# UNIVERSIDAD NACIONAL DE INGENIERÍA

FACULTAD DE INGENIERÍA ELÉCTRICA Y ELECTRÓNICA


**DESARROLLO DE SISTEMAS DISTRIBUIDOS BASADOS EN CORBA**

**CON INTEROPERABILIDAD ENTRE PLATAFORMAS HETEROGÉNEAS**

# INFORME DE SUFICIENCIA

PARA OPTAR EL TÍTULO PROFESIONAL DE:

INGENIERO ELECTRÓNICO

**PRESENTADO POR:**

**ANTONIO ROLANDO KANASHIRO MEDINA**

**PROMOCIÓN**

**1997 - II**

**LIMA – PERÚ**

**2005**

# DESARROLLO DE SISTEMAS DISTRIBUIDOS BASADOS EN CORBA CON INTEROPERABILIDAD ENTRE PLATAFORMAS HETEROGÉNEAS

**Dedico el presente trabajo a mi madre por su invalorable apoyo y a mi hija principal motivación en mi vida**

## SUMARIO

El presente trabajo tiene por objetivo dar a conocer los avances logrados en el área de Desarrollo de Sistemas Distribuidos bajo la arquitectura OMA (Object Management Architecture es la Arquitectura en la que se basan todas las tecnologías adoptadas por el OMG) del OMG(Object Management Group es el consorcio más grande del mundo dedicado a la industria de la computación, con más de 700 miembros, su sitio web: www.omg.org), específicamente el uso de CORBA(Common Object Request Broker Architecture es uno de los estándares adoptados por el OMG basado en OMA para la administración de sistemas distribuidos) que es un estándar para el desarrollo de Sistemas Distribuidos basados en objetos.

La Primera parte tiene por objetivo documentar todos los conceptos necesarios sobre Sistemas Distribuidos para poder entender el análisis del estándar CORBA, y esta conformada por los siguientes capítulos:

En el Capítulo I se describe a los sistemas distribuidos y se especifica los requerimientos que estos deben cumplir de acuerdo con las necesidades actuales.

En el Capítulo II se describe las diferentes modalidades de comunicación entre las componentes que conforman un sistema distribuido de acuerdo a la plataforma y a la arquitectura implementada.

En el Capítulo III se describe el modelo de objetos distribuidos y el proceso de comunicación de estos bajo la modalidad de peticiones y respuestas desarrollados por algunas tecnologías conocidas.

En la Segunda parte se aborda el desarrollo de sistemas distribuidos con las especificaciones del OMG y CORBA, a través de los siguientes capítulos:

En el Capítulo IV se hace una descripción general de la arquitectura OMA y de CORBA como el estándar desarrollado por el OMG que la implementa.

En el Capítulo V se aborda el tema de la interoperabilidad entre componentes heterogéneos dentro de la arquitectura CORBA a través del protocolo IIOP.

En el Capítulo VI a fin de mostrar la interoperabilidad dentro de CORBA se desarrolla una aplicación Cliente-Servidor entre dos componentes desarrollados con lenguajes diferentes como son Java y C++ luego se desarrolla una aplicación Cliente-Servidor en la que se muestra el uso del Servicio de Nombres de CORBA.

El Capítulo VII comprende las conclusiones del presente informe, donde se aborda el alcance que tienen hasta hoy los Sistemas Distribuidos y el futuro desarrollo que les espera así como el papel que tiene y tendrá CORBA en dicho desarrollo.

# ÍNDICE

**INTRODUCCIÓN**

**Capítulo I. SISTEMAS DISTRIBUIDOS**

1.1. Concepto

1.2. Ejemplos de Sistemas Distribuidos

1.2.1. Internet

1.2.2. Intranets

1.2.3. Computación móvil y ubícua

1.3. Recursos compartidos y Web.

1.4. Requerimientos.

1.4.1. Heterogeneidad

1.4.2. Extensibilidad

1.4.3. Seguridad

1.4.4. Escalabilidad

1.4.5. Tratamiento de Fallos

1.4.6. Concurrencia

1.4.7. Transparencia

1.4.8. Fiabilidad

**Capítulo II. COMUNICACIÓN ENTRE PROCESOS REMOTOS**

2.1. Características Generales de la Comunicación entre procesos

2.1.1. Uso de Sockets

2.1.2. Comunicación de Datagramas UDP

2.1.3. Comunicación de Streams TCP

2.2. Representación Externa de Datos y Empaquetado

2.2.1. Representación Común de Datos de CORBA

2.2.2. Serialización de Objetos en Java

2.2.3. Referencias a Objetos Remotos

2.3. Comunicación Cliente Servidor

**Capítulo III. OBJETOS DISTRIBUIDOS E INVOCACIÓN REMOTA**

3.1. Interfaces

3.2. Comunicación entre objetos distribuidos

3.2.1. El modelo de objetos

3.2.2. El modelo de objetos Distribuidos

**3.3.** Llamada a un procedimiento remoto

3.4. Eventos y Notificaciones

**Capítulo IV. SISTEMAS DISTRIBUIDOS CON CORBA**

4.1. Modelo de la Arquitectura de Objetos Distribuidos

4.2. ¿Qué es CORBA?

4.3. Modelo de la Arquitectura CORBA

4.3.1. El negociador de peticiones (ORB)

4.3.2. Lenguaje de definición de interfaces de OMG (IDL)

4.3.3. Cliente

4.3.4. Referencias a Objetos

4.3.5. Stub del Cliente

4.3.6. Interface de invocación dinámica (DII)

4.3.7. Skeleton del Servidor

4.3.8. Interfaces de Skeleton dinámicos

4.3.9. Adaptadores de Objeto

4.4. Servicios CORBA

4.4.1. Servicio de Nombres

4.4.2. Servicio de Eventos

4.4.3. Servicio de Persistencia de Objetos

4.4.4. Servicio de Transacción

4.4.5. Servicio de control de Concurrencia

4.4.6. Servicio de Seguridad

4.5. Lenguaje de Definición de Interfaces

**Capítulo V. INTEROPERABILIDAD CON CORBA**

5.1. Elementos para la interoperabilidad

5.2. Arquitectura de Interoperabilidad entre ORB

5.3. Soporte de Puente Inter – ORB

5.4. Protocolo General Inter-ORB (GIOP) y protocolo Inter.-ORB para Internet (IIOP)

**Capítulo VI. APLICACIONES CON CORBA**

6.1. Interoperabilidad entre Java y C++

6.2. Implementación de aplicaciones usando el Servicio de Nombres CORBA

**CONCLUSIONES**

**BIBLIOGRAFÍA**

# INTRODUCCIÓN

En las primeras épocas de la computación las computadoras operaban independientemente unas de otras sin tener comunicación entre ellas. Las aplicaciones de software eran comúnmente desarrolladas para un propósito específico. Compartir los datos entre sistemas era mínimo y se hacía de una manera muy fácil, se transportaban los medios de almacenamiento (tarjetas, cintas, discos, etc.) de un lado a otro. El próximo paso fue conectar las computadoras a través de una red usando protocolos propietarios, luego los protocolos fueron estandarizados. Luego llego la era de los sistemas abiertos y la integración de sistemas por los cuales un cliente podía elegir varios componentes de hardware de diferentes vendedores e integrarlos para crear una configuración necesaria con costos razonables.

Así siguió el paso de los años. Nuevas técnicas de desarrollo de software se fueron sucediendo una tras otra, desde la programación estructurada y modular hasta la programación orientada a objetos, siempre buscando reducir costos y aumentar la capacidad de rehúso. Si bien la programación orientada a objetos fomentó el rehúso y permitió reducir costos, no se ha logrado aún de manera definitiva el objetivo último: comprar componentes de varios proveedores e integrarlos para formar aplicaciones que a su vez se integren para formar sistemas completos.

Para lograr la integración total de componentes realizados por terceras partes es necesario llegar a un acuerdo común en el que se establezcan los mecanismos necesarios para que esa integración se haga efectiva. Será necesario especificar de manera independiente al lenguaje de programación en el que se desarrolló el componente cuáles son sus puntos de acceso (funciones), luego será necesario

establecer los mecanismos de comunicación entre componentes, que podrían estar ejecutándose en una máquina remota.

En este sentido, y buscando satisfacer esa necesidad de mecanismos estándar e interfaces abiertas, son tres los esfuerzos que más han sobresalido. Por un lado, Microsoft ha introducido en el mercado sus tecnologías COM, DCOM y COM+. Otro participante es Sun Microsystems, que ha presentado Java Beans. El tercero es el Object Management Group (OMG), un consorcio integrado por varias industrias importantes, que ha desarrollado el estándar CORBA basado en la Arquitectura OMA, tema del presente informe.

CORBA ofrece servicios de interoperabilidad e interconexión de objetos. Los servicios de interoperabilidad e interconexión son normalmente conocidos como servicios middleware.

**Servicios Middleware**

Para resolver los problemas inherentes a sistemas heterogéneos y distribuidos, que dificultan la implementación de verdaderas aplicaciones distribuidas, los proveedores de software están ofreciendo interfaces de programación y protocolos de interoperabilidad estándares. Estos servicios se denominan usualmente servicios middleware, porque se encuentran en una capa intermedia, por encima del sistema operativo y del software de red y por debajo de las aplicaciones de los usuarios finales.

Un servicio middleware es un servicio de propósito general que se ubica entre la plataforma y aplicación. Por plataforma se entiende el conjunto de servicios de bajo nivel ofrecidos por la arquitectura de un procesador y el conjunto de API's (Aplication Programing Interface, es una serie de servicios o funciones que el

sistema operativo ofrece al programador) de un sistema operativo. Como ejemplos de plataformas se pueden citar: Intel x86 y Win-32, SunSPARCStation y Solaris, IBM RS/6000 y AIX, entre otros.

Un servicio middleware está definido por las API´s y el conjunto de protocolos que ofrece. Pueden existir varias implementaciones que satisfagan las especificaciones de protocolos e interfaces. Los componentes middleware se distinguen de aplicaciones finales y de servicios de plataformas específicas por cuatro importantes propiedades:

1. Son independientes de las aplicaciones y de las industrias para las que éstas se desarrollan.
2. Se pueden ejecutar en múltiples plataformas.
3. Se encuentran distribuidos.
4. Soportan interfaces y protocolos estándar.

Debido al importante rol que juega una interfaz estándar en la portabilidad de aplicaciones y un protocolo estándar en la interoperabilidad entre aplicaciones, varios esfuerzos se han realizado para establecer un estándar que pueda ser reconocido por los mayores participantes en la industria del software. Algunos de ellos han alcanzado instituciones como ANSI e ISO; otros han sido propuestos por consorcios de industrias como son la Open Software Foundation y el Object Management Group y otros han sido impulsados por industrias con una gran cuota de mercado. Este último es el caso de Microsoft con su Windows Open Services Architecture.

CORBA es el estándar propuesto por el OMG. EL OMG fue fundado en 1989 y es el más grande consorcio de industrias de la actualidad, con más de 900 compañías que son miembros del grupo. Opera como una organización no comercial sin fines de lucro, cuyo objetivo es lograr establecer *todos* los estándares necesarios para lograr

interoperabilidad en todos los niveles de un mercado de objetos (entiéndase por objetos componentes desarrollados por diferentes proveedores).

Originalmente los esfuerzos de la OMG se centraron en resolver un problema fundamental: ¿Cómo lograr que sistemas distribuidos orientados a objetos implementados en diferentes lenguajes y ejecutándose en diferentes plataformas interactúen entre ellos?. Más allá de los problemas planteados por la computación distribuida, problemas más simples como la falta de comunicación entre dos sistemas generados por compiladores de C++ distintos que corren en la misma plataforma frenaron los esfuerzos de integración no bien comenzados. Para opacar aún más el escenario, distintos lenguajes de programación ofrecen modelos de objetos distintos. Los primeros años de la OMG estuvieron dedicados a resolver los principales problemas de *cableado*. Como resultado se obtuvo la primera versión del Common Object Request Broker, publicado en 1991. Hoy en día, el último estándar aprobado de CORBA está por la versión 2.3, y la versión 3.0 está a punto de ser lanzada.

Desde sus principios, el objetivo de CORBA fue permitir la interconexión abierta de distintos lenguajes, implementaciones y plataformas. De esta forma, CORBA cumple con las cuatro propiedades enumeradas como deseables de los servicios middleware. Para lograr estos objetivos, la OMG decidió no establecer estándares binarios (como es el caso de COM); todo está estandarizado para permitir implementaciones diferentes y permitir que aquellos proveedores que desarrollan CORBA pueden ofrecer valor agregado. La contrapartida es la imposibilidad de interactuar de manera eficiente a nivel binario. Todo producto que sea compatible con CORBA debe utilizar los costosos protocolos de alto nivel.

CORBA está constituido esencialmente de tres partes: un conjunto de interfaces de invocación, el ORB (Object Request Broker es el negociador que se encarga de gestionar las peticiones y respuestas entre los objetos dentro de un sistema distribuido) y un conjunto de adaptadores de objetos (objects adapters). CORBA va más allá de simples servicios middleware, provee una infraestructura (framework) para construir aplicaciones distribuidas orientadas a objetos. Las interfaces definen los servicios que prestan los objetos, el ORB se encarga de la localización e invocación de los métodos sobre los objetos y el object adapter es quien liga la implementación del objeto con el ORB.

Para que las interfaces de invocación y los adaptadores de objetos funcionen correctamente, se deben cumplir dos requisitos importantes. En primer lugar, las interfaces de los objetos deben describirse en un lenguaje común (IDL de CORBA). En segundo lugar, todos los lenguajes en los que se quieran implementar los objetos deben proveer un *mapeo* entre los elementos propios del lenguaje de programación y el lenguaje común. La primera condición permite generalizar los mecanismos de pasaje de parámetros a través del proceso de Marshaling (por el cual los datos son adecuados dentro de un mensaje para que puedan ser transmitidos) y Unmarshaling (por el cual los datos son desempaquetados desde un mensaje de procedencia remota al formato adecuado para quien recibe dicho mensaje). La segunda permite relacionar llamadas de o a un lenguaje en particular con el lenguaje de especificación común. Este lenguaje común fue una parte esencial de CORBA desde sus orígenes y es conocido como el OMG IDL (lenguaje en el cual se especifican las interfaces para los objetos remotos dentro de la arquitectura CORBA del OMG). Existen mapeos del OMG IDL a C, C++, Java y Smalltalk, etc.

# CAPITULO I

# SISTEMAS DISTRIBUIDOS

## 1.1 CONCEPTO

El propósito de este capítulo es el de transmitir una visión clara de la naturaleza de los sistemas distribuidos y de los desafíos que deben ser considerados.

Un sistema distribuido es aquel en el que los componentes localizados en computadores diferentes, conectados en red, comunican y coordinan sus acciones únicamente mediante el paso de mensajes a los que llamaremos invocaciones y respuestas.

Existen redes de computadores como Internet, como lo son las muchas redes de las que se compone. Las redes de teléfonos móviles, las redes corporativas, todas, tanto separadas como combinadas comparten características esenciales que las hacen elementos importantes para su estudio bajo el título de sistemas distribuidos.

Hay tres ejemplos de sistemas distribuidos:

- Internet.
- Intranet.
- Computación móvil y ubicua.

Compartir recursos es uno de los motivos principales para construir sistemas distribuidos. Estos pueden ser administrados por servidores y accedidos por clientes o pueden ser encapsulados como objetos y accedidos por otros objetos clientes.

**Recurso:** se extiende desde los componentes hardware como los discos y las impresoras hasta las entidades de software definidas como ficheros, base de datos y objetos de datos de todos los tipos.

Los desafíos que surgen en la construcción de sistemas distribuidos son la heterogeneidad de sus componentes, su carácter abierto, el tratamiento de los fallos, la concurrencia de sus componentes y la transparencia.

Consideraciones a tomar en cuenta en la implementación de Sistemas Distribuidos:

**Concurrencia:** La ejecución de programas concurrentes es la norma. La capacidad del sistema para manejar recursos compartidos se puede incrementar añadiendo más recursos (Ej. Computadores) a la red. La coordinación de programas que comparten recursos y se ejecutan de forma concurrente es también un tema importante y recurrente.

**Inexistencia de reloj global:** Cuando los programas necesitan cooperar coordinan sus acciones mediante el intercambio de mensajes. Pero resulta que hay limites a la precisión con lo que los computadores en una red pueden sincronizar sus relojes, no hay una única noción global del tiempo correcto. Esto es una consecuencia directa del hecho que la única comunicación se realiza enviando mensajes a través de la red.

**Fallos independientes:** Todos los sistemas informáticos pueden fallar y los diseñadores de sistemas tienen la responsabilidad de planificar las consecuencias de posibles fallos. Los sistemas distribuidos pueden fallar de nuevas formas. Los fallos

en la red producen el aislamiento de los computadores conectados a él, pero eso no significa que detengan su ejecución.

## 1.2 EJEMPLOS DE SISTEMAS DISTRIBUIDOS

Los siguientes ejemplos están basados en redes de computadores conocidos y utilizados ampliamente:

### 1.2.1 INTERNET

Internet es una vasta colección de redes de computadas de diferentes tipos interconectados.

El diseño y la construcción de los mecanismos de comunicación Internet (los protocolos Internet) es una realización técnica fundamental, que permite que un programa que se está ejecutando en cualquier parte dirija mensajes a programas en cualquier otra parte. Internet es también un sistema distribuido muy grande, permite a los usuarios hacer uso de servicios como el World Wide Web, el correo electrónico, y la transferencia de ficheros. A veces se confunde incorrectamente el Web con Internet. El conjunto de servicios es abierto, puede ser extendido por la adición de los servidores y nuevos tipos de servicios. Los proveedores de servicios de Internet (ISP) son empresas que proporcionan enlaces de módem y otros tipos de conexión a usuarios individuales y pequeñas organizaciones. Las intranets están enlazadas conjuntamente por conexiones troncales (backbone). Una conexión o red troncal es un enlace de red con una gran capacidad de transmisión, que puede emplear conexiones de satélite, cables de fibra óptica y otros circuitos de gran ancho

de banda. En Internet hay disponibles servicios multimedia, que permite a los usuarios el acceso de datos de audio y video.

### 1.2.2 INTRANET

Una Intranet es una porción de Internet que es administrada separadamente y que tiene un límite o frontera que puede ser configurado para hacer cumplir políticas de seguridad local. Está compuesta de varias redes de área local (LANs) enlazadas por conexiones backbone. La configuración de red de una Intranet particular es responsabilidad de la organización que la administra. Una Intranet está conectada a Internet por medio de un encaminador (router), lo que permite a los usuarios hacer uso de servicios de otro sitio (otra Intranet por ejemplo) y permitir el acceso a los servicios que ella proporciona a los usuarios de otras intranets. Muchas organizaciones necesitan proteger sus propios servicios frente al uso no autorizado de programas nocivos. El papel del cortafuegos (firewall) es proteger una Intranet impidiendo que entren o salgan mensajes no autorizados fuera de sus límites. Algunas organizaciones no desean conectar sus redes internas a Internet. La solución que se adopta es realizar una Intranet, pero sin conexiones a Internet.

### 1.2.3 COMPUTACIÓN MÓVIL Y UBICUA

Los avances tecnológicos han llevado cada vez más a la integración de dispositivos de computación pequeños y portátiles en sistemas distribuidos. Estos dispositivos incluyen:

- Computadores portátiles.

- Dispositivos de mano (handheld), entre los que se incluyen asistentes digitales personales (PDA), teléfonos móviles, etc.
- Dispositivos que se pueden llevar puestos.
- Dispositivos insertados en aparatos.

La facilidad de transporte en muchos de estos dispositivos, junto con su capacidad para conectarse adecuadamente a redes en diferentes lugares, hace posible la **Computación móvil**. Los usuarios acceden a los recursos mediante los dispositivos portátiles que llevan con ellos.

Y se está incrementando la posibilidad de que utilicen recursos, como impresoras, que están suficientemente próximos a donde se encuentren. Esto último se conoce como computación independiente de posición.

**Computación Ubicua** es la utilización concertada de muchos dispositivos de computación pequeños y baratos que están presentes en los entornos físicos de los usuarios, sin que estos se den cuenta de ellos. En otras palabras, su comportamiento computacional estará ligado con su función física de forma íntima y transparente.

La computación ubicua y móvil se solapan, pero son distintas. La computación ubicua podrá beneficiar a los usuarios mientras permanecen en un entorno sencillo como su casa o un hospital. De forma similar, la computación móvil tiene ventajas si sólo se consideran computadores convencionales y dispositivos como computadores portátiles e impresoras.

## 1.3 RECURSOS COMPARTIDOS Y WEB

Los usuarios compartimos recursos de hardware. Pero es mucho más significativo para los usuarios compartir recursos de alto nivel que forman parte de sus

aplicaciones y de su trabajo habitual y sus actividades sociales. Los patrones de compartir recursos varían en su alcance y cuán estrechamente trabajan juntos los usuarios. En un extremo, una máquina de búsqueda en el Web proporciona una función para los usuarios en todo el mundo, los usuarios no necesitan establecer contacto con los demás directamente. En el otro extremo, en un sistema de trabajo cooperativo mantenido por computador (CSCW, computer-supported cooperative working, su especificación en http://www.usabilityfirst.com/groupware/cscw.txl), un grupo de usuarios particulares determina qué mecanismos debe proporcionar el sistema para coordinar sus acciones.

Utilizamos el término *servicio* para una parte diferente de un sistema de computadores que gestiona una colección de recursos relacionados y presenta su funcionalidad a los usuarios y aplicaciones.

