo definido como un estándar por el Object Management Group (OMG)[ORG96], con el objetivo de estructurar la integración de una gran variedad de *Sistemas de basados en Objetos*.

OMG especifica como los Sistemas de Objetos Distribuidos a través de una red pueden trabajar sin importar el sistema operativo y el lenguaje de programación del Cliente y/o el Servidor.

CORBA es una plataforma de objetos distribuidos completa. Extiende las aplicaciones a través de la red, lenguajes, límites de los componentes y sistemas operativos. CORBA ***Object Request Broker*** (ORB) conecta las aplicaciones clientes con los objetos que desean usar. La ***aplicación cliente*** no necesita conocer si el objeto reside en la misma computadora o en una computadora remota en cualquier lugar sobre la red.

CORBA es un ***framework*** de aplicación que provee interoperabilidad entre los objetos, construidos en (posiblemente) lenguajes diferentes, corriendo en (posiblemente) máquinas en ambientes distribuidos heterogéneos.

OMG realiza especificaciones [OMG96] en el cuál se puede construir un ambiente desde la base de una arquitectura en un lenguaje de definición de interfase (IDL-Interface Definition Language) y Object Request Broker (ORB) de servicios y recursos que giran en un entorno de programación y ambientes de interoperabilidad.

OMG es un consorcio de vendedores de software [ORG96] y usuarios finales. Varias compañías miembros de la OMG están desarrollando productos comerciales que soportan estándares y/o desarrollando software con las especificaciones publicadas por este organismo.

CORBA proporciona:

- ✓ una arquitectura de ***interoperabilidad*** de sistemas para la producción de software,
- ✓ permite una plataforma para comercializar en una gran variedad productos en ***redes de comunicación*** con un mínimo esfuerzo.

2.1. Object Management Group (OMG)

Object Management Group es un grupo sin fines de lucro, fundado en 1989 con el propósito de promover una teórica y práctica tecnología de objetos en sistemas distribuidos de computadora. En particular, en dirigir la reducción de la complejidad, reducir los costos y acelerar las nuevas aplicaciones de software. Originalmente

estaba formada por trece (13) compañías, pero los miembros del OMG creció a más de ochocientos (800) miembros de software, desarrolladores y usuarios[ORG96] al 4/98. La diversidad de los miembros abarca integradores y vendedores de sistemas, telecomunicaciones, instituciones financieras, salud, universidades y gobiernos.

El objetivo del OMG [KKS96] es crear un estándar en el cual permita interoperabilidad y portabilidad en aplicaciones orientadas a objetos. No producen software o guías de Implementación; sólo especificaciones en el cual son ideas sugeridas de todos los miembros de la OMG, quien responden a REQUERIMIENTO DE INFORMACION (RFI-Requests For Information) y PEDIDOS DE PROPUESTAS (RFP-Requests For Proposals).

La organización está estructurada:

- ✓ **Comité de Tecnología de Plataforma**
  - ➢ Servicios de Objetos
  - ➢ Servicios comunes
  - ➢ Análisis y diseño
  - ➢ Tiempo real
- ✓ **Arquitectura**
  - ➢ Políticas y procedimientos
  - ➢ Requerimientos de usuarios finales
  - ➢ Métricas
  - ➢ Seguridad
  - ➢ Modelo referencial
  - ➢ Modelo de dominio referencial
- ✓ **Comité de Tecnología de Dominio**
  - ➢ Financiero
  - ➢ Objeto de negocio
  - ➢ Salud
  - ➢ Manufactura
  - ➢ Comercio Electrónico
  - ➢ Telecomunicaciones
  - ➢ Transporte

2.2. Modelo de Objeto de OMG

Cuando un gran grupo de colaboradores trabajan para un bien técnico en común, es necesario trabajar sobre una base consistente de entendimiento y terminologías. Para tal fin, el modelo de objeto OMG define una semántica común de objeto para la especificación visible exteriormente de las características de los objetos en un

estándar y una forma de implementación independiente. Esta semántica en común caracteriza a los objetos que existen en el sistema especificado por OMG.

Este modelo está basado en un pequeño número de conceptos básicos:

- ✓ Objetos
- ✓ Operaciones
- ✓ Clases
- ✓ Subclases

Un objeto puede representar cualquier tipo de entidad como una persona, un bote, un documento, etc. Las operaciones son aplicadas a los objetos y permiten concluir ciertas especificaciones sobre los objetos como determinar la fecha de nacimiento de una persona. Las operaciones asociadas con el objeto caracterizan el comportamiento del objeto.

Los objetos son creados como instancias de una clase. Uno puede ver a la clase como un patrón (TEMPLATE) para la creación de una instancia (objeto). Una instancia de la clase bote puede ser un BOTE ROJO, CON CAPACIDAD PARA 6 PERSONAS.

Una clase caracteriza el comportamiento de sus instancias, descriptas por las operaciones que pueden aplicarse a aquellos objetos (capítulo 1).

Pueden existir relaciones entre las clases. Por ejemplo, UNA LANCHA RAPIDA, puede estar relacionada con un bote genérico. Estas relaciones entre las clases con conocidas como subclases / superclases [WIR90].

2.3. *Motivación de CORBA*

CORBA (Common Object Request Broker Architecture), [WHI96] es la respuesta del OMG'S (Object Management Group's) a la necesidad de interoperabilidad a través de un gran número de productos de hardware y software disponibles hoy en día.

CORBA permite a las aplicaciones comunicarse entre sí sin preocuparse donde están localizadas o quién las a diseñado.

CORBA 1.1 fue introducido en 1991 por OMG y definió Lenguaje de Definición de Interfase (IDL-Interface Definition Language) y la Interfase de Programación de Aplicación (API-Aplication Programming Interfaces) que permite a objetos Cliente-Servidor [WIR90] la interacción dentro de una implementación específica de un ORB (Object Request Broker).

CORBA 2.0 [COR95] adoptó y definió en Diciembre de 1994, la especificación de interoperabilidad con el objetivo que diferentes vendedores ORB pueden ínter bópear.

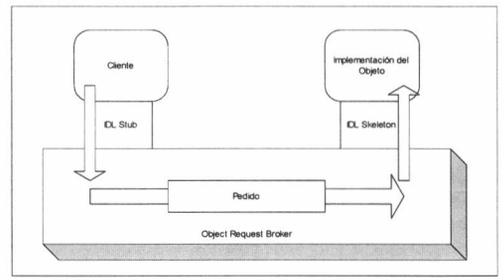
ORB es un ***middleware*** que establece relaciones Cliente-Servidor entre objetos. Usando ORB, un cliente puede transparentemente invocar un método de un objeto servidor, el cual puede estar en la misma máquina o a través de la red.

ORB intercepta la llamada y es el responsable de encontrar al objeto que puede implementar el pedido, pasándolo como parámetro, invoca este método, y devuelve el resultado.

El cliente no tiene que preocuparse de dónde está localizado el objeto, el lenguaje de programación, el sistema operativo o cualquier aspecto de sistema que no fuera parte de la interfase del objeto.

ORB provee interoperabilidad entre aplicaciones de diferentes máquinas en ambientes heterogéneos distribuidos e interconexiones transparentes de sistemas de múltiples objetos.

En el típico campo de aplicaciones Cliente-Servidor, los desarrolladores utilizan su propio diseño o reconocen estándares para definir protocolos a ser usados entre dispositivos. La definición del protocolo depende del lenguaje de programación, transporte de red y una docena de otros factores.



Un Pedido pasando desde el Cliente a la Implementación del Objeto

Con ORB, el protocolo es definido a través de interfaces de aplicaciones vía una simple implementación de especificación de un lenguaje independiente, ***IDL***; ***pedidos basados en IDL*** (Cliente) y ***servicios basados en IDL*** (Servidor). Tanto el *Cliente* como el *Servidor* pueden estar implantados en C++, Java, Smalltalk, C u otros, indistintamente.

ORB provee ***flexibilidad***. Permite a los programadores elegir el más apropiado sistema operativo, ambiente de ejecución y aún el lenguaje de programación a usar por cada una de los componentes de un sistema en construcción.

Más importante, permite la integración de componentes existentes. En la solución sobre un ORB, los desarrolladores simplemente modelan los componentes de una aplicación existente usando IDL, con el objetivo de crear nuevos objetos, luego envuelven dicha aplicación en un objeto (object wrapper) que es interpretado entre el bus de estandarización y las interfaces de las aplicaciones existentes (Legacy Systems).

CORBA es un paso en el camino de la ***estandarización*** e ***interoperabilidad*** en ambientes orientados a objetos. Con CORBA, los usuarios obtienen acceso a la información transparentemente, sin dejarles conocer donde el software o la plataforma de hardware reside o dónde está localizada en la red de la empresa.

La comunicación es el corazón de los sistemas orientados a objetos, CORBA trae una verdadera interoperabilidad al ambiente de computación de hoy en día.

CORBA provee una ***base de interoperabilidad*** basado en objetos y se construye sobre ésta una base a nivel conceptual y no jerárquica [OBJ96].

Con el objetivo de tener objetos ***plug and play*** de una manera útil, los clientes deben conocer exactamente lo que esperan de cada objeto que es llamado para un servicio deseado. En CORBA [MOW95], los servicios que un objeto provee son expresados mediante un ***contrato*** [JOH90] especificado a través de su interfase entre este y el resto de su sistema.

Este contrato sirve para dos propósitos [SIE96]:

- ✓ Informa a los clientes potenciales los servicios que el objeto provee e indica como construir el mensaje para invocar al servicio y
- ✓ Permite una infraestructura de comunicación conocida con el formato de todos los mensajes que el objeto puede recibir o enviar, permitiendo a la infraestructura traducir los formatos de datos necesarios proporcionando una conexión transparente entre el transmisor y el receptor.

Dada la definición del objeto en su funcionalidad y su sintaxis, en ningún momento se definió:

- ✓ Lenguaje de programación que se va a utilizar para implementar
- ✓ Plataforma o sistema operativo en el cual se va a correr
- ✓ ORB que se va a conectar
- ✓ Si va a correr localmente los clientes o remotamente
- ✓ El hardware de red o protocolo que se va a usar, si es remoto
- ✓ Otros aspectos incluyendo por ejemplo, niveles y provisiones de seguridad.

Cada objeto necesita un identificador (***handle***) único para identificar el cliente a través de la infraestructura para rutear el mensaje hacia el servidor. Esto no lo denominamos una dirección del objeto, sino que mantenemos el mismo handle cuando se transporta de una localización a otra. El handle como una ***clase de dirección*** que se realiza la referencia en forma automática.

En el ambiente de redes de computadoras: cada nodo es un objeto con una interfase bien definida, identificada con un único handle. Los mensajes son pasados entre el objeto transmisor y el objeto destino; el destino es identificado por su handle y el formato del mensaje es definido en una interfase definida por el sistema.

Esta información permite a la infraestructura de comunicación cuidarse de todos los detalles.

Esto potentes y simples conceptos, proveen los fundamentos para los ***componentes CORBA***.

Se describe más adelante, que las interfaces son expresadas en OMG Interface Definition Language (IDL), permitiendo accesible a los objetos escritos virtualmente en cualquier lenguaje de programación y una plataforma de arquitectura de comunicación OMA (Object Management Architecture).

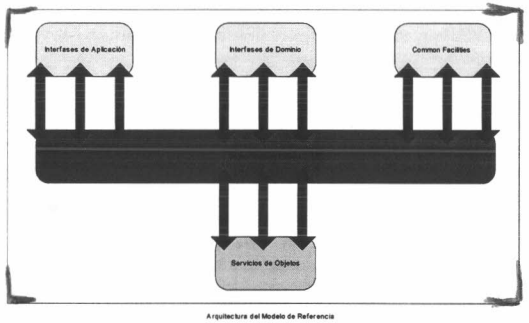
2.4. Arquitectura de Modelo de Referencia (OMA)

OMA (Object Management Architecture) es la visión de alto nivel de un ambiente distribuido completo. Está compuesto por cuatro componentes que pueden ser divididos en dos partes: componentes orientados a sistemas (ORB y Servicios de Objetos), y componentes orientados a aplicaciones (Objetos de Aplicación aplicaciones para un fin determinado denominados ***Common Facilities***).

Incluye también el ORB, del cual constituye la ***base de OMA*** y administra toda la comunicación entre los componentes. Permitiendo a los objetos interactuar en ambientes distribuidos heterogéneos, independientemente de la plataforma en el cual los objetos residen y que técnicas son usadas para implementarlos.

Esta tarea recae en ***Servicios de Objetos***, quién es el responsable para la administración de los objetos en general como crear objetos, control de acceso, mantener el camino de relocalización de objetos.

Common Facilities y aplicaciones de objetos junto con los componentes se acercan más a los usuarios finales, y sus funciones que invocan servicios de los componentes del sistema.



Arquitectura del Modelo de Referencia

✓ **Servicios de Objetos (Object Services)**

Son interfaces de dominios independientes, que son usados en muchos programas de objetos distribuidos. Por ejemplo, proveer un servicio para descubrir otros servicios disponibles, a los sumo todos los necesarios para el dominio de aplicación. Dos ejemplos de ***Object Services*** que cumplen este rol son:

Servicio de Nombre (Naming Service): permite a los clientes encontrar objetos basados en nombre.

Servicio de Tráfico (Trading Service): permite a los clientes encontrar objetos basados en sus propiedades.

Hay además especificaciones de Object Services para administración del ciclo de vida, seguridad, transacciones y notificación de eventos [OMG:95A], que se describe más adelante.

✓ **Common Facilities**

Como las interfaces de los Object Services, son horizontales, pero orientadas a aplicaciones de usuarios finales.
Un ejemplo puede ser un administrador de documentos distribuidos, permitiendo la presentación e intercambio de objetos basados en un modelo de documento, facilitando el encadenamiento de objetos de planilla de cálculo en un documento de reporte.

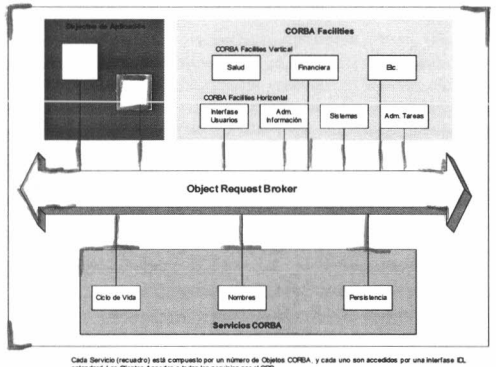
✓ **Interface de Dominios (Domain Interface)**

Estas interfaces completan los roles de Object Services y Common Facilities, pero son orientadas a un dominio de aplicación especifico. Por ejemplo, uno de

los primeros OMG RFPS para Domain Interfaces fue un administrador de datos de productos para el domino de la manufactura. Otros OMG RFPS pronto se aplicarán a los dominios de telecomunicaciones, medicina (CorbaMed) y finanzas.

✓ **Interfase de Aplicación (Aplication Interfaces)**

Estas interfaces son desarrolladas para una aplicación dada. Porque hay aplicaciones específicas y OMG no desarrolla aplicaciones (solo especificaciones), estas interfaces no están estandarizadas. Sin embargo, en un corto tiempo los servicios útiles de alguna aplicación de un dominio particular pueda ser candidata de ser una estandarización de la OMG(***Corbamed***).



Cada Servicio (recuadro) está compuesto por un número de Objetos CORBA, y cada uno son accedidos por una interfase IDL estandard. Los Clientes Acceden a todos los servicios por el ORB.

OMA resuelve la visión de OMG en un ambiente de componentes de software. de Esta arquitectura muestra como la estandarización de las interfaces los componentes pueden ingresar en la aplicación de objetos con el fin de crear ambientes de componentes de software ***plug and play*** basados en tecnologías de objetos.

Las aplicaciones de objetos, aunque no están estandarizadas por OMG, accederán a ***CORBAServices*** y ***CORBAFacilities*** a través de una interfaz estándar que provee beneficios tanto a :

- ✓ **Proveedores**
- ✓ **Usuarios Finales**

OMA especifica un conjunto de interfaces estándar y funciones para cada uno de los componentes. Diferentes implementaciones de interfaces de vendedores con sus funcionalidades luego pueden conectarse (***plug and play***) en las redes de computadoras de los clientes, permitiendo la integración de funcionalidad adicional adquiriendo módulos o desarrollos particulares.

OMA está dividido en dos grandes componentes: Nivel bajo de CORBAServices y un nivel intermedio CORBAFacilities.

- ✓ ***CORBAServicies***

  Provee una básica funcionalidad que casi cualquier objeto necesitaría: servicios de ciclo de vida de los objetos como mover, copiar, nombrarse, servicios de directorio y otros servicios básicos. Básico no significa ***simple***, sin embargo se incluye en esta categoría los accesos orientados a objetos a líneas de procesos transaccionales (***OLTP***) y objetos sofisticados de servicios de negocios.

- ✓ ***CORBAFacilities***

  Proveen servicios para aplicaciones. Por ejemplo, un administrador de documento combinados da una vía estándar de acceder a cada componente de un documento combinado. Con esta concepto, un vendedor puede fácilmente generar un conjunto de sofisticadas herramientas para manipular parte del documento.

  La arquitectura de CORBAFacilities tiene dos grandes componentes:

  - ✓ **Horizontal**

    Incluye aquellas aplicaciones como el servicio de documentos combinados, que puede ser virtual para cualquier negocio (Tareas de administración, Sistemas, etc.)

  - ✓ **Vertical**

    Administración estandarizada de información especializada a un grupo particular de industria (Salud, Financiera, etc.).

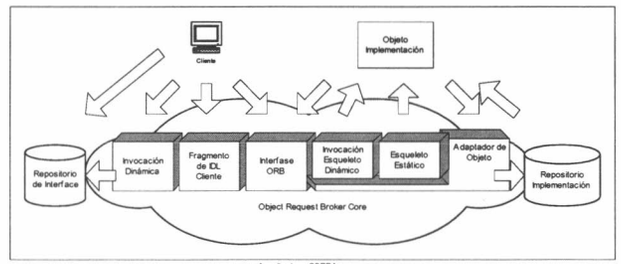
Los servicios están en el medio de la eficiencia provistos por ***Corba***: *Uno escribe su código de objeto*, ***Corba*** se encarga de cómo se identifica el objeto a sí mismo, busca otros objetos, aprende sobre los eventos de la red, maneja las transacciones de objetos a objeto y mantiene la seguridad.

2.5. Como Trabaja CORBA

ORB es un middleware que establece una relación Cliente-Servidor entre objetos. Usando ORB, el objeto cliente puede invocar un método a un objeto servidor que puede estar en la misma máquina o a través de la red.

ORB intercepta la llamada y encuentra el objeto que implementa el pedido , le pasa los parámetros, invoca el método y devuelve un resultado [NET96]. ***Corba***, como el *SQL*, provee tanto como interfaces estáticas y dinámicas para los servicios.

ORB garantiza la portabilidad e interoperabilidad de objetos a través de la red para sistemas heterogéneos. Más adelante se describirá en detalle sus componentes e interrelaciones.



Arquitectura CORBA

ORB y la arquitectura ***Corba*** provee un mecanismo para que los objetos CORBA se comuniquen. Estos objetos son pequeños componentes de software que proveen algún tipo de servicio, como acceder a la base de datos, administrar una cuenta o control de stock.

Para cualquier Cliente o Servidor que sea parte del esquema de ***CORBA***, debe incluir en el ***ORB*** la ayuda para encontrarlo y comunicarse con otros objetos ***CORBA***.

Dentro del ***ORB***, el Cliente o Servidor puede usar los servicios de cualquier objeto ***CORBA*** en cualquier Servidor o Host en la red.

Corba define el protocolo ***IIOP*** (Internet Inter- ORB Protocol), que gobierna como los objetos se comunican a través de la red. Para los que están familiarizados con el modelo OSI de protocolos de red, ***IIOP*** corre por arriba de ***TCP/IP*** sobre la capa de aplicación

2.6. Ventajas de CORBA

A continuación se detalla las ventajas desde dos puntos de vistas: Desarrolladores (personas que diseñan y producen sus aplicaciones) y los usuarios (conjuntos de requerimientos).

2.6.1. *Beneficio para Desarrolladores*

- ✓ CORBA es un ambiente que permite tomar ventajas de todas las herramientas que se han obtenido, desde el hardware hasta el desarrollo de software. Hay razones para todas las diversidades que existe en el mercado con diferentes

herramientas son usadas para diferentes trabajos y no es práctico limitar las chances cuando se enfrenta a los límites de un presupuesto, expectativas altas y una fuerte competencia sobre ruedas, se necesita una arquitectura que pueda correr en todas las plataformas de hardware y de red, y una arquitectura de interoperabilidad que una cada lenguaje de programación desde el C pasando por el Smalltalk, incluyendo herramientas de desarrollo interactuando con productividad y construcción.

- ✓ El paradigma de orientación a objetos envuelve ***Mejor Práctica*** de software desde el principio del ciclo de vida hasta el final: análisis orientado a objetos y diseño en etapas tempranas, implementando en lenguajes orientados a objetos y base de datos orientadas a objetos usando interfaces orientadas a objetos desplegado en un ambiente distribuido de objetos.

- ✓ De una interfase estándar, una capa fina de código envuelto (wrapper), y una aplicación ***LEGACY*** que permite integrarlo en un ambiente ***CORBA*** es equivalente con un nuevo componente de software. El objetivo es mantener el negocio en buen camino en un ambiente distribuida y conviviendo entre aplicaciones orientadas a objetos y convencionales, esto es esencial para permitir la integración en un solo sistema en las empresas.

- ✓ El ambiente ***CORBA*** maximiza la productividad del programador: ***CORBA*** provee una base sofisticada, con distribución transparente y fácil acceso a los componentes. ***CORBAServices*** provee la necesidad de una base orientada a objetos, mientras que ***CORBAFacilities*** estandariza la organización de la información compartida. Los desarrolladores crean o ensamblan objetos de aplicaciones en sus ambientes, tomando las ventajas de todos los componentes. Este ambiente estándar permite la interoperabilidad para ***Clientes*** de una plataforma para invocar operaciones estándar en los objetos en cualquier otra plataforma.

- ✓ La reusabilidad de código viene de dos maneras: primero, componentes son reusados como nuevos o reconfigurados dinámicamente en la aplicación; y segundo, los programadores pueden construir nuevos objetos [FOO88] haciendo modificaciones increméntales de objetos existentes sin tener que decodificar las partes que trabajan actualmente. Desde que se construye en lo que se tiene, esto permite tener librerías de componentes y código acumulado. La experiencia muestra que las compañías que codifican para reuso, ahorran entre 50 y 80 % el tiempo de desarrollo en diferentes proyectos.

- ✓ Se pueden utilizar diferentes herramientas en el mismo proyecto, desarrollar un componente desktop usando un constructor interactivo, mientras escribimos en el módulo de ***Servidor*** en un lenguaje de bajo nivel como C++. CORBA permitirá a los dos ínter operar transparentemente.

- ✓ CORBA permite reducir los costos de desarrollo, porque cada objeto es implementado y testeado, y puede ser usado una y otra vez. Esto ocurre por ser diseñado como un componente.

- ✓ CORBA provee características de seguridad como encriptamiento, autentificación y autorización para proteger los datos y controlar el acceso a los usuarios a los objetos y sus servicios.

- ✓ Todos los desarrolladores clientes necesitan conocer la definición de la interfase estándar y la descripción de lo que hace el objeto.

- ✓ CORBA provee interacción entre los objetos a través de sus interfaces. Porque la interfase y la implementación son separadas, los desarrolladores pueden modificar los objetos sin corromper las otras partes de la aplicación. Cambiando la implementación de un objeto no afectará a otros objetos o aplicaciones, porque la interfase del objeto no se modifica (concepto de la programación orientada a objeto).

2.6.2. *Usuarios*

Se necesita resolver el problema de toda la integración con el fin de sobrevivir, pero además está la necesidad de maximizar los recursos para abarcar el límite tecnológico para competir. La manera de realizarlo es: ***utilizar estándar industriales***, obteniendo una vía rápida y barata.

Para los usuarios, una aplicación ***CORBA*** es una colección dinámica de un ***Cliente*** componente y un ***Servidor*** componente de Implementación de objetos, configurado y conectado en tiempo de ejecución para atacar el problema entre manos. Esto puede incluir e integrar:

- ✓ Componentes localizados en diferentes departamentos o divisiones.
- ✓ componentes localizados dentro y fuera de la empresa, incluyendo sitios de clientes, proveedores y servicios de proveedores (cadena de producción).
- ✓ Componentes de múltiples vendedores de software.
- ✓ Componentes de vendedores externos y fábrica de software.
- ✓ Componentes embebidos con otros elementos o todo trabajando junto de una manera integrada.

Hay varias razones, en el cual deberíamos integrar en diversas plataformas, unas buenas y otras malas. Por ejemplo, en la mayoría de las compañías, en las oficinas corren PC o arquitecturas diferentes, otros corren en otros sistemas operativos, el problemas de integración envuelve a dos o más sistemas operativos y más de una red. Hay que integrar plataformas y sistemas diversos, porque necesitamos que todo corra en el negocio como una unidad única.

Esta complejidad tiene que ser resuelta con el objetivo de sobrevivir frente a los competidores y se debe responder de una manera rápida y efectiva.

CORBA protege la inversión realizada en los sistemas existentes. Se puede encapsular aplicaciones LEGACY , módulos o puntos de entrada en un ***Wrapper*** mediante una interfase estándar, a aplicaciones existentes. El objeto Wrapper permite al código LEGACY interoperabilidad con otros objetos en un ambiente distribuido de computadoras.

2.7. *Anatomía de CORBA*

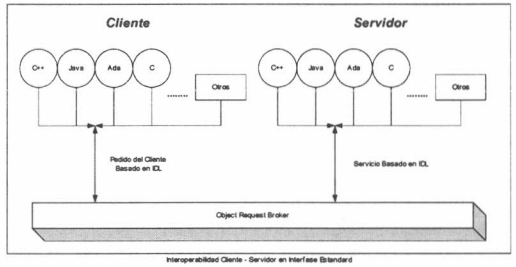
Como se ha visto en los capítulos el ***Orb*** es un middleware que establece las relaciones ***Cliente – Servidor*** entre los objetos. Usando el ***Orb*** , el objeto Cliente puede invocar transparentemente a un método sobre el objeto Servidor, que puede estar en la misma máquina o a través de la red.

Para lograr esta convivencia, a continuación se describe cada componente de la arquitectura, y las inter-relaciones entre cada uno de ellos, con el fin de permitir la integración entre las aplicaciones u objetos, haciendo de esta una único sistema corporativo.

2.7.1. *Objeto CORBA*

Un objeto ***CORBA*** se define, escribiendo u compilando en una especificación en un lenguaje de definición de interfase (IDL-Interface Definition Language). El ***IDL*** provee un lenguaje neutral para describir un objeto ***CORBA*** y los servicios que provee. ***IDL*** también permite a los componentes escritos en diferentes lenguajes comunicarse entre cada uno de ellos a través de la arquitectura ***CORBA***.

Un objeto ***CORBA*** puede residir en tipos diferentes de sistemas, incluyendo WINDOWS o servidores UNIX, IBM 3090 o mainframes DEC VAX. Pueden también estar escritos en diferentes lenguajes. Tanto que las interfaces de los servicios son escritas en IDL, los objetos pueden comunicarse y usar cada uno de los servicios a través de ***ORB*** en Clientes, Servidores, [HAR97] sistemas de base de datos, mainframes y otros sistemas en la red.