El hecho de que los servicios limiten el acceso a los recursos a un conjunto bien definido de operaciones, refleja la organización física de los sistemas distribuidos. Los recursos en un sistema distribuido están encapsulados físicamente con los computadores y sólo pueden ser accedidos desde otros computadores a través de comunicación. Para que se compartan de forma efectiva, cada recurso debe ser gestionado por un programa que ofrece una interfaz de comunicación permitiendo que se acceda y actualice el recurso de forma fiable y consistente.

El término *servidor* se refiere a un programa en ejecución (un *proceso*) en un computador en red que acepta peticiones de programas que se están ejecutando en otros computadores para realizar un servicio y responder adecuadamente. Los procesos solicitantes son llamados *clientes*. Cuando un cliente envía una petición

para que se realice una operación, decimos que el cliente *invoca una operación* del servidor. Se llama *invocación remota* a una interacción completa entre un cliente y un servidor, desde el instante en el que el cliente envía su petición hasta que recibe la respuesta del servidor.

Los términos *cliente y servidor* se aplican a los roles desempeñados en una única solicitud. Ambos son distintos, en muchos aspectos, los clientes son activos y los servidores pasivos, los servidores se están ejecutando continuamente, mientras que los clientes sólo lo hacen el tiempo que duran las aplicaciones de las que forman parte.

Muchos sistemas distribuidos, aunque no todos, pueden ser construidos completamente en forma de clientes y servidores que interaccionan. El World Wide Web. El correo electrónico y las impresoras en red concuerdan con este modelo.

Un navegador (*browser)* es un ejemplo de cliente. El navegador se comunica con el servidor web para solicitar páginas.

**EL WORLD WIDE WEB**

Este es un sistema para publicar y acceder a recursos y servicios a través de Internet. Utilizando el software de un navegador web, como Netscape o Internet Explorer, los usuarios utilizan el Web para interaccionar con un conjunto ilimitado de servicios.
El Web comenzó su vida en el centro europeo para la investigación nuclear (CERN), Suiza, en 1989 como un vehículo para el intercambio de documentos entre una comunidad de físicos. Una característica fundamental del Web es que proporciona una estructura *hipertexto* entre los documentos que almacena. Esto significa que los documentos tienen *enlaces*.

Es fundamental para la experiencia del usuario en el Web que cuando encuentra una imagen determinada o una parte de texto en un documento, esto estará acompañado de enlaces a documentos relacionados y otros recursos. La estructura de los enlaces pude de ser arbitrariamente compleja y el conjunto de recursos que puede ser añadido es ilimitado, la "telaraña" (web) de enlaces está por consiguiente repartida por todo el mundo (world-wide).

El Web es un sistema abierto. Primero, su operación está basada en estándares de comunicación y en documentos estándar que están publicados libremente e implementados ampliamente. Por ejemplo, existen muchos tipos de navegador y existen muchas implementaciones de servidores Web. Cualquier navegador implementado conforme al estándar puede recuperar recursos de cualquier servidor implementado conforme al estándar.

Segundo, el Web es abierto respecto a los tipos de recursos que pueden ser publicados y compartidos en él. En su forma más simple, un recurso es una página web o algún otro tipo de *contenido* que puede ser almacenado en un fichero y presentado al usuario, como ficheros de programa, de imágenes, de sonido y documentos en forma PostScript o PDF. Los usuarios necesitan un medio de ver imágenes en este nuevo formato, pero los navegadores están diseñados para acomodar la nueva funcionalidad de presentación en forma de aplicaciones colaboradora*s* y conectores (plug-ins).

El Web, ha evolucionado sin cambiar su arquitectura básica. El web está basado en tres componentes tecnológicos de carácter estándar básicos:

- El lenguaje de etiqueta de hipertexto HTML (*Hypertext Markup Language)* es un lenguaje para especificar el contenido y el diseño de las páginas que son mostradas por los navegadores.
- Localizadores Uniformes de Recursos (URL, *Uniform Resource Locator)* que identifica documentos y otros recursos almacenado como parte del Web
- Una arquitectura de sistema cliente-servidor, con reglas estándar para interacción (el protocolo de transferencia hipertexto-http, *HyperText Transfer Protocol)* mediante la cual los navegadores y otros clientes obtienen documentos y otros recursos de los servidores web. Una característica importante es que los usuarios pueden localizar y gestionar sus propios servidores web en cualquier parte de Internet.

A continuación se detallaran los componentes

**HTML** el lenguaje de etiquetado de hipertexto que se utiliza para especificar el texto e imágenes que forman el contenido de una página web, y para especificar cómo serán formateados para la presentación al usuario. Una página web contiene elementos estructurados como cabeceras, párrafos, tablas e imágenes. HTML se utiliza también para especificar enlaces y qué recursos están asociados con ellos.

Los usuarios producen HTML a mano, utilizando un editor de texto estándar. Un ejemplo típico de texto HTML puede ser:

< IMG SRC = *http://www.cdk3.net WebExample/Images/earth.jpg*>

1

< P >

2

¡Bienvenido a la Tierra! Los visitantes pueden estar interesados también en echar un vistazo a la

3

< A HREF. = *http://www.cdk3.net/WebExample/moon.html* >*Luna*</*A*>

4

< P >

5

(Etcétera).

6

Este texto HTML se almacena en un fichero al que puede acceder un servidor web, supongamos que es el fichero *earth.html*. Un navegador (cliente) recupera el contenido de este fichero desde el servidor web que tiene el acceso, en este caso un servidor es un computador llamado *www.cdk3.net*. El navegador lee el contenido devuelto por el servidor y lo organiza en texto formateado y en imágenes presentadas en una página web en la forma habitual. Sólo el navegador, no el servidor, interpreta el texto HTML. Pero el servidor debe informar al navegador sobre el tipo de contenido que devuelve, para distinguirlo de, por ejemplo, un documento de PostScript. El servidor puede deducir el tipo de contenido a partir de la extensión del fichero .h*tml*

Las directivas HTML, conocidas como *etiquetas*, están encerradas entre ángulos *como <P>*.

La línea 1 del ejemplo identifica un fichero que contiene una imagen para la presentación. Su URL es *http://www.cdk3.net/WebExample/Images/earth.jpg* que es

el identificador que nos permite ubicar el recurso dentro de la Web. Las líneas 2 y 5 son una directiva de comienzo de un nuevo párrafo cada una. Las líneas 3 y 6 contienen texto que será mostrado por el navegador en el formato estándar de párrafo.

La línea 4 especifica un enlace (hipervínculo) en la página. Contiene la Palabra "Luna" rodeada por dos etiquetas HTML relacionadas < A HREF...> y </A>. El texto comprendido entre dichas etiquetas es lo que aparece en el enlace tal como se presenta en la página web. La mayoría de los navegadores están configurados para mostrar el texto de los enlaces subrayados, así que lo que el usuario verá en el párrafo es:

> ¡Bienvenido a la Tierra! Los visitantes pueden estar interesados en echar un vistazo a la Luna.
>
> El navegador registra la asociación entre el texto mostrado del enlace y el URL contenido en la etiqueta <A HREF...>, en este caso:

http://www.cdk3.net/WebExample/moon.html

Cuando el usuario selecciona un texto, el navegador localiza el recurso identificado por el correspondiente URL y se lo presenta. En el ejemplo, el recurso es un fichero HTML que especifica una página sobre la Luna.

Para mayor información se puede consultar la referencia oficial del formato html en:

http://www.w3.org/TR/html401/

**URLs.** El propósito de un URL es identificar un recurso de tal forma que permita al navegador (cliente) localizarlo dentro de la Web, en otras palabras, localizar al servidor que tiene el acceso a dicho recurso.

Cada URL, en su forma global, tiene dos componentes:

Esquema: localización-específica-del-esquema.

El primer componente, el esquema, declara qué tipo de URL es. Se precisan los URLs para especificar posiciones de una variedad de recursos y también para especificar una variedad de protocolo de comunicación para recuperarlos.

El Web es abierto con respecto a los tipos de recursos que pueden ser utilizados para acceder, mediante los designadores de esquema de los URLs. Si alguien inventa un nuevo tipo de recurso llamémoslo artefacto. Como es lógico, los navegadores deben disponer de la capacidad para utilizar el nuevo protocolo artefacto, pero esto se puede conseguir añadiendo una aplicación colaboradora o conector.

Un URL http tiene dos tareas importantes que hacer: identificar qué servidor Web mantiene el recurso, e identificar cuál de los recursos del servidor es solicitado.

En general, los URLs de http son de la forma:

http://nombredelservidor [ :puerto] [/nombredelpathdelservidor] [?argumentos]

en el que los elementos entre corchetes son opcionales. Un URL de http siempre comienza con " http://" seguido por un nombre de servidor, expresado como un nombre del Servicio de Nombre de Dominio DNS (Domain Name Service). El nombre DNS del servidor esta seguido opcionalmente por el nombre del "puerto" en el que el servidor escucha las solicitudes. Después viene un nombre de recorrido opcional del recurso del servidor. Si éste no aparece se solicita la página web por

defecto del servidor. Por último, el URL finaliza opcionalmente con un conjunto de argumentos, por ejemplo, cuando un usuario envía las entradas en una forma como una página de consulta de una máquina de búsqueda.

Considérese los URLs:

http://www.cdk3.net/

http://www.w3.org/protocols/Activity.html

http://www.google.com/search?q=kindberg

Pueden ser separados de la forma siguiente:

| Nombre del servidor de DNS | Ruta en el servidor | Argumentos |
|---|---|---|
| www.cdk3.net | (por defecto) | (ninguno) |
| www.w3.org | Protocols/Activity.html | (ninguno) |
| www.google.com | search | q=kindberg |

El primer URL indica la página por defecto proporcionada por www.cdk3.net. El siguiente identifica un fichero en el servidor www.w3.org, cuyo nombre de recorrido es Protocols/Activity.html. El tercer URL especifica una consulta a una máquina de búsqueda.

Pueden haber observado también la presencia de un ancla al final de un URL, las cuales no son parte de los URL sino que están definidas como una parte de la especificación HTML. Únicamente los navegadores interpretan anclas, los navegadores siempre recuperan páginas web completas de los servidores, no partes de ellas indicadas por anclas.

El método para publicar un recurso en la web todavía es difícil de manejar y normalmente precisa intervención humana.

Publicación de un recurso: un usuario debe colocar, en primer lugar, el correspondiente fichero en un directorio al que pueda acceder al servidor web que luego será el que atenderá las peticiones realizadas por Navegadores (clientes) generalmente remotos. Existen unas ciertas convenciones para nombres de recorrido que los servidores reconocen.

**HTTP.** (Protocolo de Transferencia de HiperTexto) Define las formas en las que los navegadores y otros tipos de clientes interaccionan con los servidores web.

Principales características:

**Interacciones petición-respuesta:** HTTP es un protocolo de petición-respuesta. El cliente envía un mensaje de petición al servidor que contiene el URL del recurso solicitado. El servidor localiza el nombre de recorrido y, si existe, devuelve el contenido del fichero en un mensaje de respuesta al cliente. En caso contrario, devuelve un mensaje de error.

**Tipos de contenido:** Cuando un navegador hace una petición, incluye una lista de los tipos de contenido que prefiere. El servidor puede ser capaz de tener esto en cuenta cuando devuelve el contenido al navegador. El servidor incluye el tipo de contenido en el mensaje de respuesta de forma que el navegador sabrá cómo procesarlo. Las cadenas de caracteres que indican el tipo de contenido se llaman tipos MIME, y están estandarizados en el RFC 1521. El conjunto de acciones que un navegador tomará para un tipo de contenido dado es configurable, y el modo de configuración es propio para cada tipo de navegador.

**Un recurso por solicitud:** En la versión 1.0 de HTTP el cliente solicita un recurso por cada petición HTTP. Si por ejemplo una página web contiene nueve imágenes, por ejemplo, el navegador realizará un total de diez peticiones separadas para obtener el contenido completo de la página.

**Control de acceso simple:** Cualquier usuario con una conexión de red a un servidor web puede acceder a cualquiera de los recursos publicados. Si los usuarios desean restringir el acceso a un recurso, pueden configurar el servidor. Los usuarios correspondientes deben probar entonces que tienen derecho para acceder al recurso, por ejemplo tecleando una contraseña (password).

**Características más avanzadas, servicios y páginas dinámicas.** Hasta ahora hemos descrito cómo los usuarios pueden publicar páginas web y otros contenidos almacenados en ficheros Web. El contenido puede cambiar en el tiempo, pero es lo mismo para cualquiera. Sin embargo, mucha de la experiencia de los usuarios del Web es la de los servicios con los que el usuario puede interaccionar, el navegador envía una petición HTTP a un servidor web, que contiene los valores enviados por el usuario.

Puesto que el resultado de la petición depende de los datos introducidos por el usuario, el servidor debe procesar dichos datos. Por tanto, el URL o su componente inicial representa un programa en el servidor, no un archivo. Si los datos introducidos por el usuario son pocos, entonces son enviados como el componente final del URL, siguiendo a un carácter "?". Por ejemplo, una solicitud que contenga el URL siguiente llama a un programa llamado search en www.google.com (servidor) y especifica una consulta para kindberg (argumento):

http://www.google.com/search?q=kindberg.

El programa <<search>> produce como resultado texto HTML, y el usuario verá una lista de páginas que contienen la palabra kindberg. La diferencia entre el contenido estático localizado en un fichero y el contenido que es generado dinámicamente es transparente para el navegador (cliente).

Un programa que se ejecuta en los servidores web para generar contenido para sus clientes se suele llamar, a menudo, programa de Interfaz de Pasarela Común (CGI, Common Gateway Interface). Un programa CGI puede tener una funcionalidad específica de la aplicación. El programa consultará o modificará una base de datos cuando procesa la solicitud.

Código descargado: un programa CGI se ejecuta en el servidor. A veces los diseñadores de servicios web precisan algún código relacionado con el servicio para ser descargado y ejecutado en el navegador (cliente), en el computador del usuario. Por ejemplo, código escrito en Javascript se descarga a menudo con un formulario web para proporcionar una interacción con el usuario de mejor calidad que la proporcionada por los artefactos estándar de HTML.

Por otro lado, un applet es una pequeña aplicación que descarga automáticamente el navegador y se ejecuta cuando se descarga la página correspondiente. Los applets pueden acceder a la red y proporcionar interfaces de usuario específicas, utilizando las posibilidades del lenguaje en el que están implementados.

**Discusión sobre el web**: El extraordinario éxito del web se basa en la facilidad con la que pueden publicarse recursos, una estructura de hipertexto apropiada para la organización de muchos tipos de información, y la extensibilidad arquitectónica del sistema.

El éxito del web contradice algunos principios de diseño. Primero, su modelo de hipertexto es deficiente en algunos aspectos. Si se borra o mueve algún recurso, ocurre que los llamados enlaces descolgados a este recurso permanecen, causando cierta frustración a los usuarios.

También aparece el problema habitual de los usuarios perdidos en el hiperespacio. Los motores de búsqueda claramente imperfectos a la hora de generar lo que quiere concretamente el usuario. Los metadatos describirían los atributos de los recursos web, y serían leídos por herramientas que asistirían a los usuarios que procesaran recursos web masivamente, tal como ocurre al buscar y recopilar listas de enlaces relacionados.

Existe una necesidad creciente de intercambio de muchos tipos de datos estructurados en el Web, pero HTML se encuentra limitado en que no es extensible a aplicaciones más allá de la inspección de la información. HTML consta de un conjunto estático de estructuras tales como los párrafos, que se limitan a indicar la forma en que se presentan los datos a los usuarios.

Como arquitectura del sistema, el web plantea problemas de escala. Los servidores web más populares pueden experimentar muchos accesos por segundo, y como resultado la respuesta a los usuarios se vuelve lenta.

Finalmente, una página web no siempre es una interfaz de usuario satisfactoria. Los elementos de diálogo definidos para HTML son limitados, y los diseñadores incluyen a menudo en las páginas web, pequeñas aplicaciones, o applets, o bien muchas imágenes para darles una función y apariencia más aceptable. Consecuentemente el tiempo de carga se incrementa.

## *1.4* REQUERIMIENTOS

A la hora de diseñar un Sistema Distribuido se debe tener en cuenta cumplir con ciertos requerimientos que permitan a este prestar los servicios esperados a los usuarios del sistema.

Muchos de los desafíos que se discuten en esta sección ya están resueltos, pero los futuros diseñadores necesitan estar al tanto y tener cuidado de considerarlos.

### 1.4.1 HETEROGENEIDAD

Un Sistema Distribuido debe permitir que los usuarios accedan a servicios y ejecuten aplicaciones sobre un conjunto heterogéneo de redes y computadores. Esta heterogeneidad (es decir, variedad y diferencia) se aplica a todos los siguientes elementos:

- Redes.
- Hardware de computadores.
- Sistemas operativos.
- Lenguajes de programación.
- Implementaciones de diferentes desarrolladores.

La capa intermedia entre la aplicación y la capa de transporte que permite que sistemas heterogéneos puedan conversar se denomina Middleware.

**Middleware:** El término *middleware* se aplica al estrato software que provee una abstracción del lenguaje de programación, así como un enmascaramiento de la heterogeneidad subyacente de las redes, hardware, sistemas operativos y lenguajes de

programación. La mayoría de middleware se implementa sobre protocolos de Internet, enmascarando éstos la diversidad de redes existentes.

Aún así cualquier middleware trata con las diferencias de sistema operativo y hardware. El middleware proporciona un modelo computacional uniforme al enlace de los programadores de servidores y aplicaciones distribuidas. Los posibles modelos incluyen invocación sobre objetos remotos, acceso remoto mediante SQL y procesamiento distribuido de transacciones.

**Heterogeneidad y código móvil:** El término código móvil se emplea para referirse al código que puede ser enviado desde un computador a otro y ejecutarse en éste. Dado que el conjunto de instrucciones de un computador depende del hardware, el código de nivel de máquina adecuado para correr en un tipo de computador no es adecuado para ejecutarse en otro tipo.

La aproximación de *máquina virtual* provee un modo de crear código ejecutable sobre cualquier hardware. El compilador de un lenguaje concreto generará código para una máquina virtual en lugar de código apropiado para un hardware particular. Luego la maquina virtual en tiempo de ejecución se encarga de traducir el código en lenguaje de maquina apropiado para el hardware sobre el cual esta trabajando.

Es el modo de trabajo por ejemplo de los applets, los cuales al ser descargados en el computador del usuario son ejecutados por la maquina virtual de Java la cual interpreta el código en lenguaje de maquina apropiado para el hardware del usuario.

### 1.4.2 EXTENSIBILIDAD

La extensibilidad de un sistema de cómputo es la característica que determina si el sistema puede ser extendido y reimplementado en diversos aspectos. La extensibilidad de los sistemas distribuidos se determina en primer lugar por el grado en el cual se pueden añadir nuevos servicios de compartición de recursos y ponerlos a disposición para el uso por una variedad de programas cliente.

No es posible obtener extensibilidad a menos que la especificación y la documentación de las interfaces software clave de los componentes de un sistema estén disponibles para los desarrolladores de software. Es decir, que las interfaces clave estén publicadas. Este procedimiento es similar a una estandarización, que por lo demás suelen ser lentos y complicados.

Los sistemas diseñados de este modo para dar soporte a la compartición de recursos se etiquetan como sistemas distribuidos abiertos para remarcar el hecho de ser extensibles. Pueden ser extendidos en el nivel hardware mediante la inclusión de computadores a la red y en el nivel software por la introducción de nuevos servicios y la reimplementación de los antiguos, posibilitando a los programas de aplicación la compartición de recursos. Otro beneficio más, citado a menudo, de los sistemas abiertos es su independencia de proveedores concretos.

En resumen:

- Los sistemas abiertos se caracterizan porque sus interfaces están publicadas de manera que se busca un estándar para que cualquier nuevo recurso que se ajuste a dicha interface pueda ser acoplado al sistema.

- Los sistemas distribuidos abiertos se basan en la providencia de un mecanismo de comunicación uniforme e interfaces públicas para acceder a recursos compartidos.
- Los sistemas distribuidos abiertos pueden construirse con hardware y software heterogéneo, posiblemente de diferentes proveedores. Sin embargo, la conformidad con el estándar publicado de cada componente debe contrastarse y verificarse cuidadosamente si se desea que el sistema trabaje correctamente.

### 1.4.3 SEGURIDAD

La seguridad de los recursos de información tiene tres componentes: confidencialidad, integridad y disponibilidad.

En un sistema distribuido, los clientes envían peticiones de acceso a datos administrados por servidores, lo que trae consigo enviar información en los mensajes por la red. Por ejemplo:

1. Un médico puede solicitar acceso a los datos hospitalarios de un paciente o enviar modificaciones sobre ellos.
2. En comercio electrónico y banca, los usuarios envían su número de tarjeta de crédito a través de Internet.

La seguridad no sólo es cuestión de ocultar los contenidos de los mensajes, también consiste en conocer con certeza la identidad del usuario u otro agente en nombre del cual se envía el mensaje.

Existen dos desafíos de seguridad que no han sido cumplidos completamente:

**Ataques de denegación de servicio:** *O*tro problema de seguridad ocurre cuando un usuario desea obstaculizar un servicio por alguna razón. Esto se obtiene al bombardear el servicio con un número suficiente de peticiones inútiles de modo que los usuarios serios sean incapaces de utilizarlo. A esto se le denomina ataque de denegación de servicio.

**Seguridad del código móvil:** *E*l código móvil necesita ser tratado con cuidado. Suponga que alguien recibe un programa ejecutable adherido a un correo electrónico: los posibles efectos al ejecutar el programa son impredecibles: por ejemplo, pudiera parecer que presentan un interesante dibujo en la pantalla cuando en realidad están interesados en el acceso a los recursos locales, o quizás pueda ser parte de un ataque de denegación de servicio.

### 1.4.4 ESCALABILIDAD

Los sistemas distribuidos operan efectiva y eficientemente en muchas escalas diferentes, desde pequeñas intranets a Internet. Se dice que un sistema es escalable si conserva su efectividad cuando ocurre un incremento significativo en el número de recursos y el número de usuarios que lo utilizan. Internet proporciona un ejemplo de un sistema distribuido escalable en el que el número de computadores y servicios experimenta un dramático incremento.

El diseño de los sistemas distribuidos escalables presenta los siguientes retos:

**Control del costo de los recursos físicos:** Según crece la demanda de un recurso, debiera ser posible extender el sistema, a un costo razonable, para satisfacerla. Para

que un sistema con "n" usuarios fuera escalable, la cantidad de recursos físicos necesarios para soportarlo debiera ser como mínimo O(n), es decir proporcional a n.

**Control de las pérdidas de prestaciones:** Considere la administración de un conjunto de datos cuyo tamaño es proporcional al número de usuarios o recursos del sistema. Los algoritmos que emplean estructuras jerárquicas se comportan mejor frente al crecimiento de la escala que los algoritmos que emplean estructuras lineales. Pero incluso con estructuras jerárquicas un incremento en tamaño traerá consigo pérdidas en prestaciones: el tiempo que lleva acceder a datos estructurados jerárquicamente es O(log n), donde n es el tamaño del conjunto de datos. Para que un sistema sea escalable, la máxima pérdida de prestaciones no debiera ser peor que esta medida.

**Prevención de desbordamiento de recursos software:** No hay una solución idónea para este problema. Es difícil predecir la demanda que tendrá que soportar un sistema con años de anticipación. Además, sobredimensionar para prever el crecimiento futuro pudiera ser peor que la adaptación a un cambio cuando se hace necesario; por ejemplo las direcciones internet grandes ocupan espacio extra en los mensajes, y en la memoria de los computadores, por esta razón se debió encontrar un punto intermedio entre el tamaño de estas direcciones (uso del recurso memoria) y la cantidad de usuarios que podrían demandar el uso de este recurso, el cual actualmente comienza a ser tan grande que exige que las direcciones Internet sean extendidas.

Evitar los cuellos de botella en las prestaciones: en general, se recomienda que los algoritmos deberían ser descentralizados.

### 1.4.5 TRATAMIENTO DE FALLOS

Los sistemas computacionales a veces fallan, cuando aparecen fallos en el hardware o el software, los programas pueden producir resultados incorrectos o pudieran parar antes de haber completado el cálculo pedido.

Los fallos en un sistema distribuido son parciales; es decir, algunos componentes fallan mientras otros siguen funcionando.

Técnicas para tratar fallo:

**Detección de fallos:** *S*e pueden utilizar sumas de comprobación (*cheksums*) para detectar datos corruptos en un mensaje o un archivo. Es difícil o incluso imposible detectar algunos fallos que no pueden detectarse pero que sí pueden esperarse.

**Enmascaramiento de fallos:** *A*lgunos fallos que han sido detectados pueden ocultarse o atenuarse. Ejemplos de ocultación de fallos son:

Los mensajes pueden retransmitirse cuando falla la recepción.

Los archivos con datos pueden escribirse en una pareja de discos de forma que si uno está deteriorado el otro seguramente está en buen estado.

Eliminar un mensaje corrupto es un ejemplo de atenuar un fallo (pudiera retransmitirse de nuevo).

**Tolerancia de fallo:** La mayoría de los servicios en internet exhiben fallo; es posible que no sea práctico para ellos pretender detectar y ocultar todos los fallos que pudieran aparecer en una red tan grande y con tantos componentes. Sus clientes

pueden diseñarse para tolerar ciertos fallos, lo que implica que también los usuarios tendrán que tolerarlos.

**Recuperación frente a fallos:** La recuperación implica el diseño de software en el que, tras una caída del servidor, el estado de los datos pueda reponerse o retractarse (*roll back*) a una situación anterior. En general, cuando aparecen fallos los cálculos realizados por algunos programas se encontrarán incompletos y al actualizar datos permanentes (archivos e información ubicada en el almacenamiento persistente) pudiera encontrarse en un estado inconsciente.

**Redundancia:** Puede lograrse que los servicios toleren fallos mediante empleo redundante de componentes. Considere los siguientes ejemplos:

1. Siempre deberá haber al menos dos rutas diferentes entre cualquiera de dos *routers* (encaminadores) en internet.
2. En el Sistema de Nombres de Dominio, cada tabla de nombres se encuentra replicada en dos servidores diferentes (Primario y Secundario) como mínimo.
3. Una base de datos puede encontrarse replicada en varios servidores para asegurar que los datos siguen siendo accesibles tras el fallo de cualquier servidor concreto; los servidores pueden diseñarse para detectar fallos entre sus iguales; cuando se detecta algún error en un servidor se redirigen los clientes a los servidores restantes.

Los sistemas distribuidos proporcionan un alto grado de disponibilidad frente a los fallos del hardware. La *disponibilidad* de un sistema mide la proporción de tiempo en que está utilizable. Cuando falla algún componente del sistema distribuido solo resulta afectado el trabajo relacionado con el componente defectuoso. Así como

cuando un computador falla el usuario puede desplazarse a otro, también puede iniciarse un proceso de servicio en otra ubicación.

### 1.4.6 CONCURRENCIA.

Tanto los servicios como las aplicaciones proporcionan recursos que pueden compartirse entre los clientes en un sistema distribuido. Existe por lo tanto una posibilidad de que varios clientes intenten acceder a un recurso compartido a la vez.

Cada objeto que represente un recurso compartido en un sistema distribuido debe responsabilizarse de garantizar que opera correctamente en un entorno concurrente. De este modo cualquier programador que recoge una implementación de un objeto que no está concebido para su aplicación en un entorno distribuido deberá realizar las modificaciones necesarias para que sea seguro su uso en el entorno concurrente.
Para que un objeto sea seguro en un entorno concurrente, sus operaciones deben sincronizarse de forma que sus datos permanezcan consistentes. Esto puede lograrse mediante el empleo de técnicas conocidas como los semáforos, que se usan en la mayoría de los sistemas operativos.

### 1.4.7 TRANSPARENCIA

Se define transparencia como la ocultación al usuario y al programador de aplicaciones de la separación de los componentes en un sistema distribuido, de forma que se perciba el sistema como un todo más que como una colección de componentes

independientes y separados. Las implicaciones de la transparencia son de gran calado en el diseño del software del sistema. Algunas transparencias son:

**Transparencia de acceso.-** Permiten acceder a los recursos locales y remotos empleando operaciones idénticas.

**Transparencia de ubicación.-** Permiten acceder a los recursos sin conocer su localización.

**Transparencia de concurrencia.-** Permiten que varios procesos operen concurrentemente sobre recursos compartidos sin interferencia mutua.

**Transparencia de replicación.-** Permiten utilizar múltiples ejemplares de cada recurso para aumentar la fiabilidad y las prestaciones sin que los usuarios y los programadores de aplicaciones necesiten su conocimiento.

**Transparencia frente a fallos.-** Permiten ocultar los fallos, dejando que los usuarios y programas de aplicación completen sus tareas a pesar de fallos de hardware o de los componentes software.

**Transparencia de movilidad.-** Permiten la reubicación de recursos y clientes en un sistema sin afectar la operación de los usuarios y los programas.

**Transparencia de prestaciones.-** Permiten reconfigurar el sistema para mejorar las prestaciones según su carga. Esto significa que aunque el sistema es reconfigurado con el aumento de recursos por ejemplo los clientes no ven mermada la calidad de las prestaciones recibidas.

**Transparencia al escalado.-** Permiten al sistema y a las aplicaciones expandirse en tamaño sin cambiar la estructura del sistema o los algoritmos de aplicación.

Las dos más importantes son transparencia de acceso y la transparencia de ubicación; su presencia o ausencia afecta principalmente a la utilización de recursos distribuidos. A veces se les da nombre conjunto de *transparencia de red.*

### 1.4.8 FIABILIDAD

Una de las ventajas claras que nos ofrece la idea de sistema distribuido es que el funcionamiento de todo el sistema no debe estar ligado a ciertas máquinas de la red, sino que cualquier equipo pueda suplir a otro en caso de que uno se estropee o falle; otro tipo de redundancia más compleja se refiere a los procesos. Las tareas críticas podrían enviarse a varios procesadores independientes, de forma que el primer procesador realizaría la tarea normalmente, pero ésta pasaría a ejecutarse en otro procesador si el primero hubiera fallado.

# CAPITULO II

# COMUNICACIÓN ENTRE PROCESOS REMOTOS

## 2.1 CARACTERÍSTICAS GENERALES DE LA COMUNICACIÓN ENTRE PROCESOS

La comunicación entre procesos en sistemas con un único procesador se lleva a cabo mediante el uso de memoria compartida entre los procesos. En los sistemas distribuidos, al no haber conexión física entre las distintas memorias de los equipos, la comunicación se realiza mediante la transferencia de mensajes.

La transferencia de mensajes en un sistema distribuido plantea la implementación de los siguientes mecanismos:

Llamadas a procedimiento Remoto: RPC (Remote Procedure Call)

El diseño de un sistema operativo distribuido plantea las llamadas a procedimientos remotos o RPCs. Los RPC amplían la llamada local a procedimientos, y los generalizan a una llamada a un procedimiento localizado en cualquier lugar de todo el sistema distribuido. En un sistema distribuido no se debería distinguir entre llamadas locales y RPCs, lo que favorece en gran medida la transparencia del sistema.

La limitación del RPC más clara en los sistemas distribuidos es que no permite enviar una solicitud y recibir respuesta de varias fuentes a la vez, sino que la

comunicación se realiza únicamente entre dos procesos. Por motivos de tolerancia a fallos, bloqueos, u otros, sería interesante poder tratar la comunicación en grupo.

Sincronización:

La sincronización en sistemas de un único ordenador no requiere ninguna consideración en el diseño del sistema operativo, ya que existe un reloj único que proporciona de forma regular y precisa el tiempo en cada momento. Sin embargo, los sistemas distribuidos tienen un reloj por cada ordenador del sistema, con lo que es fundamental una coordinación entre todos los relojes para mostrar una hora única. Los osciladores de cada ordenador son ligeramente diferentes, y como consecuencia todos los relojes sufren un desfase y deben ser sincronizados continuamente. La sincronización no es trivial, porque se realiza a través de mensajes por la red, cuyo tiempo de envío puede ser variable y depender de muchos factores, como la distancia, la velocidad de transmisión o la propia saturación de la red, etc.

La sincronización no tiene por qué ser exacta, y bastará con que sea aproximadamente igual en todos los ordenadores. Hay que tener en cuenta, eso sí, el modo de actualizar la hora de un reloj en particular. Es fundamental no retrasar nunca la hora, aunque el reloj adelante. En vez de eso, hay que demorar la actualización del reloj, frenarlo, hasta que alcance la hora aproximadamente. Existen diferentes algoritmos de actualización de la hora.

Modelos de Acceso a los Archivos

Debido a la complejidad del acceso a los archivos a través de todo el sistema distribuido, surgen dos modelos para el acceso a los archivos: el modelo carga / descarga, y el modelo de acceso remoto. El primer modelo simplifica el acceso permitiendo únicamente las operaciones de cargar y descargar un archivo. El acceso a cualquier parte del archivo implica solicitar y guardar una copia local del archivo completo, y sólo se puede escribir de forma remota el archivo completo. Este método sería especialmente ineficaz a la hora de realizar pequeñas modificaciones en archivos muy grandes, como podrían ser bases de datos. El modelo de acceso remoto es mucho más complejo, y permite todas las operaciones típicas de un sistema de archivos local.

Memoria compartida distribuida

La memoria compartida distribuida o DSM (Disributed Shared Memory) es una abstracción que se propone como alternativa a la comunicación por mensajes.

El esquema de DSM propone un espacio de direcciones de memoria virtual que integre la memoria de todas las computadoras del sistema, y su uso mediante paginación. Las páginas quedan restringidas a estar necesariamente en un único ordenador. Cuando un programa intenta acceder a una posición virtual de memoria, se comprueba si esa página se encuentra de forma local. Si no se encuentra, se provoca un fallo de página, y el sistema operativo solicita la página al resto de computadoras. Los fallos de página se propagan al resto de ordenadores, hasta que la petición llega al ordenador que tiene la página virtual solicitada en su memoria local.

Una alternativa al uso de páginas es tomar el objeto como base de la transferencia de memoria. Aunque el control de la memoria resulta más complejo, el resultado es al mismo tiempo modular y flexible, y la sincronización y el acceso se pueden integrar limpiamente. Todos los accesos a los objetos compartidos han de realizarse mediante llamadas a los métodos de los objetos, con lo que no se admiten programas no modulares y se consideran incompatibles.

Modelos de Consistencia: La duplicidad de los bloques compartidos aumenta el rendimiento, pero produce un problema de consistencia entre las diferentes copias de la página en caso de una escritura. Si con cada escritura es necesario actualizar todas las copias, el envío de las páginas por la red provoca que el tiempo de espera aumente demasiado, convirtiendo este método en impracticable. Para solucionar este problema se proponen diferentes modelos de consistencia, que establezcan un nivel aceptable de acercamiento tanto a la consistencia como al rendimiento. Nombramos algunos modelos de consistencia, del más fuerte al más débil: consistencia estricta, secuencial, causal, PRAM (http://wwwdi.ujaen.es/asignaturas/ASO/Teoria/4MemDistr.pdf), del procesador, débil, de liberación y de entrada.

### 2.1.1 USO DE SOCKETS

Los sockets no son más que puntos o mecanismos de comunicación entre procesos que permiten que un proceso hable (emita o reciba información) con otro proceso incluso estando estos procesos en distintas máquinas.

Un **socket** es al sistema de comunicación entre ordenadores lo que un buzón o un teléfono es al sistema de comunicación entre personas: un punto de comunicación entre dos agentes (procesos o personas respectivamente) por el cual se puede emitir o recibir información. La forma de referenciar un socket por los procesos implicados es mediante un **descriptor** del mismo tipo que el utilizado para referenciar ficheros. Debido a esta característica, se podrá realizar redirecciones de los archivos de E/S estándar (descriptores 0,1 y 2) a los sockets y así combinar entre ellos aplicaciones de la red. Todo nuevo proceso creado heredará, por tanto, los descriptores de sockets de su padre.

La comunicación entre procesos a través de sockets se basa en la filosofía **CLIENTE-SERVIDOR**: un proceso en esta comunicación actuará de **proceso servidor** creando un socket cuyo nombre conocerá el **proceso cliente**, el cual podrá "hablar" con el proceso servidor a través de la conexión con dicho **socket nombrado**.

El proceso crea un socket sin nombre cuyo valor de vuelta es un descriptor sobre el que se leerá o escribirá, permitiéndose una comunicación **bidireccional**, característica propia de los sockets y que los diferencia de los **pipes**, o canales de comunicación unidireccional entre procesos de una misma máquina. El mecanismo de comunicación vía sockets tiene los siguientes pasos:

1) El proceso servidor crea un socket con nombre y espera la conexión.
2) El proceso cliente crea un socket sin nombre.

3) El proceso cliente realiza una petición de conexión al socket servidor.
4) El cliente realiza la conexión a través de su socket mientras el proceso servidor mantiene el socket servidor original con nombre.

Es muy común en este tipo de comunicación lanzar un proceso hijo dentro del servidor, una vez realizada la conexión, que se ocupe del intercambio de información con el proceso cliente mientras el proceso padre servidor sigue aceptando conexiones. Para eliminar esta característica se cerrará el descriptor del socket servidor con nombre en cuanto realice una conexión con un proceso socket cliente.

Todo socket viene definido por dos características fundamentales:
El **tipo** del socket, que indica la naturaleza del mismo, el tipo de comunicación que puede generarse entre los sockets.

El **dominio** del socket especifica el conjunto de sockets que pueden establecer una comunicación con el mismo.

Tipos de sockets

Definen las propiedades de las comunicaciones en las que se ve envuelto un socket, esto es, el tipo de comunicación que se puede dar entre cliente y servidor. Estas pueden ser:

- Fiabilidad de transmisión.
- Mantenimiento del orden de los datos.
- No duplicación de los datos.

- El "Modo Conectado" en la comunicación.
- Envío de mensajes urgentes.

Los tipos disponibles son los siguientes:

- Tipo **SOCK_DGRAM**: sockets para comunicaciones en **modo no conectado**, con envío de **datagramas** de tamaño **limitado** (tipo telegrama). En dominios Internet como la que nos ocupa el protocolo del nivel de transporte sobre el que se basa es el **UDP**.
- Tipo **SOCK_STREAM**: para comunicaciones **fiables en modo conectado**, de **dos vías** y con tamaño **variable** de los mensajes de datos. Por debajo, en dominios Internet, subyace el protocolo **TCP**.
- Tipo **SOCK_RAW**: permite el acceso a protocolos de más bajo nivel como el **IP** ( nivel de red)
- Tipo **SOCK_SEQPACKET**: tiene las características del SOCK STREAM pero además el tamaño de los mensajes es fijo.

<u>El dominio de un socket.</u>

Indica el formato de las direcciones que podrán tomar los sockets y los protocolos que soportarán dichos sockets.

La estructura genérica es:

**struct sockaddr {**

```
        u  short   sa__family;      /* familia */
        char       sa__data[14];    /* dirección */
};
```

Pueden ser:

- Dominio **AF_UNIX** (Address Family UNIX):

El cliente y el servidor deben estar en la misma máquina. Debe incluirse el fichero cabecera **/usr/include/sys/un.h**. La estructura de una dirección en este dominio es:

```
struct sockaddr__un {
  short       sun__family;  /* en este caso AF  UNIX */
  char        sun__data[108]; /* dirección */
  };
```

* Dominio **AF_INET** (Address Family INET):

El cliente y el servidor pueden estar en cualquier máquina de la red Internet. Deben incluirse los ficheros cabecera **/usr/include/netinet/in.h, /usr/include/arpa/inet.h, /usr/include/netdb.h**. La estructura de una dirección en este dominio es:

```
        struct in   addr {
        u__long     s__addr;
```

```
};

struct sockaddr__in {
short        sin_family;  /*en este caso AF  INET */
u__short     sin_port;   /*numero del puerto */
struct in__addr        sin__addr; /*direcc Internet */
char         sin_zero[8];    /*campo de 8 ceros */
};
```

Estos dominios van a ser los utilizados en xshine. Pero existen otros como:

- Dominio **AF NS**:

Servidor y cliente deben estar en una red XEROX.

- Dominio **AF CCITT**:

Para protocolos CCITT, protocolos X25, etc.

**FILOSOFIA CLIENTE-SERVIDOR:**

**El Servidor**: Vamos a explicar el proceso de comunicación servidor-cliente en **modo conectado**, modo utilizado por las aplicaciones estándar de Internet (telnet, ftp). El servidor es el proceso que crea el socket no nombrado y acepta las conexiones a él. El orden de las llamadas al sistema para la realización de esta función es:

1°) int **socket** ( int *dominio*, int *tipo*, int *protocolo*)

Crea un socket sin nombre de un dominio, tipo y protocolo específico

*dominio* : AF_INET, AF_UNIX

*tipo* : SOCK__DGRAM, SOCK__STREAM

*protocolo* : 0 ( protocolo por defecto )

2°) int **bind** ( int *dfServer*, struct sockaddr* *direccServer*, int *longDirecc* )

Nombra un socket: asocia el socket no nombrado de descriptor *dfServer* con la dirección del socket almacenado en *direccServer*. La dirección depende de sí estamos en un dominio AF UNIX o AF INET.

3°) int **listen** ( int *dfServer*, int *longCola* )

Especifica el máximo número de peticiones de conexión pendientes.

4°) int **accept** ( int *dfServer*, struct sockaddr* *direccCliente*, int* *longDireccCli*)

Escucha al socket nombrado servidor *dfServer* y espera hasta que se reciba la petición de la conexión de un cliente. Al ocurrir esta incidencia, crea un socket sin nombre con las mismas características que el socket servidor original, lo conecta al socket cliente y devuelve un descriptor de fichero que puede ser utilizado para la comunicación con el cliente.

**El Cliente:** Es el proceso encargado de crear un socket sin nombre y posteriormente enlazarlo con el socket servidor nombrado. O sea, es el proceso que demanda una conexión al servidor. La secuencia de llamadas al sistema es:

1°) int **socket** ( int *dominio*, int *tipo*, int *protocolo*)

Crea un socket sin nombre de un dominio, tipo y protocolo específico

*dominio*: AF_INET, AF_UNIX

*tipo*: SOCK__DGRAM, SOCK__STREAM

*protocolo*: 0 ( protocolo por defecto )

2°) int **connect** ( int *dfCliente*, struct sockaddr* *direccServer*, int *longDirecc* )

Intenta conectar con un socket servidor cuya dirección se encuentra incluida en la estructura apuntada por *direccServer*. El descriptor *dfCliente* se utilizará para comunicar con el socket servidor. El tipo de estructura dependerá del dominio en que nos encontremos.