Interoperabilidad Cliente - Servidor en Interfase Estandard

Un objeto ***CORBA*** reparte la inteligencia permitiéndole vivir en cualquier lugar de la red. Se empaqueta como componentes binarios que los clientes remotos (pueden ser a través de la red o en la misma máquina) pueden acceder vía la invocación de métodos. Tanto el lenguaje como el compilador usados para crear los objetos ***Servidores*** son totalmente transparentes a los Clientes. El ***Cliente*** no necesita conocer como reside el objeto distribuido o sobre que sistema operativo se esta ejecutando. Puede ser en el mismo proceso o sobre la misma máquina que se sitúa a través de toda la red.

El ***Cliente*** no necesita conocer como esta implementado el objeto ***Servidor***. Por ejemplo, un objeto ***Servidor*** puede ser implementado como un conjunto de clases en C++ o puede ser implementado con millones de líneas de código COBOL; el objeto ***Cliente*** no conoce la diferencia (**Transparencia en la implementación**). Lo único que tiene que conocer el objeto ***Cliente*** es la interfase que publica el objeto ***Servidor***. Esta interfase establece el contrato entre el Servidor y el Cliente.

En la definición de interfase se especifica las operaciones que el objeto está preparado a responder, cada uno de los parámetros de entrada y salida requeridos y cualquier excepción que se pueda generar en esta vía.

Esta interfaz constituye un contrato con los clientes del objeto, quién usa la misma definición de interfase para construir y despachar invocaciones, como los objetos de implementación usa para responder y recibir. Este diseño provee una gran flexibilidad y muchos beneficios, como forzar el encapsulamiento y permite a los clientes acceder a las implementaciones de objetos independientemente de cada uno de los lenguajes de programación, sistemas operativos, plataformas de hardware, representación de datos, localización en la red, protocolos nativos y otros factores.

Como se enunció en el **capítulo 1.6**, un objeto de negocio debe ser flexible y tener bien definida su interfase con el fin de poder ser implementado independientemente.

Además debe tener la capacidad de reconocer eventos en su ambiente, modificar sus atributos e interactuar con otros objetos de negocios. Como un objeto ***CORBA***,

un objeto [ORF97] de negocio expone su interfase a sus clientes mediante ***IDL*** y se comunica con otros objetos utilizando el ***ORB.***

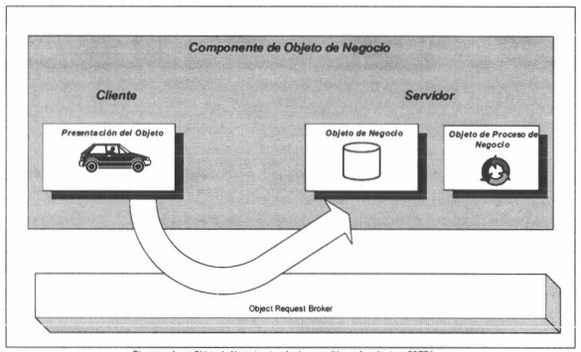
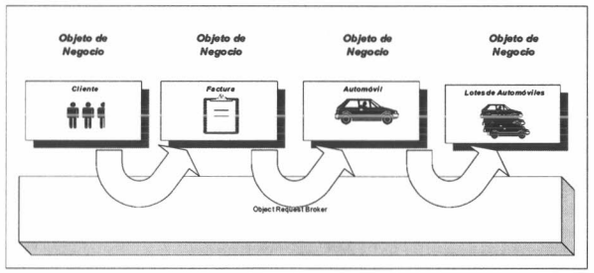


Diagrama de un Objeto de Negocio a través de un pedido en Arquitectura CORBA

Un objeto de negocio puede tener diferentes presentaciones a través de diferentes clientes. Tanto el objeto de negocio como su proceso puede residir en uno o más servidores. La ventaja de la arquitectura ***CORBA*** que todos los objetos que la componen tienen una interfase ***IDL*** definida y pueden correr a través de un ***ORB***. No interesa si el objeto corre en la misma máquina o en diferentes máquinas. Los Clientes también definidos como componentes, pueden ser aún factorizados en diferentes máquinas, y pueden utilizar el servicio de concurrencia y transacción para mantener la integridad del estado del objeto de negocio, y esto está dado por el ***ORB***.

En un sistema de objetos distribuidos, la unidad de trabajo y distribución es un ***componente***. La infraestructura de objetos distribuidos ***CORBA*** permite a los ***componentes*** fácilmente ser más autónomos, auto administrables y colaborativos.

La tecnología de objetos distribuidos ***CORBA*** permite poner juntos sistemas de información ***Cliente-Servidor*** complejos mediante simples componentes de extensión [HAR97] y conexión. Permite modificar un objeto sin afectar el resto de los componentes en el sistema o como interactúan. Una aplicación Cliente-Servidor se convierte en una colección de componentes que colaboran entre si.



Cooperación entre objetos de Negocio

2.7.2. *Lenguaje de Definición de Interfase (IDL)*

Para el cliente o usuario, ***OMG IDL*** representa un compromiso: cuando el cliente envía una invocación apropiada a un objeto a través de su interfase, el espera que la respuesta vuelva. Para el programador del objeto, la interfase representa una obligación: el debe implementar en algún lenguaje de programación, todas las operaciones especificadas en la interfase [SIE96].

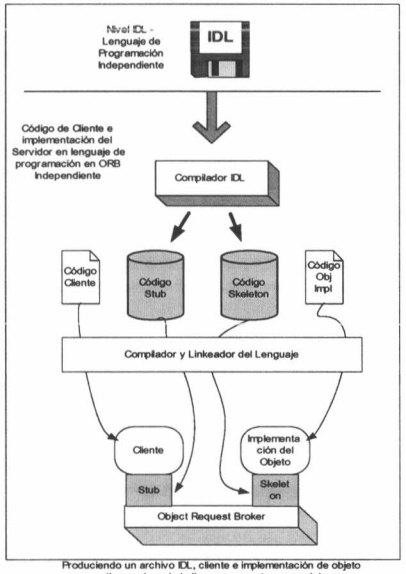
Escribir un contrato (en ***OMG IDL***) y completarlo son usualmente dos pasos separados en escribir un objeto ***CORBA***, aunque algunos vendedores de productos ***CORBA*** generen automáticamente ***OMG IDL*** de su código fuente o la información de diseño de aplicación.

Para la mayoría de los lenguajes de programación, mapenado un lenguaje estándar ***OMG***, especifica como las invocaciones de tipos y métodos ***OMG IDL*** convierten en lenguajes de tipos y funciones. Esto es como el esquema del ***OMG IDL*** y la implementación de los objetos vienen juntos: el compilador ***OMG IDL*** usa el mapa de especificaciones para generar el conjunto de llamadas de función desde las operaciones de ***OMG IDL***.

Los programadores probablemente asistidos por una herramienta automática o semi-automática, referida a un archivo de ***OMG IDL*** y usa un lenguaje de mapeo para generar el correspondiente conjunto de sentencias de funciones. Después de la compilación y el linking, esta resolución permite al esquema hacer las correctas llamadas para invocar las operaciones en su implementación de objetos.

ORBs comparte las definiciones de interfaces ***OMG IDL***, cuales son mantenidas en sus repositores de interfaces, para permitir formatos de datos a ser traducidos cuando los requerimientos y respuestas crucen los limites del sistema.

El cliente y la implementación de objetos están separados del ORB mediante una interfase ***OMG IDL***. ***CORBA*** requiere que todos los objetos de interfase estén expresados en una ***OMG IDL***.



Produciendo un archivo IDL, cliente e implementación de objeto
(Lo sombreado indica componente generado)

El cliente sólo puede ver las interfaces de objetos, nunca cualquier detalle de implementación. Esto garantiza sustituir la implementación detrás de la interfase (componente plug and play para un ambiente de software).

Aunque las interfaces ***IDL*** son una programación de lenguaje independiente, el lenguaje ***IDL*** en sí mismo tiene apariencia (pero no en la semántica) de ***ANSI C++*** en muchos aspectos.

En la interfaz ***IDL*** se especifica todas las operaciones que el objetos va a ejecutar, sus parámetros de entrada y de salida, y devuelve un valor, y toda excepción que se pueda generar.

2.7.2.1. Ejemplo de IDL

La manera más fácil de aprender los elementos básicos de un ***IDL*** es trabajando a través de un ejemplo. En esta sección vamos a focalizarnos en la gramática del ***IDL*** y sus capacidades.

En el ejemplo que se muestra a continuación está relacionado con un almacén con terminales de punto de ventas. Esta interfase del objeto de punto de venta usa para comunicarse con un objeto de lector de código de barra, objetos de teclado y un objeto de impresión de recibo:

```
// POS Ejemplo de Objeto IDL
module POS {
        typedef string Barcode;

        interface InputMedia {
                typedef string OperatorCmd;
                void barcode_input(in Barcode item);
                void keypad_input(in OperatorCmd cmd);
        };
        interface OutputMedia {
                boolean output_text(in string string_to_print);
        };
        interface POSTerminal {
                void end_of_sale();
                void print_POS_sales_summary();
        };
};
```

2.7.2.2. Módulos IDL, tipos y alcance

OMG IDL es un lenguaje fuertemente tipado de interfase, esto es cualquier tipo de variable debe ser declarada para un tipo en particular. Esto permite al ***ORB***

convertir variables desde un formato de una plataforma a otra, para transferir mensajes a través de redes heterogéneas.

IDL provee el tipo ***ANY*** un formato alternativo para la restricción de la imposición del tipamiento fuerte. Internamente, ***ORB*** asocia cualquier tipo con su valor, permitiendo a ellos proveer el mismo nivel de servicio para cualquiera, como hacen para todos los otros tipo ***IDL***. En la red, un conjunto estándar de código de tipos asegura que cualquier paso desde cualquier ***ORB*** a otros son interceptados correctamente.

Las definiciones de tipos, constantes, excepciones, interfaces y módulos son tomados en cuenta, esto es, solo toman efecto dentro de la sección donde fueron definidos, a menos que tengan un operador que les permite realizar una importación desde un alcance externo.

En el ejemplo anterior, la variable **Barcode** es alcanzado por el **módulo POS**. En este módulo, **Barcode** es usado en dos interfaces: **Interface InputMedia** y **Interface POSTerminal**. El alcance asegura que la definición es validada para ambos casos.

Los módulos no solamente definen alcances, también definen Interfaces, estructuras, uniones, operaciones y excepciones.

2.7.2.3. Definiendo una Interfaz

Una interfaz se construye en un archivo ***IDL***, que colecta un número de operaciones que forman un grupo natural. Dado que varios objetos pueden tener más de una interfase, usualmente no todas hacen a las operaciones de un objeto, pero por lo menos un subconjunto.

La palabra **keyword** en el ejemplo anterior , indica que comienza un nuevo **scope**, dentro se pueden definir a otros conjuntos de alcances y un conjunto de operaciones, obviamente esto es lo que define la interfase del objeto.

En el ejemplo muestra tres interfaces: **InputMedia** con dos operaciones, **OutputMedia** con una operación y **POSTerminal** con dos operaciones. **InputMedia** además define otro nuevo tipo, **OperatorCmd**.

2.7.2.4. Operaciones

El formato de una sentencia de una operación tiene tres partes que se requieren

- ✓ Nombre de la operación
- ✓ Valor de Retorno
- ✓ Lista de parámetros (de Entrada, de Salida, de Entrada / salida)

Si miramos la operación **output_text** en **interface OutputMedia**. La sentencia comienza por declarar el tipo de retorno. La operación devuelve un valor **Boolean**. El comando **void** permite no definir un valor de retorno.

El ***nombre de la operación*** , en el compilador ***IDL*** va a usar el nombre para construir un nombre para el lenguaje de mapeo.

La ***lista de parámetros***, permite definir para cada uno si es **IN**, **OUT**, **INOUT** (entrada, salida, entrada salida), el tipo del parámetro y el nombre del parámetro.

ORB usa estas declaraciones en las sentencias de las operaciones para manipular los datos como pedidos a través de la red.

2.7.2.5. Mapeo de IDL a un lenguaje de programación

Diferentes lenguajes de programación orientados a objetos o los no orientados a objetos prefieran acceder a los objetos ***CORBA*** de diferentes maneras.

Para los lenguajes orientados a objetos, es deseable ver a los objetos ***CORBA*** como objetos de lenguaje de programación. Aún para los lenguajes que no son orientados a objetos, es buena idea esconder la representación exacta del ***ORB*** de la referencia del objeto, nombre de métodos, etc.

Un particular mapping de una ***IDL*** debe ser la misma para todas las implementaciones del ***ORB***. El lenguaje de mapeo, incluye la definición de un tipo de datos de un lenguaje especifico y las interfaces del procedimiento para acceder a los objetos a través del ORB.

Un lenguaje de mapeo también define la interacción entre la invocación de objetos y el hilo de control en el cliente o en la implementación del objeto.

2.7.2.6. Herencia

Usando herencia, uno puede generar una nueva interface de una o más interfaces existentes. La interface derivada *hereda* todos los elementos de las interfaces en las que se basa y se puede agregar cualquier tipo de elementos (constantes, tipos, atributos, operaciones, etc.) que se necesita. Uno no puede redefinir las ***interfaces bases***. Esto significa que el Cliente escrito originalmente invocado garantiza poder invocar las interfaces derivadas, dado que todas las operaciones esperadas están incluidas.

La sintaxis de la herencia se realizan mediante los dos puntos (:):

```
Interface ejemplo1
        Long operacion1 (in long arg1);
};
```

```
Interface ejemplo2:ejemplo1 {
        void operacion2 (in long arg2, out arg3);
};
```

interface ejemplo2 además incluye, a través de la ***herencia***, **operacion1**. Aún cuando no aparece explícito en el ***IDL***, el lenguaje de mapeo va a generar el código para el cliente y el objeto de implementación para **ejemplo2** deber ser preparado para responder las invocaciones de la **operacion1** como también la de **operacion2**.

Otra característica importante de ***IDL*** es que permite múltiple herencia.

2.7.2.7. Resumen

OMG IDL define los tipos de objetos especificando su interface. Una interface consiste un conjunto de nombre de operaciones y sus parámetros. Aunque ***IDL*** provee un framework conceptual para describir objetos a manipular por el ***ORB***, no es necesario en este punto que el código fuente de ***IDL*** esté disponible por el ***ORB*** para trabajar. Tan pronto como la información equivalente esté disponible en la forma de una rutina o en un repositorio de interface para ejecutarse, un ***ORB*** puede disponer de la función correctamente (se va a profundizar en la siguiente sección).

IDL describe consistentemente las interfaces de los objetos en una forma común. ***IDL*** no es otro nuevo lenguaje de programación. Es un lenguaje para expresar tipos, específicamente tipos de interfaces. No tiene flujo de control o iteradores.

CORBA IDL es un código fuente para un traductor ***IDL*** (típicamente llamado compilador). Este traductor (lenguaje de interface) toma las sentencias ***IDL*** como entrada y luego lo convierte (mapping) al lenguaje destino. Esto es como ***CORBA*** e ***IDL*** permite tener componentes plug and play en un ambiente ***Cliente – Servidor***.

IDL permite [FAQ97] separar la interface de la implementación y es un concepto importante en orientación a objetos en general y particularmente en ***CORBA***.

2.7.3. *ORB (Object Request Broker)*

El ***ORB*** es un bus de objetos. Permite a los objetos transparentemente realizar pedidos a - o recibir respuestas desde - otros objetos localizados localmente o remotamente. El Cliente no tiene que preocuparse como es el mecanismo utilizado para comunicarse, activarlo o almacenarlo en el objeto Servidor.

Un ***ORB*** provee una gran variedad de servicios middleware distribuidos. El ***ORB*** permite a los objetos descubrirse entre ellos en tiempo de ejecución (*Runtime*) e invocarse servicios entre ellos. Un ***ORB*** es una de las alternativas que provee middleware Cliente / servidor incluyendo el tradicional ***Remote Procedure Calls***

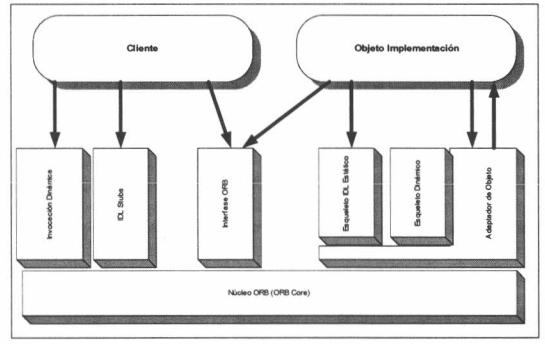
(RPC), ***Middleware Orientados a Mensajes*** (MOM - Message Oriented Middleware), procedimientos almacenados de base de datos, ***Distribuited Computing Environment*** (DCE) Y servicios puerto a puerto (peer to peer).

2.7.3.1. Estructura de un ORB

Como se enunció en punto anteriores un objeto ***CORBA*** realiza un pedido es desde el cliente a una implementación del objeto a través del ***ORB***. El cliente es una entidad que desea ejecutar una operación de un objeto (implementación) que contiene el código y los datos que actualmente lo implementa.

El ***ORB*** es el responsable de todos los mecanismos requeridos para encontrar la implementación del objeto para el pedido, para preparar a la implementación del objeto a recibir el pedido y para comunicar el dato que confecciona el pedido. La vista de la interface del cliente es completamente independiente de dónde el objeto está localizado, que lenguaje de programación está implementado o cualquier otro aspecto que no se vea reflejado en la interface del objeto.

A continuación se describe la estructura individual de un ***ORB***:



Estructura de un ORB individual

Para realizar el pedido, el cliente puede usar la interface de invocación dinámica (Dynamic Invocation Interface-DII), que es la misma interface independiente de la interface del objeto destino, o el repositorio (stub) ***OMG IDL*** (Static Invocation Interface-SII), el stub especifico depende de la interface del objeto destino. El cliente puede directamente interactuar con el ***ORB*** para algunos servicios.

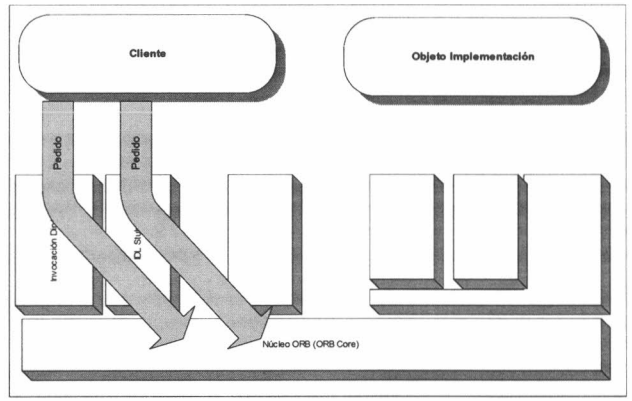
El objeto implementación recibe el pedido como una llamada (desde el ***ORB***) a través del esquema del ***OMG IDL*** generada (Esqueleto IDL) o a través del *esqueleto dinámico*. El objeto implementación puede llamar al *objeto adaptador* y al ORB mientras procesa un pedido u otros servicios.

La *interfaz ORB* provee servicios para ambos (el cliente y el objeto implementación), accediendo al repositorio de interface e implementación, y algunas operaciones de la referencia del objeto que solamente pueda ejecutar el ***ORB***.

Las definiciones de las interfases de los objetos pueden ser definidas de dos maneras. Las interfaces pueden ser definidas estáticamente en ***IDL***. Este lenguaje define los tipos de objetos acorde a las operaciones que puede ejecutar sobre ellos y los parámetros de esas operaciones. Alternativamente , o además, las interfaces pueden ser agregadas a un servicio de *repositorio de interfaces* (Interface Repository), estos servicios representan los componentes de una interface como objetos, permitiendo el acceso en tiempo de ejecución (runtime) a estos componentes. En cualquier implementación ***ORB***, el ***IDL*** y el repositorio de interface tiene igual poder expresivo.

El cliente ejecuta un pedido teniendo acceso a la ***Referencia del Objeto***, para un objeto y conociendo el tipo de objeto y la operación deseada para ser ejecutada. El cliente inicia un pedido llamando a rutinas del *stub* que son específicas del objeto o construyendo el pedido dinámicamente

La interfaz dinámica y el stub para invocar un pedido requiere la misma semántica del pedido, y el receptor del mensaje no puede decir cómo el pedido fue invocado.

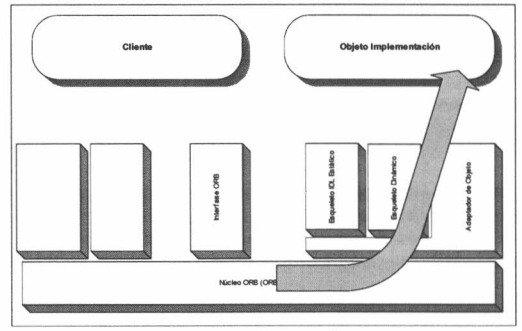


Cliente usando invocación Dinámica y Stub

Por otro lado, el ***ORB*** localiza el apropiado código de implementación, transmitiendo los parámetros y transfiriendo el control al objeto implementación a través del ***IDL*** o el ***esquema dinámico***.

Para ejecutar un pedido, el objeto implementación puede obtener algunos servicios desde el ***ORB*** a través del objeto adaptador. Cuando el pedido es completado, el control y los valores de salida son devueltos al cliente.

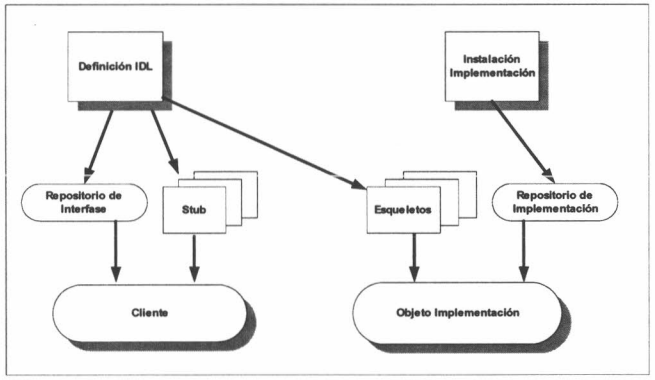
El objeto implementación puede elegir con cuál *objeto adaptador* va a usar. Esta decisión esta basada en que clase de servicios el objeto implementación requiera.



Objetp Implementación recibiendo un pedido

La interface y la información de la implementación está disponible a los clientes e implementaciones de objeto. La interfase está definida en ***OMG IDL*** y/o en ***Repositorio de Interface***; la definición es usada para generar los *stubs del cliente* y los esquemas de la implementación del objeto.

La información de la implementación del objeto es provista en tiempo de instalación y está almacenada en el ***repositorio de implementación*** para usar durante el reparto de pedidos.



Repositorios de Interfase e Implementación

En la arquitectura, el ***ORB*** no es requerido para ser implementado como un simple componente, pero está definido por sus interfaces. Cualquier implementación de ***ORB*** que provea apropiadas interfaces es aceptable. Las interfases están organizadas en tres categorías:

- ✓ Las operaciones son las mismas para todas las implementaciones de ***ORB***
- ✓ Las operaciones son específicas para un particular tipo de objeto.
- ✓ Las operaciones son específicas para un particular estilo de implementación de objetos.

Diferentes ***ORBs*** pueden hacer apenas diferentes elecciones de implementaciones, y, juntas con el compilador *IDL*, *repositorios*, y varios *adaptadores de objetos*, proveen un conjunto de servicios a los clientes y las referencias de las implementaciones de los objetos y diferentes significados para la ejecución de las invocaciones.

Puede ser posible para un cliente tener acceso simultáneamente a dos referencias de objetos manejados por diferentes implementaciones de ***ORB***. Cuando dos ***ORBs***

intentan trabajar juntos, esos ***ORBs*** deben poder distinguir sus referencias de objeto; ***no es responsabilidad del cliente hacer esto***.

El núcleo del ***ORB*** es una parte del ***ORB*** que provee la representación básica de los objetos y comunicación de los pedidos. ***CORBA*** está diseñado para soportar diferentes mecanismos de objetos, y esto hace estructurar al ***ORB*** con componentes arriba del núcleo ***ORB,*** cual provee interfaces que pueden enmascarar las diferencias entre los núcleos de ORB.

2.7.3.2. Cliente y el IDL Stub

Un cliente de un objeto tiene acceso a un objeto a través de la ***referencia del objeto***, e invoca operaciones sobre el objeto. Un cliente conoce solamente la estructura lógica de un objeto acorde a las interfaces y experiencias del comportamiento del objeto mediante sus invocaciones. Aunque generalmente se considera a los clientes como un programa o pedido que inicia un proceso en un objeto, es importante reconocer que puede ser un cliente relativo a un objeto en particular. Por ejemplo, la implementación de un objeto puede ser un cliente de otros objetos.

El cliente generalmente ve a los objetos y las interfaces de ***ORB*** a través de la perspectiva de un mapeo de lenguaje (*Language Mapping*), trayendo al ***ORB*** hacia arriba hasta el nivel de programación. El cliente es máximamente portable y debería poder trabajar sin cambiar de fuentes en cualquier ***ORB*** que soporte el mapeo de lenguaje deseado con cualquier instancia de objeto que implementa la interface deseada. Los clientes no tiene el conocimiento de la implementación del objeto, *el objeto adaptador* es usado por la implementación o el ***ORB*** es usado para accederlo.

El rol de cliente es simplemente requerir servicios invocando operaciones. Hay operaciones no estándar ***CORBA*** para la administración del objeto implementación; activación del objeto, desactivación, suspensión y otras operaciones más son ejecutadas automáticamente por el ***ORB*** y por los servicios customizados localizados fuera del cliente.

El código del cliente trata exclusivamente con el problema entre manos, resultando una máxima portabilidad e interoperabilidad.

El cliente accede al objeto implementación mediante su interfaz ***IDL***, especificando la instancia del objeto destino vía su ***referencia de objeto***. La ***referencia del objeto*** aísla al cliente de la localización del objeto.

El ***stub*** une al cliente de un lado y al ***ORB*** del otro lado. El **stub** es generado por el compilador ***IDL*** y no es escrito por el programador, y no es necesario porque los ***stubs*** son intercambiables. (IDL es intercambiable; uno puede correr a través del compilador IDL para generar el stub que uno quiere)

La interfaz *Cliente-Stub* es definida por el estándar del lenguaje mapeador OMG para el lenguaje de programación que uno elige. Esto quiere decir que el código fuente se puede portar de un vendedor ORB a otro para el mismo lenguaje, esta portabilidad generada por un archivo ***IDL*** está definido por el estándar que por un vendedor particular.

2.7.3.3. Implementación de objeto

Una implementación de un objeto provee la semántica de un objeto, usualmente define los datos para la instancia del objeto y el código para los métodos del objeto. Generalmente la implementación va a usar otros objetos o software adicional para implementar el comportamiento del objeto. En algunos casos, la función primaria del objeto es tener efectos colaterales en otras cosas que no son objetos.

Una variedad de implementaciones de objetos pueden ser soportados, incluyendo servidores separados, librerías, un programa por método, una aplicación encapsulada, una base de datos orientada a objetos, etc. A través del uso de adaptadores de objetos adicionales, es posible soportar virtualmente cualquier estilo de implementación de objeto.