Una vez establecida la comunicación, los descriptores de ficheros serán utilizados para almacenar la información a leer o escribir.

| SERVIDOR | CLIENTE |
|---|---|
| descrServer = **socket** ( dominio, SOCK_STREAM,PROTOCOLO) | descrClient = **socket** (dominio, SOCK_STREAM,PROTOCOLO) |
| **bind** (descrServer, PuntSockServer,longServer) | |
| | do { |
| **listen** (descrServer, longCola) | |

| | |
|---|---|
| descrClient = **accept** (descrServer,PuntSockClient,longClient) | result = **connect** (descrClient, PuntSockServer,longserver) |
| [ **close** (descrServer) ] | } while ( result == -1 ) |
| < DIALOGO > | < DIALOGO > |
| **close** (descrClient) | **close** (descrClient) |

## COMPARACIÓN SOCKETS-PIPES COMO MECANISMOS DE COMUNICACIÓN ENTRE PROCESOS

| SOCKETS | PIPES |
|---|---|
| refenciado por descriptores | referenciado por array de descriptores |
| admite comunicación entre procesos de distintas máquinas | sólo admite comunicación entre procesos de la misma máquina |
| comunicación bidireccional | comunicación unidireccional |
| filosofía cliente-servidor | simple intercambio de información |

## 2.1.2 COMUNICACIÓN DE DATAGRAMAS UDP

UDP (Protocolo de Datagrama de Usuario) es un protocolo no orientado a la conexión, utilizado comúnmente con IP. Es más simple que TCP ya que confía en un servicio de red seguro, por lo que las funciones de recuperación frente a errores y desorden no las posee. El UDP incluye toda la información en cada mensaje y se emplea para la gestión remota de la red y para servicios de acceso por mnemónico.

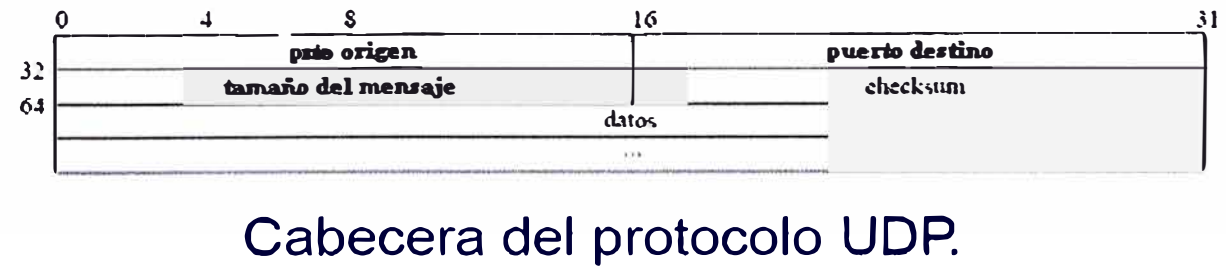
Características

- Es casi una replica en el nivel de transporte de IP.
- Cada datagrama UDP se encapsula en un paquete IP. Tiene una cabecera corta que incluye los números de puerto origen y destino (las direcciones correspondientes a los hosts están en la cabecera IP), un campo de longitud y otro de suma de comprobación.
- No ofrece garantía de entrega. Los datagramas pueden desecharse en caso de congestión o error de la red.
- No ofrece ningún mecanismo adicional de fiabilidad excepto la suma de comprobación que es opcional. Si el valor de la suma de comprobación es distinto de cero, el host receptor calcula el valor de comprobación para el contenido del paquete y lo compara con el valor recibido en el paquete, desechándolo en el caso de que estos no coincidan.
- Proporciona un modo de comunicación entre procesos con mensajes de hasta 64Kbytes (máximo permitido por IP), con un coste adicional y un retardo de transmisión mínimos sobre aquellos debidos a la transmisión IP.

- No existe un coste asociado a la configuración y no necesita mensajes de reconocimiento.
- Su uso es restringido a aquellas aplicaciones y servicios que no requieran una entrega fiable de mensajes simples o múltiples.

Formato del datagrama UDP

Cada datagrama UDP se envía en un sólo datagrama de IP. Aunque el datagrama IP se fragmente durante la transmisión, la implementación de IP que lo reciba lo reensamblará antes de pasárselo a la capa de UDP. Todas las implementaciones de IP deben aceptar datagramas de 576 bytes, lo que significa que si se supone un tamaño máximo de 60 bytes para la cabecera IP, queda un tamaño de 516 bytes para el datagrama UDP, aceptado por todas las implementaciones. Muchas implementaciones aceptan datagramas más grandes, pero no es algo que esté garantizado. El datagrama UDP tiene una cabecera de 16 bytes que se describe en la figura 2.1:




Cabecera del protocolo UDP.

**Figura 2.1**

Donde:

Puerto origen:

Indica el puerto del proceso que envía el datagrama. Es el puerto al que se deberían dirigir las respuestas.

Puerto destino:

Especifica el puerto destino en el host de destino.

Tamaño del mensaje:

Es la longitud(en bytes) del mismo datagrama de usuario, incluyendo la cabecera.

Checksum

Es un campo opcional consistente en el complemento a uno de 16 bits de la suma en complemento a uno de una pseudocabecera IP, la cabecera UDP y los datos del datagrama UDP. La pseudocabecera IP contiene las direcciones IP de origen y destino, el protocolo y la longitud del datagrama UDP.

### 2.1.3 COMUNICACIÓN DE STREAMS TCP

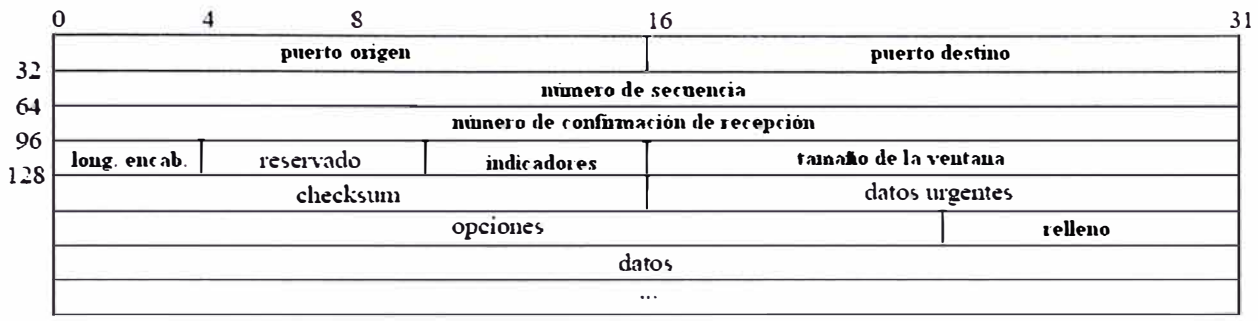
TCP (Transport Control Protocolo) es el protocolo que proporciona un transporte fiable de flujo de bits entre aplicaciones. Esta pensado para poder enviar grandes cantidades de información de forma fiable, liberando al programador de aplicaciones de la dificultad de gestionar la fiabilidad de la conexión (retransmisiones, perdidas de paquete, orden en que llegan los paquetes, duplicados de paquetes, ...) que gestiona el propio protocolo. Pero la complejidad de la gestión de la fiabilidad tiene un coste en eficiencia, ya que para llevar a cabo las gestiones anteriores se tiene que añadir bastante información a los paquetes a enviar. Debido a que los paquetes a enviar

tienen un tamaño máximo, cuanta más información añada el protocolo para su gestión, menos información que proviene de la aplicación podrá contener ese paquete. Por eso, cuando es más importante la velocidad que la fiabilidad, se utiliza UDP, en cambio TCP asegura la recepción en destino de la información a transmitir.

Características

- Proporciona un servicio de transporte mucho más sofisticado. Proporciona entrega fiable de secuencias de bytes arbitrariamente grandes por vía de la abstracción de la programación basada en streams.
- La garantía de fiabilidad implica la entrega al proceso receptor de todos los datos confiados al protocolo TCP por el proceso emisor y en el mismo orden.
- Esta orientado a conexión, por lo tanto antes de transferir cualquier dato, el proceso emisor y el receptor deben cooperar para establecer un canal de comunicación bidireccional.
- Los nodos intermediarios como los routers no tienen conocimiento de las conexiones TCP, y los paquetes IP que transfieren los datos de una transmisión TCP no tienen que seguir necesariamente el mismo camino.

Encabezado del segmento TCP:




Cabecera del protocolo TCP

**Figura 2.2**

Donde:

Puerto origen y puerto destino identifican los puntos terminales locales de la conexión.

Número de secuencia; para identificar los paquetes.

Numero de confirmación de recepción; especifica el siguiente byte esperado.

Longitud del encabezado (long. encab.); indica el número de palabras de 32 bits contenidas en el encabezado TCP. Esta información es necesaria ya que el campo de opciones es de longitud variable, por lo que el encabezado también.

Reservado; campo de 6 bits que no se usa, protocolos anteriores lo habrían usado para corregir errores del diseño original.

Indicadores; 6 bits de control.

Tamaño de la ventana; indica la cantidad de bytes que pueden enviarse comenzando por el byte cuya recepción se ha confirmado. Es valido recibir un campo tamaño de ventana 0, e indica que se han recibido los bytes hasta número de confirmación -1, inclusive, pero que el receptor actualmente necesita un descanso y quisiera no recibir mas datos por el momento.

Cheksum o suma de verificación; es una suma de verificación del encabezado y sirve para agregar confiabilidad.

Opciones; ofrece una forma de agregar características extra no cubiertas por el encabezado normal. La opción más importante es la que permite que cada host especifique la carga útil TCP máxima que esta dispuesto a aceptar.

## 2.2 REPRESENTACIÓN EXTERNA DE DATOS Y EMPAQUETADO.

La información almacenada dentro de los programas en ejecución se representa mediante estructuras de datos (por ejemplo, por conjunto de objetos interrelacionados) mientras que la información transportada en los mensajes consiste en secuencias de bytes. Independientemente de la forma de comunicación utilizada. Las estructuras de datos deben ser aplanadas (convertidas a una secuencia de bytes antes de su transmisión y reconstruidas en el destino. Los tipos de datos primitivos transmitidos en los mensajes pueden tener valores de muchos tipos distintos y no todos los computadores almacenan los valores primitivos tales como los enteros, en

el mismo orden. También la representación de los números en coma flotante es diferente para cada arquitectura. Otro problema es el conjunto de códigos utilizado para representar los caracteres: por ejemplo UNIX utiliza la codificación ASCII con un byte por carácter, mientras que el estándar Unicode permite representar textos en la mayoría de los lenguajes y utiliza dos bytes por carácter. Existen dos variantes en la ordenación de enteros: la llamada big-endian en la que el byte más significativo va al principio y la llamada Little – endian, en la que va al último.

Para hacer posible que dos computadores puedan intercambiar datos se puede utilizar uno de los dos métodos siguientes:

- Los valores se convierten a un formato externo acordado antes de la transmisión y se revierten al formato local en la recepción: si los dos computadores son del mismo tipo y lo saben se puede omitir la transformación al formato externo.
- Los valores se transmiten según el formato del emisor, junto a una indicación del formato utilizado y el receptor los convierte si es necesario.

Hay que hacer notar, sin embargo, que los bytes no son alterados durante la transmisión. Para soportar RMI o RPC, cualquier tipo de dato que pueda ser pasado como un argumento o devuelto como resultado debe ser capaz de ser aplanado y cada uno de los tipos de datos primitivos representados en una representación de datos acordada. Al Estándar acordado para la representación de estructuras de datos y valores primitivos se denomina representación externa de datos.

El empaquetado (marshalling) consiste en tomar una colección de ítems de datos y ensamblarlos de un modo adecuado para la transmisión en un mensaje. El desempaquetado (unmarshalling) es el proceso de desensamblado en el destino para

producir una colección equivalente de datos y los valores primitivos en una representación externa de datos externa y reconstruir las estructuras de datos.

Las dos alternativas para la representación externa de datos y el empaquetado son:

- La representación común de datos de CORBA, que incumbe a una representación externa para los datos estructurados y primitivos que pueda ser pasada con argumento y resultado de la invocación remota de métodos en CORBA. Puede ser utilizada por una gran variedad de lenguajes de programación.
- La serialización de objetos JAVA que esta relacionado con el aplanado y la representación externa de datos de cualquier objeto simple o un árbol de objetos que tienen que ser transmitiros en un mensaje o almacenados en disco. Es de uso exclusivo de Java.

En ambos casos, el empaquetado y el desempaquetado son llevados a cabo por una capa de middleware sin ninguna participación del programador de la aplicación. Debido a que el empaquetado requiere consideración de todos los detalles de bajo nivel de la representación de los componentes primitivos de los objetos compuestos, el proceso es propenso a fallar cuando se hace a mano. La eficiencia es otro criterio que resulta deseable en el diseño de los procedimientos de empaquetado generados de forma automática.

Las dos propuestas mencionadas aquí empaquetan los tipos de datos primitivos de forma binaria. Una alternativa es empaquetar todos los objetos a transmitir en texto ASCCI, lo cual es relativamente simple de implementar, pero implica un empaquetado generalmente más grande. El protocolo HTTP tratado en el apartado 1.3 es un ejemplo de esta última aproximación.

### 2.2.1 REPRESENTACIÓN COMUN DE DATOS DE CORBA (CDR)

CORBA CDR es la representación externa de datos definida en CORBA y puede representar todos los tipos de datos a utilizar como argumentos o como resultados en las invocaciones remotas de Corba. Consta De 15 tipos primitivos, que incluyen los tipos short (16 bit) Long (32 bit) unsigned short (sin signo) unsigned long, flota (32 bits) double (64 bits) char boolean (TRUE FALSE) octet (8 bit) y any (el cual puede representar cualquier tipo básico o compuesto); junto con un abanico de tipos compuestos, que se describen en la tabla mas adelante. Cada argumento o resultado en una invocación remota se representa como una secuencia de bytes en el mensaje de la invocación o en el del resultado.

Tipos primitivos: CDR define una representación tanto para el orden big-endian como para el little-endian. Los valores se transmiten en el orden del emisor, que se especifica en cada mensaje. El receptor lo traduce si es necesario a un orden diferente. Por ejemplo, un short de 16 bits ocupa dos bytes en el mensaje, y en el orden big -endian los bits más significativos ocupan el primer byte y los menos significativos el segundo. Cada valor primitivo se coloca en una posición en la secuencia de bytes desde cero. Entonces un valor primitivo cuyo tamaño es n bytes (donde n= 1,2,4 u 8) será añadido a la secuencia en una posición que sea un múltiplo de n en el flujo de bytes.

Los valores de coma flotante siguen el estándar IEEE en el cual el signo, el exponente y la parte fraccionaria están en los bytes 0-n para el orden big-endian y en

el orden inverso para el little-endian. Los caracteres se representan por un conjunto de caracteres acordado entre el cliente y el servidor.

Tipos compuestos: Los valores primitivos que componen cada tipo compuesto se añaden a la secuencia en un orden particular según se muestra en la Tabla 2.1 que viene a continuación.

| Tipo | Representación |
|---|---|
| sequence | Longitud (unsigned long-entero largo sin signo) seguida de los elementos en orden |
| string | Longitud (unsigned long) seguida de los caracteres en orden (también puede tener caracteres anchos -2 bytes-) |
| array | Elementos de la cadena en orden (no se especifica la longitud porque es fija) |
| struct | En el orden de declaración de los componentes |
| enumerated | Unsigned long (los valores son especificados por el orden declarado) |
| Union | Etiqueta de tipos seguida por el miembro seleccionado |

**Tabla 2.1 - Tipos compuestos CDR de CORBA**

Otro ejemplo de una representación externa de datos es el estándar XDR de Sun, el cual está especificado en el RFC 1832 (Srinivasan 1995b) y descrito en www.cdk3.net/ipc Fue desarrollado por Sun para utilizarlo en el intercambio de mensajes entre clientes y servidores en Sun NFS.

El tipo de un elemento de datos no acompaña a la representación de los datos en el mensaje, ni en el CDR de CORBA ni el XDR de Sun. Esto es porque se supone que tanto el emisor como el receptor tienen un conocimiento común del orden y de los

tipos de datos de los ítems en el mensaje. En particular para RMI o RPC, cada invocación de método pasa argumentos de unos tipos concretos y el resultado es un valor de un tipo dado.

**Empaquetado en CORBA:** Las operaciones de empaquetado se pueden generar automáticamente a partir de las especificaciones de los tipos de datos de los ítems que tienen que ser transmitidos en un mensaje. Los tipos de las estructuras de datos y los tipos de los ítems de datos básicos están descritos en CORBA IDL (el cual es tratado a profundidad en el apartado 4.5), que proporciona una notación para describir los tipos de los argumentos y los resultados de los métodos RMI.

La interfaz del compilador CORBA genera las operaciones de empaquetado y desempaquetado apropiadas para los argumentos y el resultado de los métodos remotos a partir las definiciones de los tipos de sus parámetros y resultados.

### 2.2.2 SERIALIZACIÓN DE OBJETOS EN JAVA

En Java RMI tanto los objetos como los datos primitivos pueden ser pasados como argumentos y resultados de la invocación de los métodos. Un objeto es una instancia de una clase Java. Por ejemplo: primero se muestra la representación externa definida en IDL y luego la clase Java equivalente a struct persona:

```
Struct persona {
String nombre;
String lugar;
```

```
Long año;
 }:

Public class persona implements Serializable{
Private String nombre;
Private String lugar;
Private int año;
Public Personal (String unNombre.  String unLugar,int unAño) {
nombre = unNombre ;
lugar = unLugar
año =  unaño ;
 }
 // Seguido por los métodos para acceder a los campos
```

La clase en Java declara implementar la interfaz Serializable, la cual no tiene métodos. Declarar que una clase implementa la interfaz Serializable (que viene dada en el paquete java.io) tiene el efecto de permitir que sus instancias sean serializables. En Java, el término serialización se refiere a la actividad de aplanar (dar formato de secuencia de bytes) un objeto o un conjunto relacionado de objetos para obtener una forma lineal adecuada para ser almacenada en disco o para ser transmitida en un mensaje, por ejemplo, como argumento o como resultado de un RMI. La deserialización consiste en restablecer el estado de un objeto o un conjunto de objetos desde su estado lineal. Se asume que el proceso que realiza la deserialización no tiene conocimiento previo de los tipos de los objetos en la forma lineal. Por lo

tanto, debe incluirse en la forma lineal alguna información sobre la clase de cada objeto. Esta información posibilita al receptor la carga de la clase apropiada cuando un objeto es deserializado.

La información sobre la clase se compone del nombre de la clase y un número de versión. El número de versión está pensado para cambiar y reflejar con este cambio modificaciones importantes en la clase. Puede ser establecido por el programa o calculado de forma automática como un valor dependiente del nombre de la clase, sus campos métodos e interfaces. El proceso que deserializa un objeto puede comprobar que tiene la versión correcta de la clase.

Los Objetos Java pueden contener referencias a otros objetos. Cuando un objeto es serializado todos los objetos a los que referencia son serializados con él para asegurarse de que cuando el objeto sea reconstruido, todas sus referencias pueden ser rellenadas en el destino. Las referencias se serializan como apuntadores (handlers) (en este caso, el apuntador es una referencia a un objeto dentro de la forma serializada, por ejemplo, el siguiente número en una secuencia de enteros positivos). El procedimiento de serialización debe asegurarse de que existe una correspondencia 1-1 entre las referencias y los apuntadores. También debe asegurarse de que cada objeto sea escrito sólo una vez (en la segunda o subsecuentes ocurrencias de un objeto, se escribirá el apuntador en lugar del objeto).

Para serializar un objeto, se escribe la información de su clase, seguida de los tipos y los nombres de los campos. Si los campos pertenecen a una nueva clase, entonces también se escribe la información de la clase, seguida de los nombres y los tipos de sus campos. Este procedimiento recursivo continúa hasta que se haya escrito la información de todas las clases necesarias y de los nombres y tipos de sus campos.

Cada clase recibe un apuntador, y ninguna clase se escribe más de una vez en el flujo de bytes (en su lugar se escribe su apuntador cuando sea necesario)

Los contenidos de los campos que sean tipos de datos primitivos, como enteros, caracteres booléanos, bytes y enteros largos, se escriben en un formato binario transportable utilizando métodos de la clase ObjetctOutputScream. Las cadenas de caracteres y los caracteres se escriben con los métodos llamados writeUTF utilizando el Formato de Transferencia Universal (Universal Transfer Format UTF) que hace que los caracteres ASCII sean representados sin cambios(en un byte)mientras que los caracteres Unicote (ver la referencia en http://www.ifla.org/IV/ifla65/papers/079-155s.htm) se representan por varios bytes. Las cadenas de caracteres se preceden por el número de bytes que ocupan en el flujo.

### 2.2.3 REFERENCIA A OBJETOS REMOTOS

Cuando un cliente invoca un método en un objeto remoto, se envía un mensaje de invocación al proceso servidor que alberga el objeto remoto. Este mensaje necesita especificar el objeto particular cuyo método se va a invocar. Una referencia a un objeto remoto es un identificador para un objeto remoto que es válida a lo largo y ancho de un sistema distribuido. En el mensaje de invocación se incluye una referencia a objeto remoto que especifica cual es el objeto invocado, el capítulo 3 explica que las referencias a objetos remotos también pasan como argumentos y se devuelven como resultados de la invocación de métodos remotos, que cada objeto

remoto tiene una única referencia al objeto remoto y que las referencias a objetos remotos pueden compararse para saber si se refieren al mismo objeto remoto.

Ahora trataremos sobre la representación externa de las referencias a objetos remotos.

| 32 bits | 32 bits | 32 bits | 32 bits | |
|---|---|---|---|---|
| Dirección internet | Número de puerto | Tiempo | Número de objeto | Interfaz de objeto remoto |

**Tabla 2.2 - Representación de una referencia de objeto remoto**

Las referencias a objetos remotos deben generarse de modo que se asegure su unicidad sobre el espacio y el tiempo. En general, existirán varios procesos alojando objetos remotos, de modo que las referencias a objetos remotos deber ser únicas entre todos los procesos en los computadores de un sistema distribuido. Es importante que la referencia al objeto remoto no sea reutilizada incluso después de que el objeto remoto asociado a una referencia a objeto remoto haya sido borrado, ya que los potenciales invocadores podrían tener referencias remotas obsoletas. Cualquier intento de invocar objetos borrados debería producir un error en lugar de permitir el acceso a un objeto diferente.