Generalmente, las implementaciones de objetos no dependen de un ***ORB*** o como los clientes invocan al objeto. Las implementaciones pueden seleccionar interfaces a servicios dependientes del ***ORB***, eligiendo el objeto adaptador.

2.7.3.4. Referencia de Objeto

La referencia de objeto es una información necesitada para especificar a un objeto dentro de un ***ORB***. Tanto clientes e implementaciones de objetos tienen una noción opaca de la referencia de un objeto con respecto a un lenguaje de mapeo, y esto lo separa de la actual representación de cada uno de ellos. Dos implementaciones de ***ORB***, pueden diferir en la elección de la representación de la ***referencia del objeto***.

La representación de una referencia de objetos manejada por un cliente es solamente válida para el tiempo de vida de ese cliente.

Todos los ***ORBs*** debe proveer el mismo lenguaje de mapeo para una referencia de objeto (usualmente referida a un objeto) para un particular lenguaje de programación. Esto permite a un programa escrito en un lenguaje particular acceder a referencias de objetos independientemente de un ***ORB*** en particular. El lenguaje de mapeo puede además proveer una manera adicional de acceder a referencias de objetos en un tipo determinado por conveniencia del programador.

Hay que distinguir la referencia del objeto, garantizando ser diferentes a todas las otras referencias, que no denota ningún objeto.

2.7.3.5. Interfaz de Invocación Dinámica

Una interfaz esta también disponible para permitir una construcción dinámica de una invocación de objeto, esto es, más que llamar a una rutina de stub que especifica una operación particular para un determinado objeto, un cliente puede especificar un objeto para ser invocado, una operación a ser ejecutada, y un conjunto de parámetros de la operación a través de una llamada o una secuencia de llamadas. El código del cliente debe suministrar la información acerca de la operación a ser llamada y los tipos de parámetros que están siendo pasados (tal vez obteniéndolos de un repositorio de interface o de otra fuente en tiempo de ejecución). La naturaleza del ***DII*** puede variar substancialmente de un lenguaje de programación de mapeo a otro.

2.7.3.6. Interfase Estática

Con el objetivo de invocar una operación sobre un objeto, el cliente tiene que realizar una llamada, y está estáticamente unida con su correspondiente stub. El desarrollador determina que stub del cliente contiene el código. Esta interface no puede acceder a nuevo tipos de objetos que se agregaron luego en el sistema.

Hay un número de importante diferencias entre la interface dinámica y la estática; una que se destaca es la interface dinámica pospone la selección del tipo de objeto y la operación hasta el tiempo de ejecución, mientras que la interfase estática requiere que la selección se realice en tiempo de compilación.

Formalmente, se puede decir que la invocación de la interface dinámica permite tipamiento dinámico, mientras que la invocación estática requiere tipamiento estático. (Ambos permiten ***binding dinámico***, esto es, no se tiene que tener seleccionado la instancia del objeto destino hasta el tiempo de ejecución).

Hay otras diferencias: como consecuencia del binding dinámico, el ***DII*** no puede chequear los tipos de argumentos correctamente en tiempo de compilación, mientras que ***SII*** si puede. Estructuralmente, el ***ORB*** requiere un stub separado para cada interface estática (generado por el compilador ***IDL***), pero solamente una interface ***DII*** es provista por el ***ORB***. Mientras que ***SII*** son generalmente sincrónicas (se bloquean, a menos que la operación sea declarado para un lado solo en la definición del ***IDL***), ***DII*** puede invocar tanto en sincrónico, asincrónico o modos diferidos de sincronismo.

2.7.3.7. Implementación del esquema

Para un particular lenguaje de mapeo, y posiblemente dependiendo del objeto adaptador, habrá una interface para los métodos que implementan cada uno de los tipos de objetos. La interface generalmente puede ser una interface hacia el cliente o implementación del objeto, en el cual la implementación del objeto escribe rutinas que conforman la interface y el ***ORB*** los llama a través del esquema.

La existencia de un esquema no implica la existencia de un stub cliente correspondiente (los clientes pueden además hacer pedidos vía DII).

Es posible escribir un objeto adaptador que no usa esquemas para invocar métodos de implementación. Por ejemplo, puede ser posible crear implementaciones dinámicamente para lenguajes como el Smalltalk.

2.7.3.8. Esquema de Interfaz Dinámica

Un interfaz está disponible cuando permite un manejo dinámico de una invocación de objeto. Esto es, más que siendo accedido a través de un esquema; especificando una operación en particular, la implementación del objeto es alcanzada a través de una interface que provee acceso al nombre de la operación y a los parámetros en una manera análoga que del lado del cliente en una ***DII***. Puramente conocimiento estático de aquellos parámetros pueden ser usados o un conocimiento dinámico (tal vez determinado a través del *Repositorio de Interface*-IR), para determinar los parámetros.

El código de implementación debe proveer una descripción de todos los parámetros de las operaciones hacia el ***ORB***, y el ***ORB*** provee los valores de cualquier parámetro de entrada para ser usado en la ejecución de una operación. El código de implementación provee valores de cualquier parámetro de salida, o una excepción, hacia el ORB después de efectuar la operación. La naturaleza del Esquema de Interface Dinámica puede variar sustancialmente desde un mapeo de un lenguaje de programación o un objeto adaptador a otro.

2.7.3.9. Objeto Adaptador

Un objeto adaptador es la primer manera que una implementación de objetos accede a los servicios provistos por el ***ORB***. Se espera que haya algunos objetos adaptadores que estén disponibles, con interfaces que son propiamente de una clase específica de objetos. Los servicios provistos por el ***ORB*** a través de un objeto adaptador generalmente incluye: generación e interpretación de referencias de objetos, métodos de invocación, seguridad en la interacción, objetos e implementación de activación y desactivación, mapeando a referencias de objetos a implementaciones, y registración de implantaciones.

Hay un gran rango de objetos de granularidad, ciclos de vida, políticas, estilos de implementación y otras propiedades que hacen dificultoso para el ***núcleo del ORB*** proveer en una simple interface que es conveniente y eficiente para todos los objetos. Quiere decir, que a través de ***objetos adaptadores***, es posible para el ***ORB*** hacia grupos particulares de implementación de objetos destino tener requerimientos similares con interfaces que lo toleren.

2.7.3.10. Interfaz del ORB

La interfaz del ***ORB*** es la interface que va directamente hacia el ***ORB***, que es la misma para todos los ***ORBs*** y no depende de los objetos de interface o de los objetos adaptadores. Porque la mayoría de las funcionalidades del ***ORB*** es proveer a través de *objetos adaptador, stubs, esquemas o skeleton o mediante una invocación dinámica*, algunas operaciones que son comunes a través de todos los

objetos. Estas operaciones son útiles tanto para los clientes y las implementaciones de los objetos.

2.7.3.11. Repositorio de Interfaces

El ***repositorio de interfaces*** (IR) es un servicio que provee las persistencia de objetos que representa la información del ***IDL*** en una forma disponible en tiempo de ejecución. La información del ***IR*** puede ser usada por el ***ORB*** para ejecutar pedidos.

Más aún, usando la información en el ***IR***, es posible para un programa contar con un objeto cuya interface no fue conocida cuando el programa fue compilado, aún, disponiendo de determinadas operaciones que son válidas en el objeto y realizando la invocación en él.

Agregando a este rol en el funcionamiento del ***ORB***, el ***IR*** es un lugar común para almacenamiento adicional de información asociadas a las interfaces para los objetos del ***ORB***. Por ejemplo, información de debugging , librerías de esquemas o stubs, rutinas que pueden ser formateadas o ver particulares clases de objetos, etc, pueden ser asociadas con el ***IR***.

2.7.3.12. Repositorio de Implementación

El ***repositorio de implementación*** (IIR) contiene información que permite al ***ORB*** localizar y activar las implantaciones de los objetos. Aunque la mayoría de la información del ***IIR*** está especificada en el ORB o en el ambiente de operación, el ***IIR*** es un lugar convencional para registrar ciertas informaciones.

Ordinariamente, la instalación de implementaciones y control de políticas relacionadas con la activación y ejecución de implementaciones de objetos está hecho a través de operaciones en el ***IIR***.

Agregando a este rol en el funcionamiento del ***ORB***, el IIR es un lugar común para almacenar información adicional asociadas a implementaciones de objetos de ORB. Por ejemplo, información de debugging, control administrativo, localización de recursos, seguridad, etc., debe ser asociado con el ***IIR***.

2.7.3.13. Conclusiones

En este capítulo se describió los componente que están involucrados en un ***ORB***. En el capítulo siguiente se describe como funciona tanto el Cliente como el objeto de implementación interactuando con el ***ORB***.

A continuación se describe una caracterización y ventajas que permite los ***ORB*** con respecto a entornos de ejecución:

- ✓ **Invocación de Métodos Estáticos y Dinámicos**

ORB permite definir invocación de métodos en tiempo de compilación o también permite realizarlo dinámicamente, descubriendo estas invocaciones en tiempo de ejecución. También la implementación de cada uno de los métodos puede ser fuertemente tipados (strong type) en tiempo de compilación o asociando una mayor flexibilidad mediante el late binding (débilmente tipado), resolviendo en tiempo de ejecución. La mayoría de los Middleware solo soporta tipificación estática.

✓ **Alcances en Lenguajes de Alto Nivel**

ORB permite invocar métodos en objetos Servidores usando cualquier lenguaje de alto nivel deseado. No importa en que lenguaje esta escrito el objeto *Servidor*. ***CORBA*** separa la implementación de la interface y provee un lenguaje neutral con tipos de datos que hace posible la llamadas o invocaciones de los objetos a través de los limites del lenguaje y sistema operativo. En contraste, otros tipos de Middleware típicamente proveen lenguajes específicos de bajo nivel, y librerías API (Aplication Program Interface).

✓ **Sistemas Auto descriptivos**

CORBA provee un meta data en tiempo de ejecución para describir todas las interfaces de los Servidores en el sistema. Todo ***ORB*** debe soportar el **Repositorio de Interfaces** (Interface Repository) que contiene en tiempo real la información describiendo las funciones que provee un *Servidor* y sus *parámetros*. Los clientes utilizan el meta data para descubrir como invocar los servicio en tiempo de ejecución. Además es una herramienta de ayuda para generar código dinámicamente (cuando se esta corriendo una aplicación). El meta data es generado tanto como el lenguaje ***IDL*** precompilado o por compiladores que conocen como generar directamente ***IDL*** desde un *lenguaje orientado a objetos*. Por ejemplo, el compilador **Metaware C++** genera directamente ***IDL*** desde la definición de las clases C++; **Visigenic/Netscape's Caffeine** genera ***IDL*** directamente desde **Java Bytecodes**. No hay otra forma Middleware Cliente-Servidor que provea este tipo de meta data en tiempo de ejecución y la definición de un lenguaje independiente para todos los que proporciona servicios.

✓ **Transparencia Local - Remota**

Un ***ORB*** puede correr en modo monousuario (stand-alone) en una notebook o puede estar interconectado con otros ***ORB*** (se verá más adelante) en un universo usando servicios CORBA 2.0's Internet Inter-ORB Protocol (***IIOP***). Un ***ORB*** puede proporcionar llamadas inter-objetos dentro de un simple proceso, múltiples procesos corriendo en la misma máquina o múltiples procesos corriendo a través de la red y sistemas operativos. Esto es completamente transparente para todos los objetos. En general, programadores de CORBA Cliente / servidor no tiene que preocuparse sobre el transporte, localización de Servidores, activación de objetos, ordenamiento de bytes a través de

plataformas distintas o sistemas operativos diferentes - ***CORBA*** permite que todo sea transparente.

✓ **Seguridad y Transacciones Incorporadas**

El ***ORB*** incluye un contexto de información en los mensajes que permite manejar la seguridad y la transacción a través de la máquina y los límites del ***ORB***.

✓ **Mensajes Polimórficos**

En contraste con otros Middleware, un ***ORB*** no es una simple invocación a una función remota, sino que invoca una función a un objeto determinado. Esto significa que la llamada a la misma función puede tener diferentes efectos o respuestas dependiendo del objeto que lo recibe (característica de la programación orientada a objetos). Por ejemplo, el método **configuración**, que al ser invocado puede tener distinto comportamiento cuando es aplicado a un objeto de una base de datos o a un objeto de impresión.

✓ **Coexistencia con Sistemas Existentes**

La separación de la definición e implementación de un objeto ***CORBA*** es perfectamente encapsulada por las aplicaciones existentes. Usando ***IDL***, se puede escribir códigos existentes que se parezcan a objetos en el ***ORB***, aún si la implementación son procedimientos almacenados, CICS, IMS o COBOL. Esto permite a ***CORBA*** una solución de evolución. Permite escribir nuevas aplicaciones como objetos puros y encapsularlos en aplicaciones existentes ocultándolo mediante la definición de ***IDL***.

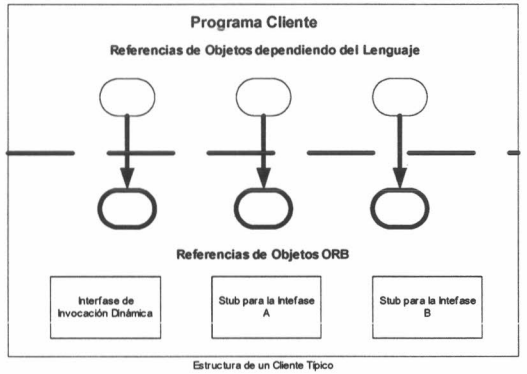
2.7.4. *Estructura del Objeto Cliente*

Un cliente es un objeto que tiene una referencia de objeto que se refiere a ese objeto. Una referencia de objeto es un ***TOKEN*** que puede ser invocado o pasado como parámetro a una invocación de un objeto diferente. La invocación de un objeto envuelve la especificación del objeto a ver invocado, la operación a ser ejecutada y los parámetros a ser dados para esa operación y ser devueltos a él.

El ***ORB*** administra la transferencia de control y datos hacia la implementación del objeto y vuelve al cliente. En un evento que el ***ORB*** no puede completar la invocación, una respuesta de excepción es devuelta. Ordinariamente, un cliente llama a una rutina en un programa que ejecuta una invocación y vuelve cuando la operación se haya completado.

El Cliente accede a un específico tipo de objeto como rutinas de librerías en su programa. El programa Cliente ve las rutinas que puede invocar en una forma normal en el lenguaje de programación. Todas las implementaciones proveerán el tipo de dato específico del lenguaje a ser usado para hacer ***referencia al objeto***, usualmente un puntero. El Cliente luego pasa la referencia del objeto a una rutina

del ***stub*** para iniciar una invocación. El ***stub*** tiene acceso a la representación de la referencia del objeto e interactúa con el **ORB** para ejecutar la invocación.



Estructura de un Cliente Típico

Una alternativa del conjunto de códigos de librerías está habilitado para ejecutar invocaciones sobre los objetos, por ejemplo cuando en el objeto no fue definido en tiempo de compilación. En este caso, el programa Cliente provee información adicional para nombrar el tipo de objeto y el método a ser invocado, y ejecutar la secuencia de llamadas para especificar los parámetros e iniciar la invocación.

Los *Clientes* comúnmente obtienen las referencias de los objetos recibiendo de ellos parámetros de salida desde las invocaciones de otros objetos por el cual tienen referencias. Cuando un *Cliente* es también una implementación, recibe las referencias de objetos como parámetros de entrada sobre las invocaciones para los objetos que implementa. Una referencia de objeto puede ser convertida a una cadena para ser almacenada en archivos o preservarla o comunicada por diferentes significados y cambiarla dentro de la referencia del objeto por el ***ORB*** que produce la cadena.

2.7.4.1. Funcionamiento de la Interfaz de Invocación Dinámica

DII permite en unas líneas de código para iniciar cada invocación de objetos una libertad para conseguir el objeto destino, interface y la operación en tiempo de ejecución.
DII da al Cliente la capacidad en cualquier momento, de invocar cualquier operación sobre cualquier objeto que puede acceder a través de la red. Esto incluye objetos por el cual el Cliente no tiene ***stub***.

El ***ORB*** es el responsable de preparar el pedido dinámico que tiene la misma forma que el pedido estático, antes de transmitir el pedido al objeto implementación.

Hay cuatro pasos para la ***invocación dinámica*** [SIE96]:

- ✓ Identificar el objeto que uno quiere invocar
- ✓ Capturar la interfaz destino
- ✓ Construir la invocación
- ✓ Invocar un pedido DII

2.7.4.2. Funcionamiento del Repositorio de Interfaz

El ***repositorio de interfase*** es crucial para la operación de ***CORBA***. El ***IR*** [WHI96] provee:

- ✓ Interoperabilidad entre diferentes implementaciones de ***ORB***
- ✓ Chequeos de tipos de requerimientos, si el pedido fue en forma estático o dinámica
- ✓ Chequear col rectitud en el grafo de herencia
- ✓ Usar utilitarios provistos por diferentes vendedores de ***ORB***
- ✓ Compartir el ***IR*** por más de un ***ORB***

El ***IR*** se puede usar para:

- ✓ Administrar la instalación y distribución de las definiciones de la red
- ✓ Durante el proceso de desarrollo, se puede modificar las definiciones de las interfaces u otra información almacenada en el **IDL**

La implementación de un ***IR*** requiere alguna forma de almacenamiento de persistencia de objetos. Par aun ***ORB*** que sirven a un número de diferentes interfaces, a pesar de si la fuente es local o remota, esto puede crecer para ser un componente sustancial. Afortunadamente para los implementadotes de ***ORB***, la especificación de OMG permite a los vendedores la libertad para implementar el ***IR***, en la mejor forma con respecto a la plataforma destino y el sistema operativo, mientras que el ***IDL*** permite la interoperabilidad y portabilidad.

Cuando se evalúa un ***ORB*** , se cheque como administra la información del ***IR***.

2.7.4.3. Interfaz del ORB

El objeto interfase visto en el ***IDL***, es parte de la interface del ***ORB*** . Lo parte más importante de la interface del ***ORB*** es la inicialización del componente. Hay una parte para el ***Cliente*** y otro la ***Implementación del Objeto***.

Cuando el Cliente inicializa, necesita las referencias de objetos para su ***ORB***, ***Repositorio de Interface***. La OMG sintió que esta inicialización común debería ser estandarizada, desde cada Cliente pueda invocarla y que el formato deba permitir al Cliente especificar cual ***ORB***, ***Repositorio de Interface*** quiere.

La interfaz del ***ORB*** provee el acceso a los repositorios de Interface e Implementación, y a algunas operaciones de los objetos de referencia que solamente el ***ORB*** puede ejecutar.

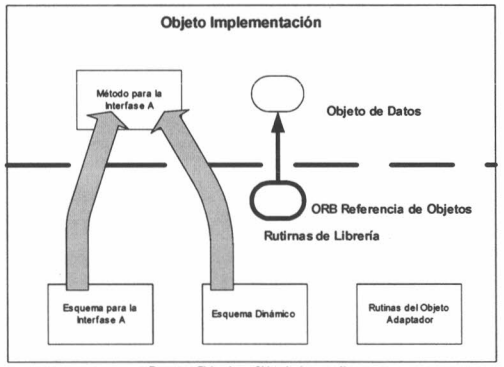
2.7.5. *Estructura del Objeto Implementación*

Un ***objeto implementación*** provee el actual estado y el comportamiento de un objeto. El ***objeto implementación*** puede ser estructurado en una variedades de formas. A pesar de definir los métodos para las operaciones, una implementación usualmente definirá procedimientos para activar o desactivar objetos y usará otros objetos o aplicaciones con un propósito en particular que no son objetos con el objetivo de hacer el estado del ***objeto persistente***, para el control de acceso del objeto como también la implementación del método.

El ***objeto implementación*** interactúa con el ***ORB*** en una variedad de formas para establecer su identidad, para crear nuevos objetos y obtener los servicios dependientes del ***ORB***.

Primeramente lo realiza accediendo vía un ***objeto adaptador***, cual provee la interface para el servicio del ***ORB*** que es conveniente para el estilo particular del objeto implementación.

Porque el rango de posibles objetos implementación, es dificultoso ser definitivo acerca del conocimiento general de la estructura del objeto implementación.



Estructura Típica de un Objeto Implementación

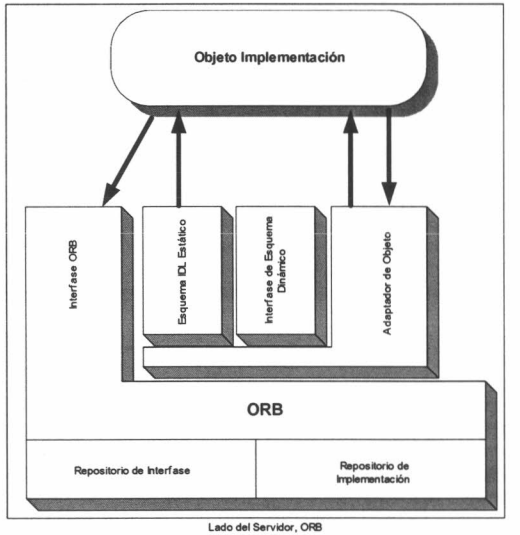
Cuando la invocación ocurre, el núcleo del ***ORB*** , el objeto adaptador y el esquema arreglan la llamada para que sea hecha al método apropiado de la implementación.

Un parámetro del método especifica el objeto siendo invocado, que el método puede usar para localizar el dato para el objeto. Parámetros adicionales son provistos acorde a la definición de los esquemas. Cuando el método está completo, lo devuelve, y ocasiona que los parámetros de salida o excepciones va a ser trasmitidas de vuelta al ***Cliente***.

Cuando un nuevo objeto es creado, el ***ORB*** puede ser notificado con el objetivo que conozca dónde encontrar la implementación para el objeto. Usualmente, la implementación además registra como los objetos se implementan para una interface en particular, y especifica como comienzan la implementación si no está todavía corriendo.

2.7.5.1. Componentes del Servidor

En la siguiente figura, se puede visualizar los componentes del objeto implementación, el cual trabaja con el ***ORB*** y el ***adaptador de objeto*** para proveer las funciones de administración.



Lado del Servidor, ORB

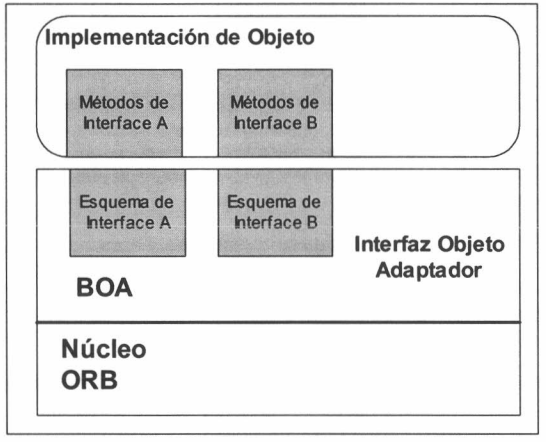
El ***adaptador de objeto*** provee la interfaz entre el ***ORB*** y el objeto, permitiendo al ***ORB*** preparar al objeto para recibir el pedido, y permitiendo también a los objetos

notificar al ***ORB*** que ellos están listos (o no están listos) para procesar el pedido. El componente del adaptador del objeto entre el ***ORB*** y los objetos está en una posición para igualar los servicios que proveen varios ***ORBs***, permitiendo al mismo conjunto de objetos funcionar con ***ORBs*** de diferentes niveles de servicios.

El esquema ***estático IDL*** es el lado del servidor equivalente al ***stub del cliente***, creado por el compilador IDL para realizar un puente entre el ***ORB*** y el ***objeto implementación***.

2.7.5.2. Adaptadores de Objetos

En ***CORBA***, el objeto adaptador provee la última chance de ***campo de juego de nivel***, en la interfaz del ***objeto ORB***. Esto se necesita porque algunos objetos van a residir en sus propios procesos, y requieren la activación mediante el ***ORB*** antes que ellos puedan ser usados; otros dentro del núcleo del proceso con algunos y no todos en sus clientes, y pueden no estar sujetos a activación, todavía otros pueden ser manejados por base de datos orientadas a objetos y requieren un diferente conjunto de servicios a hacer invocado desde diferentes ***ORB***.



Estructura típica de un adaptodor de objeto. Hay una inerfaz de adaptor de objeto que sirve para todas las implementaciones de objetos. Puede haber múltiples esquemas por implementación de objeto y múltiples implementaciones de objetos por Adaptador de Objeto.

Los adaptadores de objetos son responsables de:

- ✓ Registro de implementaciones
- ✓ Generar e interpretar referencias de objetos

- ✓ Mapear referencias de objetos a sus implementaciones correspondientes
- ✓ Activar y desactivar implementaciones de objetos
- ✓ Invocar métodos, vía esquema o ***DSI***
- ✓ Coordinamiento la interacción de la seguridad

La mayoría de los servicios son accedidos a través de la interfaz del Objeto Adaptador, los desarrollados de ***ORB*** pueden localizar estos servicios en el propietario del ***ORB*** o en el Adaptador de Objeto transparentemente en la implementación del objeto. ***ORB*** diferentes pueden proveer diferentes niveles de servicios, con algunas funciones internas y otras agregadas por el adaptador de objeto.

2.7.5.3. Adaptador de Objeto Básico

Para configurar en que objeto reside procesos distintos desde el ***ORB***, el ***BOA*** provee básicamente las funcionalidades básicas que los objetos y servidores requieren. ***BOA*** provee:

- ✓ Generación e interpretación de referencias de objetos
- ✓ Activación y desactivación de implementaciones de objetos
- ✓ Activación y desactivación de objetos individuales
- ✓ Invocación de métodos vía esquemas

BOA soporta implementaciones de objetos que son construidos desde uno o más programas. También diferencia de un servidor, que se corresponde a una **unidad de ejecución** o a un **objeto**, el cual implementa un método o una interfaz. Los servidores pueden contener múltiples objetos, aunque hay categorías de servidores que soportan solamente uno.

Hay que diferenciar el concepto de **creación de objeto** y de **activación de objeto**. El primero son creados por algún programa durante una ejecución.; el acto de creación genera una nueva instancia y una nueva referencia de objeto. Los objetos son activados por el ***ORB***, no todos, solo los objetos que residen en un proceso servidor conectado a un adaptador.

La interfaz del ***BOA*** soporta cuatro políticas de activación [MOW95]:

- ✓ **Política de servidor compartidos**

  Un servidor activado por un ***BOA*** envuelve múltiples objetos activados.

- ✓ **Política de servidor persistente**

  Como el servidor compartido excepto que el servidor es activado fuera del ***BOA*** y registrado en el proceso de instalación.

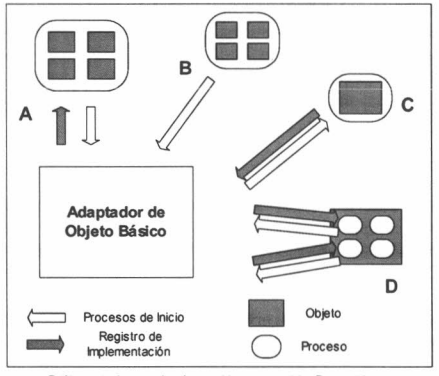
- ✓ **Política de servidor no compartido**

Sólo un objeto para una implementación dada a la vez puede ser activada en el servidor.

✓ **Política de servidor por método**

BOA inicia un servidor separado por cada método invocado; el servidor completa el requerimiento y luego finaliza.

Los servidores compartidos y persistentes son probablemente lo más comunes. La ventaja en los servidores compartidos es la reducción de la sobrecarga de la iniciación del proceso comparado con el **no compartido**; la desventaja es que el sistema operativo no está habilitado para asegurar memoria y seguridad aislada de un objeto a otro. Los servidores persistentes tienen la ventaja que cualquier instalación y mantenimiento son tan complejos que el ***BOA*** no puede realizar todo y requiere scripts separados revisados por un administrador humano. Candidatos para esta posición incluye la mayoría de los sistemas de procesamiento de transacciones, base de datos comerciales. Durante el proceso de instalación, un elemento en el script puede adjuntar varios objetos al ***BOA*** o especializar el adaptador de objeto, registrar su implementación y señalizar el ***ORB*** aquellos objetos que están listos para procesar pedidos. Los servidores no compartidos son apropiados para servir exclusivamente recursos como impresoras, mientras que los servidores por método permiten cargar balanceadamente o aislada de servidores entre ellos por seguridad o razones administrativas.



Políticas de Activación: A, servidor compartido; B, servidor persistente; C, servidor no compartido; D, servidor por método

Las operaciones del ***BOA*** usadas por los objetos y servidores


```
Void impl_is_ready(in implementacionDef impl);
Void deactivate_impl(in implementacionDef impl);
Void object_is_ready( in Object obj, in ImplementationDef impl);
Void deactivate_object(in Object obj);
```

El servidor compartido y el persistente va a notificar al ***BOA*** que los objetos han completado su procesos de inicialización mediante la invocación **impl_is_ready**.

BOA luego envía el pedido de activación del objeto hasta que recibe la llamada **deactivate_impl**. El ***BOA*** llamará a la rutina de activación del servidor de objeto para un objeto antes de realizar la llamada del objeto, y asumirá que el objeto se mantendrá activo y listo para responder invocaciones hasta que el servidor ejecute **deactivate_object** para el objeto.

2.7.5.4. Esquema de IDL Estático

El esquema de IDL estática sobre el lado del servidor juega un rol correspondiente al **stub** sobre el lado del cliente. Se conecta hacia el servidor realizando el mapeo para los lenguajes de programación, y hacia el **OA** mediante la interfaz propietaria.

Las invocaciones pasan a través de los esquemas desde el ***OA*** hacia la implementación; los pedidos vuelven por la correspondiente ruta de regreso.
Donde el núcleo de la implementación de objeto está en el mismo proceso como su ***ORB*** y ***OA*** , los esquemas pueden tener un pequeño plano y crean virtualmente, sin avisar la carga sobre la invocación. Sin embargo, donde el servidor reside en su propio proceso, el esquema tiene que manipular una comunicación **IPC** usando memoria compartida o la red de comunicación. Desde que los usuarios esperan una respuesta instantánea aún desde sistemas distribuidos, la configuración que ubica al ***ORB*** y la implementación en el mismo proceso tiene a dominar en el mercado.

2.7.5.5. Interfaz de esquema dinámico

Usando ***DSI***, un ***ORB*** puede repartir pedido de cualquier implementación de objeto o un proxy puede contactarlo sobre la red, y no solamente que uno se conecte estáticamente al stubs. Esta capacidad va a ser crucial sobre un dinamismo, que se está construyendo una red distributiva.

DSI entre en la arquitectura de ***CORBA*** a través de la interoperabilidad, y por una razón: permite a dos ***ORBs*** construir un puente que los comunica por una invocación a un objeto de ***ORB*** remoto, y devuelve la respuesta.

2.7.6. *Interoperabilidad CORBA*

Hay cuatro asignaciones de usuarios para la interoperabilidad ***CORBA***:

- ✓ Utilización de objetos distribuidos
- ✓ Seleccionar, adquirir, configurar, instalar o mantener la red de objetos distribuidos
- ✓ Programar Clientes y Objetos para su sistema, para sus Clientes o para el mercado abierto
- ✓ Uno debe ser el desarrollador del ***ORB***, y actualmente debe programar la interoperabilidad del ***ORB***.

Cada una de estas asignaciones o roles requiere un conjunto y tipos de conocimiento de cómo ***CORBA*** trabaja la interoperabilidad.

Cuando se utiliza ***CORBA*** , se determina quien tiene asignado cada uno de los roles anteriormente mencionados. Se describe diferentes tipos de usuarios, con el objetivo de utilizar ***CORBA***, con el objetivo de permitir la interoperabilidad en sistemas distribuidos heterogéneos o ambientes heterogéneos.

2.7.6.1. Interoperabilidad para Usuarios Finales

Para los usuarios finales, la interoperabilidad ***CORBA*** es una tecnología transparentemente accesible. Cuando se ejecuta una aplicación distribuida, no hay nada que permita revelar el modo de interoperabilidad o protocolo de la red. Se tiene un grado de libertad que resulta desde la distribución transparente [PET94] a nivel de aplicación, pero aplicaciones individuales pueden proveer la misma función usando otros transportes [TAM91].

Para algunas invocaciones, se puede totalmente despreocupar sobre la localización del objeto destino, lo que es importante es tener disponible ***servicios*** que permitan capturar la localización del objeto deseado como impresoras, escáner, etc..

2.7.6.2. Interoperabilidad para Administradores de Sistemas y compradores de ORB

Si uno adquiere, instala, configura o mantiene ambientes de ***ORBs***, primero hay que realizar el mismo tipo de análisis y configuración que se haría como cualquier otro sistema ínter operable. El protocolo de red va a ser la diferencia para el administrador de sistemas, porque va a tener que configurar la red para que los protocolos de los usuarios pueden correr, asegurar el adecuado ancho de banda y proveer ***gateways*** [TAM91] donde se puede cambiar de acuerdo a las necesidades. Además hay que tener en cuenta la seguridad entre las comunicaciones entre los distintos ***ORBs***. Para lograr este objetivo, es necesario adquirir herramientas y utilitarios con el fin de realizar la administración tanto de la red como de los ***ORBs***.

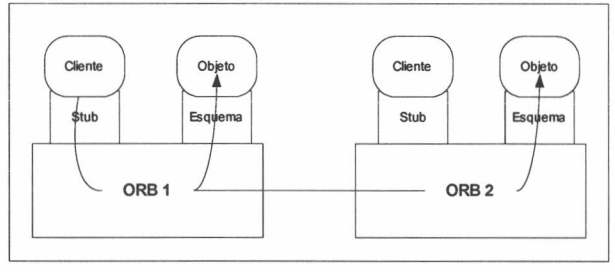
2.7.6.3. Interoperabilidad para programadores

La interoperabilidad ***CORBA*** es totalmente transparente para cualquier aplicación. Una vez que se tiene un objeto con una interfaz ***IDL***, su localización se convierte transparente, y el administrador o el instalador puede situarlo en cualquier lugar sobre la red.

Cuando se diseña aplicaciones para distribución con el objetivo de obtener la combinación de performance y funcionalidad que un buen programa tiene que repartir. El mejor diseño deja lugar para la flexibilidad a instalar o ejecutar, permitiendo el mismo conjunto de ejecutables correr bien en diferentes sitios con diferentes requerimientos y recursos. Teniendo en cuenta lo anterior, se dividen las aplicaciones en objetos. Tomar ventaja para la práctica de buena programación que se aprende de aplicaciones que no están basados en ***CORBA***: diseñar para minimizar el tráfico de la red, utilizar cache [PET94] para datos usados frecuentemente, y utilizar el buen sentido de uno para asegurar una buena performance y flexibilidad.

2.7.6.4. Comunicación ORB a ORB

La interoperabilidad está basada en la comunicación ***ORB*** a ***ORB***, utilizando las interfaces IDL, los repositorios de interfaces e implementación, y todos los aspectos de ***CORBA*** que han sido diseñado para la interoperabilidad.



Interoperabilidad comunicación de ORB a ORB. Todos los clientes se conectan al ORB 1 pueden acceder al objeto implementación tanto en ORB 1 y ORB 2. La misma condición está para los clientes conectados al ORB 2. La arquitectura escala para cualquier número de ORBs conectados.

¿ Cómo trabaja la interoperabilidad ?: una invocación desde un cliente del ***ORB 1***, es a través del IDL stub a través del ***núcleo del ORB***. El ***ORB*** examina la referencia

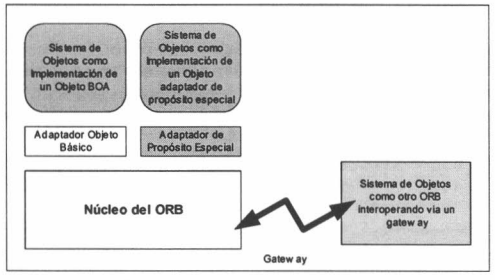
del objeto y busca la localización de la implementación en el repositorio de implementación. Si la implementación es local, el ***ORB*** pasa la invocación a través del esquema del objeto para ser servido. Si la implementación es remota, el ***ORB 1*** pasa la invocación a través del camino de comunicación con el ***ORB 2***, quien rutea hacia el objeto. Porque hay una sola manera de invocación y la referencia de objeto es ***opaca***, la implementación del objeto no tiene forma de conocer (y no le interesa) si el clientes es local o remoto.

Este escenario trabaja a pesar de la plataforma, protocolo, y diferencia de formato que debería existir entre el ***ORB 1*** y el ***ORB 2***. Esto requiere que ambos ***ORBs*** conozcan los suficiente acerca de las invocaciones o respuestas que permiten a ellos traducir los datos donde es necesario como transferir el pedido de una plataforma a otra y viceversa.

Si se observa los enunciado anteriormente, tanto el ***Cliente*** y la ***Implementación del Objeto*** no están involucrados en los pasos de la comunicación. En ***CORBA***, la comunicación siempre va de un ***ORB*** a otro. El Cliente tiene que especificar la ***referencia de objeto*** de un destino remoto, pero el tipo de la referencia de objeto es opaco al Cliente y también no sabemos si el destino es local o remoto. Y la implementación del objeto no recibe información alguna acerca de la invocación del Cliente cuando le llega el pedido (La seguridad es tratada dentro del ORB antes de alcanzar el destino).

2.7.6.5. Integración con otros Sistemas

La arquitectura del ***ORB*** está diseñada para permitir la ínter operación con una gran variedad de sistemas de objetos. Porque hay varios sistemas de objetos existentes, con un deseo común de poder habilitar a los objetos en aquellos sistemas ser accesibles vía el ***ORB***.



Diferentes formas de integrar Sistema de Objetos

Para los sistemas de objetos que simplemente quieren mapear sus objetos hacia los objetos del ORB y recibir invocaciones a través del ORB, un posibilidad es tener

aquellos sistemas de objetos aparezcan ser implementaciones de los correspondientes objetos del ORB. El sistema de objetos debería registrar sus objetos con el ORB y manejar los pedidos entrantes, y poder actuar como un cliente y ejecutar los pedidos salientes.

Un objeto adaptador puede ser diseñado para objetos que son creados en conjunto con el ORB y que son primariamente invocados a través del ORB. Otro sistema de objetos puede desear crear objetos sin consultar al ORB, y esperaría la mayoría de las invocaciones que ocurran dentro de él que a través del ORB. En algunos casos, es más apropiado que el ***adaptador de objeto*** permita que los objetos sean registrados implícitamente cuando son pasados a través del ORB [SIE96].

Hay dos maneras de que los ***ORBs*** dialoguen entre sí:

- ✓ Tener todos los ***ORBs*** para hablar en el mismo protocolo, son el objetivo que pueden dialogar entre ellos directamente.
- ✓ Si los ***ORBs*** hablan diferentes protocolos, se pueden instalar puentes (bridges – [TAM91]) para traducir de un protocolo a otro.

La especificación del ***CORBA 2.0*** [OMG95A] provee las dos soluciones. La primera es simple, fácil de administrar, y eficiente en su uso. Todos los ***ORBs*** compatibles con ***CORBA 2.0*** hablan el protocolo mandatario ***IIOP*** (Internet Inter-ORB protocol), permitiendo a una organización estandarizarse y poder elegir cualquier ***ORB*** del mercado con una total interoperabilidad. Pero se puede optar DCE CIOP (Common Inter-ORB protocol) o un protocolo propietario.

Pero hay veces que un protocolo común no es posible o no es deseable por alguna razón. En estos casos, el ***bridging*** permite la interoperabilidad transparente entre los límites de los dominios.

2.7.6.6. Dominios en CORBA 2.0

En cualquier organización, se realizan divisiones claras y diferencias en distintos ***tipos de dominios***. Esto resulta desde las diferencias de tecnología, y otras desde la clase de trabajo o de los recursos disponibles. También esta clasificación dependa de los niveles de seguridad y acceso.

La especificación ***CORBA 2.0*** de lista dominios de tecnologías, de administración y de políticas y se definen algunas generalizaciones:

- ✓ Existencia por alguna razón
- ✓ Aplicaciones y datos en general van a ser compartidos a través de los límites de dominios
- ✓ Todo conjunto es un dominio tiene su correspondencia en otro dominio, luego se construye un puente para ejecutar la traducción dónde es necesario

- ✓ Los límites del ORB define sólo una pequeña fracción de los dominios en un sistema

Tomando estos factores en consideración, la interoperabilidad está basada en conceptos de ***construir un puente*** (bridging) que es más general que la comunicación inter-ORB.

La especificación define:

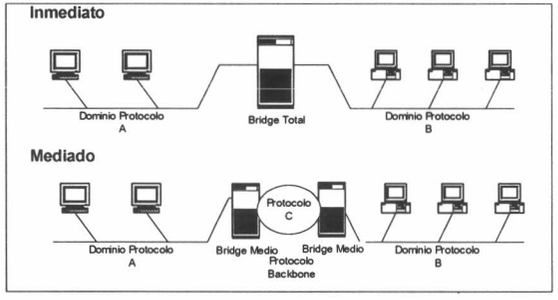
Dos clases para definir ***bridging*** y dos clases de puentes (***bridge***) [SIE96].

- ✓ Tecnología y Políticas

  Hay tendencias que asumen entre los límites de los dominios, que deben ser transparentes y hacen referencia a la tecnología como protocolos de red, formatos de datos o plataformas de ***ORB***. Y otros límites que son administrativos que separan a la organización con diferentes políticas, objetivos y recursos con el objetivo de controlar el flujo de trabajo por diferentes tipos de razones. Esto necesita políticas de mediación, para monitorear los conceptos que son enviados de un lugar a otro.

- ✓ Puente inmediato y mediado

  Hay dos formas de construir puentes, ***inmediato*** y ***mediado***.



Generalmente especificamos el puente entre diferentes protocolos de red. En el ***puente inmediato*** dos dominios charlan directamente a través de un simple puente que traduce cualquier parte del mensaje requerido. El ***puente mediado***, todos los dominios están unidos a un protocolo simple en común. Cuando un mensaje pasa a través del primer puente del dominio originado, la parte que requiere es traducida al protocolo en común.

2.7.6.7. Referencias de Objetos Ínter operable

Es usado solo en las invocaciones ***inter-ORB***, y por lo tanto es emitido y aceptado por los ***ORBs*** que hablan en la red, y es usado por lo puentes que están definidos entre ellos.

La información en el ***IOR*** permite el pasaje de la invocación de un ***ORB*** a otro, de distintos vendedores. Estos requieren la siguiente información:

¿ Que tipo es el objeto ?

ORBs deben poder saber el tipo de objeto con el objetivo de preservar la integridad del tipo de sistema.

¿ Que protocolo puede invocar el ORB usado ?

El ***IOR*** lista el protocolo o los protocolos aceptados por el ***ORB***. Pero como cruzó el puente, se convierte disponible sólo por lo protocolos aceptados por el puente. Esto requiere puentear el reconocimiento de los ***IORs*** y actualizar esta información como cuando pasan a través de él.

¿ Que servicios del ORB están disponibles ?

La invocación puede involucrar servicios extendidos del ***ORB***.

¿ Es nula la referencia del objeto ?

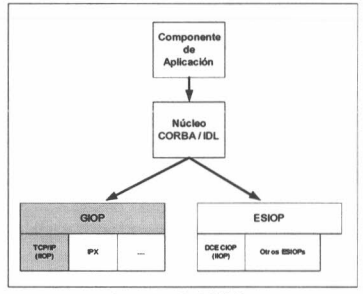
Reconociendo nulos, el puente puede evitar demasiado trabajo innecesario.

Cada protocolo usado por ***CORBA*** en la comunicación tiene un *tag* y un *profile*. Los *profile* contienen toda la información de un ***ORB*** remoto necesita para ejecutar la invocación usando el protocolo.

2.7.7. Especificación de Interoperabilidad

A continuación se muestra la estructura de la especificación de Interoperabilidad ***CORBA 2.0***. Si el objeto de referencia destino es un ***objeto local***, su invocación permanecerá local, pero si es remota, luego su invocación será remota.

Tanto el ***Cliente*** como el ***objeto implementación*** habla mediante los ***ORBs***, y no con la red. Toda la comunicación de la red es de un ***ORB*** dialogando con otro. Esta es la forma que trabaja ***CORBA***.



Estructura de la especificación CORBA 2.0

GIOP, o ***General Inter- ORB Protocol*** (Protocolo inter-ORB general) contiene las especificaciones para los protocolos de mensajes inter-ORB generales de ***CORBA***, que ha sido diseñado para ser implementado en cualquier transporte confiable.

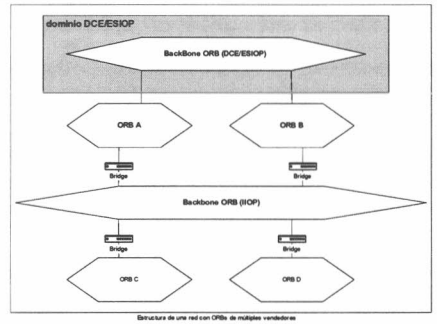
GIOP especifica los formatos de requerimientos y protocolos de transmisión para permitir la interoperabilidad ***ORB a ORB***.

Debajo del ***GIOP*** están los conjuntos de protocolos confiables, con el ***TCP/IP*** como mandatario, que especifica un protocolo de interoperabilidad estandarizada para Internet. No es necesario utilizar ***RPC***.

ESIOP, o ***Envioronment-Specific Inter-ORB***. Cada ***ESIOP*** permite definir su protocolo independientemente tanto como se encuentra con los requerimientos para la comunicación ***ORB a ORB*** de pedidos y respuestas de objetos.

El ***DCE ESIOP*** especifica la comunicación ***ORB a ORB*** basados en invocaciones y respuestas ***DCE RPC***.

Los protocolos estándar deben direccionarse tan cerca posible a la estructura del ***GIOP*** permitiendo el bridging hacia el ***IIOP*** estándar y otros protocolos basados en ***GIOPs***.



Estructura de una red con ORBs de múltiples vendedores

2.7.7.1. Protocolo inter-ORB general (GIOP) y Protocolo inter-ORB internet (IIOP)

IIOP, el cual es ***GIOP*** [SIE96] sobre ***TCP/IP***, es un protocolo mandatario para ***CORBA 2.0***. La especificación ***GIOP/IIOP*** fue diseñada para cumplir los siguientes objetivos:

- ✓ Gran Disponibilidad Posible

  IIOP está basado en ***TCP/IP***, el más ampliamente usado y disponibilidad de un mecanismo flexible de comunicación de transporte, y define solamente unas mínimas capas adicionales para transferir requerimientos ***CORBA*** entre ***ORBs***.

- ✓ Simplicidad

  Trabajando dentro de otros objetivos necesarios, ***GIOP*** se mantuvo lo más simple posible.

- ✓ Escalabilidad

  Está diseñado para escalar al tamaño de hoy en día de Internet y más allá.

- ✓ Bajo Costo

  Tanto la reingeniería e implementación de ***ORB*** y el tiempo de ejecución soportan costos, por lo tanto fueron minimizados lo más posible.

- ✓ Generalidad

  El ***GIOP*** fue diseñado para la implementación de cualquier confiable, protocolo orientado a conexión, no sólo ***TCP/IP***.

✓ Neutralidad de Arquitectura

GIOP hace mínimas asunciones acerca de la arquitectura e implementación de los ***ORBs*** y los puentes que lo soportan.

GIOP consiste de tres especificaciones:

- Definición de representación común de datos (CDR)
- Formatos de mensajes ***GIOP***
- Asunciones de transporte ***GIOP***

La especificación ***IIOP*** agrega:

- Transporte de mensajes **Internet *IOP*** (no es una especificación separada, sino que está mapeado al ***GIOP*** que especifica el transporte***TCP/IP***

2.7.7.2. GIOP – Representación de Datos Común (CDR)

El ***CDR*** define la representación de todos los tipos de datos del ***OMG IDL***, incluyendo códigos de tipos y tipos de construcciones como **struct, sequence** y **enum**.

La especificación toma en cuenta el ordenamiento de bytes y alineamiento. El receptor tiene la correcta semántica, por lo tanto evita la innecesaria traducción del mensaje entre máquinas con los mismos ordenamiento de bytes.

Este características es importante tenerla en cuenta en el momento de diseñar un ***ORB*** propio. El punto es que el ***CDR*** es para todos los ***GIOP*** que comparten la representación de datos común, por lo tanto el bridging de esta parte del mensaje debería ser fácil.

2.7.7.3. GIOP – Formatos de Mensajes

GIOP define siete mensajes distintos para la comunicación ***ORB a ORB***. Este permite llevar a cabo requerimientos, localización de objetos de implementación, y administrar los canales de comunicación. Ellos soportan todas las funciones y comportamientos requeridos por ***CORBA***.

Un comentario frecuente acerca de este mecanismo es que es **sólo otro RPC**, pero en realidad, estos siete mensajes con sus características asociadas son diferentes a **RPC**. Uno puede programar estos siete mensajes sólo si programa su propio ***ORB***.

Los siete mensajes simplifican la comunicación. El acuerdo de binding y de formato son llevados a cabo sin negociación. En la mayoría de los casos, los mensajes desde un ***Cliente ORB*** a un ***Objeto ORB*** puede ser enviado lo más pronto posible a una conexión establecida. Algunas características de la transferencia de mensaje son:

✓ La conexión es asimétrica

Los roles del Cliente y Servidor son distintos , asignados en la conexión. El Cliente origina la conexión, y puede enviar el pedido pero no la contesta. El Servidor acepta la conexión, y puede enviar la respuesta, pero no el pedido.

✓ El pedido puede ser multiplexado

Múltiples Clientes dentro de un ***ORB*** pueden compartir una conexión hacia un ***ORB*** remoto. La información en el pedido los diferencia sobre la misma conexión.

✓ El pedido puede solaparse

Si es asincrónico, cualquier orden de pedido y respuesta puede soportar identificadores de pedido / respuesta.

Los siete mensajes se caracterizan por:

Cliente	***Servidor***	***Ambos***
Request	Reply	MessageError
CancelRequest	LocateReply	
LocateRequest	CloseConnection	

Estos mensajes están definidos generalmente, a menos que se construya un ***ORB***, por lo tanto uno tiene que estudiar internamente tanto en estructura como de comportamiento, pero no es el alcance de este capítulo.

2.7.7.4. GIOP – Requerimientos de Mensajes de Transporte

Los requerimientos de mensajes de transporte ***GIOP*** son elegidos específicamente para mapear al ***TCP/IP***, pero también se puede considerar ***Novell IPX***.
Básicamente, ***GIOP*** requiere:

- Protocolo orientado a conexión
- Reparto confiable (ordenamiento de bytes debe ser preservado y el asentimiento de reparto debe estar disponible)
- Los participantes debe ser notificados de la pérdida de conexión
- El modelo de inicio de una conexión debe cumplir ciertos requerimientos

2.7.7.5. ESIOPs

Un protocolo que no es ***GIOP*** es un ***ESIOP*** si está basado en ***CORBA 2.0***, que es la arquitectura básica que incluye dominio y bridging, IOR, y las interfaces de interoperabilidad incluyendo DSI.

Protocolos basados en ***DCE*** fueron adoptados por la ***OMG*** como parte de ***CORBA 2.0*** como ***ESIOP***. Esto conforma todos los requerimientos ***CORBA*** que fueron listados, además, usa el mismo ***CDR*** como ***GIOP***, una característica que facilitará el bridging entre él y los dominios ***GIOP***.

El ***DCE-CIOP*** remplaza la funcionalidad de los siete mensajes definidos anteriormente con dos llamadas ***DCE-RPC***, *locate* y *invoke* y es reemplazado el transporte ***GIOP***, para manejar únicamente estos dos mensajes.

2.7.7.6. Estructura de especificación DCE-CIOP

El ***DCE-CIOP*** es un protocolo para la comunicación ***ORB*** a ***ORB***, donde tiene el mismo rol que el ***IIOP***. El Cliente y el objeto implementación está interactuando solamente con su ***ORB*** local.

DCE-CIOP corre sobre ***RPC*** que es trabaja con protocolos específicos ***DCE***:

- Define ambos protocolos orientados a conexión y sin conexión
- Soporte múltiples protocolos de transporte incluyendo TCP/IP
- Soporta múltiples pedidos de múltiples objetos CORBA sobre la misma conexión
- Soporta fragmentación de mensajes, una ventaja para largo pedidos y respuestas

2.7.8. Conclusiones

En esta sección se describió la arquitectura ***CORBA*** y cada uno de sus componentes, para que las aplicaciones puedan ejecutarse en un ambiente distribuidos de objetos.

Hasta ahora se mostró como los componentes de software dialogan entre ellos y se trató a ***CORBA*** como un bus de interoperabilidad y lo que hace para componentes inteligentes, luego se habló del modelo de objetos ***CORBA*** y la infraestructura que lo soporta. El surgimiento de esta arquitectura se debe a la búsqueda de un estándar (OMG), con el objetivo que las aplicaciones dialoguen de una forma transparente e independiente en el entorno en dónde se están ejecutando.

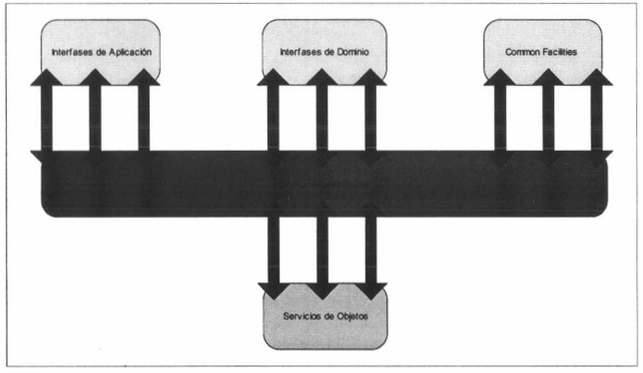
2.8. Arquitectura por Componentes

OMA (Object Management Architecture) [OMG95A] es la visión de alto nivel de un ambiente distribuido completo. Está compuesto por cuatro componentes que pueden ser divididos en dos partes: componentes orientados al comportamiento y arquitectura de sistemas en general (ORB y Servicios de Objetos), y componentes orientados a aplicaciones (Objetos de Aplicación aplicaciones para un fin determinado denominados ***Common Facilities***).

Incluye también el ORB, del cual constituye la ***base de la arquitectura*** [COR95] definida por la ***OMG*** y administra toda la comunicación entre los componentes. Permite a los objetos interactuar en ambientes distribuidos heterogéneos, independientemente de la plataforma en el cual los objetos residen y que técnicas son usadas para implementarlos.