Existen varios modos de asegurarse que una referencia a un objeto remoto es única. Un modo es construir una referencia a objeto remoto concatenando la dirección Internet de su computador y el número de puerto del proceso que lo creó junto con el instante de tiempo de su creación y un número de objeto local como muestra la Tabla 2.2. El número de objeto local se incrementa cada vez que el proceso crea un objeto.

El número de puerto junto al tiempo constituyen un identificador único en ese computador. Con esta aproximación, las referencias a objetos remotos podrían representarse según un formato como el mostrado de la Figura anterior. En las implementaciones más simples de RMI, los objetos remotos viven en el proceso que los crea y sobreviven sólo mientras que el proceso continúa la ejecución. En estos casos, la referencia a objeto remoto puede utilizarse como una dirección del objeto remoto. En otras palabras, los mensajes de invocación se envían a la dirección internet de la referencia remota y dentro del computador al proceso identificado por el número de puerto.

Para permitir que los objetos remotos sean recolocados en distintos procesos en computadores diferentes, las referencias de objetos remotos no debieran utilizarse como dirección del objeto remoto. Más adelante se tratará sobre un formato de referencia de objetos remotos que permite que los objetos puedan ser activados en diferentes servidores a lo largo de su tiempo de vida.

El último campo de la referencia de objeto de objeto remoto mostrada en la figura contiene cierta información sobre la interfaz del objeto remoto, por ejemplo el nombre de la interfaz. Esta información es relevante para cualquier proceso que reciba una referencia de objeto remoto como argumento o resultado de una invocación remota ya que necesitará conocer los métodos que ofrece el objeto remoto.

## 2.3 COMUNICACIÓN CLIENTE SERVIDOR

Esta forma de comunicación está orientada a soportar los roles y el intercambio de mensajes de las interacciones típicas cliente-servidor. En el caso normal la

comunicación petición – respuesta es síncrona, ya que el proceso cliente se bloquea hasta que llega la respuesta del servidor. Esta comunicación también puede ser fiable ya que la respuesta del servidor es, en efecto, un acuse de recibo para el cliente. La comunicación cliente-servidor asíncrona es una alternativa que puede ser útil en situaciones donde los clientes pueden recuperar las respuestas más tarde.

En los siguientes párrafos se describen los intercambios cliente-servidor en términos de las operaciones envía y recibe del API Java para datagramas UDP, aunque muchas implementaciones actuales utilizan streams TCP como es el caso de CORBA.

Un protocolo construido sobre datagramas evita las sobrecargas asociadas con el protocolo de streams TCP. En concreto:

- Los reconocimientos son redundantes ya que las peticiones son seguidas por las respuestas.
- El establecimiento de la conexión involucra dos pares de mensajes extra además del par necesario para la petición de conexión y la consiguiente respuesta.
- El control del flujo es redundante en la mayoría de las invocaciones, ya que estas pasan solamente una pequeña cantidad de información sobre argumentos y resultados.

El protocolo petición – respuesta: Este protocolo está basado en un trío de primitivas de comunicación: Haz operación, dame Petición y envía respuesta.

Es de esperar que la mayoría de los sistemas RMI y RPC estén soportados por un protocolo similar. El que se describe aquí se construye a medida para dar soporte

RMI en la que se le pasa una referencia a un objeto remoto sobre el que hay que invocar el método indicado en el mensaje de petición.

Este protocolo de petición – respuesta diseñado especialmente hace corresponder a cada petición una respuesta. Podría estar diseñado para obtener ciertas garantías de entrega. Si se utiliza diagramas UDP, las garantías de entrega deben venir dadas por el protocolo petición – respuesta donde el mensaje de respuesta del servidor sirve como acuse de recibo del mensaje de petición del cliente.

La primitiva *Haz Operación* se utiliza en los clientes para invocar las operaciones remotas. Sus argumentos especifican el objeto remoto y el método a invocar, junto con la información adicional (argumentos) requerida por el método. Su resultado es una respuesta RMI. Se supone que el cliente que usa *Haz Operación* empaqueta los argumentos en una cadena de bytes y desempaqueta el resultado de la cadena de bytes que le es devuelta. El primer argumento de haz Operación es una instancia de la clase Remote Object ref. que representa las referencias de los objetos remotos, como se mostró en la tabla de referencia de un objeto remoto anteriormente. Esta clase proporciona los métodos para conseguir la dirección Internet y el puerto del servidor del objeto remoto. La primitiva *Haz Operación* envía un mensaje de petición al servidor cuya dirección Internet y puerto se especifican en la referencia de objeto remoto dada como argumento. Después de enviar el mensaje de petición *Haz Operación* invoca el método *recibe* para conseguir el mensaje respuesta, del que se extrae el resultado y lo devuelve a su invocador. El invocador de *Haz Operación* se bloquea hasta que el objeto remoto en el servidor ejecuta la operación solicitada y transmite el mensaje respuesta al proceso cliente.

La Primitiva *Dame Peticiones* se usa en el servidor para hacerse con las peticiones de servicio. Cuando el servidor ha invocado el método sobre el objeto especificado utiliza el método *envía respuesta* para mandar el mensaje de respuesta al cliente. Cuando el cliente recibe el mensaje de respuesta, desbloquea la operación *haz Operación* y continúa la ejecución el programa cliente.

La información transmitida en un mensaje de petición y de respuesta se muestra en la Tabla 2.3

| Tipo Mensaje | Int (0 = Petición, 1=Respuesta |
|---|---|
| IdPetición | Int |
| ReferenciaObjeto | Remote Objet Ref |
| IdMétodo | Int o Method |
| Argumentos | // cadena de bytes |

**Tabla 2.3 - Estructura de un mensaje Petición -respuesta**

El primer campo indica cuando el mensaje es una petición o una respuesta. El segundo campo *idPetición* contiene un identificador de mensajes. Una primitiva *Haz Operación* en el cliente genera una *IdPetición* por cada mensaje de petición y el servidor lo copia en el correspondiente mensaje de respuesta. Esto hace posible que *Haz Operación* pueda comprobar que un mensaje de respuesta es el resultado de la petición actual, no de una invocación anterior que se ha retrasado. El tercer campo es una referencia de objeto remoto empaquetada según la forma mostrada anteriormente en el apartado 2.2.3. El cuarto campo es un identificador del método a invocar, por

ejemplo los métodos en una interfaz pueden numerarse 1, 2. 3, ...; Si el cliente y el servidor utilizan un lenguaje común que soporte la reflexión, entonces se puede colocar en este campo una representación del método mismo (en Java se puede colocar en este campo una instancia de Method).

Identificadores de mensaje: Cualquier esquema que involucre la gestión de mensajes para proporcionar propiedades adicionales como la entrega fiable de mensaje o una comunicación petición-respuesta necesita que cada mensaje tenga un identificador único por el que pueda ser referenciado. Un identificador de mensaje tiene dos partes:

1. Un identificador de petición. *IdPetición* que proporciona el proceso emisor de una sucesión de enteros creciente.
2. Un identificador para el proceso emisor, por ejemplo, su puerto y su dirección internet.

La primera parte hace que el identificador sea único para el emisor, y la segunda lo hace único para el sistema distribuido (La segunda parte se puede obtener independientemente, por ejemplo si se usa UDP desde el mensaje recibido).

Cuando el valor del campo *IdPetición* alcanza el valor máximo para un entero sin signo por ejemplo ($2^{32}$-1) vuelve a cero. La única restricción impuesta es que el tiempo de vida de un identificador de mensaje deberá ser mucho menor que el tiempo que se necesita para dar la vuelta a esta secuencia de enteros.

# CAPITULO III

# OBJETOS DISTRIBUIDOS E INVOCACIÓN REMOTA

## 3.1 INTERFACES

La mayoría de los lenguajes de programación modernos proporcionan medios para organizar un programa en conjunto de módulos que puedan comunicarse unos con otros. La comunicación entre los módulos se puede realizar mediante llamadas a procedimientos entre los módulos o accediendo directamente a las variables de otro módulo. Para controlar las interacciones posibles entre los módulos se define explícitamente una interfaz para cada módulo. Los módulos se implementan de forma que se oculte toda la información excepto aquella que se haga disponible a través de su interfaz. De este modo, mientras la interfaz permanezca inalterada, la implementación podrá cambiar sin afectar a los usuarios del módulo.

Las interfaces en los sistemas distribuidores: En un programa distribuido, los módulos pueden lanzarse en procesos separados. No es posible para un módulo que se ejecuta en un proceso acceder a las variables de un módulo que está en otro proceso. Asimismo, la interfaz de un módulo escrita para RPC o RMI no puede especificar el acceso directo a variables. Hay que notar que las interfaces pueden especificar atributos, lo que parece violar su regla. Sin embargo, los atributos no son accedidos directamente sino mediante ciertos procedimientos de escritura y lectura que se añaden automáticamente a la interfaz.

Los mecanismos de paso de parámetros, por ejemplo la llamada por valor y la llamada por referencia, utilizados en las llamadas a procedimientos locales, no son

adecuados cuando el que llama y el procedimiento llamado están en procesos diferentes. La especificación de un método o procedimiento en la interfaz de un módulo en un programa distribuido describe los parámetros como entrada o salida o a veces, ambos. Los parámetros de entrada se pasan al módulo remoto mediante el envío de los valores de los argumentos en el mensaje de petición y posteriormente se proporcionan como argumentos a la operación que se ejecutará en el servidor. Los parámetros de salida se devuelven en el mensaje de respuesta y se sitúan como la respuesta de la llamada o reemplazando los valores de las correspondientes variables argumento en el entorno de la llamada. Cuando se proporciona un parámetro tanto como para entrada como para salida(mas adelante se vera que los parámetros son tres tipos in, out o inout siendo este ultimo al que me refiero), el valor debe transmitirse tanto en los menajes de la petición como en los de la respuesta.

Otra diferencia entre los módulos locales y remotos es que los punteros en un proceso dejan de ser válidos en el remoto. En consecuencia no pueden pasarse punteros como argumentos o como valores retornados como resultado de las llamadas a los módulos remotos.

Los próximos dos párrafos discuten las interfaces empleadas en el modelo cliente-servidor original para RPC y en el modelo de objetos distribuidos para RMI:

**Interfaces de Servicio**.- En el modelo cliente-servidor, cada servidor proporciona un conjunto de procedimientos disponibles para su empleo por los clientes. Por ejemplo un servidor de archivos proporcionará procedimientos para leer y escribir archivos. El término interfaz de servicio se emplea para referirse a la especificación de los

procedimientos que ofrece un servidor, y define los tipos de los argumentos de entrada y salida de cada uno de los procedimientos.

**Interfaces remotas.-** En el modelo de objetos distribuidos, una interfaz remota especifica los métodos de un objeto que están disponibles para su invocación por objetos de otros procesos y define los tipos de los argumentos de entrada y de salida de cada uno de ellos. Sin embargo, la gran diferencia es que los métodos en las interfaces remotas pueden pasar objetos como argumentos y como resultados de los métodos. Además también se puede pasar referencias a objetos remotos; lo cual no debe confundirse con los punteros, que se refiere a ubicaciones específicas de la memoria.

Ni las interfaces de servicio ni las interfaces remotas pueden especificar un acceso directo a variables. En este último, esta prohibido el acceso directo a las variables de las instancias de un objeto.

Lenguajes de definición de interfaces: Se puede integrar un mecanismo RMI con un lenguaje de programación concreto si incluye una rotación apropiada para definir interfaces, que permita relacionar los parámetros de entrada y de salida con el uso habitual de los parámetros en ese lenguaje. Java RMI es un ejemplo en el que se ha añadido un mecanismo RMI a un lenguaje de programación con orientación a objeto. Esta aproximación es útil cuando se puede escribir cada parte de una aplicación distribuida en el mismo lenguaje. También es conveniente dado que permite al programador emplear un solo lenguaje para las invocaciones locales y remotas.

Sin embargo muchos servicios útiles se encuentran escritos en C++ y otros lenguajes. Sería beneficioso que se pudieran acceder a ellos remotamente, desde muchos otros programas escritos en su gran variedad de lenguajes, incluyendo Java.

Los lenguajes de definición de interfaces (IDL) están diseñados para permitir que los objetos implementados en lenguajes diferentes se invoquen unos a otros. Un IDL proporciona una notación para definir interfaces en la cual cada uno de los parámetros de un método se podrá describir como entrada o de salida además de su propia especificación de tipo.

Otros ejemplos incluyen el lenguaje de definición de interfaz para el sistema RPC en el Entorno de computación Distribuida de OSF(DCE, Distributed Computing Environment:http://www.lions.odu.edu/docs/dce/app_gd_core_18.html) que emplea la sintaxis del lenguaje C y se denomina IDL y DCOM (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomarch.asp) IDL que se basa en DCE IDL (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/midl/midl/midl_language_reference.asp) y que se emplea en el Método de Objetos y Componentes Distribuidos (Distributed Component Object Model) de Microsoft.

## 3.2 COMUNICACIÓN ENTRE OBJETOS DISTRIBUIDOS

### 3.2.1 EL MODELO DE OBJETOS

Un programa orientado al objeto por ejemplo en Java o C++ consta de un conjunto de objetos que interaccionan entre ellos, cada uno de los cuales consiste en un conjunto de datos y un conjunto de métodos. Un objeto se comunica con otro objeto invocando sus métodos, generalmente pasándole argumentos y recibiendo resultados. Los objetos pueden encapsular sus datos y el código de sus métodos. Algunos lenguajes por ejemplo Java y C++ permiten que los programadores definan objetos en los que las variables de sus instancias estén accesibles de modo directo. Pero para su uso en un sistema distribuido de objetos, los datos de un objeto sólo deberían ser accesibles mediante sus métodos.

Importante en este punto hacer un esquema que consolide el papel de los elementos vistos anteriormente dentro de un modelo de objetos:

Referencias a Objetos: Se puede acceder a los objetos mediante referencias de objetos. Una variable que aparenta contener un objeto en realidad maneja una referencia a ese objeto. Para invocar un método en un objeto, se proporciona la referencia del objeto y el nombre del método, junto a cualquier argumento que sea necesario. El objeto cuyo método se invoca se denomina unas veces el objetivo (target) y otras el receptor (receiver). Las referencias a objetos son valores de primera clase, lo que significa que, por ejemplo, puede ser asignadas a variables pasadas como argumentos y devueltas como resultados de métodos.

Interfaces: Una interfaz proporciona una definición de las signaturas de un conjunto de métodos (es decir, los tipos de sus argumentos, valores devueltos y excepciones) sin especificar su implementación. Un objeto proporcionará una interfaz particular si su clase contiene código que implemente los métodos de esa interfaz. En java una clase puede implementar varias interfaces, y los métodos de una interfaz pueden ser implementados por cualquier clase. Una interfaz también define un tipo que pueda usarse para declarar el tipo de las variables o de los parámetros y valores devueltos de los métodos de otras interfaces. Las interfaces no tienes constructores.

Acciones: La acción es un programa orientado al objeto que inicia en un objeto que invoca un método de otro objeto. Una invocación puede incluir información adicional (argumentos) necesaria para llevar a cabo el método. El receptor ejecuta el método apropiado y entonces devuelve el control al objeto que lo invoca a veces aportando un resultado. Una invocación a un método puede tener dos efectos:

1. Puede cambiar el estado del receptor
2. Pueden tener lugar más invocaciones sobre métodos de otros objetos.

Dado que una invocación puede traer consigo más invocaciones de métodos en otros objetos una acción es una cadena de invocaciones de métodos relacionados, cada uno de los cuales retorna eventualmente. Claro sin tener en cuenta los casos en que ocurren excepciones que se explican a continuación.

Excepciones: Los programas pueden encontrarse con muchos tipos de errores y condiciones inesperadas de diversa gravedad. Durante la ejecución de un método, pueden descubrirse muchos problemas diferentes, por ejemplo valores inconscientes en las variables de un objeto, o fallo en los intentos de leer o escribir sobre archivos o sockets de red. Cuando los programadores necesitan insertar comprobaciones en su código para tratar con todos los casos inusuales o erróneos posibles se hace a costa de la claridad del código del caso normal (ósea se debe incluir código adicional que maneje la ocurrencia de estos casos). Las excepciones proporcionan una forma limpia de tratar con las condiciones de error sin complicar el código. Además cada cabecera de método lista explícitamente como excepciones las condiciones de error que pudiera encontrar, permitiendo a los usuarios del método tratar con ellas. Puede decirse que un bloque de código lanza (throw) una excepción cuando quiera que aparezcan condiciones o errores inesperados. Esto conlleva que el control pase a otro bloque de código que capture ( (catch) la excepción. El control no vuelve al lugar donde se lanzó la excepción.

Compactación automática de la memoria: Se hace necesario proporcionar mecanismos de liberación del espacio ocupado por aquellos objetos que ya no se necesitan. Un lenguaje, por ejemplo Java, que pueda detectar automáticamente cuando un objeto ya no es accesible puede recuperar el espacio y ponerlo a disposición de su asignación para otros objetos. Este proceso de denomina compactación automática de la memoria. Cuando un lenguaje (por ejemplo C++) no da soporte para la compactación automática de la memoria, el programador debe encargarse de la liberación del espacio asignado a los objetos. Ésta puede ser una gran fuente de errores.

### 3.2.2 EL MODELO DE OBJETOS DISTRIBUIDOS

El estado de un objeto consta de los valores de sus variables de instancia. En el paradigma de la programación basada en objetos visto en el apartado anterior, el estado de un programa se encentra fraccionado en partes separadas, cada una de las cuales está asociada con un objeto. Dado que los programas basados en objetos están fraccionados lógicamente, la distribución física de los objetos en diferentes procesos o computadores de un sistema distribuido en una extensión natural.

Los sistemas de objetos distribuidos pueden adoptar la arquitectura cliente-servidor. En este caso los objetos están gestionados por servidores, y sus clientes invocan sus métodos utilizando una invocación de métodos remota. En RMI, la petición del cliente de invocación de un método de un objeto se envía en un mensaje al servidor y el resultado se devuelve al cliente en otro mensaje. Para permitir las cadenas de invocaciones relacionadas, se permite que los objetos en los servidores se conviertan en clientes de objetos de otros servidores.

El disponer de objetos clientes y servidores en diferentes procesos promueve la encapsulación. Esto es, el estado de un objeto está accesible sólo para los métodos del objeto, lo que quiere decir que no es posible que los métodos no autorizados actúen sobre el estado. Por ejemplo, la posibilidad del RMI concurrente desde objetos en diferentes computadores implica que se pueda acceder concurrentemente a un objeto. De este modo, aparece la posibilidad de accesos conflictivos.

Sin embargo, el hecho que los datos sobre un objeto sólo sean accesibles por sus propios métodos permite a los objetos proporcionar métodos para protegerse contra accesos incorrectos (control de concurrencia). Por ejemplo se puede emplear

primitivas de sincronización como variables de condición para proteger el acceso a sus variables de instancia.

Otra ventaja de tratar el estado compartido de un programa distribuido como un conjunto de objetos es que se pueda acceder a un conjunto de objetos vía RMI o también copiarse en un caché local y ser accedidos directamente, dado que la implementación de la clase está disponible localmente.

El hecho de que los objetos sean accedidos únicamente mediante sus métodos nos da otra ventaja para los sistemas heterogéneos en los que se pueden usar diferentes formatos de datos en diferentes lugares; estos formatos pasarán inadvertidos para aquellos clientes que usan RMI para acceder a los métodos de los objetos debido al uso de la representación externa y el empaquetado y desempaquetado de datos como ya se explico.

Ahora pasaremos a discutir las extensiones al modelo de objetos para hacerlo aplicable a los objetos distribuidos. En el modelo de objetos distribuidos cada proceso contiene un conjunto de objetos, alguno de los cuales pueden recibir tanto invocaciones locales como remotas, las invocaciones de métodos entre objetos en diferentes procesos, tanto si están en el mismo computador o no, se conoce como invocaciones de métodos remotas. Las invocaciones de métodos entre objetos del mismo proceso son invocaciones de métodos locales.

A los diferentes objetos que puedan recibir invocaciones remotas los conocemos como objetos remotos. Todos los objetos pueden recibir invocaciones locales, a pesar de que sólo puedan recibirlas de otros objetos que posean referencias a ellos. Ahora

pasaré a tratar los conceptos fundamentales que son el corazón del modelo de objetos distribuidos:

Referencia a objetos remotos: Otros objetos pueden invocar los métodos de un objeto remoto si tiene acceso a su referencia de objeto remoto. La noción de referencia a objeto en el modelo de objeto se extiende para permitir que cualquier objeto que pueda recibir un RMI tenga una referencia a objeto remoto. Una referencia a objeto remoto es un identificador que puede usarse a lo largo de todo un sistema distribuido para referirse a un objeto remoto particular único. Las referencias a objetos remotos son análogas a las locales en cuanto a que:

1. El objeto remoto donde se recibe la invocación del método remoto se especifica mediante una referencia a objeto remoto.
2. Las referencias a objetos remotos pueden pasarse como argumentos y resultados de las invocaciones de métodos remotos.

Interfaces remotas: La clase de un objeto remoto implementa los métodos de su interfaz remota, por ejemplo las instancias públicas en Java. Los objetos en otros procesos pueden invocar solamente los métodos de objetos remotos a través de su interfaz remota.