El objetivo de la arquitectura [MOW95] es permitir a las aplicaciones que proveen funcionalidades básicas, las provean mediante una interfaz estándar. Esto habilita al mercado de componentes de software el desarrollo tanto desde el alto y bajo nivel de la interfaz.

Por ***debajo del nivel***, múltiples implementaciones intercambiables de funcionalidades básicas aportando diferencias en performance, precio o adaptaciones para ejecutarse en plataformas especializadas, mientras que por ***arriba del nivel*** tenemos componentes especializados que vienen para un mercado, mediante el cual se integran mediante interfaces estándares.



Arquitectura del Modelo de Referencia

Los ***servicios de objetos*** especifican servicios básicos que al menos todos los objetos necesitan; esto es parte de la arquitectura de la ***OMG*** que empezó primero.

Common Facilities provee los servicios de nivel intermedio para todas las aplicaciones y se acercan más al usuario.

La transparencia de accesos y manipulación de los datos o aplicaciones provistos por los **servicios de objetos (Corbaservices y Corbafacilities)** y **common facilities** minimizarán el gasto de tiempo y esfuerzo dedicado sobre la infraestructura y permitir a los programadores concentrarse sobre los problemas en sus dominios de trabajo y no en problemas estructurales como sea integración, interrelación, etc.

A continuación se describe las características de cada uno de los componentes de la arquitectura definida por la ***OMG***.

2.8.1. Servicios de Objetos (CORBAservices)

Los servicios de objetos ***CORBA*** son una colección de servicios de ***nivel de sistema*** empaquetados con una interfaz específica en ***IDL***. Se puede pensar como un complemento de las funcionalidades de un ***ORB***. Se puede utilizar para crear un componente, darle un nombre e introducirlo dentro del ambiente. ***OMG*** ha definido servicios estándares.

- Servicio de ciclo de vida

Define las operaciones básicas de los objetos para crear, copiar, mover y borrar sobre el bus de integración. Esta convención [SIE96] permite a los clientes ejecutar operaciones de ciclo de vida sobre objetos en diferentes localizaciones, y especificar las localizaciones requeridas donde sea necesario, usando interfaces estándar y sin violar el principio de transparencia referencial en el cual fue construido ***CORBA***.

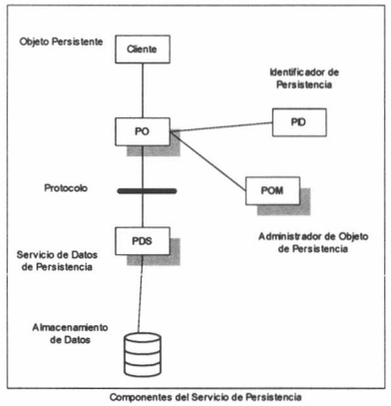
- Servicio de Relación

Provee una manera de crear asociaciones dinámicas o enlaces entre componentes que no se conocen entre ellos [SIE96]. Además, provee mecanismos para recorrer los enlaces que agrupa a los componentes. Su utilización refuerza las consistencias de referencia de integridad, seguimiento de relaciones existentes y para cualquier tipo de enlaces dentro de cada componente definido como un grupo. Las aplicaciones pueden usar las relaciones en varias maneras diferentes y se puede pensar que cualquier objeto puede estar involucrado en una relación al mismo u otro tiempo.

- Servicio de persistencia

Provee una interfaz simple para almacenar otros componentes persistentemente [MOW95] sobre una variedad de servidores de almacenamiento, incluyendo base de datos orientadas a objeto [ELM94], base de datos relacionales [ELM94] y archivos simples. Estandariza los servicios de almacenamiento y recuperación de las implementaciones de los objetos. Con el objetivo de proveer flexibilidad y

estandarización con el mismo servicio, la arquitectura de este servicio se divide en un número de componentes.



Componentes del Servicio de Persistencia

- Servicio de Externalización

Provee una manera estándar de obtener y proporcionar datos desde un componente [SIE96]. Define interfaces, protocolos y convenciones para almacenar y obtener estados de objetos en una cadena estandarizada de datos que puede ser capturado por cualquier medio de almacenamiento o enviado sobre la red. El servicio de externalización usa el servicio de relación para visualizar los grupos relacionados de objetos y manipular operaciones sobre la composición de los mismos.

- Servicio de Nombres

Permite a los componentes sobre el bus localizar otros componentes por el nombre [ORF97]. Está diseñado para tener una sintaxis independiente, estructura de nombre jerárquica que puede ser utilizado en cualquier establecimiento de nombramiento convenido.

- Servicio de Trader

Permite a los objetos publicar sus servicios y ofrecerse para trabajos. Tiene un comportamiento parecido a las páginas amarillas. Permite ir de compras de objetos en línea. Los servidores de objetos registrarán los ***servicios ofrecidos*** en su comercio local (servidores).

- Servicio de Evento

Permite a los componentes sobre el bus, registrar dinámicamente o eliminar la registración de determinados eventos específicos [OF97]. El servicio define un objeto bien conocido como ***canal de evento*** que colecta y distribuye los eventos a todos los componentes registrados y no se conocen entre ellos. Se definen dos actores: abastecedor y el consumidor.

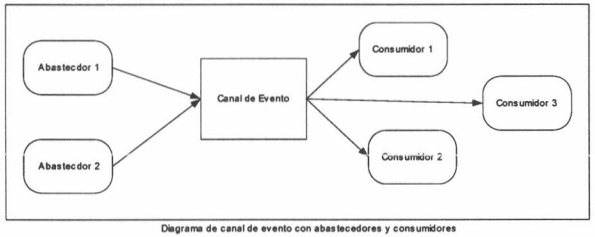


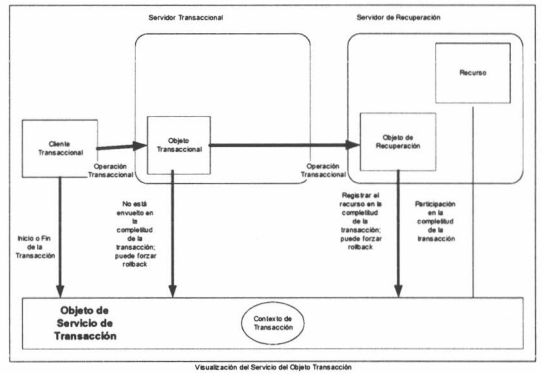
Diagrama de canal de evento con abastecedores y consumidores

- Servicio de Concurrencia

Proveer la administrador de lockeos [PET94] que permite obtener lockeos en forma directa desde una transacción o threads. Frecuentemente, una base de datos debe asegurar que solamente un cliente a la vez puede acceder a un registro; o a un sistema de archivo [PET94] va a tener un acceso restringido de escritura a un archivo con el objetivo de prevenir conflictos de edición. Este servicio fue diseñado específicamente para el servicio de transacción para entender el contexto de la transacción.

- Servicio de Transacción

Provee la coordinación del two-phase commit [ELM94] para la recuperación de componentes usando transacciones simples o anidadas. El servicio de transacción mantiene el contexto de transacción que está asociados todas las operaciones [SIE96].



Visualización del Servicio del Objeto Transacción

2.8.2. CORBAfacilities

CORBA provee una arquitectura base e interoperabilidad, y ***CORBAservices*** construye una base de soporte de los componentes permitiendo la comercialización independiente por diferentes vendedores facilitando la integración de los mismos. Esto desde el punto de vista del programador, pero desde el usuario final, no es necesario conocer estos detalles, por lo tanto se definen las ***CORBAfacilities*** que proveen servicios para el trabajo aplicado a un fin específico. Esta arquitectura está dividida en:

- Facilities Horizontal

Componentes generales, que pueden ser utilizados en diferentes contextos y dominios como sistemas de administración, entidades de ayuda, etc. Las ***CORBAfacilities*** están divididas en ***interfaz de usuario***, ***administración de información***, ***administración de sistemas***, ***administración de tareas***.

- Facilities Verticales

Componentes para un área especializada del mercado como Salud, telecomunicaciones y servicios financieros. Para lograr este consenso para cada dominio en particular, es necesario una participación de audiencia, y es convocada por la ***OMG***. Entre ellas se puede observar ***CORBAMED***.

2.9. *Conclusiones*

Básicamente, ***CORBA*** es un estándar para objetos distribuidos. Permite a una aplicación realizar un pedido de una operación para ser ejecutada por un objeto distribuido y recibe el resultado de la operación requerida. La aplicación se comunica con el objeto distribuido que actualmente ejecuta la operación. Esto es la funcionalidad básica del ***Cliente – Servidor***, dónde el ***Cliente*** realiza un pedido hacia un ***Servidor*** y este último le responde al ***Cliente***. El dato puede pasar desde el ***Cliente*** hacia el ***Servidor*** y estar asociado con una operación en particular sobre un objeto en particular. El dato es devuelto hacia el ***Cliente*** sobre la forma de la respuesta.

CORBA es una herramienta que con mínimo esfuerzo permite escribir aplicaciones distribuidas orientadas a objetos, brindando la posibilidad de integración multiplataforma y multilenguaje.

Para los desarrolladores, ***CORBA*** provee una clara interfaz orientada a objeto hacia un acceso a la red de bajo nivel. Esto hace que al escribir aplicaciones distribuidas sea menos dificultoso que escribir aplicaciones **stand-alone** [PET94].

CORBA reconoce dos roles de las aplicaciones: ***Cliente*** y ***Servidor***. El ***Servidor*** expone a los objetos [WIR90]. Los ***Clientes*** invocan métodos sobre esos objetos [WIR90]. Notar que una aplicación puede jugar los dos roles, esto es, un ***Servidor*** puede además actuar como ***Cliente*** de otro ***Servidor***.

Cliente y ***Servidor*** pueden residir sobre diferentes computadoras. Pueden correr en diferentes sistemas operativos, y pueden estar escritos en diferentes lenguajes. Todas estas variaciones son transparente hacia el desarrollador.

CORBA provee una infraestructura escalable ***Servidor a Servidor***. Un conjunto de servidores de objetos de negocio (aplicados a dominios en particular) pueden comunicarse usando ***CORBA-ORB***. Estos objetos pueden correr sobre múltiples servidores dando balance sobre la carga de los pedidos entrantes de los ***Clientes***. Los objetos ***CORBA*** sobre el ***Servidor*** interactúan con otros objetos usando **ORB**.

Como se ha visto, el ***ORB*** puede despachar los pedidos hacia el primer objeto disponible y agregar más objetos de acuerdo al incremento de la demanda. ***CORBA*** permite a los ***Servidores de Objetos*** actuar usando los alcances de las transacciones y los servicios de ***CORBA*** relacionados, esto a través del ***ORB***.

CORBA provee ***servicios*** generales y específicos de dominio con el fin de lograr una interoperabilidad e integración clara y transparente, focalizando sobre el problema a resolver, en contraposición al hecho de encontrar primero las herramientas de cómo resolver el problema planteado.

2.9.1. Evaluación de ***CORBA***

Para utilizar esta tecnología, primero hay que realizar un estudio de evaluación. En otras palabras, no hay una forma de responder a la hora de saber si es conveniente de elegir este tipo de tecnología con el objetivo de lograr la integración de datos y aplicaciones en una organización. La definición de un plan de sistemas está íntimamente relacionado con las consideraciones de negocios de la organización (estrategia de negocio), a la hora de elegir una tecnología determinada. La vasta mayoría de estas decisiones son dominadas por las ***consideraciones de negocio*** y no por ***consideraciones técnicas***.

Tomando en cuenta estos conceptos, en el momento de encarar un plan de sistemas, y evaluando como este caso a ***CORBA*** como un estándar de infraestructura de aplicaciones para una organización determinada, los items a estudiar son:

- Curva de aprendizaje

  Tiempo que se emplea para aprender esta tecnología debe ser amortizada a través de varios proyectos. ***CORBA*** introduce características que pueden ser nuevas o no dependiendo del nivel de desarrolladores que tiene la organización:

  - ✓ Conceptos: referencias de objetos, proxies, adaptadores de objetos, etc.
  - ✓ Componentes y herramientas: lenguajes de definición de interfaces, compiladores ***IDL***, ***ORB*** (brokers de objetos), etc.
  - ✓ Funcionalidades: manejo de excepciones y herencia de interfaces

- Interoperabilidad

  La forma de comunicar las aplicaciones e integración de la información en toda la organización. Esto va a depender de la estructura tanto de los ***Servidores*** y de los ***Clientes*** a la hora de realizar integraciones. Las implementaciones de los ***Servicios*** que se necesitan para abordar a soluciones determinadas. Existen vendedores independientes de ***ORB***, mediante el cuál le da la posibilidad de comparar a los usuarios de cómo resuelven los servicios que se necesitan.

- Portabilidad

  Hay una relación muy estrecha con la curva de aprendizaje, dado que la portabilidad va a depender en los diferentes escenarios que necesitan ser integradas las aplicaciones y datos. Va a depender de los servicios que proveen los vendedores para las funcionalidades que se necesitan resolver. Es una característica importante dado las aplicaciones tienen que evolucionar paralelamente, con las decisiones estratégicas de la organización, en consecuencia se tiene que garantizar el crecimiento a un bajo costo.

- Limitaciones

CORBA no direcciona sobre las claves inherentes a la complejidad de la computación distribuida, como ***retardos***, ***tolerancia a fallas***, ***orden casual***, ***abrazos mortales*** (deadlocks) [PET94]. Esto tiene que ser tomado en cuenta con mucho cuidad al ponerlo en práctica. ***CORBA*** aún no soporta operaciones asincrónicas ni operaciones no bloqueantes; esto es dejando a la ***calidad de servicio***, definido en capítulos anteriores, sobre la implementación. Implementaciones actuales de ***CORBA*** carecen de soportes para la transferencia de volúmenes de datos.

- Performance

  La performance puede verse afectada, de acuerdo a las características de ***CORBA*** como:

  - ✓ Invocaciones remotas adicionales
  - ✓ Sobrecarga de relaciones de parámetros
  - ✓ Copias de datos
  - ✓ Administración de memoria (debido a la interacción sobre la red)

  Afectación en la performance dada la ecuación ***extensibilidad***, ***robustez*** y ***mantenibilidad***, sobre la eficiencia a bajo nivel, dado el contraste sobre la optimización a nivel general dado por el ***ORB.***

CORBA permite la colaboración entre vendedores independientes y aplicaciones sin importar en que plataformas están implementadas.

2.9.2. Beneficios de *CORBA*

Para el desarrollo de aplicaciones se convierte en parte crítica, la figura del ***middleware*** con el fin de facilitar la integración en una organización, sin preocuparse sobre la comunicación a bajo nivel para integrar su aplicaciones presentes y futuras. Hay un número de razones mediante el cuál ***CORBA*** permite estas características [ORG96]:

- ✓ ***IDL*** une los lenguajes de programación, sistemas operativos, redes y sistemas de objetos.
- ✓ Facilidad para definir, implementar y utilizar interfaces. Estas interfaces son orientadas a objetos, facilitando el mantenimiento.
- ✓ Cada Servidor puede contener varios objetos; la comunicación es directa desde el llamador al objeto destino y los objetos pueden ser de cualquier tamaño.
- ✓ Interacción con diferentes middleware.
- ✓ Los servicios de ***CORBA*** provee un conjunto de extensión opcional que direcciona a áreas que el núcleo por si mismo no puede llevar a cabo. Por ejemplo, transacciones, eventos, etc.

- ✓ Integración con otras tecnologías, como base de datos, sistemas de mensajes, sistemas de generación de interfaces de usuario, etc.
- ✓ Aplicación para varios dominios. Es decir, puede ser utilizado para los diferentes ámbitos de la industria, resolviendo problemas de bajo y alto nivel.
- ✓ ***IDL*** permite el mapeo separado para cada lenguaje de programación, con el fin de ser usado para cada lenguaje en forma natural.
- ✓ Existencia de un protocolo de acuerdo, como el ***IIOP***, para la comunicación entre ***ORB***.
- ✓ Está bien establecido y adoptado como un estándar, para ser escrito y mantenido por un procedimiento abierto.
- ✓ Hay implementaciones compitiendo, logrando así una continua innovación y actualización.

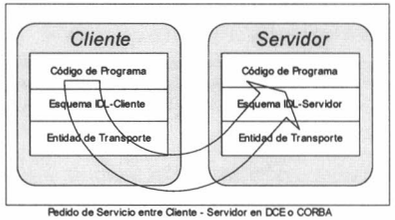
Capítulo 3: DCE y CORBA

Mucha gente observa a ***DCE*** [ROS92] y ***CORBA*** [MOW95] como dos tecnologías en competencia. Sin embargo, ambas soportan la construcción e integración de aplicaciones *Cliente – Servidor* en ambientes distribuidos heterogéneos. Las comparaciones se focalizan en diferencias entre capacidades individuales o en diferencias de madurez de especificaciones y productos que lo conforman. Hay diferencias fundamentales entre ***DCE*** y ***CORBA*** [HAR97], sin embargo no hay una explicación clara sobre estos criterios como una base de seleccionar una plataforma de computación distribuida.

Este capítulo comienza con una introducción general de la similitud de las dos arquitecturas y continúa con una descripción de ***DCE***, junto con su arquitectura conceptual y explicación de cada uno de sus componentes. Para finalizar con presentar las diferencias más importantes con ***CORBA***. El desarrollo se extiende desde las capacidades individuales, maduración de especificaciones y productos, y concluye con una revisión de cómo una organización debe seleccionar la tecnología más apropiada para cumplir los objetivos de computación distribuida.

3.1. Introducción General

Si se observa la arquitectura del manejo de los pedidos entre llamadas de ***DCE*** y ***CORBA*** no se encuentra diferencias en un nivel superior.



Pedido de Servicio entre Cliente - Servidor en DCE o CORBA

Si analizamos cada uno de los niveles, ambas arquitecturas presentan los mismos componentes:

- Lenguaje de Definición de Interfaces
- Almacenamiento de IDL en Stubs de Cliente y Servidor
- Secuencia de llamadas de pedidos de Cliente a Servidor
- Mismos conceptos de transmisión de pedidos de servicios

Hay muchas similitudes entre ***DCE*** y ***CORBA***, pero el propósito es examinar que y como difieren una arquitectura de otra. Las diferencias se pueden establecer en:

- Capacidades Individuales
- Madurez de las especificaciones
- Productos que la conforman

Se va a presentar las diferencias entre ***DCE*** y ***CORBA*** en todos los niveles como ***estilo y soporte de programación***, ***integración***. Como se describió en el capítulo anterior ***CORBA***, se va a explicar las características de ***DCE*** para continuar con las diferencias de estas dos tecnologías.

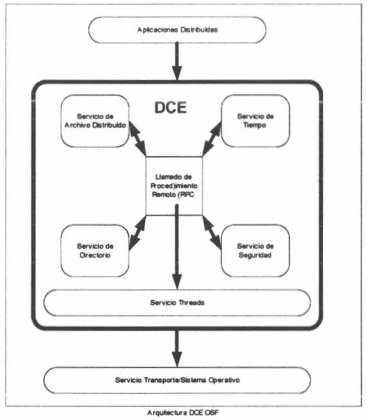
3.2. Examinando DCE

En 1990, la Open Software Foundation (OSF) – un consorcio que lo integran un número de vendedores de hardware – crean una colección de herramientas para desarrollo y administración de sistemas Cliente – Servidor a gran escala. El foco inicial fue un conjunto fundamental de bloques de construcción que operaría transparentemente a través de diferentes plataformas de diferentes vendedores.

Incluidos en el paquete – llamado *ambiente de computación distribuida* (DCE - Distribuited Computing Enviorement), debería soportar comunicación, seguridad, nombramiento, tiempos y otros servicios básicos de computación distribuida. El esfuerzo fue sin precedentes para el número de vendedores independientes que tuvieron que trabajar juntos, la diversidad de plataformas de hardware y la magnitud de la tarea.

DCE 1.0 fue la primera versión en 1992 por IBM [IBM98], Hewllett-Packard y Transarc Corporation [TRA98].

La visión común fue utilizar ***DCE*** para construir sistemas Cliente – Servidor a gran escala y que permita el próximo escalón lógico en la evolución de una arquitectura abierta en una organización. [TRW98]



Arquitectura DCE OSF

La arquitectura ***DCE*** se puede clasificar en:

- ***DCE Executive***
- ***DCE Servicios Extendidos***
- ***Aplicaciones Distribuidas***
- ***Servicios de red y Sistema Operativo***

DCE Executive consiste en los siguientes componentes:

- Servicio de Seguridad
- Servicio de Directorio
- Servicio de Distribución de Tiempos
- Llamada de Procedimiento Remoto (RPC)
- Servicio de Threads

DCE Servicios Extendidos consiste en el siguiente componente:

Servicio de Archivo Distribuido (DFS)

Aplicaciones Distribuidas se corresponden con las aplicaciones que pueden dialogar con un Servidor mediante DCE.

Servicios de red y Sistema Operativo representa el componente en dónde corren cada una de las aplicaciones de red y permite la comunicación entre los Clientes y Servidores.

Aún cuando los usuarios de ***DCE***, se encontraron con poderosos bloques de construcción ofrecidos por ***DCE***, todavía se presentaba desafíos en la computación a gran escala.

Por ejemplo, mecanismos de administración avanzados de fallas como transacciones que requieren sistemas críticos de negocios [TRA98], más allá del alcance de ***DCE***. Además, muchas áreas relacionadas con la ejecución, ejecución y administración de negocios críticos, las aplicaciones basadas en ***DCE*** no llegan directamente por servicios ***DCE***. Estos puntos y otros dejan a algunas organizaciones en la búsqueda de soluciones externas.

Cuando consideramos ***DCE,*** es importante recordar que ***DCE*** no es el punto final. La mejor manera que ver ***DCE*** es como un importante primer paso hacia diferentes y deseables finales. Desde la perspectiva, es fácil ver que ***DCE*** permite:

- Reparte servicios fundamentales de computación distribuida en un paquete integrado.
- Tiene compromisos de vendedores, que está generalmente disponible y es completamente compatible a nivel de ***API*** a través de diversas plataformas, desde Windows a OS/2 a UNIX a MVS.
- Fue diseñado desde abajo para soportar las necesidades de computación distribuida.

3.3. Entendiendo DCE

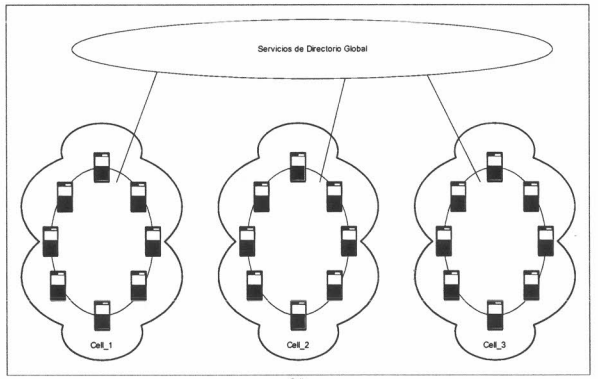
DCE fue diseñado para proveer la llave de bloques de funciones requeridos para cualquier sistema distribuido. Una manera fácil de imaginar los servicios necesarios es pensar a un *mainframe* que ha sido dividida en varias piezas. Los *servicios de comunicación* son requeridos para comunicar las piezas. Los *servicios de seguridad* son necesarios para proteger la comunicación y accesos autorizados. Los ***servicios de nombres*** permiten a las aplicaciones encontrar dónde está corriendo la aplicación correspondiente. Los ***servicios de tiempos*** aseguran que los eventos puedan ser sincronizados en todas las piezas. Los ***servicios de archivos distribuidos*** permiten los datos compartidos entre las piezas.

En este capítulo se va a detallar cada de las características y componentes de ***DCE***:

- Celdas (Cells)
- Servicios Threads
- Llamada de Procedimiento Remoto (RPC)
- Servicios de Seguridad
- Servicio de Celdas (Cells) Directorio
- Servicio de Tiempos
- Sistema de Archivo Distribuido

3.2.1. *Cells*

En un ambiente ***DCE***, máquinas individuales son agrupadas en un ***cluster*** (división) autosuficientes (***stand-alone***) denominado ***cells***. Los ***cells*** son diseñados alrededor de dos conceptos básicos: máquinas dentro de un **cell** debería ser fácil administrar como un grupo, y las aplicaciones dentro de un **cell** debería poder encontrar la mayoría de los servicios que necesitan dentro de su propio **cell**. Por ejemplo, todas las operaciones administrativas como seguridad o recursos de nombramiento son ejecutadas en el nivel de **cells**. Además una compañía que tiene dos localizaciones en diferentes partes del país normalmente definiría un **cells** para cada localización. Porque es posible para las aplicaciones en un **cells** acceder a servicios en otro **cells** usando ***Servicios de Directorio Global*** [TAM96]. ***DCE*** ofrece gran escalabilidad.

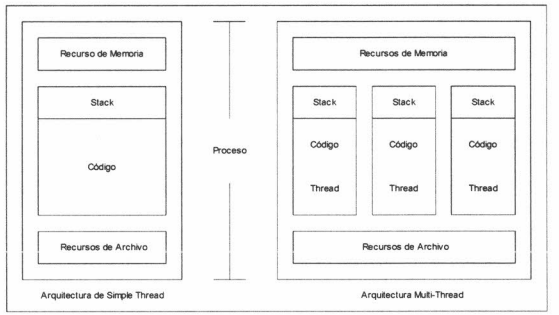


Cells

3.2.2. *Servicios Threads*

Sistemas basados en ***DCE*** deben poder ejecutar diferentes operaciones concurrentemente, por nivel de aplicación de usuario y de ***DCE*** en sí mismo. Por ejemplo, ***DCE*** necesita un monitor con varios puertos de comunicación diferentes por actividad, y las aplicaciones pueden querer ejecutar diversas operaciones en paralelo. Sobre varias plataformas, la unidad nativa de concurrencia es el ***proceso***, y los procesos son generalmente demasiados caros desde la perspectiva del recurso para soportar grandes números de actividades concurrentes. Por esta razón, ***DCE*** provee una implementación de ***threads*** que permite múltiples tareas para correr dentro de un simple proceso.

Los ***threads*** [GER99] ofrecen una perfomance excelente en hardware mono-procesadores y proveen aún mejor performance en plataformas multi-procesadores, como las máquinas con arquitectura SMP [PET94]. El ***servicio de threading*** es usado extensiblemente por ***DCE*** internamente y están disponibles para los programadores que necesitan obtener una máxima eficiencia y velocidad.

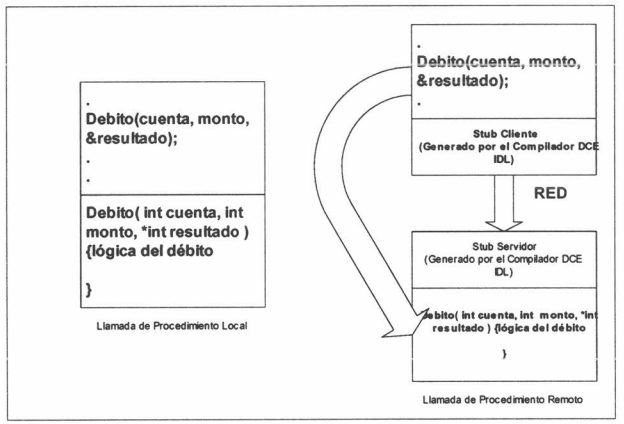


Servicios de Threads

3.2.3. *Llamada de Procedimiento Remoto (RPC)*

Todos los sistemas distribuidos necesitan una forma para los ***Clientes*** puedan comunicarse con los ***Servidores***. Dentro del ambiente ***DCE***, las comunicaciones Cliente – Servidor son ejecutadas con ***Llamada de Procedimiento Remoto*** (***RPC***). ***RPC*** [TRW98] se asemeja a una llamada de un procedimiento local por todo un lenguaje de programación estructurado. La única diferencia significante entre los dos es que la llamada del procedimiento remoto ejecuta el procedimiento llamado en un espacio de direccionamiento diferente, típicamente en una máquina diferente a la del Cliente.
Las funciones requeridas para llevar los pedidos y respuestas entre las máquinas son ejecutadas transparentemente por los componentes llamados ***stubs*** [TAM91]. Los ***stubs*** son generados por un utilitario del ***DCE*** llamado compilado de ***Lenguaje de Definición de Interfaz*** (***IDL***). Realmente, uno de los grandes beneficios del mecanismo ***DCE RPC*** es que los programadores que conocen como se construye una llamada de procedimiento en C [STR93], Pascal [NOR90] o Cobol todavía conoce la mayoría de los elementos para usar ***DCE RPCs***.

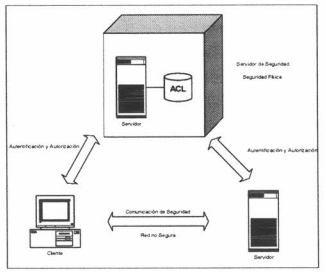
Un Cliente puede usar un servicio de directorio para acceder a un Servidor de su interés en particular en tiempo de ejecución, y el Cliente y Servidor puede usar el servicio de seguridad para garantizar los niveles deseados de autenticación, ,autorización, integridad y privacidad [TRW98]. ***RPC*** aísla al Cliente de los detalles de dónde el servidor está localizado en la red, tipo de hardware y sistema operativo en el cuál es ejecutado, las diferencias de la representación de datos entre la plataforma del Cliente y el Servidor y su particular transporte de red utilizado.



Llamada de Procedimiento Remoto Vs Local

3.2.4. *Servicio de Seguridad*

Con Bridge (puentes), routers (ruteadores) y monitores de red, es difícil asegurar los niveles de seguridad requeridos para el ambiente de red de computación en una organización hoy en día. ***DCE***, sin embargo ofrece servicios de seguridad basados en el modelo de Kerberos (Cliente y Servidor puede conocer quienes son) para la autenticación, autorización (Servidores pueden usar una lista de control de accesos para determinar si el cliente está autorizado para obtener el servicio dado)[PET94], integridad (un checksum [TAM91] que garantiza que la información es recibida como transmitida) y privacidad (DES – Data Encryption Standard – que protege información sensible durante la transmisión entre el Cliente y Servidor [TAM91]).

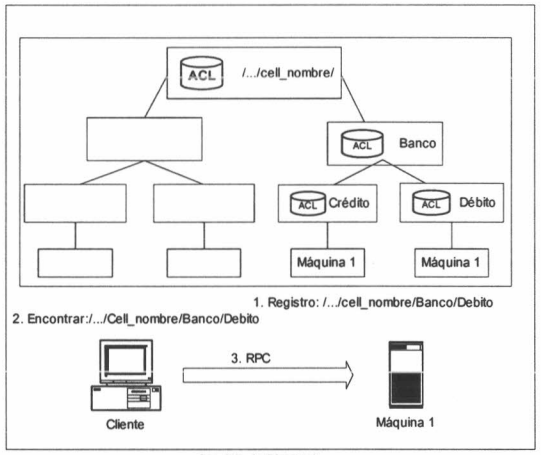


En un ambiente ***DCE***, el servicio de seguridad es usado para mantener la lista de usuarios y grupos, junto con sus privilegios. Obteniendo accesos a los servicios es un proceso de dos partes. Un cliente debe primero autentificar su identidad por un login en el ***servidor de seguridad*** con su nombre y clave. Una vez autentificado, los clientes ganan el acceso a las aplicaciones del servidor preguntando al servicio de seguridad que los provee con tickets para los servicios deseados. Los tickets no son autorizados a menos que el cliente tiene los privilegios apropiados almacenados en una ***lista de control de acceso*** que es administrada sobre el servidor de seguridad. Sofisticados mecanismos de criptografía prevén que los tickets sean falsificados. Porque los Servidores deben además proveer sus identidades para el Servidor de Seguridad y poder chequear los tickets, Clientes conocen que dialogan con los servidores reales y los Servidores conocen que dialogan con Clientes reales y autorizados.

DCE además contiene mecanismos incorporados para proveer funciones avanzadas de seguridad como paquetes de encriptamiento y pedidos de chequeos de datos. Y todos los servicios de seguridad mencionados anteriormente son usados directamente por ***DCE*** para proteger recursos como información de archivos y de nombre.

3.2.5. Servicio de Celdas (Cells) Directorio

En sistemas distribuidos a gran escala y complejos, los recursos (servicios de aplicaciones y archivos) pueden ser localizados en diferentes localizaciones físicas. Además, los recursos pueden ser replicados sobre múltiples máquinas. La localización es sujetas a cambios y puede cambiar un poco frecuentemente. Cualquier aplicación que usa una información de localización física embebida en el programa del Cliente rápidamente se convierte inmanejable.

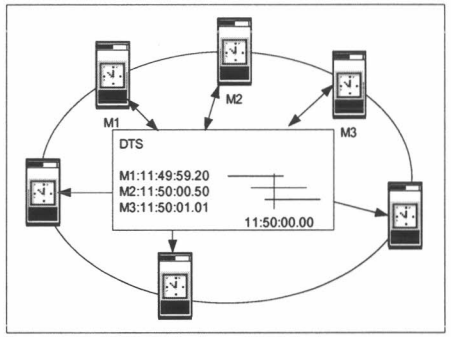


Servicio de Directorio

Para acceder a este ítem, ***DCE*** provee un ***Servicio de Celdas Directorio*** (CDS), que permite al Cliente buscar la localización del recurso por el nombre en vez del lugar. Más específicamente, ***CDS*** es una aplicación repositorio que corre dentro del ***cells*** y mantiene los nombre de la información en un ***árbol jerárquico***. Cuando una aplicación Servidor se inicializa, se ingresa el nombre y la localización del ***CDS***. Luego cuando un Cliente quiere llamar a la aplicación Servidor, pasa el nombre del servicio al ***CDS*** y este luego devuelve la dirección física.

3.2.6. *Servicios de Tiempos*

Con varios Clientes y Servidores diferentes involucrados en un sistema basado en ***DCE***, es importante para todas las máquinas de un mismo ***cells*** tengan un concepto común de tiempo. Administrar manualmente que los relojes estén sincronizados es impráctico. ***DCE*** direcciona esta problemática mediante el ***Servicio de Tiempo*** (DTS). ***DTS*** determina el valor de tiempo común sirviendo el reloj de un grupo de máquinas maestras previamente designadas y luego computa con el valor que más se ajusta.

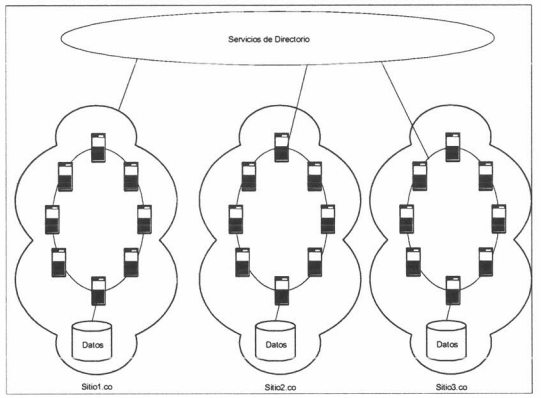


Servicio de Tiempo Distribuido

Este valor es usado luego por el ***DTS*** para acelerar o desacelerar el reloj en cada una de las máquinas dentro del ***cells*** hasta alcanzar la convergencia. Los relojes individuales nunca saltan hacia delante o hacia atrás, porque esto puede traer problemas con las aplicaciones que utilizan valores de tiempos para el login o sincronización. Una vez que está la convergencia, los relojes de cada una de las máquinas del ***cells*** son mantenidos dentro de 200 milisegundos como valor común de tiempo.

3.2.7. *Sistema de Archivo Distribuido*

Los datos compartidos pueden convertirse en un ítem considerable, en un ambiente distribuido geográficamente, ***DCE*** ofrece una alta performance, en un sistema de archivo distribuido llamado ***DFS*** para compartir información dentro de los ***cells*** o hacia otros ***cells***.

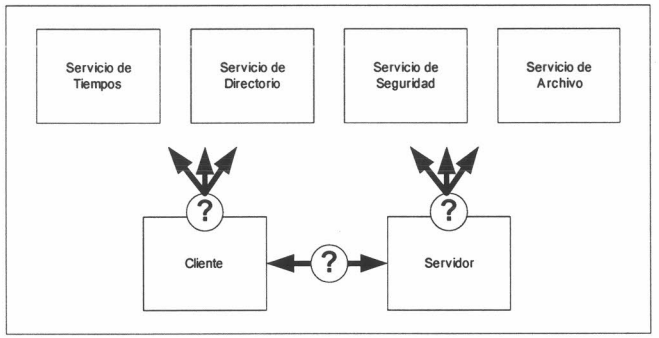


3.2.8. *Conclusiones*

Estas básicas explicaciones introducen el poder y funcionalidad de ***DCE***. Hay además un extensivo conjunto de APIs que los programadores pueden usar para escribir aplicaciones ***DCE***. Porque las APIs se mantienen consistentes a través de todas las plataformas e implementaciones, aplicaciones basados en ***DCE*** disfrutan de gran portabilidad, y simples aplicaciones pueden expandirse en plataformas de hardware de varios vendedores.

3.3. Alcance de DCE

Como es de esperar, el poder y la flexibilidad de ***DCE*** viene del costo de algunas complejidades. ***DCE*** fue diseñado para ser comprensivo en un conjunto de servicios para computación distribuida. Esto lo coloca con muchas limitaciones en lo que los programadores pueden hacer. Una vez instalado, los servicios ***DCE*** están disponibles en la red. Los programadores deben construir los clientes y servidores de aplicación, y requiere de decisiones acerca de los detalles como sintaxis de nombres, niveles de seguridad y programación.



Implicaciones del uso de DCE

Por ejemplo, cuando se hace un ***RPC*** hacia un Servidor, el Cliente primero debe realizar:

- Inicio como un Cliente ***DCE***
- Selección del protocolo de red
- Búsqueda de servicio destino en el ***CDS***
- Pedido de autorización desde el servicio de seguridad

Del lado del Servidor, hay además responsabilidades de programación. Por ejemplo, el proceso de un ***RPC***, el Servidor debe realizar:

- Inicio como un Servidor ***DCE***
- Ingreso de nombre en ***CDS***
- Validación del ***RPC*** entrante con el servicio de seguridad

Servidor debe manejar estos ítems como revalidación de sus propias credenciales de seguridad como la expiración o renombramiento de la información de nombres desde el ***CDS*** cuando se baja el Servidor. ***DCE*** no incluye aplicaciones orientada a estructuras para ejecutar o administrar aplicaciones una vez construidas. El soporte de funciones, como inicio, parada, replicación y administración de aplicaciones de usuario son dejados a los programadores que las construye.

DCE además no direcciona directamente la administración de fallas. Mientras que la administración de fallas suena como un simple concepto, las implicaciones de una

administración de fallas en un ambiente distribuido puede ser muy amplio. Puede extenderse desde reiniciar los servidores fallados hasta recuperar las fallas de comunicación. Por ejemplo, un sistema dónde el Cliente hace ***DCE RPCs*** hacia dos Servidores, cada uno pude actualizará el mismo dato en diferentes bases de datos.

Esto es muy común en una estructura a gran escala mediante el cuál accede a diferentes recursos y la estructura además tiene varios arquitecturas de objetos paralelas dónde el mecanismo de almacenamiento no es conocido por el objeto que realiza la llamada.

3.4. Comparación entre DCE y CORBA

En este capítulo se describe las diferencias entre ***DCE*** y ***CORBA*** en el esquema de diseño de aplicaciones Clientes y de Servidor, esquemas de comunicación, identificación de componentes, paradigma de programación, esquemas de pasaje de parámetros de funciones y procedimientos, tipificación de datos.

Se puede observar que la mayor diferencia que existe entre las dos arquitecturas (***DCE*** y ***CORBA***) radica que el primero fue diseñado para el ***soporte de programación en procedimiento*** mientras que la segunda para el soporte de ***programación orientada a objetos*** [WIR90].

También hay características que ***DCE*** se solapa con las capacidades de la programación orientada a objetos como encontrar componentes en tiempo de ejecución o definiciones de tipos de objetos.

3.4.1. Diferencia fundamental entre *DCE* y *CORBA*

Las diferencias fundamentales entre ***DCE*** y ***CORBA*** que ***DCE*** fue diseñado para soportar programación en procedimientos (procedural programming), mientras que ***CORBA*** fue diseñado para soportar programación orientada a objetos tiene las siguientes características:

- Encapsulamiento
- Abstracción
- Herencia
- Polimorfismo
- Creación nuevas clases en tiempo de ejecución
- Late binding
- Referencias de objetos
- Reusabilidad

CORBA soporta todas las características y estilos de programación descriptos anteriormente.

La programación por procedimientos en ambientes distribuidos como ***DCE*** soportan diferentes capacidades que las enunciadas. El acercamiento básico para distribuir un programa de procedimiento es:

- División de los datos de programa y las funciones que manipulan los datos en los Servidores
- Distribución de aquellos Servidores a través de múltiples Hosts
- Cambios de llamadas de función a ***RPCs***

Este estilo de programación que junta los datos y funciones en los Servidores, es la única manera de acceder a los datos a través de interfaces ***RPCs*** en los Servidores. Esto no protege cualquier dato dentro de un Servidor desde el acceso por cualquier función en el Servidor, sin embargo, no soporta abstracción, herencia, polimorfismo o un estilo dinámico de programación.

DCE tiene capacidades adicionales que comienzan con el solapamiento con capacidades tradicionales de los sistemas orientados a objetos:

Un Cliente ***DCE*** puede determinar en tiempo de ejecución un Servidor específico con el cuál quiere conectarse y hacer ***RPCs*** aunque sea fija el soporte de interfaces sea definido en tiempo de compilación [SHI92].

Se puede definir en el Servidor diferentes recursos que pueden ser administrados por él mismo. Esto se denota ***identificadores Unicos Universales*** (UUID) [SHI92].

Lo que permite ***UUID*** es una forma de abstracción y polimorfismo, pero no es programación orientada a objetos aunque se identifique las características inherentes al paradigma.

3.4.2. Capacidades Individuales

En este punto describiremos individualmente las características de cada una de las arquitecturas.

DCE soporta varios tipos de datos [APR93] que no soporta ***CORBA***:

- Arreglo de tamaño fijo, pero en ***CORBA*** se puede obtener un comportamiento similar pero no hay un equivalente.

- Permite pasajes parámetros largos en bloques mediante una sola operación, en cambio ***CORBA*** se puede implementar mediante una serie de operaciones, simulando que es una sola operación conceptualmente.

- ***DCE*** soporta contextos [APG93], que es un mecanismo para mantener el estado del Servidor durante una serie de pedidos relacionados lógicamente para un Cliente. En ***CORBA*** no hay mecanismo que se corresponda, el programador es el responsable de administrar la información de contexto explícitamente.

- ***DCE*** utiliza punteros dentro de los parámetros de las operaciones. En ***DCE-IDL*** se puede tener un puntero como parámetro. ***CORBA*** no soporta punteros. El conjunto de tipos de datos básicos de ***CORBA-IDL*** [SIE96] y la construcción de tipos de datos complejos no incluye punteros. La complejidad se puede tratar dado que se pueden componer estructuras mediante objetos.

- ***CORBA*** soporte el tipo de dato ***ANY*** mientras que ***DCE*** no lo soporta. Esto permite pasar cualquier valor arbitrario entre el Cliente y el Servidor.

- ***DCE-IDL*** no soporta herencia de interfaz, mientras que ***CORBA-IDL*** soporta herencia múltiple.

- ***CORBA*** define un repositorio de interfaz, visto en el capítulo 2, que contiene la información equivalente a los archivos IDL y pueden ser consultados en tiempo de ejecución. ***DCE*** no define este repositorio.

- ***DCE*** no define una interfaz de invocación dinámica.

Los servicios que provee ***DCE*** son más limitados comparados a los componentes de la arquitectura ***CORBA*** y provee a los desarrolladores un conjunto más rico de capacidad para construir aplicaciones.

3.4.3. Madurez de especificaciones

Claramente, las especificaciones ***DCE*** están más completa que las especificaciones de ***CORBA***, pero dado que hoy en día la ***OMG*** está desarrollando aceleradamente la normalización que se extiende más allá de un modelo de arquitectura, sino que establece estándares en diferentes dominios de la industria, esto hace que sea más amplia su aceptación y seguimiento por las diferentes organizaciones.

3.5. Conclusiones

Se enunciaron las características más importantes entre ***DCE*** y ***CORBA***, pero la fundamental es el paradigma de programación: ***DCE*** fue diseñado para soportar programación distribuida de procedimientos, mientras que ***CORBA*** fue diseñado para soportar programación orientada a objetos.

Tomando las capacidades individuales de cada una de las arquitecturas para el desarrollo es más rica ***CORBA***, que provee un ambiente más poderoso y se extiende también a estandarización de dominios tanto sea verticales como horizontales permitiendo no solo definir como construir aplicaciones sino también como dialogar aplicaciones con la misma regla de negocio pero diferenciando su comportamiento.

Hay una aceptación que se va incrementando día a día sobre ***CORBA*** debido a su característica de ser implementado orientado a objetos, y dado la llegada el año

2000, permite que programas realizados en Cobol en diferentes arquitecturas y sistemas operativos no sean obsoletos y permiten acceder mediante llamadas a objetos que ponen una máscara sobre estas aplicaciones en forma transparente.

Capítulo 4: RMI y CORBA

En este capítulo se discutirá sobre el modelo ***RMI*** (Remote Method Invocation – Invocación de Método Remoto) [JAV99] y ***CORBA***. Ambas tecnologías soportan la construcción de aplicaciones *Cliente – Servidor* en ambientes distribuidos. A diferencia de los modelos vistos hasta ahora como ***DCE*** y ***CORBA*** estos permiten trabajar en ambientes heterogéneos mientras que ***RMI*** corre en ambiente ***Java*** [OBJ98].

Este capítulo comienza con una introducción general de la similitud de las dos arquitecturas y continúa con una descripción de ***RMI*** junto con su arquitectura conceptual. Para finalizar con presentar las diferencias más importantes con ***CORBA***.

Se estudiará las siguientes características:

- Lenguaje de implementación
- Plataforma
- Estándar
- Definición de Interfaz
- Unicidad de Objetos
- Facilidad de Desarrollo
- Escalabilidad

3.4. *Introducción General*

RMI provee una forma para que las aplicaciones *Cliente – Servidor* puedan invocar métodos a través de una red distributiva de ***Clientes*** y ***Servidores*** corriendo la ***Máquina Virtual de Java*** [JAV99].

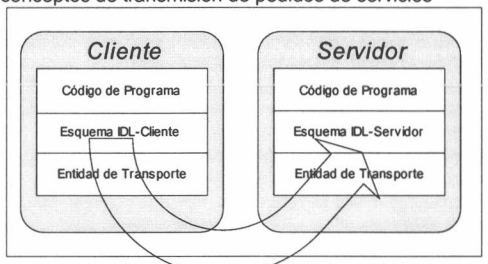
Mientras que las distintas organizaciones buscan posicionarse en tecnologías competitivas, la realidad es que las tecnologías de objetos distribuidos son aproximadamente similares, con diferencias especificas dependiendo sobre la tecnología principal y propietaria (legacy systems) [OBJ98] que dispone la organización.

Ellas proveen un bus (transporte) de objetos distribuidos, dando la posibilidad de descubrir en tiempo de compilación o ejecución propiedades, métodos y eventos en objetos remotos.

Si analizamos cada uno de los niveles, ***RMI*** y ***CORBA*** presentan los mismos componentes (como se presentó en el capítulo anterior):

- Lenguaje de Definición de Interfaces
- Almacenamiento de IDL en Stubs de Cliente y Servidor
- Secuencia de llamadas de pedidos de Cliente a Servidor

- Mismos conceptos de transmisión de pedidos de servicios



Pedido de Servicio entre Cliente - Servidor en RMI o CORBA

A pesar que ***RMI*** es considerado menos robusto y menos poderoso que ***CORBA*** (***RMI*** está implementado únicamente en Java), ***RMI*** todavía provee una lista de algunas características únicas, como la distribución, administración automática de objetos y la habilidad de pasar objetos entre ellos de una máquina a otra máquina. A continuación se visualiza los componentes básicos de la arquitectura ***RMI***.

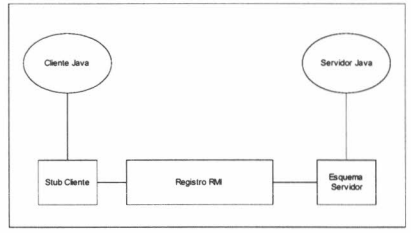
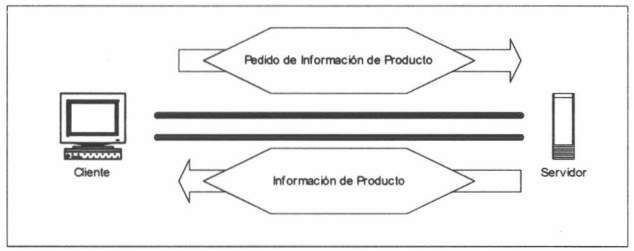


Diagrama RMI

El stub del Cliente y el esquema del Servidor son creados por un objeto de interfaz común. La diferencia entre los dos componentes es que el Cliente está conectado al ***Registro RMI*** [JAV99] mientras que el *esquema del Servidor* está unido por las operaciones del método activo.

3.5. Objetos Remotos: El rol del Cliente y Servidor

Como se enunció en capítulos anteriores uno quiere capturar información localmente en una computadora Cliente y enviar la información a través de una red a una computadora Servidor.



Transmisión de Objetos entre un Cliente y un Servidor

La transmisión entre el Cliente y el Servidor se puede utilizar una conexión ***socket*** [PET94] que envía una cadena de bytes entre el un extremo a otro. El programador para lograr el envío de un extremo a otro, y el receptor interprete la cadena de información transmitida, necesita tener las apropiadas formas de codificar los datos y el protocolo de transmisión entregar los datos.

En los programas de Java, es natural implementar el pedido y la respuesta en objetos. Usando ***RMI***, pueden ser transportados como objetos de un extremo a otro.

Además permite tener colecciones de objetos heterogéneos y cada uno de ellos sabe como responder a cada uno de los pedidos y con quién colabora.

Otro de los beneficios, es que al invocar métodos sobre objetos que residen en otras computadoras sin importar como mover estos objetos de un lado a otro. Estos métodos se denominan **invocación remota** [COR97].

El concepto central en la implementación de objetos remotos en Java es ***invocación método remoto*** o ***RMI***. El código sobre una computadora Cliente invoca un método sobre un objeto en el Servidor. Es importante darse cuenta que la terminología Cliente / servidor se aplica solamente a la llamada de un ***simple método***. La computadora que corre el código Java que llama al método remoto es el Cliente de esa llamada y la computadora que reside el objeto que procesa la llamada es el Servidor para esa llamada. Es posible que los roles se inviertan en otras circunstancias, por ejemplo que Servidor de la llamada previa puede haberse convertido en Cliente cuando invoca el método remoto sobre un objeto que reside en otra computadora.

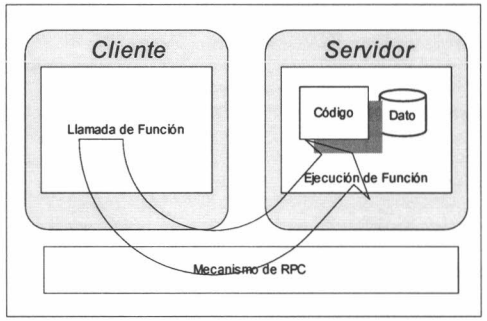
3.6. Examinando RMI

De acuerdo a lo visto hasta ahora los sistemas distribuidos necesitan realizar cálculos en diferentes espacios de dirección, y requieren de comunicación entre servidores.

Se presentará las diferencias entre el uso de ***sockets*** [LEN94] y ***RPC***.

En Java [JAV99], soporta ***sockets***, que da las facilidades de comunicación a un nivel general. Se necesitan que tanto el Cliente como el Servidor participen en el nivel de aplicación del protocolo para codificar y decodificar mensajes.

Una alternativa a los ***sockets*** es el ***RPC*** [TAM91], que permite la abstracción de comunicación al nivel de una llamada de procedimiento. Se realiza la llamada de un procedimiento local y es empaquetado y llevado al destino remoto.



RPC maneja la llamada de función. El código y el dato residen en el mismo lugar.

RPC, sin embargo, no traduce bien en un sistema de objetos distribuidos, dónde la comunicación entre los objetos que residen en diferentes espacios de dirección. Hay que pensar que solamente se manejan objetos y no procedimientos. Para lograr la conversación entre objetos remotos, mediante la invocación de la semántica, se requiere de ***invocación de métodos remotos***.

Los objetos tendrán su ***stub***, mediante el cual pueden administrar el objeto remoto mediante el cual necesitan establecer la comunicación.

El sistema de invocación remota de Java descrito en esta introducción ha sido específicamente diseñado para operar en un ***ambiente Java***. Mientras que otros sistemas pueden adoptar el manejo de objetos Java, estos sistemas no logran una gran integración con sistemas Java, debido requieren inter-operabilidad con otros lenguajes.

En ***CORBA*** es un ambiente de multilenguaje y heterogéneo, y debe tener un ***lenguaje neutral*** de modelo de objetos. En cambio, en Java, ***RMI*** asume un

ambiente homogéneo en la ***máquina virtual de Java*** y el sistema puede seguir el modelo de objetos de Java cuando se posible. Esta es la diferencia fundamental entre estas dos tecnologías, ***RMI*** y ***CORBA***.

3.7. Objetivos del Sistema

Los objetivos para soportar objetos distributivos en el lenguaje Java son [JAV99]:

- Soporte invocación remota sobre los objetos en diferentes máquinas virtuales
- Integración del modelo de objetos distribuido en Java de una manera natural manteniendo la semántica de los objetos Java
- Diferenciación de apariencia entre el modelo de objetos distribuido y el modelo de objeto local de Java.
- Implementación minimizada en complejidad tanto para el Cliente como el Servidor. El primero usa el objeto remoto y el segundo lo implementa
- Preservación de seguridad provisto por el ambiente de ejecución de Java

En general, lo que se busca realmente como un requerimiento general es que el modelo ***RMI*** sea simple de usar y se adecue bien dentro del lenguaje.

RMI tiene otras propiedades:

- Garbagge collection [SMA92] para objetos remotos
- Replicación de Servidor
- Activación de objetos persistentes para servir a una invocación

Esto trae transparencia al Cliente y agrega mínimos requerimientos de implementación sobre el Servidor. A esto se debe agregar las siguientes propiedades que debe dar soporte:

- Mecanismos de varias invocaciones
- Referencias semánticas para objetos remotos
- Garbage collection distribuido para objetos activos
- Capacidad de soportar múltiples transportes

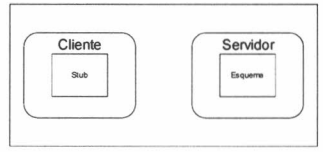
Estas condiciones tiene que cumplirse para lograr comunicación entre objetos remotos a través de distintas aplicaciones en Java en una forma transparente y segura, respetando las propiedades ofrecidas por el lenguaje.