El sistema CORBA proporciona un lenguaje de definición de interfaces (IDL) que permite definir interfaces remotas. Las clases de los objetos remotos y los programas de los clientes pueden implementarse en cualquier lenguaje C cono C++ o Java para lo cual se dispone de un compilador IDL. Los clientes CORBA no necesitan emplear el mismo lenguaje que el objeto remoto para invocar sus métodos

remotamente (esta es una de las principales bondades de CORBA: la transparencia al lenguaje de implementación de los objetos).

En Java RMI, las interfaces remotas se definen de la misma forma que cualquier interfaz en Java. Adquieren su capacidad de ser interfaces remotas al extender una interfaz denominada Remote.

Ambos CORBA IDL y Java soportan herencia múltiple para sus interfaces. Esto es, se permite que una interfaz extienda una o más interfaces.

Acciones en un sistema de objeto distribuido: Como en el caso del modelo de objetos, una acción se inicia mediante la invocación de un método perteneciente a un objeto, que pudiera resultar en consiguientes invocaciones sobre métodos de otros objetos. Pero en el caso distribuido, los objetos, involucrados en una cadena de invocaciones relacionadas pueden estar ubicados en procesos o en computadoras diferentes. Cuando una invocación cruza los límites de un proceso o un computador, se emplea una RMI, la referencia remota al objeto se hace disponible para hacer posible la RMI. Las referencias a un objeto remoto pueden obtenerse como resultado de una invocación a otro objeto remoto. Como por ejemplo el objeto A podría obtener una referencia remota al objeto F a partir de una invocación sobre un método del Objeto B.

Compactación automática de memoria en un sistema de objetos distribuidos: Si un lenguaje, por ejemplo Java, soporta compactación automática de memoria entonces cualquier sistema RMI asociado debiera permitir la compactación automática de memoria para objetos remotos.

La compactación automática de memoria, distribuida, se logra usualmente mediante la cooperación entre el compactador automático de memoria local y un modulo adicional que realiza una forma de compactación automática de memoria, distribuida, basada generalmente en una cuenta de referencias a objetos.

Excepciones: Cualquier invocación remota puede fallar por razones relativas a que el objeto invocado está en un proceso o computador diferente de la del objeto que lo invoca. Por ejemplo, el proceso que contiene el objeto remoto pudiera malograrse o estar demasiado ocupado para responder o pudiera perderse el mensaje resultante de la invocación. Es así, que una invocación a un método remoto debiera ser capaz de lanzar excepciones tales como ***timeouts*** (cuando se pasa el tiempo de espera máxima para una respuesta) debido a la distribución así como aquellos lanzados durante la ejecución del método invocado. Ejemplos de estos últimos son: un intento de lectura pasado el fin de un archivo, o un acceso a un archivo sin los permisos adecuados.

CORBA IDL proporciona una dotación para las excepciones específicas del nivel de aplicación, y el sistema subyacente genera excepciones estándar cuando ocurren errores debido a la distribución. Los programas Clientes CORBA deben ser capaces de gestionar las excepciones, por ejemplo, un programa cliente C++ empleará los mecanismos de excepciones de C++.

## 3.3 LLAMADA A UN PROCEDIMIENTO REMOTO

Una llamada a un procedimiento remoto es muy similar a una invocación a un método remoto, en la que un programa cliente llama a un procedimiento en otro

programa en ejecución en un proceso servidor. Los servidores pueden ser clientes de otros servidores para permitir cadenas de RPC. Un proceso servidor define en su interfaz en servicio los procedimientos disponibles para ser llamados remotamente. RPC se implementa usualmente sobre un protocolo petición – respuesta como el que se discutió en el apartado 2.3("Modelo Cliente Servidor"), que se encuentra simplificado por la omisión de la referencia, a objetos remotos en la parte de los mensajes de petición. Los contenidos de los mensajes de petición y respuesta son los mismos que se mostraron para RMI ("Estructura de un mensaje petición respuesta"), excepto que se omite el campo Referencia Objeto.

El cliente que accede a un servicio incluye un procedimiento de resguardo para cada procedimiento en la interfaz del servicio. El papel de un procedimiento de resguardo es similar al de un proxy. Se comporta como un procedimiento local del cliente, pero en lugar de ejecutar la llamada, empaqueta el edificador del procedimiento y los argumentos en un mensaje de petición, que se envía vía a su módulo de comunicación al servidor. Cuando llega el mensaje de respuesta, desempaqueta los resultados. El proceso servidor contiene un distribuidor junto a un procedimiento de resguardo de servidor y un procedimiento de servicio para cada procedimiento de la interfaz del servicio. El servidor selecciona uno de los procedimientos de resguardo según el identificador de procedimiento del mensaje de petición. Un procedimiento de resguardo de servidor es como un método de esqueleto en el que se desempaquetan los argumentos en el mensaje de petición, se llama al procedimiento de un servicio correspondiente y se empaquetan los datos con el resultado para el mensaje de respuesta. Los procedimientos de servicio implementan los procedimientos en la interfaz del servicio.

Los procedimientos de resguardo del cliente y del servidor y el distribuidor pueden generarse desde la definición de la interfaz del servicio por medio de un compilador de interfaces

## 3.4 EVENTOS Y NOTIFICACIONES

La idea bajo el uso de eventos es que un objeto pueda reaccionar a un cambio que ocurre en otro objeto. Las notificaciones y los eventos son esencialmente asíncronos y son determinados por sus receptores. En particular, en las aplicaciones interactivas, las acciones que el usuario realiza sobre los objetos, por ejemplo al manipular un botón con el ratón o al introducir texto en una caja de texto mediante el teclado, se ven como eventos que provocan cambios en los objetos que mantienen el estado de la aplicación. Los objetos responsables de mostrar el aspecto del estado actual son notificados cuando cambia el estado.

Los sistemas distribuidos basados en eventos extienden el modelo local en eventos al permitir que varios objetos en diferentes ubicaciones puedan ser notificados de los eventos que tienen lugar en un objeto. Emplean el paradigma publica-suscribe, en el que un objeto que genera eventos publica el tipo de eventos que ofrece para su observación por otros objetos. Los objetos que desean recibir notificaciones de un objeto que haya publicado sus eventos se suscriben a los tipos de eventos que sean de interés para ellos. Los objetos que representan los eventos se denominan notificaciones o anuncios. Las notificaciones se pueden almacenar, enviarse en los mensajes, consultarse y aplicarse en variedad de órdenes a diferentes cosas. Cuando un anunciante experimenta un evento los suscriptores que hayan expresado interés en

ese tipo de evento recibirán las notificaciones. La acción de suscribirse a un tipo particular de evento se denomina también registrar el interés por ese tipo de evento.

Los eventos y las notificaciones pueden emplearse en una gran variedad de aplicaciones diferentes, por ejemplo para comunicar que se añade una forma a un dibujo, una modificación a un documento, el hecho de que una persona haya encontrado o dejado una sala, o que una parte del equipamiento o un libro etiquetado electrónicamente se encuentra en un nuevo lugar. Los dos últimos ejemplos se hacen posibles con el uso de distintos activos o dispositivos empotrados.

Los sistemas distribuidos basados en eventos presentan dos características importantes:

**Heterogéneos.-** Cuando se emplean modificaciones como mecanismo de comunicación entre objetos distribuidos, es posible hacer funcionar conjuntamente aquellos componentes del sistema distribuido que no han sido diseñados con características de interoperabilidad. Todo lo que se requiere es que los objetos generadores de eventos publiquen los tipos de eventos que ofrecen y que otros objetos se suscriban a los eventos y proporcionen una interfaz para recibir las notificaciones. Por ejemplo se pueden emplear los sistemas basados en eventos para conectar componentes heterogéneos en Internet. Un sistema en que las aplicaciones son advertidas de las ubicaciones y actividades de sus usuarios, tales como utilizar los computadores, las impresoras o libros marcados electrónicamente. Los autores prevén su uso futuro en el contexto de una red doméstica con operaciones como *si los niños vuelven a casa, conecta la calefacción general.*

**Asíncronos.-** Las notificaciones se envían asincrónicamente desde los objetos generadores de eventos a todos los objetos que se hayan suscrito a ellos para prevenir que el que los anunciantes necesiten sincronizarse con los suscriptores: es preciso desacoplar los anunciantes de los suscriptores. Mushroom por ejemplo es un sistema distribuido basado en eventos diseñados para dar soporte al trabajo colaborativo, en el que la interfaz de usuario muestra objetos que representan a usuarios y objetos de información tales como documentos y libros de notas en el interior de espacios de trabajo llamados lugares de red. El estado de cada lugar se encuentra replicado en los computadores de los usuarios que concurren en ese lugar. Los eventos se emplean para describir cambios sobre los objetos y cambios en el foco de interés de un usuario. Por ejemplo, un evento podría especificar que un usuario concreto ha entrado o salido de un lugar, o ha realizado una acción particular sobre un objeto. Cada réplica de cualquier objeto para el que sean relevantes ciertos tipos de eventos se suscribe a ellos y recibe notificaciones cuando estos ocurran. Pero los suscriptores se encuentran desacoplados de los objetos que experimentan los eventos, dado que en momentos diferentes los usuarios activos son diferentes.

Una situación en la que pueden ser útiles los eventos es la que se muestra en el siguiente ejemplo sobre una sala de contratación.

Sistema simple de una sala de contratación: Considere un sistema simple de una sala de contratación cuya tarea es permitir que los tratantes hagan uso de los computadores para consultar la información más reciente sobre los precios del mercado de las mercancías con que comercian. El precio del mercado para una sola mercancía conocida se representa mediante un objeto con varias variables de

instancia. La información llega a la sala de contratación desde diferentes fuentes externas en forma de actualizaciones de algunos o todas las variables de instancia de los objetos que representan las mercancías y es recolectada por procesos que llamamos proveedores de información. Los tratantes están interesados solamente en aquellas mercancías en que están especializados. Un sistema de sala de contratación podrá modelarse mediante procesos con dos tareas diferentes.

- Un proceso proveedor de información recibe continuamente nueva información comercial desde una sola fuente externa y le aplica los objetos de la mercancía apropiada. Cada una de las actualizaciones a un objeto de mercancía se contempla como un evento. El objeto de mercancía que experimenta tales eventos notifica a todos los tratantes que se hayan suscrito a la mercancía correspondiente. Habrá un proceso proveedor de información separado para cada fuente externa.

- Un proceso tratante crea un objeto para representar cada mercadería que el usuario pueda visualizar. Este objeto local se suscribe al objeto que representa esa mercancía en el proveedor de información relevante. Entonces recibe toda la información enviada al interior mediante las notificaciones y la muestra al usuario.

**Tipos de eventos:** Una fuente de eventos puede generar eventos de uno o más tipos diferentes. Cada evento tiene atributos que especifican información sobre ese evento, tal como el nombre o el identificador del evento que lo generó, la operación y sus parámetros y el tiempo (o un número de secuencia). Los tipos y los atributos se

usan tanto para en la suscripción como en la notificación. Cuando nos suscribimos a un evento, se especifica el tipo del evento, a veces modificado con ciertos criterios sobre los valores de los atributos. Cuando quiera que aparezca un evento que concuerde con los atributos, se notificará a las partes interesadas. En el ejemplo de la sala de contratación hay un tipo de evento (la llegada de una actualización de una mercancía) y los atributos podrían especificar el nombre de la mercancía, su precio actual y la última elevación o caída de precio. Los tratantes podrán por ejemplo especificar si están interesados en todos los eventos relacionados con una mercancía con un nombre concreto.

# CAPITULO IV

# SISTEMAS DISTRIBUIDOS CON CORBA

## 4.1 MODELO DE LA ARQUITECTURA DE OBJETOS DISTRIBUIDOS

Todas las tecnologías adoptadas por el OMG incluido el estándar CORBA se deben ajustar a la arquitectura OMA.

OMA ("Object Management Architecture") pretende definir en un alto nivel de abstracción las reglas necesarias para la distribución de la computación orientada a objetos (OO) en entornos heterogéneos. OMA se compone de dos modelos, **el Modelo de Objetos y el Modelo de Referencia**, el primer modelo define como se deben describir los objetos distribuidos en un entorno heterogéneo y el segundo modelo caracteriza las interacciones entre dichos objetos.

En el modelo de Objetos de OMA, un Objeto es un ente encapsulado con una única identidad y por intermedio de una interfaz bien definida es posible acceder a sus servicios (cuando un cliente solicita los servicios de un objeto, la localización y la implementación de dicho objeto son transparentes al cliente y el cliente no necesita saberlo, simplemente solicita el servicio a través de la interfaz publica de dicho objeto).

Este modelo se restringe a definiciones abstractas las cuales no condicionan la sintaxis de las interfaces de los objetos, ni de sus implementaciones ni de los ORBs (negociadores de las peticiones entre objetos).El modelo nos provee las bases para CORBA pero es más relevante para los diseñadores e implementadotes de ORBs que para los desarrolladores de aplicaciones con objetos distribuidos.

Los componentes del Modelo de Referencia de OMA lo podemos apreciar en la figura 4.1, su núcleo es el ORB ("Object Request Broker"- el negociador de peticiones de objeto), además de dar transparencia en la localización y activación de objetos, se encarga de facilitar la comunicación entre clientes y dichos objetos.

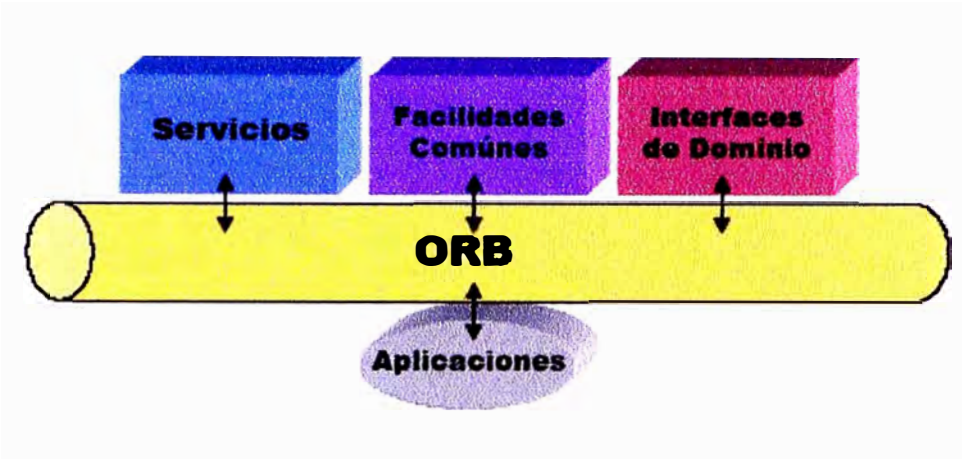



**Figura 4.1 - Modelo de Referencia de OMA**

Modelo de referencia del OMA donde el ORB es el centro que provee el soporte para el desarrollo de aplicaciones con objetos distribuidos

Al ser el ORB el centro de toda la arquitectura es quien se encarga de ser el bus de comunicación entre en los diferentes componentes, asimismo gestionar una serie de servicios los cuales son accedidos a través de sus respectivas interfaces.

Vamos ahora a describir el conjunto de interfaces que utilizan el ORB en el modelo de referencia en cuestión:

l

Servicios:

- Proporcionan funciones elementales necesarias para cualquier tipo de entorno distribuido, independientemente del entorno de aplicación.
- Permiten encontrar y administrar los objetos y la información, asimismo coordinar la ejecución de operaciones más complejas.
- Concede un valor añadido sobre otras tecnologías (por ejemplo RMI).
- Están pensados para grandes sistemas.

Facilidades Comunes:

- Proporcionan funciones, al igual que los servicios, válidas para varios dominios pero más complejas. Están orientadas a usuarios finales (no al desarrollo de aplicaciones).
- Un ejemplo de este tipo de funciones es el DDCF (Distributed Document Component Facility: http://www.mitre.org/tech/domis/omg/facilities.html#specs) formato de documentación basado en OpenDoc.
- También hay facilidades para intercambio de Datos, facilidades para Agentes móviles, facilidades para impresión, etc.
- También denominadas Facilidades Horizontales.

Interfaces de Dominio:

- Proporcionan funciones complejas, al igual que las Facilidades, pero restringidas a campos de aplicación muy concretos, es decir, orientadas al usuario final. Por ejemplo, telecomunicaciones, aplicaciones médicas o financieras, etc.

- Muchos grupos de interés (SIGs, grupos especiales de interés que dentro de la OMG se dedican al desarrollo de algunas tecnologías muy específicas, así como sus estándares y especificaciones) trabajan sobre estas especificaciones.
- También denominadas Facilidades Verticales.

Aplicaciones:

- El resto de funciones requeridas por una aplicación en concreto. Es el único grupo de objetos que OMG no define, pues está compuesto por los objetos propios de cada caso concreto y que pueden estar implementados de diversas maneras.
- Estos son los objetos que un sistema concreto tiene que desarrollar. El resto (servicios, facilidades) pueden venir dentro del entorno de desarrollo.

## 4.2 ¿QUÉ ES CORBA?

CORBA (Common Object Request Broker Arquitecture) es un estándar para sistemas de objetos distribuidos que ha sido desarrollado por el OMG ( Object Management Group) y que esta basado en la Arquitectura OMA. Es la propuesta concreta del OMG para el desarrollo de aplicaciones distribuidas.

CORBA da las especificaciones necesarias para los mecanismos a través de los cuales los objetos CORBA (de los que hablaré más adelante) hacen peticiones y reciben respuestas, de forma transparente para el cliente gracias al núcleo de su arquitectura que es el ORB. El ORB de CORBA es entonces, una aplicación que proporciona interoperabilidad entre diferentes objetos posiblemente programados en diferentes lenguajes, corriendo bajo sistemas operativos distintos y en general en diferentes máquinas. Este es el punto básico en el que se basa la arquitectura CORBA.

De lo anterior se deduce que CORBA tendrá todas las características de la arquitectura OMA como toda propuesta de la OMG, pero con especificaciones concretas como son el lenguaje de Interfaces IDL de CORBA, los condicionantes en la implementación de un ORB, el formato de las referencias a objetos, etc.

## 4.3 MODELO DE LA ARQUITECTURA CORBA

CORBA es un estándar computacional abierto de objetos distribuidos, que ha sido especificada por la OMG, con el ánimo de describir todas las características del ORB de OMA. En la figura 4.2 podemos apreciar cada uno de los componentes de CORBA, y a continuación se describirá cada uno de ellos.

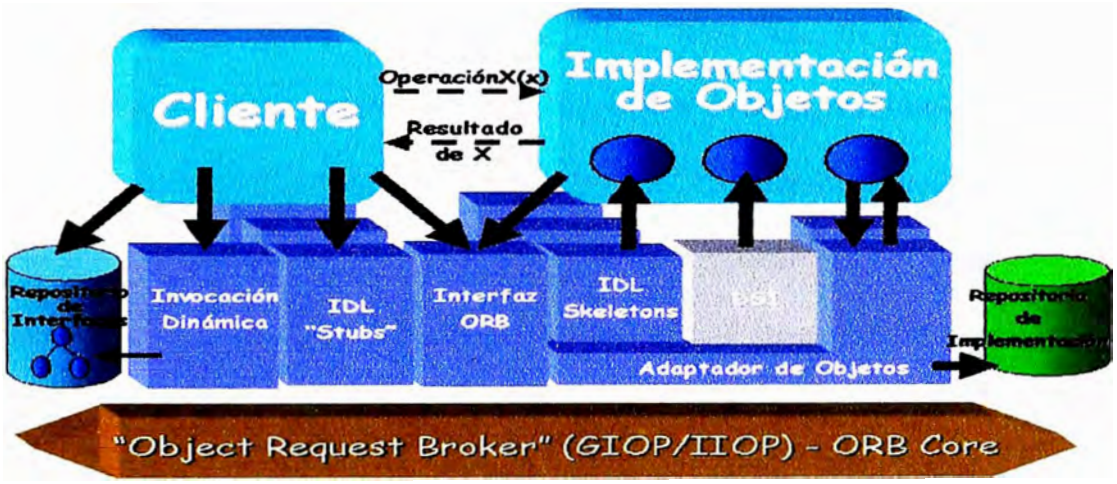



**Figura 4.2 - Arquitectura de CORBA**

### 4.3.1 EL NEGOCIADOR DE PETICIONES (ORB):

El ORB es el bus de mensajería que transmite las peticiones y los resultados entre los objetos CORBA donde quiera que estos este localizados y como quiera que este implementados.

La principal ventaja del ORB es la transparencia con la que realiza la comunicación cliente/objeto.

El objeto que un cliente desea y al que el ORB envía sus requerimientos, es llamado el "*target object*". Mucha de la transparencia a la que antes me refiero sé ve reflejada en que comúnmente el ORB se encarga de la localización de los objetos ya que el cliente no la conoce, ni la necesita (un objeto puede estar en un proceso corriendo en otra maquina dentro de la red o sobre la misma maquina, corriendo en diferente o el mismo proceso), el cliente tampoco conoce la implementación de los objetos con los que desea interactuar, ni el lenguaje de programación en que están escritos, ni el sistema operativo, ni el "hardware" sobre el cual están corriendo; el cliente tampoco se preocupa de la activación de los objetos requeridos, ya que el ORB es el encargado de activar los objetos si fuese necesario, además, el cliente no necesitará conocer los mecanismos de comunicación (TCP/IP, llamada de métodos locales, etc.) que sé utilizan, simplemente el ORB pasa los requerimientos de los clientes a los objetos servidores y envía una respuesta a quien hizo el requerimiento; por otra parte, la transparencia del ORB permite que los desarrolladores se preocupen más de sus aplicaciones y menos de los asuntos que tengan que ver con programación de sistemas distribuidos a bajo nivel.