3.8. Detalle de Implementación RMI

A igual que ***CORBA***, el sistema ***RMI*** tiene una estructura de diálogo entre el Cliente y el Servidor similar.

RMI es la acción de invocar un método de una interfaz remota sobre un objeto remoto. Más importante, la invocación del método sobre el objeto remoto tiene la misma sintaxis como la invocación del método sobre un objeto local.

RMI está basado en un modelo Cliente – Servidor. Las aplicaciones respectivas residen en el Cliente y en el Servidor. El Cliente se comunica con el sistema ***RMI*** a través del ***stub***. El Servidor se comunica con el sistema ***RMI*** a través del esquema [JAV99].



La comunicación entre el Cliente y el Servidor es realizada mediante ***sockets*** (esto es transparente para el programador).

Una aplicación que usa ***RMI***, primero hace contacto con el objeto remoto encontrándolo en un registro sobre el Servidor. El registro vuelve al stub del objeto que es conectado al objeto remoto [JAV99]. Este stub puede ser usado para manipular el objeto remoto, aunque el stub parece ser local.

Cuando un método remoto es ejecutado, los argumentos son codificados en tiempo de ejecución y enviados sobre la red hacia el Servidor. El Servidor decodifica los argumentos, invoca el método, codifica el resultado, y los envía devuelta. El Cliente luego decodifica el resultado recibido si hay algún dato que se devuelve.

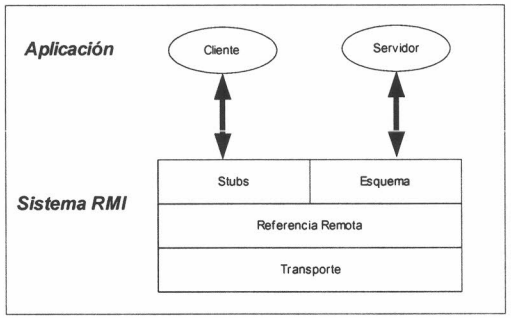
Para entender cada uno de los componentes, se verá en la siguiente sección, la arquitectura del Sistema y la interacción de cada una de sus niveles.

3.6. Arquitectura RMI

El sistema ***RMI*** consiste de tres niveles:

- Stub y Esquema
- Referencia remota
- Transporte

Los límites de cada nivel está definido por su interfaz específica y protocolo, cada nivel es independiente del próximo y puede ser reemplazado por una implementación alternativa sin afectar los otros niveles en el sistema [JAV99].



Arquitectura del sistemaRMI

El nivel de ***stub*** se corresponde del lado del Cliente y del ***esquema*** al Servidor, el nivel de ***referencia remota***, permite el comportamiento de referencia remota (invocación a un simple objeto o al objeto replicado) y el nivel de ***transporte*** permite la inicialización, administración y seguimiento del objeto remoto.

Una invocación del método remoto desde un Cliente a un objeto del Servidor remoto viaja sobre hacia abajo en el Sistema de ***RMI*** al nivel de transporte y luego sube desde el nivel de transporte del lado del Servidor.

Un Cliente invoca un método sobre un objeto de un servidor remoto mediante un ***stub*** para el objeto remoto como un conducto al objeto remoto. El Cliente toma la referencia hacia el objeto remoto que está referenciado al stub local

Los métodos del ***stub*** sobre el Cliente construye un bloque de información que consiste en:

- Identificación del objeto remoto a ser usado
- Número de operación, que describe el método a ser llamado

- Disposición de parámetros

Luego se envía esta información al Servidor. Sobre el lado del Servidor, hay un ***objeto esquema*** [COR97] que le da sentido a la información que contiene el paquete enviado. Específicamente el ***esquema*** ejecuta cinco acciones para todas las llamadas de métodos remotos:

- Captura de parámetros
- Llama al método deseado sobre el objeto real remoto que reside sobre el Servidor
- Captura el valor de retorno o la excepción de la llamada sobre el Servidor
- Disposición del valor
- Envía el paquete devuelta al stub del Cliente.

La ***referencia remota*** se encarga del nivel más bajo de interfaz de protocolo. Esta etapa es además responsable de llevar la referencia remota específica de protocolo que es independientemente del ***stub*** del Cliente y ***esquema*** del Servidor.

El ***transporte*** en el sistema ***RMI*** es responsable por:

- Configuración de conexiones en los espacios de dirección remota
- Administración de las conexiones
- Monitorización de las conexiones activas
- Captura de las llamadas entrantes
- Mantenimiento de la tabla de objetos remotos que reside en el espacio de dirección
- Configuración de las conexiones de las llamadas entrantes

3.7. Construyendo Aplicaciones RMI

Hay tres pasos que el programador debe hacer para construir aplicaciones ***RMI***:

- Definición las interfaces remotas
- Implementación de las interfaces por las clases creadas
- Creación de las clases de ***stub*** y ***esquema*** para las clases implementadas

Todos los métodos tienen que ser declarados como parte de la interfaz para ser ejecutados remotamente. Esta interfaz no declara cualquier método como propio, sino que es usado para identificar todos los objetos remotos.

Hay dos clases que deben ser creadas que son las clases ***stub*** y ***esquema***. Estas proveen la interfaz con los programas de aplicaciones. La clase ***stub*** son las imágenes del lado del Cliente de las clases de objetos remotas. Las clases de ***esquema*** son las instancias del lado del Servidor que son los métodos remotos hacia el Cliente. ***Stubs*** implementa las mismas interfaces como las clases remotas y envía los métodos invocados sobre sus instancias hacia sus correspondientes

instancias remotas. El esquema intercepta el método remoto requerido desde el Cliente y luego llama al actual método sobre la instancia de implementación. Luego este captura el valor de retorno y envía el resultado de vuelta al ***stub*** del Cliente.

3.8. Conclusiones

RMI y ***CORBA*** tiene similares características y capacidades como también algunas diferencias. Esta sección resume algunas de las similitudes y diferencias para ayudar a entender estas tecnologías mejor y hacer la elección más fácil entre ellas.

- ***RMI*** es una solución íntegra de Java para objetos remotos, aportando todas las ventajas del lenguaje Java [JAV99] (escribir una vez, corre en cualquier lado – write once, run anywhere). Los Servidor y Clientes desarrollados con ***Java RMI*** pueden ser desarrollados en cualquier lugar de la red sobre cualquier plataforma que soporta ***ambientes de ejecución Java***. ***CORBA***, en contraste, está basado en un estándar de la industria para la invocación de objetos remotos escritos en cualquier lenguaje de programación. Como resultado, ***CORBA*** provee la forma de conectar aplicaciones ***legacy*** que todavía son utilizados en importantes sectores de la organización, pero son escritos en cualquier lenguaje que no sean ***Java***.

- ***RMI*** y ***CORBA*** actualmente utilizan diferentes protocolos para comunicarse entre objetos sobre diferentes plataformas. ***CORBA*** utiliza ***IIOP*** que comparte todos los ***ORBs*** compatibles con objetos ***CORBA***. ***IIOP*** permite a los objetos estar en diferentes plataformas y ser escritos en diferentes lenguajes que interactúan de una manera estándar. ***RMI*** utiliza actualmente ***Java Remote Messaging Protocol*** (JRMP) [COR97], que es un protocolo desarrollado específicamente para objetos de remotos Java.

- ***CORBA***, el cliente interactúa con los objetos remotos por ***referencia***. El Cliente nunca obtiene una copia del objeto servidor que es propio en tiempo de ejecución, sino utiliza el ***stub*** en ejecución local que manipula el objeto servidor que reside sobre la plataforma remota. A diferencia de ***RMI***, permite que el Cliente interactúe con un objeto remoto por referencia o capturarlo y manipularlo en el ambiente de tiempo de ejecución local. Esto es porque todo los objetos ***RMI*** son objetos Java. ***RMI*** utiliza la serialización de los objetos desde el Servidor hacia el Cliente. Los objetos ***CORBA*** no pueden tener la ventaja de la característica del lenguaje de programación Java (write once, run anywhere).

Para versiones futuras, anuncia la capacidad que ***RMI*** pueda usar el protocolo ***IIOP*** para comunicarse con objetos remotos ***CORBA***. Y en futuras versiones de especificaciones ***CORBA*** va a incluir protocolos para pasar objetos por valor.

Con ***CORBA*** no estamos obligados a utilizar únicamente el lenguaje de programación ***Java*** y se tiene más posibilidades de implementaciones y capacidades desde diferentes vendedores para llegar a la solución deseada.

Además, no hay una respuesta a la pregunta sin conocer las consideraciones de negocio de la organización que decide que tecnología utilizar. La vasta mayoría de las decisiones son dominadas por las consideraciones de negocios y no por las consideraciones técnicas, aunque en este capítulo se estableció claramente las diferencias técnicas de estas tecnologías.

Capítulo 5: Ejemplo: Arquitectura de un CPR sobre Objetos Distribuidos

Se presentará una arquitectura de objetos distribuidos, mostrando la integración de un CPR [VEM98] (registros de pacientes computarizados) en una institución de salud [SES98].

Mediante distintas tecnologías incluyendo ***CORBA*** permite la integración de modelos de sistemas de información sobre una plataforma independiente. En las Instituciones de Salud existen diferentes departamentos con reglas de negocios que afectan al área asistencial y administrativa, en consecuencia en la mayoría de los casos no existen aplicaciones en toda la organización del mismo vendedor; por lo tanto la necesidad de integración de la información en las diferentes aplicaciones es crítica para la toma de decisiones.

A continuación se describe una estrategia de integración, con el fin de mostrar el análisis tanto de tecnología como de los conceptos anteriormente mencionados, logrando así, una interacción y comunicación transparente, de datos y aplicaciones.

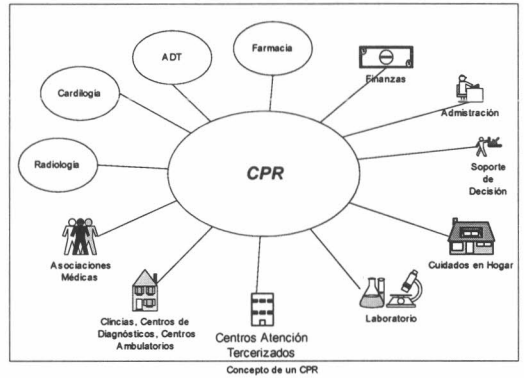
5.1. Introducción

En las instituciones de salud, desde el punto de vista de su organización, necesitan definir un modelo corporativo de datos como de aplicaciones.

Un registro de paciente computarizado es una colección de documentos electrónicos que forman la historia de salud longitudinal de una persona. Estos documentos incluyen datos en múltiples formatos, por ejemplo, imágenes, gráficos, textos, etc. La institución de salud, también posee aplicaciones independientes que resuelven determinados problemas en cada uno de sus departamentos de su organización, dando así múltiples repositorios datos para la información del paciente tanto clínica como administrativa. La adquisición, identificación y presentación de estos datos son las claves para la definición de un modelo corporativo, que integre distintas sistemas operativos, administradores de base de datos y redes de comunicación en una forma transparente y consistente.

Un ***CPR*** se convierte en el nexo que permite la unión de toda la organización. Más allá de la unión que tiene que cumplir la arquitectura de un ***CPR*** y los aspectos humanos, hay otros puntos que son importantes:

- Informática clínica, telecomunicaciones, ingeniería de administración y sistemas de información son parte fundamental para lograr la implementación de un ***CPR***
- Participación y cooperación de las distintas disciplinas a través de toda la organización



Concepto de un CPR

En 1998, el ***CPRI*** (Instituto de Registro de Paciente Computarizado) dio a conocer y actualizar la definición de un ***CPRs*** [CPR98]:

*Un registro computarizado de paciente (**CPR**) es información mantenida electrónicamente acerca de los cuidados y estados de salud de la vida de los individuos. **CPRs** no solamente una automatización de registros médicos basados en papel, sino que envuelve el alcance completo de la información de salud en todos los medios de almacenamiento. **CPRs** incluye historia médica, medicaciones actuales, resultados de exámenes de laboratorio, imágenes de rayos X, etc.*

CPR debe contener información que encuentre las necesidades de las múltiples disciplinas dentro de la organización. La información existente no puede estar más aislada en cada uno de los departamentos, en consecuencia se vuelva a un modelo corporativo: dato longitudinal, completo y libre de error.

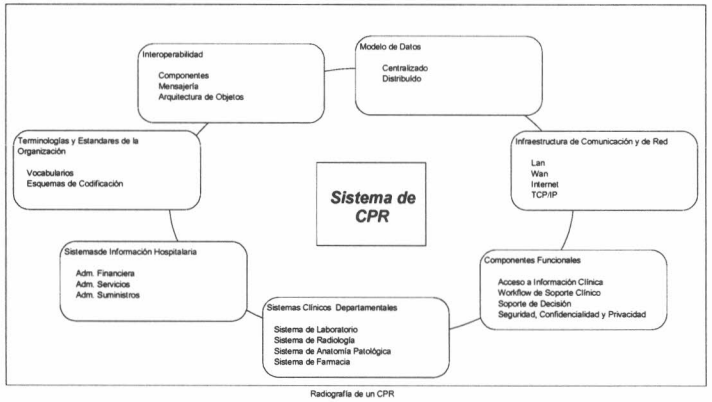
Un ***CPR*** es un ***sistema de sistemas***, y no un solo sistema.

5.2. Registro Computarizado de Pacientes (CPR)

Para entender el rol de un ***CPR*** en una organización de salud, es necesario entender que un ***CPR*** es un ***sistema de sistemas***. Una definición útil de un ***CPR*** viene desde el ***CPRI*** [CPR98]. El ***CPRI*** fue creado como resultado de un estudio del Instituto de Medicina sobre el uso de la tecnología de información en salud en 1991. Generalmente, las aplicaciones de sistemas Clínicos forman parte del ***CPR***. Basado sobre las discusiones de los miembros, se extiende la siguiente definición del ***CPR*** [CPR98]:

*Un sistema **CPR** facilita la captura, almacenamiento, procesamiento, comunicación, seguridad y presentación de información de salud no redundante. No son bases de datos masivas, sino sistemas de computación independientes en sitios de cuidados individuales, el cual con mínimos requerimientos de conectividad y seguridades apropiadas, para acceder a los datos específicos desde cualquier sistema sobre la autorización de paciente. Un sistema de **CPR** provee la disponibilidad de los datos de pacientes completos y sin errores, recordatorios y alertas clínicos, soporte de decisión, y uniones a cuerpos de datos relacionados y bases de conocimientos. Un sistema **CPR** puede advertir a los médicos cuando hay una alergia cuando se está prescribiendo un medicamento, puede proveer la última modalidad de tratamiento y puede organizar volúmenes de información acerca de la condición de pacientes crónicos.*

Las instituciones de Salud, pueden tener varios centros de atención, en diferentes localizaciones geográficas, creando así, un ambiente ***Multi-Institución***, obligando así que un ***sistema de CPR*** tenga que ser rápidamente escalable como el crecimiento de cada una de sus aplicaciones, y tanto los datos locales como remotos. El papel de las comunicaciones y la integración de los datos y aplicaciones va a ser vital para mejorar la eficiencia y brindar el acceso a la información clínica (puede residir en distintos repositorios de datos y localizaciones), en forma transparente.



Radiografía de un CPR

Es evidente que los sistemas de información es una parte integral para llevar a cabo el negocio (al menos en cualquier modelo de negocios) [HUC95], pero no explica como el negocio es realizado, dado que pueden existir distintas estrategias, y los sistemas tiene que adaptarse a los diferentes cambios.

El objetivo de toda institución de salud, sobre la implementación de un ***CPR***, es proveer un mecanismo de captura, administración y presentación de la información requerida a través de la atención de una manera optimizada, dado que intervienen diferentes aplicaciones, y los datos pueden estar en diferentes formatos y repositorios de datos a lo largo de toda la organización.

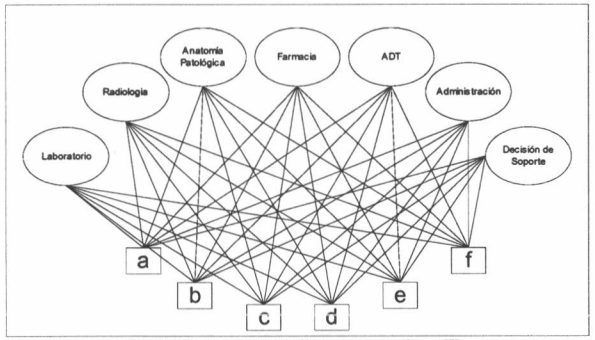
Un ***CPR***, tiene que tener un comportamiento unificado, brindando la información de los sistemas departamentales asistenciales (farmacia, laboratorio, etc.) y administrativos (contabilidad, tesorería, etc.).

Cada uno de estos sistemas trabajan con sus propios datos por los cuales son responsables en su administración. El desafío es lograr el intercambio de datos y transacciones para cada uno de los departamentos logrando un nivel de transparencia en toda la organización.

Dado que cada departamento de la institución tiene definidos sus procesos y en ciertos casos en los procesos pueden intervenir más de un departamento, es importante asegurar una visualización funcional que asegure la integración.

Las funciones a encapsular pueden ser:

a. Información demográfica de paciente
b. Cobertura de atención de pacientes
c. Ordenes de exámenes médicos, medicaciones y tratamiento terapéutico
d. Información clínica, asistencial y administrativa
e. Información externa (datos no generados por sistemas de la organización, sino obtenidas de terceros)
f. Brindar información de alertas y recordatorios sobre información de pacientes que es generado a lo largo de la organización



Relación de Funciones de los Sistema Clínicos a Integrar en un Sistema CPR

Cada una de estas funciones establecen una relación de muchos a muchos con los departamentos de la institución. Viendo esta relación es necesario contar con una arquitectura que satisfaga el modelo de negocio en una forma transparente.

5.3. Arquitectura de un CPR distribuido

Para lograr la integración, se necesita tener una arquitectura abierta y un sistema orientado a objetos o componentes, el cuál provee métodos de accesos a los datos para cada proceso que se define en la organización.

Con el objetivo de proveer la información de los datos definidos en las funciones presentadas anteriormente, se tiene que construir una arquitectura por la cual cada aplicación definida en cada una de los departamentos brinde la información requerida para cada proceso de negocio, estableciendo así un ***repositorio de datos distribuido*** brindando información básica tanto clínica como administrativa.

Cada una de las aplicaciones que corren en cada departamento tienen repositorio de datos propios, y existe la necesidad de intercambio de datos con el fin de que sea definido de una forma única y no esté repetido en cada área, dando así una inconsistencia en cada momento se requiera de un dato en particular.

Es necesario disponer de un sistema orientado a funcionalidades a través de toda la organización para que brinde la información en forma adecuada y consistente. Cada departamento (ejemplo: ***Laboratorio***) va a brindar métodos de acceso y

manipulación de datos generados dentro de él mediante el encapsulamiento de las funcionalidades que brinda (ejemplo: ordenes de laboratorio, reportes de laboratorio, etc.), mediante la implementación de ***componentes*** u ***objetos***.

El concepto anterior, sobre la utilización de componentes u objetos en cada uno de los departamentos, se puede definir como ***método centralizados en departamentos o funcionalidades***, pero también se puede definir el ***método centralizado en funciones genéricas***, dado que la misma función genérica se puede utilizar en cada departamento, logrando así la colaboración de objetos y componentes y maximizar el desarrollo de cada una de las funciones a través del encapsulamiento de funciones genéricas a ser utilizada por cada ***objeto y componente departamental*** (ejemplo: entrada de órdenes, reporte de resultados, etc.). Los beneficios y características de la orientación a objetos (Clases, herencia y polimorfismo), permite la comunicación y colaboración de objetos dentro de una arquitectura estándar, sin importar como se realiza cada función o como está implementada las mismas.

La ventaja de definir las funciones por dominio o departamento trae la oportunidad de utilizar sistemas existentes o nuevos desarrollos, siempre y cuando permita la metodología de integración usada para soportar las características que un ***CPR*** debe cumplir, brindando:

- Distribución

Soportar un modelo de objetos de datos que es distribuido en toda la organización en una red local o regional.

- Plataforma independiente

Cada aplicación puede correr en distintas plataformas, desde mainframes hasta servidores, PCS y sistemas operativos.

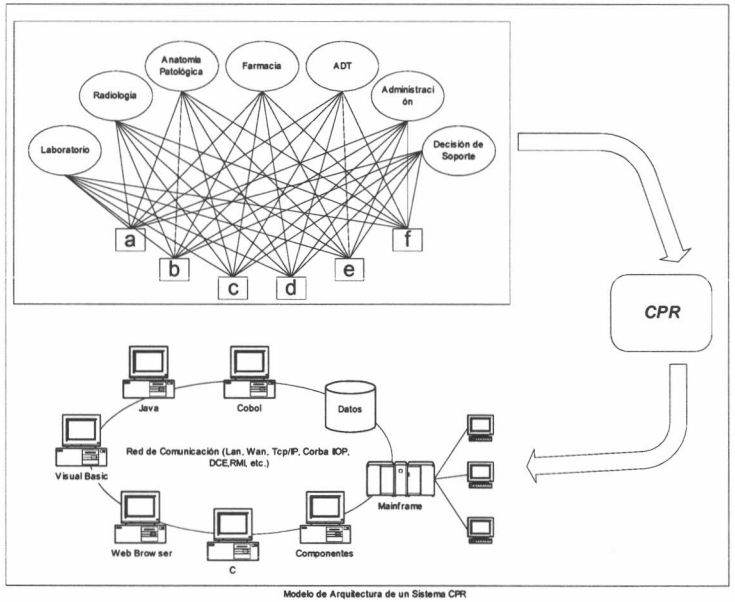
- Heterogeneidad

Diferentes tipos y clases de plataformas de diferentes vendedores.

CORBA es un sistema abierto y multiplataforma el cuál mejor se adapta a estos requerimientos. Logrando así una comunicación de componentes y objetos de forma transparente a través de ***ORB***. Cada requerimiento tiene que ser definido en una interfaz estándar para ser soportado en cualquier ambiente distribuido que se defina.

Con el objetivo de utilizar todas las aplicaciones existentes de la organización, para maximizar su efectividad, un ***CPR*** debe permitir ser visualizado desde cualquier tipo de plataforma de computación, desde una estación clínica hasta una PC. Consecuentemente, se puede pensar tanto en ***clientes delgados*** (thin client), permitiendo separar las ***reglas de negocio*** de la presentación de la información (***three tier***) [HAR97] o tener ***clientes pesados*** que tienen la presentación y la regla

de negocio en un solo componente. De esta manera se pueden generar una interacción de distintas aplicaciones como ***legacy systems***, applet de Java, browsers, aplicaciones en cobol, en visual basic, etc., dando productividad a la organización para lograr el modelo corporativo integrado.



Modelo de Arquitectura de un Sistema CPR

Las ventajas de estas características enunciadas son:

- Disponibilidad de utilización de los sistemas de información ***legacy*** para visualizar el ***CPR***. Esto permite obtener la visualización de la información requerida en estaciones de trabajo orientadas a la especialidad del usuario.

- Disponibilidad de utilizar PCS a bajo costo y redes de arquitecturas abiertas para distribuir y visualizar el ***CPR*** a través de la organización.

- Disponibilidad de utilizar el ambientes de computación existente en la organización lo más posible para soportar las funciones del ***CPR***.

Se observa la definición de una modularización independiente de cada una de las aplicaciones o accesos a repositorios de datos de la organización, obteniéndose para cada una de ellas un bus de comunicación mediante el cual van a dialogar,

llevando así a un esquema de componentes u objetos, por el cuál lleva a la definición de interfaces independientes, con el objetivo de dar un ambiente de comunicación, integración e interacción independiente, sin importar ***como*** resuelve la regla de negocio requerida de cada uno de los elementos que componen la red corporativa.

5.4. Integración de Componentes de un CPR

Una consideración clave es como realizar la integración de cada una de las aplicaciones y accesos a los datos de toda la organización en el modelo corporativo para lograr así un sistema de ***CPR***, con información consistente y sin ambigüedad, sin atentar con la modularización y la independencia de los dominios que la componen.

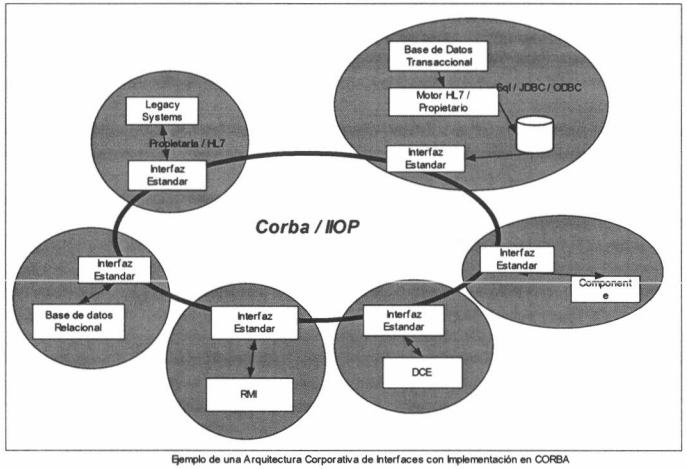
Toda regla de negocio tiene dominios de datos, por lo tanto se tiene que definir interfaces estándar sin preocuparse de ***cómo lo hace***, otorgando así la libertad de la implementación de la solución.

Todas las aplicaciones y repositorios de datos de la organización se comportarán como componentes u objetos, estableciendo claramente la definición de acceso al comportamiento deseado, sin importar la forma en que resuelve la regla del negocio.

La situación ideal es que todas las aplicaciones y reglas de negocio estén definidas en un solo repositorio de datos, en un mismo sistema operativo, en un mismo protocolo de comunicación, etc., pero en la realidad no ocurre; entonces está la necesidad de definir interfaces estándar para acceder a los componentes dialogando sobre un bus de comunicación estándar. Esto permite mediante las interfaces, que los componentes u objetos sean implementados sobre cualquier plataforma y lenguajes de programación, teniendo así un modelo corporativo de múltiple plataforma y múltiples repositorios de datos.

Los componentes pueden involucrar a:

- Bases de datos relacionales
- Bases de transaccionales
- Sistemas implementados con mensajería HL7 [MEN98]
- Legacy Systems
- Objetos Corba
- Servicios DCE
- Objetos RMI
- Applets de Java
- Aplicaciones en diferentes lenguajes de aplicación
- Componentes de ISVs (Vendedores independientes de Software)



Ejemplo de una Arquitectura Corporativa de Interfaces con Implementación en CORBA

Para esta situación, se requiere de protocolos estándar de comunicación, mecanismo de acceso de datos, mecanismos de activación de aplicaciones y ***CORBA*** cumple estas características para lograr la convivencia de estos elementos.

5.5. Conclusiones

Los sistemas de información y telecomunicación has sido encapsuladas para lograr la implementación de un ***CPR***. Aunque pueda parecer que estas tareas están focalizadas estrictamente en aspectos técnicos, va más profundo que ello. Esto lleva a un análisis de las tecnologías de los sistemas implementados en la organización de salud, y que ajustes se tienen que realizar para lograr una total integración en un modelo corporativo único, que permitan la escalabilidad y expansión desde el punto de vista tecnológico como de dominio del negocio.