Para que un cliente pueda solicitarle al ORB el envío de sus peticiones hacia un determinado objeto debe contar con una referencia hacia dicho objeto, la cual puede tenerla desde la implementación o conseguirla durante el tiempo de su ejecución de manera dinámica, estas dos formas se explicaran mas adelante.

### 4.3.2 LENGUAJE DE DEFINICIÓN DE INTERFACES DE OMG (IDL):

Antes de que un cliente pueda realizar peticiones sobre un objeto debe conocer los tipos de operaciones que este ofrece así como sus respectivas firmas(argumentos de

entradas, de salida o ambos, tipo devuelto, excepciones). La interfaz de un objeto especifica sus operaciones y tipos que soporta y por tanto, define las peticiones que pueden ser hechas a ese objeto. IDL es el lenguaje en el que se especifican las interfaces de los objetos CORBA. Es un lenguaje declarativo, así que fuerza a los interfaces a ser definidos separados de sus implementaciones. Esto permite que los objetos sean construidos utilizando distintos lenguajes de programación y que aún así puedan comunicarse ya que una misma interfaz puede tener varios tipos de implementaciones diferentes siendo esto transparente al cliente que hace la petición.

En la descripción de la interfaz con IDL, el programador deberá plasmar todos los elementos del objeto que serán accesibles a los clientes (como un fichero de cabecera en C++). La sintaxis del lenguaje IDL es similar a la de C++.

Lo que se hace necesario es un mapeo entre los tipos de datos que maneja un determinado lenguaje de programación y los tipos dentro de IDL, para eso esta el compilador IDL que se encarga de generar los mecanismos que hacen esto automáticamente, así tendremos compiladores IDL para cada lenguaje, como se muestra en la figura 4.3:

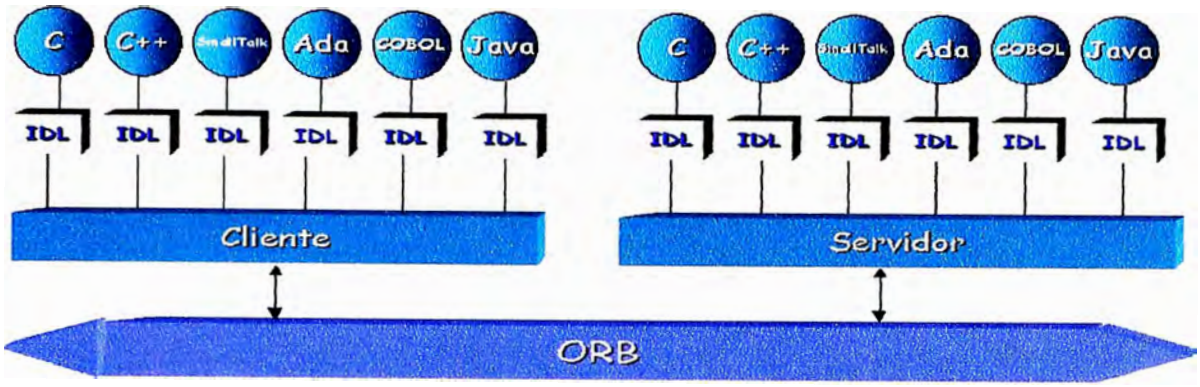



**Figura 4.3**

La interfaz IDL permite que diferentes lenguajes puedan comunicarse de manera transparente

4.3.3 CLIENTE

Es la entidad que invoca operaciones sobre un objeto dentro del sistema distribuido, la localización de dicho objeto es transparentes al llamante, bastaría simplemente con invocar un método sobre un objeto, de tal forma que un objeto remoto para una entidad cliente (local y llamante) se comporta como si fuese un objeto local.

### 4.3.4 REFERENCIAS A OBJETOS CORBA

Una "referencia de objeto" es opaca ya que oculta al cliente todas las operaciones de conexión y localización realizados por el ORB para transmitir las peticiones y devolver las respuestas, y sirve para identificar a un objeto. Las referencias de objetos pueden pasarse entre procesos. El ORB convierte las referencias a objetos a un formato que puede ser transmitido a través de la red, CORBA especifica un formato para las referencias a objetos remotos que es adecuada para su uso, tanto si el objeto es activable remotamente como si no. Las referencias que utilizan este formato se denominan referencias a objetos remotos interoperables IOR del cual hablare mas adelante.

### 4.3.5 STUB DEL CLIENTE

Cuando un cliente desea invocar una operación definida en la interfaz IDL de un objeto remoto como si este fuese local, este cliente debe enlazarse al stub Cliente de dicha interfaz, el cual se encarga de convertir los valores que pasa el cliente dentro de su invocación a los tipos de valor que maneja el lenguaje IDL, esto se conoce como el mapeado. El stub para el cliente es generado cuando se compila la interfaz IDL del objeto y se realiza de acuerdo al lenguaje de programación del cliente. Por ejemplo si el cliente esta en Java, el compilador IDL debe generar el stub para un cliente Java.

Asimismo el stub del cliente se encarga de convertir de manera inversa los valores contenidos en la respuesta a la invocación hacia los tipos de valor propios del lenguaje del cliente.

### 4.3.6 INTERFACE DE INVOCACIÓN DINÁMICA (DII)

Cuando un cliente recibe una referencia a un objeto remoto para el cual no conoce su interfaz en tiempo de ejecución y por lo tanto no tiene un stub, la invocación sobre hacia dicho objeto remoto se debe realizar a través de DII.

El cliente envía a través de una de las interfaces del ORB una petición de la interfaz de invocación dinámica de un objeto, la cual le provee de operaciones que le permiten conocer el nombre de los métodos a invocar del objeto remoto, los tipos de argumentos y respuestas. Esto le permite al cliente construir una invocación sin contar con un stub.

### 4.3.7 SKELETON DEL SERVIDOR

Una vez que la invocación ha alcanzado al servidor que aloja a uno o más objetos, debe haber un mecanismo que permita identificar el método correcto dentro del código de la implementación que corresponde al objeto invocado, asimismo que se encargue de hacer la conversión de tipos en los argumentos de la petición hacia los tipos del lenguaje en el que esta hecha la implementación, así como la representación

de los mismos en el hardware del servidor. Este mecanismo es el Skeleton que viene a ser el análogo al stub del Cliente pero en el lado del servidor.

Asimismo al igual que los stubs, los Skeletons se generan a través del compilador IDL y para el lenguaje y plataforma específica en el que esta implementado el servidor.

### 4.3.8 INTERFACES DE SKELETON DINÁMICOS:

El servidor puede atender invocaciones que han sido hechas de manera genérica como se explicó anteriormente en el apartado 4.3.6, interpretando la operación invocada y sus argumentos así como la semántica de manera dinámica, este mecanismo es lo que llamaremos Interfaz de Skeleton Dinámica (DSI "*Dinamic Interface Skeleton*"), el cual provee operaciones que permiten obtener información del mensaje de invocación.

### 4.3.9 ADAPTADORES DE OBJETO

Recordemos que un objeto servidor estará implementado en determinado lenguaje y bajo cierto soporte y que recibe todas las peticiones a través del ORB, pero esto no ocurre de manera directa. El adaptador de objeto (Object Adapter) es el componente de contacto entre el ORB y la implementación de los objetos y es quien acepta requerimientos en nombre de los objetos servidores, dicho adaptador se encarga en tiempo de ejecución de activar, instanciar, pasar requerimientos y generar referencias de dichos objetos. También, de colaborar con el ORB para que todos los

requerimientos que se hagan desde múltiples conexiones, sean recibidos sin ningún tipo de bloqueo. El adaptador de objetos tiene tres interfaces asociadas, una al DSI , una al IDL "*skeleton*" y otra a la implementación de objetos. Todo esto con él animo de aislar la implementación de los objetos del ORB tanto como sea posible.

En el lado del servidor, el adaptador de objetos es la parte mas importante del ORB y básicamente tiene las siguientes funciones:

- Genera referencias a objetos remotos (identificadores únicos) y las asigna a los objetos creados.
- Soporta las llamadas entrantes de los clientes y las envía a los respectivos Skeletons.
- Autentifica las peticiones entrantes, de manera que para cada objeto, será fácilmente identificable el cliente.
- Activa y desactiva objetos servidores y sus implementaciones de acuerdo a ciertas políticas.
- Los objetos están siempre presentes para el sistema CORBA, pero no activos, por lo que pueden ser llamados directamente. Antes de cualquier petición, el adaptador de objetos activa el elemento en cuestión.

La OMG estandarizo un adaptador de objetos llamado BOA ("*Basic Object Adaptator*"), dicho BOA y los servidores, tiene la posibilidad de soportar más de un adaptador de objetos. OMG actualmente ha lanzado una especificación que mejora algunos defectos de portabilidad del adaptador de objetos BOA y es llamada POA (*"Portable Object Adaptator*" ).

## 4.4 SERVICIOS CORBA

CORBA incluye la especificación para los servicios que pudieran necesitarse en los objetos distribuidos. En concreto, el servicio de Nombres es una parte esencial para cualquier ORB, asimismo hay que aclarar, que de acuerdo a los requerimientos de cada desarrollador de ORBs estos implementaran algunos de los servicios especificados por CORBA. A continuación paso a detallar los principales servicios dentro de un sistema distribuido con CORBA

### 4.4.1 SERVICIO DE NOMBRES

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

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

La misma referencia a objeto puede ser almacenada varias veces con diferentes nombres, pero cada nombre identifica únicamente a una referencia. Un contexto de nombres (naming context) es un objeto que almacena relaciones entre referencias y

nombres. Este servicio se puede estructurar como un sistema de ficheros jerárquico en el que los contextos de nombres se comportarían como directorios.

Un contexto inicial de nominación proporciona una raíz para un conjunto de enlaces. En la práctica, cada instancia de un ORB tiene un único contexto inicial de nominación, pero los servidores de nombres asociados con diferentes ORB pueden formar federaciones de tal manera que todo el manejo de nombres es distribuido y se puede trabajar de manera análoga a los dominios en el DNS.

Los nombres que emplea el Servicio de Nombres de CORBA son del tipo NameConponent y constan de 2 cadenas, una para el nombre y otra para la clase del objeto: este proporciona un solo atributo cuyo uso se previó para las aplicaciones y puede contener cualquier información descriptiva de utilidad pero realmente el servicio de nombres no lo usa.

Los clientes emplean el método resolve para buscar referencias a objetos por su nombre. El tipo del valor devuelto es Object que es un tipo general de todos los objetos CORBA. Por lo tanto el tipo del resultado deberá ser estrechado (significa que deberemos aplicar el método narrow() para convertir el Object general en una referencia de objeto específica para el objeto que se esta invocando), antes de poder ser utilizado para invocar un método en un objeto remoto de la aplicación.

E1 argumento de resolve es de tipo Name, que está definido como una secuencia de componentes de nombre. Esto viene a decir que el cliente debe construir una secuencia de componentes de nombre antes de realizar la llamada si es que la referencia del objeto esta dentro de una jerarquía de contextos.

Los servidores de los objetos remotos emplean la operación bind para registrar los nombres de sus objetos y unbind para eliminarlos. La operación bind enlaza cierto nombre con una referencia a un objeto remoto y se invoca en el contexto en el que se añade el enlace.

La operación bind-new-context se usa para crear un contexto nuevo y para enlazarlo con el nombre dado en el contexto sobre el que fue invocada. Otro método llamado bind_context enlaza un nombre de contexto, dado a un nombre dado en el contexto sobre el que fue invocado. El método unbind puede usarse para contextos y nombres.

La operación list está preparada para ojear la información disponible en un contexto en el Servicio de Nombres. Devuelve una lista de enlaces desde el NameContext objetivo. Cada enlace consta de un nombre y un tipo; un objeto o un contexto. Algunas veces, un nombre de contexto puede contener un número muy grande de enlaces, en cuyo caso sería deseable que pudiera proporcionarlos como resultado de una sola invocación. Por esta razón, el método list devuelve cierto número máximo de enlaces como resultado de la llamada, si es el caso de que faltan enlaces por enviar, se puede arreglar para que envíe los resultados en tandas. Esto es posible dado que retorna un iterador como resultado secundario. El cliente usa el iterador para recuperar el resto de los resultados, uno cada vez.

El tipo bindingList es una secuencia de enlaces, cada uno de las cuales contiene un nombre y su tipo, que es bien un contexto o bien una referencia a un objeto remoto. El tipo BindingIterator proporciona un método next_n para acceder al próximo conjunto de enlaces; su primer argumento especifica cuántos enlaces se desean y en el segundo recibimos una secuencia de enlaces. El cliente llama al método lista

dándole como primer argumento el número máximo de enlaces que se recibirán inmediatamente a través del segundo argumento. El tercer argumento es un iterador, que podrá ser usado para obtener el resto de los enlaces, si los hubiera.

El espacio de nombres de CORBA permite la federación de Servicio de Nombres, usando un esquema en el que cada servidor proporciona un subconjunto del grafo de nombres. Similar a como el DNS hace la delegación de la administración por dominios.

La implementación Java de Servicio de Nombres de CORBA es muy simple y se denomina transitoria porque almacena todos sus enlaces en memoria volátil.

### 4.4.2 SERVICIO DE EVENTOS

La especificación del Servicio de Eventos de CORBA define interfaces que permiten a los objetos de interés, denominados proveedores, comunicar notificaciones a sus subscriptores, llamados consumidores. Las notificaciones se comunican como argumentos o resultados de invocaciones síncronas ordinarias de métodos remotos CORBA.

Las notificaciones pueden propagarse impulsadas (pull) por el proveedor hacia el consumidor, en este caso, el consumidor implementa la interfaz PushConsumer que incluye un método Push que toma cualquier tipo de dato CORBA como argumento. Los consumidores registran sus referencias a objetos remotos con los proveedores. Los proveedores invocan el método push, pasando una notificación como argumento. Lo pueden hacer como extraídas (pull) por parte del consumidor desde el proveedor. Este implementa la interfaz PullSupplier, que incluyen método pull que recibe cualquier tipo de dato CORBA como valor de retorno. Los proveedores registran sus

referencias a objetos remotos frente a los consumidores. Los consumidores invocan el método pull y reciben como resultado una notificación.

La notificación en sí misma se transmite como un argumento o resultado cuyo tipo es any, que indica que los objetos que intercambian notificaciones deben haberse puesto de acuerdo sobre los contenidos de las notificaciones. Los programadores de aplicaciones sin embargo, podrán definir sus propias interfaces IDL con notificaciones de cualquier tipo deseado.

Los canales de eventos son objetos CORBA que pueden emplearse para permitir que múltiples proveedores se comuniquen con múltiples consumidores de modo asíncrono. Un canal de eventos actúa como búfer entre los proveedores y los consumidores. También puede multidifundir las notificaciones a los consumidores.

Los procesos proveedores en la aplicación ofrecen ellos mismos las suscripciones obteniendo los proxy de consumidor desde el canal de eventos y conectando los consumidores a ellos.

La presencia de proxy de proveedores y proxy de consumidores hace posible construir cadenas de canales de eventos en las que cada canal suministra notificaciones que serán consumidas por el siguiente canal.

Estos pueden programarse para llevar a cabo algunos de los papeles de los observadores. Sin embargo, las notificaciones no conllevan forma alguna de identificadores, y así los patrones de reconocimiento o de filtrado de notificación deberán basarse en la información de tipo puesta en las notificaciones por las aplicaciones.

### 4.4.3 SERVICIO DE PERSISTENCIA DE OBJETOS

Según el cual los objetos persistentes pueden almacenarse en una forma pasiva en un almacén de objetos persistentes, mientras no están siendo usados, y pueden activarse cuando se los necesite. A pesar de que un ORB activa objetos CORBA con referencias a objetos persistentes, obteniendo sus implementaciones desde el repositorio de implementación, no se hacen responsables de almacenar y restaurar el estado de los objetos CORBA. En su lugar, cada objeto CORBA es responsable de guardar y recuperar su propio estado, por ejemplo mediante archivos. El Servicio de Objetos Persistentes (POS) de CORBA está dispuesto para servir como almacén de objetos persistentes de objetos CORBA. El POS puede implementarse de variedad de formas.

La arquitectura de POS permite que se disponga de un conjunto de almacenes de datos; cada objeto persistente tiene un identificador persistente que incluye la identidad de su almacén de datos y un número de objeto dentro de este almacén. Tanto el cliente como el objeto CORBA pueden decidir cuándo almacenar su estado, enviando una petición al POS. Para permitir que el cliente controle su persistencia, un objeto CORBA hereda una interfaz que incluye opciones para almacenarlo, restaurarlo o eliminarlo.

La implementación de un objeto CORBA persistente debe elegir un protocolo de comunicación con un almacén de datos.

La especificación POS ofrece una selección de protocolos alternativos para esta tarea, proporcionando un abanico de funcionalidades.

### 4.4.4 SERVICIO DE TRANSACCIÓN

El Object Transaction Service permite que los objetos distribuidos CORBA participen en transacciones tanto sencillas como anidadas. El cliente especifica una transacción, como una secuencia de llamadas RMI, que comienzan por begin (comenzar) y terminan por commit (consumar) o rollback (abortar, retractar). El ORB adhiere un identificador de transacción a cada invocación remota y trata con las peticiones begin, commit, rollback (abort). Los clientes también pueden suspender y retomar las transacciones. El servicio de transacciones lleva a cabo un protocolo de consumación en dos fases.

### 4.4.5 SERVICIO DE CONTROL DE CONCURRENCIA

Emplea bloqueos para aplicar el control de concurrencia al acceso a objetos CORBA. Puede utilizarse desde el interior de las transacciones o aisladamente.

### 4.4.6 SERVICIO DE SEGURIDAD

Este servicio incluye:

Autenticación de principales (usuarios y servidores); generando credenciales para los principales (esto es certificados confirmando sus derechos).

Se puede aplicar control de acceso a los objetos CORBA cuando reciben invocaciones remotas.

- Auditoria de las invocaciones a métodos remotos por parte de los servidores.

- Medios para el no repudio. Cuando un objeto lleva a cabo una invocación remota en nombre de un principal, el servidor crea y amacena las credenciales que prueban que se hizo la invocación al servidor en representación del principal peticionario.

Para garantizar la aplicación correcta de la seguridad a las invocaciones de métodos remotos, el servicio de seguridad requiere cooperación en nombre del ORB. Para realizar una invocación segura a un método remoto, se envían las credenciales del cliente en el mensaje de petición.

Si las credenciales son válidas, se utilizan para decidir si el principal tiene derecho a acceder al objeto remoto utilizando el método del mensaje de petición. Esta decisión se hace consultando un objeto que contiene información sobre qué principales tienen derecho a acceder a cada método del objeto destino. Si el cliente tiene suficientes derechos, se lleva a cabo la invocación y se devuelve el resultado al cliente, junto con las credenciales del servidor, si fuera necesario. El objeto destino también podría almacenar los detalles de la invocación en un histórico o guardar las credenciales de no repudio.

CORBA permite especificar una variedad de políticas de seguridad según los requisitos. Una política de protección del mensaje declara si deben autenticarse el cliente o el servidor (o ambos).

Se proporciona a los usuarios un tipo especial de credencial, denominada privilegio, a la medida de sus funciones. Los objetos se agrupan en dominios. Cada dominio tiene una única política de control de acceso que especifica los derechos de acceso a los objetos del dominio por parte del conjunto de usuarios con privilegios concretos.

Para dar cabida a una variedad impredecible de métodos, cada método se clasifica en términos de uno de cuatro métodos genéricos: 1) Los métodos get sólo devuelven partes del estado de un objeto, 2) los métodos set alteran el estado del objeto, 3) los métodos use hacen que el objeto realice algún trabajo, 4) los métodos manage realizan funciones especiales no proyectadas para este uso general. Dado que los objetos CORBA tienen una variedad de interfaces diferentes, hay que especificar los derechos de acceso de cada nueva interfaz en términos de los métodos genéricos anteriores. Esto implica que los diseñadores de la aplicación estarán involucrados en la aplicación de control de acceso, la determinación de los atributos de privilegio apropiados (por ejemplo: grupos o funciones) y en ayudar al usuario en la obtención de los privilegios apropiados para su tarea.

## 4.5 LENGUAJE DE DEFINICIÓN DE INTERFACES

El IDL de CORBA es un elemento clave para la interoperabilidad. Separa la interfaz de la implementación permitiendo definir las interfaces de los objetos CORBA.

El IDL ofrece una forma de especificar las interfaces de los objetos implementación de una forma independiente de los lenguajes de programación elegidos para implementar los métodos de los objetos implementación. Las interfaces se especifican en el lado cliente como ficheros IDL. En el lado del servidor, los servants se tienen como especificaciones abstractas de los objetos implementación. Los servants pueden ser clases propias del lenguaje soportado por el ORB, como por ejemplo C++ o Java.

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

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

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

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

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

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

**Tipos de datos simples**

Su significado es similar al que tienen en lenguajes de programación C/C++ o Java. Llama la atención que no existe tipo int como ocurre en otros lenguajes de programación. Una variable de tipo any permite almacenar valores de cualquier tipo CORBA IDL, incluyendo tipos definidos por el usuario... Realmente hay pocas

ocasiones donde el tipo any sea necesario, aunque podría ser interesante cuando no se conoce en tiempo de compilación qué tipo de CORBA-IDL se va a necesitar.

| | | | |
|---|---|---|---|
| void | boolean | char | wchar |
| unsigned long | short | unsigned short | long |
| double | float | enum | any |
| **String** | **octet** | | |

**Tipos de datos simples en CORBA**

**Tipos de datos estructurados**

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

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

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

Las uniones IDL tienen una pequeña diferencia con las que existen en C, en IDL son discriminadas, es decir, tienen un campo que en cada momento indica qué elemento

de los que contiene está en uso. También soportan opcionalmente un caso por defecto (default:).