En este capítulo se describió las características de un sistema de ***CPR*** que debe cumplir mediante la implementación de objetos distribuidos en ***CORBA***. Esta arquitectura permite a cada uno de los sistemas ser usados en un ambiente heterogéneo de plataformas, sistemas operativos, lenguajes de programación, etc., permitiendo un alto grado de integración. Esto preserva la inversión realizada en las aplicaciones existentes, y lo único que se tienen que ocupar en investigar de

cómo realizar la integración, solamente preocupándose de ***QUE*** hace cada uno de los componentes de los dominios definidos en la organización.

CORBA provee una arquitectura de objetos distribuidos con el que puede ser aplicado sobre un sistema de ***CPR***. La ejecución de esta estrategia, es la definición de los métodos de cada ***wrappers*** orientados a objetos en las aplicaciones, accesos a las base de datos transaccionales y relacionales resolviendo los problemas de confiabilidad, disponibilidad, integridad de datos y performance en un ambiente de objetos distribuidos.

Esta arquitectura permite la participación de diferentes vendedores en todos los niveles de la integración, desde el nivel de comunicación hasta el de presentación.

No existe una única solución, pero se puede destacar las consideraciones que se tienen que realizar para lograr la meta de crear un sistema de ***CPR*** en un ambiente heterogéneo:

- ***CPR*** debe encontrar las necesidades de múltiples disciplinas dentro de la organización
- Estándar de comunicación y terminologías
- Reconocimiento de las necesidades de procesos departamentales e inter-departamentales de la organización
- Bus de comunicación único
- Definición de interfaces claras y consistentes
- Diseño modular de las reglas de negocio y aplicaciones

Conclusiones

El espíritu de esta tesis es la presentación de recomendaciones a la hora de realizar cualquier tarea o estudio de integración o desarrollo de sistemas.

El proceso de cambio del dato hacia la información y esta hacia el conocimiento requiere alineación y unión de los sistemas de información hacia los procesos de negocio dentro y fuera de una organización.

En el mundo de hoy, en dónde conviven las aplicaciones tradicionales, Internet y la necesidad de acortar los tiempos de desarrollos cuando los procesos y reglas de negocios son dinámicos (de acuerdo a distintas condiciones), es imperioso disponer de arquitecturas y modelos que van más allá de los datos.

En este documento comenzó con descripciones teóricas de diseño de software y la presentación de los distintos paradigmas de programación para concluir con el modelo de objetos. A partir de este modelo, se presentaron ***frameworks*** [FOO88], con la característica que ayuda a agregar valor al negocio (sea de una organización a resolver un problema o a la generación de un producto).

El común denominador en los distintos modelos que se presentaron, es el trabajo mediante componentes, que hace más fácil la interacción y compresión de los problemas a resolver.

El interés en este trabajo es la ***integración de aplicaciones***, porque se puede dar en distintos niveles, el de ***datos*** y de ***ejecución***, basados en computación distribuida, procesamiento de transacciones y mensajería. Dada la importancia de la integración a cualquier nivel, los departamentos de tecnología de información de las organizaciones disponen de un área denominada ***EAI (Entreprise Aplication Integration)***, con la responsabilidad de integración y definición de la arquitectura de sistemas de información, garantizando la interoperabilidad en los distintos ambientes y departamentos.

Arquitectura

La definición de ***Arquitectura*** no es un nuevo concepto en la ingeniería de software. [BAL99]

Se definen distintos tipos de arquitectura: sistemas, aplicaciones y de red.

La arquitectura de sistema define como la mayoría de los componentes de un sistema (Hardware, sistemas operativos, etc) trabajan juntos, la arquitectura de aplicación define como las aplicaciones deberían desarrollarse para usar la arquitectura de sistema y la arquitectura de red define como físicamente los componentes de hardware separados se comunican entre sí.

Aunque la arquitectura de datos no se enunció como una arquitectura aparte, sino que es parte de la arquitectura de sistema y aplicaciones, es importante focalizar el

objetivo del éxito de la integración en la información que se procesa y cómo se procesa.

De lo enunciado de este trabajo se describe las características de una arquitectura general [ZAC87] en una organización:

- ✓ La implementación de sistemas de información debe mostrar las características del negocio.
- ✓ Los datos tienen que ser consistentes de aplicación en aplicación.
- ✓ Hardware y sistemas de software deben ser compatibles en todos los sentidos.
- ✓ Las reglas de negocio deben ser consistentes a través de las implementaciones.
- ✓ Los sistemas tienen que ser definidos lógicamente e independientemente de las restricciones tecnológicas.
- ✓ El concepto de cambio o actualización debe ser incorporado como criterio de diseño.

Para cubrir los requerimientos de información para ser soportados en los sistemas para una organización, se pueden reclasificar la definición de arquitectura dado anteriormente en:

- ✓ Arquitectura de Datos
- ✓ Arquitectura de Procesos
- ✓ Arquitectura de Aplicaciones
- ✓ Arquitectura de Tecnología
- ✓ Arquitectura de Integración
- ✓ Arquitectura de Conocimiento

En este trabajo se enunció las características a cumplir, enfocado a la ***arquitectura de aplicaciones y de integración***, aunque implícitamente se vean afectadas las otras, dado que lo que se busca es la transformación del dato en conocimiento a través de las aplicaciones.

Arquitectura de aplicación

Los objetivos y beneficios de la integración de aplicaciones varia enormemente en organización en organización. En general, a la hora de iniciar las integraciones de las aplicaciones se tienen en cuenta los siguientes puntos:

- ✓ Mejora en las operaciones mediante la sincronización de datos y eliminando la múltiples entrada de datos.

- ✓ Distribución del valor agregado a través de la información de la organización (disponibilidad del mismo dato y funcionalidad en todas las aplicaciones).
- ✓ Asistencia en la transformación del dato a través de las aplicaciones permitiendo una administración de la misma en toda la organización.
- ✓ Mejora en el acceso de la información para los usuarios finales, resultando una disponibilidad de recursos altamente valorable.
- ✓ Facilidad de distribuir las funcionalidades en las aplicaciones, reduciendo los costos relacionados con la administración y mantenimiento de sistemas.
- ✓ Reducción la dependencia en un solo vendedor o varios, permitiendo aprovechar la transparencia de usabilidad de funcionalidades en distintas aplicaciones.
- ✓ Minimiza el tiempo de desarrollo y esfuerzo relacionado con la integración.
- ✓ Soporte de rediseño en procesos operacionales a través de la automatización de flujos de trabajo y programación orientada a eventos.
- ✓ Mejora en la administración de información a través de la centralización y estandarización de tareas de administración de datos comunes.
- ✓ Fuerza políticas organizacionales en seguridad y confidencialidad a través de todas las fuentes de información de todas las aplicaciones utilizadas en la organización, como un solo elemento.
- ✓ Distribución de tareas de información y procesamiento manteniendo un modelo de información uniforme y de estrategia uniforme.

Esta investigación y desarrollo hace referencia a las características que deben cumplir una correcta arquitectura de aplicación, con el objetivo continuar con la mantenibilidad, escalabilidad y la transparencia en el funcionalmente de las aplicaciones en ambientes heterogéneos.

Arquitectura de integración

Las características que deben cumplir las aplicaciones para que dialoguen entre sí, y se comparó con diferentes estándares: ***CORBA, DCE, RMI***.

La utilización de cada una de estas arquitecturas dependen en gran parte de la filosofía de la organización, el nivel de conocimiento de los integrantes del departamento de sistemas de información, etc. .

No existe ***él modelo*** que permita garantizar el éxito de toda implantación de software o el diseño de buenas aplicaciones que permitan la actualización y la transparencia de comunicación e ínter operación (***concepto de bala de plata***), sino que depende de cuán efectivo son los diseñadores y desarrolladores de software de poner en práctica los diferentes conceptos enunciados aquí, con el objetivo de trabajar en ***ambientes heterogéneos en una forma transparente, segura y a bajo costo***.

El primer paso para llevar a cabo una ***estrategia de integración***, es definir un modelo de información de la organización. Este modelo define objetos y relaciones entre objetos y permitir la integración de las aplicaciones (plataforma, lenguajes de programación, redes, base de datos) que ocurren dentro del contexto que se reconocen las entidades de negocios y funciones.

El segundo paso es decidir dónde está el mayor beneficio que puede dar a través de la integración, esto tiene que estar relacionado con la estrategia de administración de la información y que todos los componentes de aplicación deben estar sintonizados para la integración, a través de la definición de una arquitectura estándar, y ***esto motivó este trabajo de tesis***.

El tercer paso es decidir sobre que herramientas o combinación de herramientas se lleva a cabo la integración, y esta respuesta debe ser basada en los conceptos en que se basó la construcción de las aplicaciones.

Claramente, la integración juega un rol significante en la administración de información. Realmente, creo que la integración de aplicaciones es requerida para ejecutar la administración de información en ambientes heterogéneos. Y todos estos conceptos son aplicables, a la hora de empezar un desarrollo desde cero de aplicaciones, o ya disponer de aplicaciones funcionando para reutilizar sus funcionalidades.

Interoperabilidad

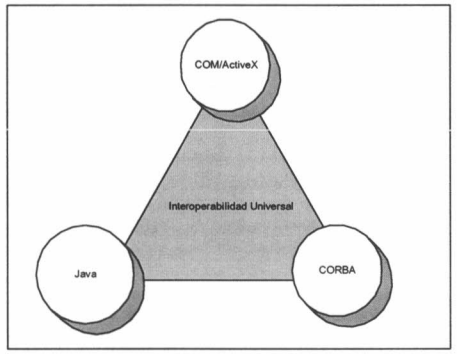
En un mundo de sistemas de objetos distribuidos, diversidad e interoperabilidad son el ***nombre del juego***. Recientemente se debatía la forma de construir sistemas heterogéneos si es ***COM*** o ***CORBA***, y saber quién ganaría esta lucha, para ganar mercado o establecer normas para elevar a los componentes al objetivo de tener desarrollos de sistemas de aplicación extensibles.

En el desarrollo de la tesis, no existe un claro ganador, sino que cada una de estas arquitecturas deberían convivir juntas e ínter operar para distribuir el valor verdadero del negocio.

Dado la complejidad de las reglas de negocios de las organizaciones, y el número de aplicaciones que se están ejecutando, es necesario definir estrategias de integración de aplicaciones a través de la organización (EIA), quien es el

combustible para establecer y manejar la interoperabilidad entre los diferentes actores (COM, CORBA, RMI, etc.).

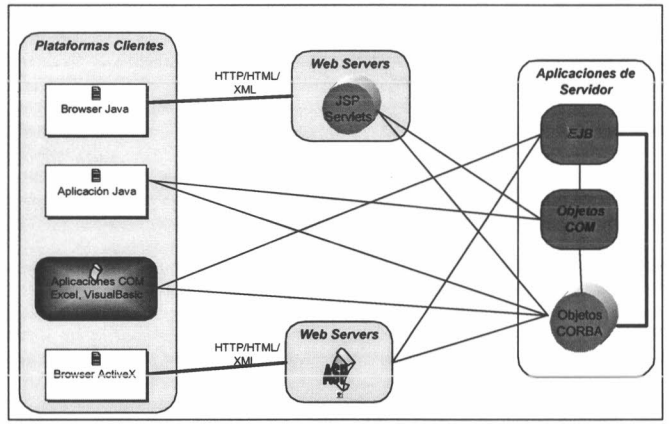
Más importante, la interoperabilidad permite a las organizaciones elevar los procesos de negocio a través de múltiples canales de accesos hacia los servicios y funcionalidades de la organización. Esto permite que el mismo conjunto de componentes soporten el acceso desde distintos clientes como clientes Java, browsers Active X, Clientes CORBA y aplicaciones COM, llegando a la conclusión de que cada modelo de aplicación coexiste entre cada una de ellas arribando a una ***interoperabilidad universal***.



Una Solución completa de integraciín e interoperabilidad deberá incorporá los componentes a partir de estos tres modelos de componentes

Los productos de interoperabilidad son habilitadores claves para la construcción de sistemas de aplicaciones que se expanden en una variedad de componentes más allá en los que se abordó en este trabajo como ***Servlets, Java Servers Pages (JSP), Enterprise JavaBeans (EJB), Active Server Pages (ASP) y Active Server Componentes,*** como también tecnologías como ***Servidor de Aplicaciones*** y ***Servidores WEB***, que pueden ser material de estudio para una futura tesis.

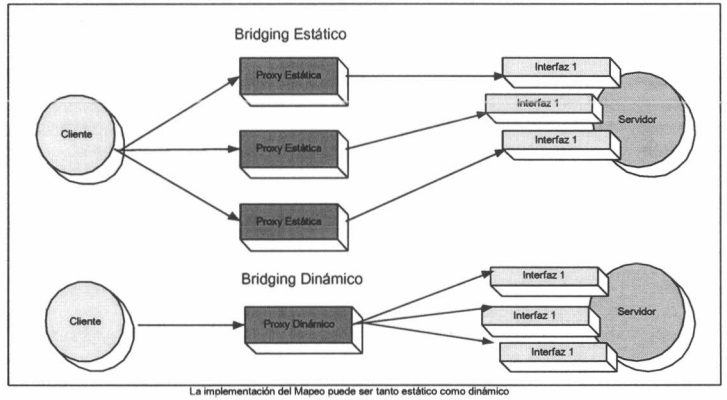
En la siguiente figura se visualiza los caminos claves de la interoperabilidad como también el cuadro claro de diversidad y variedad de los puntos de conexión entre componentes de los diferentes modelos de componentes.



Productos Actuales soportan una variedad de conectividad de componentes para el camino de la interoperabilidad e integración (Java/CORBA, COM/CORBA, Java/COM)

Fundamentos

El objetivo de la interoperabilidad entre los componentes o ***bridging***, es proveer un punto entre dos modelos de componentes. Este mapeo permite a los clientes de un sistema de componente acceder a componentes de otro sistema de componente en una manera no intrusa y transparente proveyendo al componente destino una semántica y una construcción adecuada. En los capítulos de CORBA, RMI, DCE se describen mapeos entres cada uno de sus sistemas de componentes, llevándolo a una forma más general se aplica entres distintos modelos de componentes.



La implementación del Mapeo puede ser tanto estático como dinámico

Ninguna aplicación es una isla. En el mundo de hoy que aumenta la complejidad, es imperativo que las aplicaciones dialoguen y cooperen. Y para llevar a cabo los objetivos de negocio en el mundo de hoy, incluido ***Internet***, hay una necesidad adicional para un vocabulario común.

Las aplicaciones necesitan dialogar entre si u con otras dentro de la organización y entre las entidades de negocio, como también a través de una red de área local, Internet y ahora en redes inalámbricas.

Esto es un problema que perdura día a día en la industria de la tecnología. Todos los años una nueva ***bala de plata*** surge prometiendo resolver todos los problemas de integración.

La integración entre aplicaciones puede ser vista como un problema de comunicación. Las aplicaciones deben tanto hablar el mismo lenguaje como tener un traductor. Una vez que las aplicaciones hablan el mismo lenguaje, debería haber un mecanismo para soportar el intercambio.

La motivación de este trabajo es la presentación distintos modelos de componentes para crear una arquitectura de aplicaciones en ambientes heterogéneos, pero se llega a la conclusión que no existe un solo modelo sino una combinación de varios para lograr los objetivos, considerando tres aspectos para la comunicación Inter.-aplicación:

- ✓ Transporte (Como obtener la información a través del cable)
- ✓ Protocolo (Como un paquete de información se envía a través del cable)

- ✓ Mensaje (La información en si misma)

GLOSARIO

Análisis de Afinidad
Analizar la frecuencia de interacción entre dos o mas entidades.

Atributo
Una pieza de un estado que esta visible implícitamente y recuerda operaciones de procedimientos para un objeto.

Adaptador Básico de Objeto (Basic Object Adapter-BOA)
Provee para una implementación de objeto un acceso a las funciones del ORB (Object Request Broker).

Browsers
Navegadores de Acceso a información y datos.

Calidad de Servicio
Recurso de tolerancia que permite a una aplicación determinada el objetivo de prevenir si la aplicación cumple con los objetivos requeridos.

Clase
La descripción estructural de un objeto que incluye una administración de grupo, extensión y operaciones durante el ciclo de vida para sus miembros.

Cliente
Alguien que solicita servicios, recursos o ambos.

Common Objetc Model (COM)
Es un producto de un acuerdo entre Microsoft y Digital Equipment Corp para proveer funcionalidades de objetos distribuidos OLE (Object Linking and Embedding).

Common Object Request Broker Architecture (CORBA)
Es una especificación del Object Management Group que define un framework (cuerpo) para la construcción de sistemas de objetos distribuidos. Las especificaciones de CORBA incluye definiciones para invocaciones de interfaces estáticas y dinámicas, repositorio de interfaces y el ORB.

Comportamiento
El intento de una semántica para un contrato con un objeto.

Distributed System Object Model (DSOM)
Implementación de IBM para especificaciones CORBA.

Encapsulamiento
Un mecanismo de abstracción que esconde estructura de datos desde el punto de vista público y permite el manipulamiento mediante la definición de operaciones

(conceptualmente están relacionados). Agrupar información y operaciones que están conceptualmente relacionadas.

Extensión o Extensibilidad
La habilidad de agregar a un sistema nuevas objetos implementados que define el comportamiento de la aplicación. Existen aplicaciones clientes que luego pueden ser usadas en nuevas implementaciones sin requerir cambios en el código de aplicación.

Filtro
Un proceso que ejecuta una función fija sobre una cadena desde un proceso, pasándolo luego procesado a otro.

Firma (Signature)
Una completa descripción de un método que incluye un mensaje (nombre de parámetros y tipos) y la respuesta (tipo) o las condiciones que se esperan después de la invocación del método.

Granularidad
Es un alcance abstracto que es usado para discutir en los objetos y su tamaño en términos de toda la funcionalidad y en términos de servicios, que ellos ofrecen.

Herencia
Es un mecanismo por el cual las clases pueden ser especializadas desde una clase mas general. Las propiedades del nuevo objeto incluye aquellas de la clase en que se basó para su creación.

Implementación
La definición de un objeto y la forma que provee servicios para aplicaciones clientes. Una implementación de objeto incluye una especificación de la estructura de datos que representa los atributos del objeto y la definición de las operaciones que el objeto puede ejecutar.

Interfaz
Un protocolo completo usado por una clase para todos los mensaje
Interface Definition Language (IDL)
Un lenguaje de programación abstracta usado por una interfase de una aplicación cliente de objetos. IDL es independiente del lenguaje actual de programación usado en la implementación tanto del cliente o del objeto.

Interfaz de Invocación Dinámica (Dynamic Invocation Interface-DII)
Es una interfase que permite a los clientes crear pedidos en tiempo de ejecución (Runtime) como sean necesarios. DII es útil cuando la interfase exacta u operación no es conocida en tiempo de compilación.

Invariantes
Los aspectos de una clase que son verdaderos y nunca varia.

Invocación
Ejecutar una operación de un objeto dado.

Lado del Cliente (Client Side)
Una colección de código es normalmente un Cliente o Servidor. Sin embargo, el cliente designa el código que generalmente requiere servicios. Por lo tanto, los objetos (lo que satisfacen esos pedidos) que viven en el Cliente son lado cliente (client-side) y los objetos que viven en el Servidor son lado servidor (Server-side).

Lado Servidor
Ver Client-side

Latencia
Es el tiempo desde que se requirió el pedido hasta que se obtuvo la respuesta.

Método
Un simple pedido o mensaje disponible por un Server.

Modelo de Fallas
El conjunto de todos los tipos y condiciones bajo cual sistema puede fallar (particularmente distribuido).

Objeto
Elemento básico en la programación orientada a objetos, consistiendo en un código de aplicación y datos. Los objetos son definidos por su comportamiento y conocen como ejecutar ciertas operaciones.

Object Request Broker (ORB)
Un sistema que es responsable de repartir pedidos de objetos en un ambiente distribuido. ORB establece y administra las comunicaciones entre la aplicación cliente y los objetos.

Orientación a Objetos
Un paradigma de programación que ve a un sistema como un conjunto de entidades autónomas (objetos) con un conjunto específicos de operaciones y funciones más que una serie de procesos discretos.

Polimorfismo
La habilidad de un objeto de mostrar distintos comportamientos dado el mismo mensaje.

Pre-condición
Aquellas condiciones que se tiene que cumplir antes de la invocación de un método

Protocolo de Mensaje
Una descripción de un mensaje.

Post-condición
Aquella condición que debe ser válida después de que un método es invocado.

Referencia de Objeto
Un handle (manejador) de un objeto que permite a la aplicación cliente invocar y hacer pedidos sobre un objeto. Una referencia de objeto identifica de forma única a un objeto.

Reglas de Negocio
Es un conjunto de secuencia de operaciones que ejecutan servicios que tienen significado para el negocio.

Repositorio de Interfaces (Interface Repository)
Es una base de datos que provee un almacenamiento persistente de definiciones IDL clientes usadas para acceder a los objetos en un ambiente de objetos distribuidos.

Servidor
Una entidad proveedora de servicios o recursos.

Sockets
Es un canal a través del cual las aplicaciones se pueden conectar con cada una de ellas y comunicarse.

Static Invocation Interface (SII-Interface de Invocación Dinámica)
Es una interfase básica de objetos, en el cual es definida por IDL. SII crea una relación fija entre el client y el objeto, que no cambia sin ser redefinida por el desarrollador.

Tipo abstracto de datos (TDA)
Es un encapsulamiento entre la estructura de datos y los procedimientos públicos que pueden manipular la estructura de datos en una entidad.

3-TIER
Un sistema Client-Server que muda la lógica del negocio en un nivel separado (tier)

2-TIER
Un sistema Client-Server donde la lógica de negocio esta contenida dentro del cliente o el Server o ambos.

Tipo
Una abstracción usada para describir propiedades de un entidad y como un mecanismo de protección usado para verificar la consistencia esperada para cada una de las entidades en tiempo de compilación.

Bibliografía

[APR93]= Application Development Reference, Revision 1.0, Prentice Hall, 1993.

[BRO95]=Migrating Legacy Systems, Brodie,M, McGraw-Hill,1995.

[COR95]=Object Management Group. The Common Object Request Broker; Architecture and Specification, Revision 2.0, OMG, 1995.

[COR97]= Core Java, Gary Cornell & Cay S. Horstmann, second edition, Sunsoft Press A Prentice Hall Title, 1997.

[CPR98]= Computer-based Patient Record Institute (CPRI). Definición de CPR 1998:http://www.cpri.org/what.html.

[ELM94]=Fundamentals of Database Systems, Elsmasri/Navathe, second edition, addison wesley, 1994.

[FAQ97]= www.cerfnet.com/mpcline/Corba-FAQ.html, frequently asking questions about CORBA, 1997

[FOO88]=Designing Reusable Classes. Ralph Johnson, Brian Foote. Journal of Object Oriented Programming, Junio/Julio 1988, Vol.1, Numero 2, Pagina 22-35.

[GAM95]=Desing Patterns-Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Addison-Wesley,1995.

[GER99]= COM-CORBA INTEROPERABILITY, Ronan Geraghty, Sean Joyce, Tom Moriarty and Gary Noone, Prentice Hall series on Microsoft Technologies, Prentice Hall, 1999.

[HAR97] = The Essential Distribuited Objects Survival Guide, Robert Orfali, Dan Harkey, Jeri Edwards, Segunda Edición, John Wiley & Son, 1997.

[HUC95]= Using object thinking to automate the business process, Jerry Huchzermeir, white paper, Proforma Corporation, 1995.

[IBM98]= International Bussiness Machinery, IBM, Inc. http://www.ibm.com

[JAV99]= http://java.sun.com/products/jdk

[JOH90]=Surveying-Current Research in Object-Oriented Design. Ralph Johnson, Rebecca J. Wirfs-Brock. Communications of the ACM. Septiembre 1990, Vol.3, Nro.9, Pagina 104-123.

[JOU95] = Dr. Dobb's Journal, "Interoperable Objects", Enero1995.

[Jus97]= Just Java, Peter van der Linden, Sunsoft Press A Prentice Hall Title, 1997.

[KKS96]=a Brief Tutorial on CORBA. www.cs.indiana.edu, 1996.

[MEN98]= Health Level Seven, "Estándar HL7"; Ann Arbor, MI: versión mantenida electrónicamente en http://www.mcis.duke.edu/standards/HL7/hl7.htm

[MOW95]=The Essential CORBA, Mowbray T. y Zahavi R. John Wiley and sons, 1995

[NET98]=Netscape Communications Corporation. htp://developer.netscape.com

[NOR90]= Guia del Programador para el IBM PC y PS/2, Perter Norton, Richard Wilton, Microsoft Press, Anaya, 1990.

[OBJ96]=Object Magazine, Octubre 1996, Pag. 52-55.

[OBJ98]= Object Magazine, Enero de 1998, Sigs Publications.

[OEG]= Open Engineering Inc. http:// www.openeng.com

[OMG95A] =Common Object Request Broker Architecture, OMG, Julio 1995.

[ORF97] = The Essential Client/Server Survival Guide, Robert Orfali, Dan Harkey, Jeri Edwards, Segunda Edición, John Wiley & Son, 1997.

[ORG96]=Object Management Group. www.omg.org

[PET94]= Sistemas Operativos-Conceptos fundamentales, J. Peterson, A. Silberschatz y P. Galvin, Tercera edición, Addison Wesley, 1994.

[ROS92]= Understanding DCE, Rosenberry, W., D. Kenney, and G. Fisher, O'Reilly & Associates, 1992.

[SES98]= CPR Architecture Based on Distribuited Objects and Web Technologies, Session 58, HIMSS, 1998.

[SHI92]= Guide to Writing DCE applications, Shirley, J., W. Hu, y D. Magid, Second Edition, O'Really & Associates, 1992.
[APG93]= Application Development Guide, Revision 1.0, Prentice Hall, 1993.

[SIE96]=CORBA, Fundamentals and Programming, Jon Siegel. John Wiley & Son, 1996.

[SMA92]= Smalltalk, User Guide.

[STR93]=El C++ Lenguaje de Programación, segunda edición, Bjarne Stroustrup, Addison-Wesley/Diaz de Santos, 1993.

[TAM91] = Redes de Ordenadores, Andrew S. Tanenbaum, Segunda Edición, Prentice Hall, 1991.

[TAM96] = Computer Networks, Andrew S. Tanenbaum, Tercera Edición, Prentice Hall, 1996.

[TRA98]= Transarc Corporation, Inc. http://www.transarc.com

[TRW98]= Extending DCE for Business-Critical Computing with Encina Monitor, Transarc Corporation, Inc, White Paper, 1998.

[T1199]= Orientación a objetos y desarrollo de componentes de software, tutorial 11, Pablo Madril, Luis Quelves da Silva, PEP99, Prontuario Eletronico do Paciente 99, Rio de Janeiro,Brasil.

[WAY94]=What is Object-Oriented Desing ?, Wayne Haythorn, Joop Magazine, Marzo-Abril 1994, Pagina 67-78.

[WHI96]=What is CORBA ?. Object Management Group 1996.

[WIR90]=Designing Object-Oriented Software. Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener. 1990. Prentice Hall