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

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

**Tipos de datos contenedores**

CORBA tiene en las matrices y en las secuencias los tipos contenedores elementales, estando representados por las palabras clave array y sequence respectivamente. Los arrays son similares a los de C/C++, pero deben ser declarados como tipos definidos por el usuario (typedef string estacion[4];).

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

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

**Excepciones**

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

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

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

**Métodos, atributos y módulos**

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

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

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

IDL no permite la sobrecarga de funciones. Todos los parámetros han de tener en su declaración un calificador que indique el sentido que tendrá la información que va a pasar a través de ellos. Las palabras reservadas son in (entrada), out (salida) e inout (entradasalida).

Es obligatorio poner el nombre de los parámetros en la declaración, cosa que no es necesaria en C++.

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

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

```
attribute float radio;
```

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

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

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

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

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

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

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

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

Es fundamental mantener el lenguaje IDL tan simple como sea posible. Esto significa que sólo contiene tipos que puedan trasladarse a los lenguajes de programación de uso más común.

Debido a la heterogeneidad de los sistemas de objetos distribuidos, la simplicidad de IDL es crítica para el éxito de CORBA como una tecnología de integración. Los compiladores de IDL trasladan las construcciones en IDL al lenguaje que se utilice para implementar las operaciones definidas en la interfaz IDL.

En la figura 4.4 se resume el proceso de creación y utilización de los ficheros IDL en una aplicación CORBA.

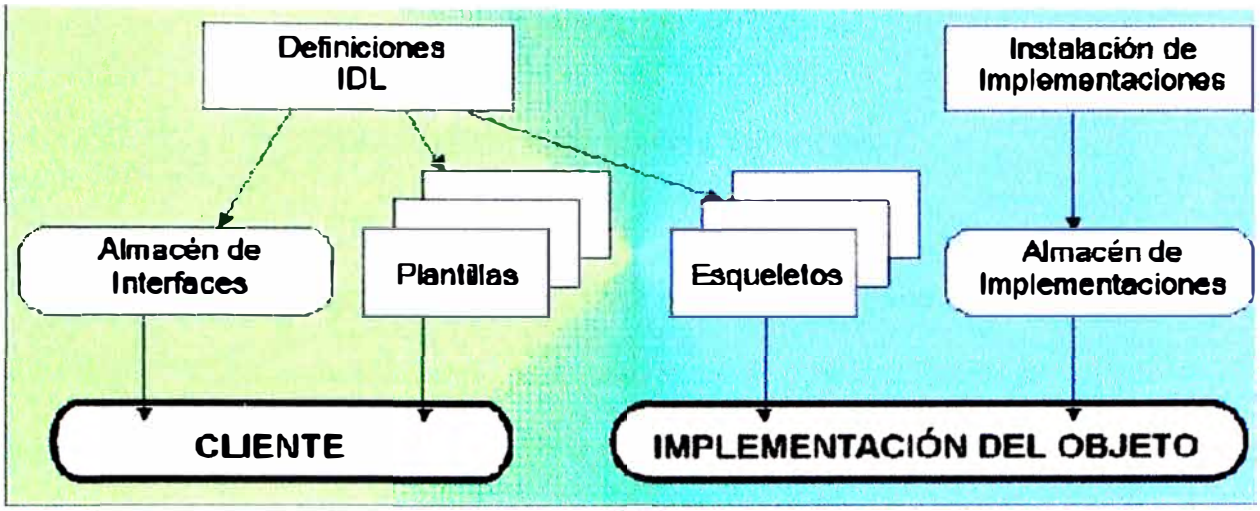



**Figura 4.4 a) Uso del IDL de CORBA**

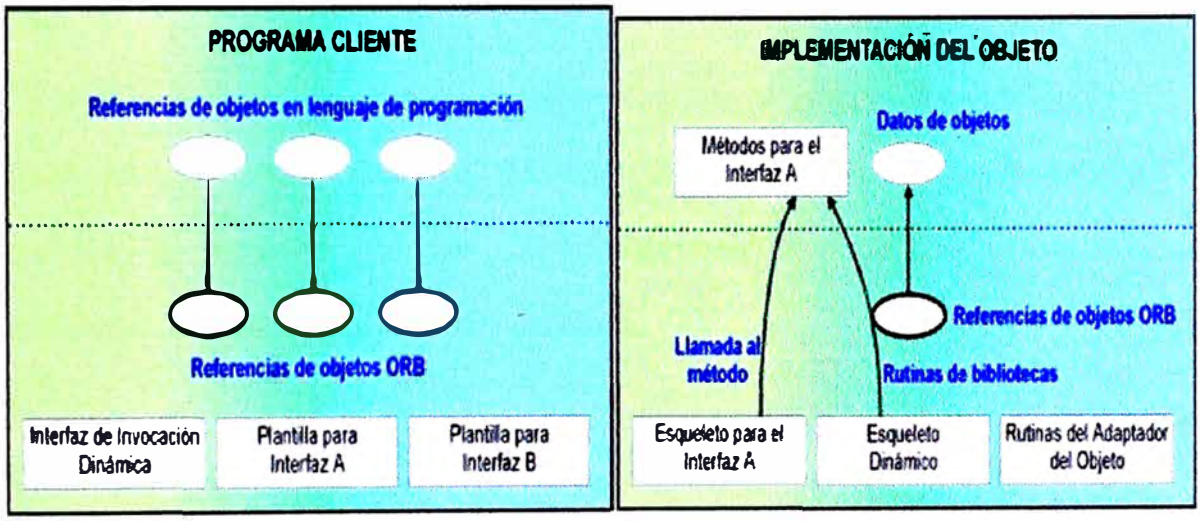



**Figura 4.4 b) Parte cliente y parte servidor de una aplicación**

# CAPITULO V

## INTEROPERABILIDAD CON CORBA

El estándar de interoperabilidad de CORBA fue desarrollado para permitir que diferentes ORBs(implementados para diferentes lenguajes, sistemas operativos, hardware, etc.) pudieran comunicarse. La interoperabilidad de CORBA ofrece una pasarela de infraestructura que hace que diferentes implementaciones de ORBs sean compatibles. Esta parte del estándar se asienta en la capa de transporte del modelo OSI.

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

**Condiciones de la capa de transporte para poder transmitir mensajes GIOP:**

- Debe ser orientado a conexión.
- Las conexiones son *full-duplex*.
- Las conexiones son simétricas.
- Comunica si se produce una pérdida de conexión.

El acercamiento a la interoperabilidad entre ORBs es universal, porque sus elementos se pueden combinar de muchas maneras para satisfacer una gama muy amplia de necesidades.

## 5.1 ELEMENTOS PARA LA INTEROPERABILIDAD

Los elementos de interoperabilidad son los siguientes:

- Arquitectura de Interoperabilidad entre ORBs.
- Soporte para puentes Inter-ORB
- Protocolo General Inter-ORBs (GIOP) y protocolo para Internet Inter-ORBs (IIOP).

Cada uno de estos elementos son los que paso a desarrollar a continuación.

## 5.2 ARQUITECTURA DE INTEROPERABILIDAD ENTRE ORBS

La arquitectura de la interoperabilidad de ORB proporciona un marco conceptual para definir los elementos de la interoperabilidad y para identificar sus puntos compatibilidad. También caracteriza nuevos mecanismos y especifica convenciones necesarias para alcanzar interoperabilidad entre ORBs desarrollados independientemente.

En el apartado anterior vimos que el protocolo general de interoperabilidad GIOP debe esta basado en un protocolo de transporte orientado a la conexión, si se tratará del protocolo TCP como es el más común de los casos entonces el protocolo se denomina IIOP (Internet Inter. ORB Protocol).

Específicamente, la arquitectura introduce los conceptos de puenteo mediato e inmediato entre dominios de ORB. El Inter-ORB protocolo de Internet (IIOP) forma la base común para la mayoría de puenteos mediatos. El puenteo inter-ORB puede

ser usado para implementar tanto puenteo inmediato como "medio-puente" para lo que es el puenteo mediato.

A través del uso de las técnicas del puenteo, los ORBs pueden interoperar sin conocer los detalles internos de su implementación propia, tales como que tipo de comunicación interproceso usa o que protocolo especifico usa internamente.

EL IIOP (Internet Inter. ORB protocol) puede ser usado para puentear dos o mas ORBs a través del "medio-puente". También puede trabajar con ORBs en modo "stand-alone" y con ORBs interconectados que usan su propio ESIOP (Environment-Specific Inter. ORB Protocol) que viene a ser un protocolo de interoperabilidad entre ORBs desarrollado de manera propietaria. El IIOP también puede usarse para implementar la comunicación interna de un ORB.

Ya que no es necesario de que los ORBs usen el IIOP internamente las implementaciones pueden desarrollar libremente sus propios protocolos internos y esto no importa a la hora de interoperar con otros ORBs, mientras respete las especificaciones para el puenteo mediato vía IIOP hacia el exterior (es decir, para la comunicación a nivel de ORBs).

## 5.3 SOPORTE PARA PUENTE INTER-ORB:

La arquitectura de interoperabilidad identifica claramente el papel de las diversas clases de dominios para la información específica en cada ORB. Estos Dominios pueden incluir dominios de referencias de objeto, dominios de tipo, dominios de

seguridad (por ejemplo el alcance de un identificador principal), un dominio de la transacción, etc.

Cuando dos ORBs están en el mismo dominio, pueden comunicarse directamente. En muchos casos, éste es el modo preferible. Sin embargo, esto no siempre es cierto, puesto que las organizaciones necesitan a menudo establecer dominios de control local.

Cuando la información dentro de una invocación debe dejar su dominio, la invocación debe atravesar un puente. El papel de un puente es asegurarse de que el contenido y la semántica sea mapeada desde la forma apropiada para un ORB origen hacia la forma apropiada del ORB destino, de tal manera que los usuarios de cualquiera de los dos ORBs solo vean el contenido y la semántica apropiada.

Los elementos que soportan el puente Inter-ORBs especifican ORB APIs (Application Programming Interfaces) y convenciones que permiten una construcción fácil de los puentes de interoperabilidad entre los dominios de ORB. Tales productos que prestan el servicio de puenteo pueden ser desarrollados por los vendedores de ORB, los integradores de sistema, u otros terceros.

Debido a que las extensiones requeridas para apoyar los puentes Inter-ORB son en gran parte generales en naturaleza, no afecte la otra operación del ORB, y puede ser utilizado para muchos otros propósitos además de la construcción de los puentes, ellos son apropiado para soportar a todos los ORBs. Otros usos incluyen eliminar errores, interposición de objetos, implementación de objetos con los intérpretes y los lenguajes de scripts, generación dinámica de implementaciones.

La técnica de punteo inter-ORB se puede también utilizar para proveer de interoperabilidad con otros sistemas que no pertenecen a CORBA, tales como modelo componente de objeto de Microsofts (COM). La facilidad de hacer esto dependerá del grado con el cual esos sistemas se compatibilizan con el modelo de objeto de CORBA.

## 5.4 GENERAL INTER-ORB PROTOCOL (GIOP) INTERNET INTER- ORB PROTOCOL (IIOP)

El GIOP especifica una sintaxis de transferencia estándar (representación de datos en bajo-nivel) y un conjunto de formatos de mensaje para las comunicaciones entre ORBs. El GIOP esta hecho específicamente para interacciones de ORB a ORB y se diseña para trabajar directamente sobre cualquier protocolo transporte orientado a la conexión que cumpla un número mínimo de requerimientos como los mencionados anteriormente en la introducción de este capítulo. No requiere ni depende del uso de mecanismos RPC de alto nivel. El protocolo es simple, escalable y relativamente fácil de implementar. Esta diseñado para permitir portabilidad, poco uso memoria de (footprints) y una razonable performance, con una mínima dependencia en el software de soporte pero si en la capa de transporte subyacente.

La especificación de este protocolo busca:

- Amplia disponibilidad
- Simplicidad
- Escalabilidad

- Bajo costo
- Generalidad
- Neutralidad arquitectural Consta de 3 especificaciones:
- Definición del CDR(Common Data Representation) de CORBA explicado en el apartado 2.2.1
- Los mensajes GIOP: Las características de la transferencia de mensajes es : La conexión es asimétrica, las solicitudes pueden ser multiplexadas. Los mensajes que se emplean son

**Tipos de mensajes**

| **Tipo de mensaje** | **Emisor** |
|---|---|
| Request | Cliente |
| Reply | Servidor |
| CancelRequest | Cliente |
| LocateRequest | Cliente |
| LocateReply | Servidor |
| Coseconnecion | Cliente o Servidor |
| MessageError | Cliente o Servidor |
| fragment | Cliente o Servidor |

**Request message:** Enviado por el cliente para hacer una solicitud **Reply:** Enviado por el servidor para responder a una solicitud

- Cancel request : Enviado por el cliente para cancelar una solicitud
- Locate request : Enviado para el cliente para ubicar un proveedor de servicios

- Locate reply: Enviado por un servidor para responder a la solicitud inmediatamente anterior.
- Close connection : Enviado por el servidor para terminar la conexión
- Message error: Puede ser enviado por cualquiera de los dos en caso de que ocurra algún error en la comunicación.

- Supuestos sobre el transporte : Se espera que el transporte sea proporcionado por un protocolo orientado a conexión, que de un despacho confiable, los participantes deben ser notificados de la perdida de la conexión , el modelo para iniciar la conexión debe ser como el empleado por TCP/IP.

Formato de un mensaje GIOP

| Cabecera (12 bytes) | Cuerpo (Variable) |
|---|---|

- La cabecera especifica
- Bytes 0..3: GIOP
- Bytes 4..5: versión del protocolo (ej.: 1.1)
- Byte 6: flags
- Bit menos significativo: big-endian o little-endian
- Segundo bit menos significativo: fragmento o { fin de fragmento / mensaje completo }
- Byte 7: tipo de mensaje
- Bytes 8-11: tamaño del mensaje (sin contar la cabecera)

CDR (COMMON DATA REPRESENTATION)

Recordemos como lo explicado en el apartado 2.2.1 que define el formato binario en el que se transmiten los tipos IDL. Soporta representación big-endian y little-endian.

El emisor especifica el tipo de orden de bytes.

El receptor cambia el orden de los bytes si es preciso.

Enfoque más eficiente que otras codificaciones (ej.: XDR) que transmiten siempre los datos en un determinado orden (ej.:big-endian)

Los datos nunca se etiquetan

Los stubs y skeletons generados son consistentes con las definiciones IDL

INTERNET INTER- ORB PROTOCOL (IIOP)

Este protocolo especifica cómo se intercambian los mensajes GIOP usando conexiones de TCP/IP. El IIOP especifica un protocolo estandardizado de interoperabilidad a través de Internet, proporcionando la capacidad de interoperar con otros ORBs compatibles y basado en el protocolo de la capa de transporte mas estandarizado como es tcp/ip.

Es el protocolo mas conveniente y apropiado a usar por cualquier ORB que necesite interoperar en dominios bajo el protocolo TCP/IP a menos que un protocolo alternativo sea necesario por un diseño especifico para el sistema o un ambiente de operación previsto para el ORB. En este sentido representa el protocolo básico Inter.-ORBs para ambientes TCP/IP que es el caso mas extendido.

La relación entre el IIOP y el GIOP es similar a la relación entre un lenguaje específico y el OMG IDL; el Protocolo GIOP se puede "mapear" con varios tipos de protocolos de transportes siempre que estén orientados a la conexión, y especifica los elementos del que son comunes a todos estos "mappings". De esta manera El GIOP por sí mismo no proporciona interoperabilidad completa, como una IDL no se puede utilizar para construir programas completos. El protocolo IIOP así como otros mappings similares a diversos protocolos de transporte, son realizaciones concretas de las definiciones abstractas de GIOP, según se muestra en la figura 5.1:

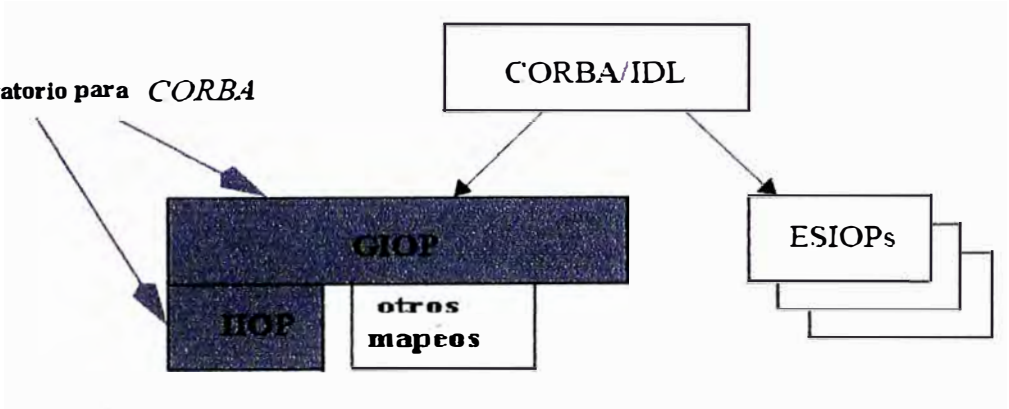



**Figura 5.1 – Jerarquía de los Protocolos de Interoperabilidad de CORBA**

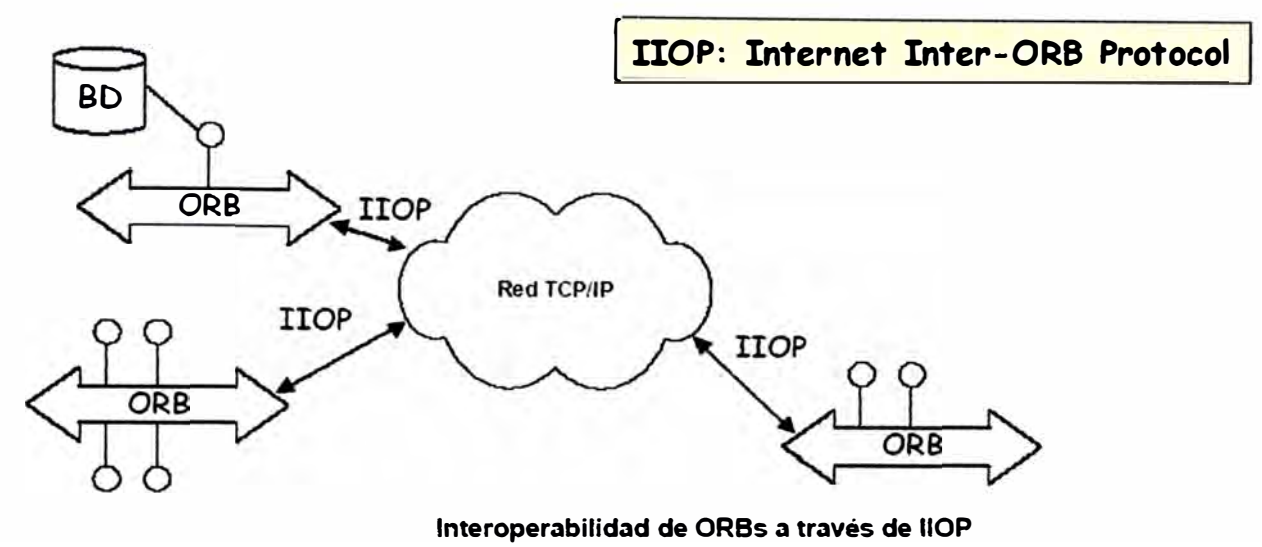



**Figura 5.2**

GIOP SOBRE TCP/IP

Como vimos anteriormente un IOR es una referencia de objetos para ser transmitida a lo largo del sistema distribuido, y muchas veces debe cruzar las fronteras del dominio de un determinado ORB, por lo tanto es el mecanismo de transporte y comunicación de mensajes entre los ORBs.

El IOR únicamente es usado en invocaciones inter-ORBs, por lo que es emitido y aceptado por los ORBs que están hablándole a la red, y usado por los puentes entre ORBs. La información que se encuentra en el IOR es :

- ¿De qué tipo es el objeto ? : Esto es necesario para preservar la integridad del objeto.
- ¿Qué protocolos podría usar el ORB que esta haciendo la invocación ?
- ¿Qué servicios del ORB están disponibles?

- ¿Es una referencia al objeto nulo? : Para evitar sobrecarga en el tráfico.

El IOR esta compuesto por una identificación de tipo, seguido por uno o varios perfiles etiquetados, cada protocolo que sea CORBA tiene un perfil y una etiqueta única que son asignados por la OMG. El perfil contiene toda la información que un ORB remoto necesita para realizar una invocación usando el protocolo.

Al recibir una invocación de un cliente el ORB determina si el objeto es local o remoto, si es remoto se comunica con el otro ORB, como siempre para el cliente la localización del objeto es transparente.

Para permitir la comunicación entre ORBs el estándar CORBA permite que todos los ORB hablen en el mismo protocolo, o que los ORBs hablen diferentes protocolos, en este caso se emplearan puentes para traducir entre los protocolos. Por ejemplo si el ORB A maneja un protocolo propietario y el ORB B maneja otro protocolo entonces pueden interoperar mapeando cada uno sus IORs hacia un protocolo intermediario como por ejemplo IIOP, es decir cada uno realiza un medio puente. También podría implementarse un puenteo directo pero no es muy común.

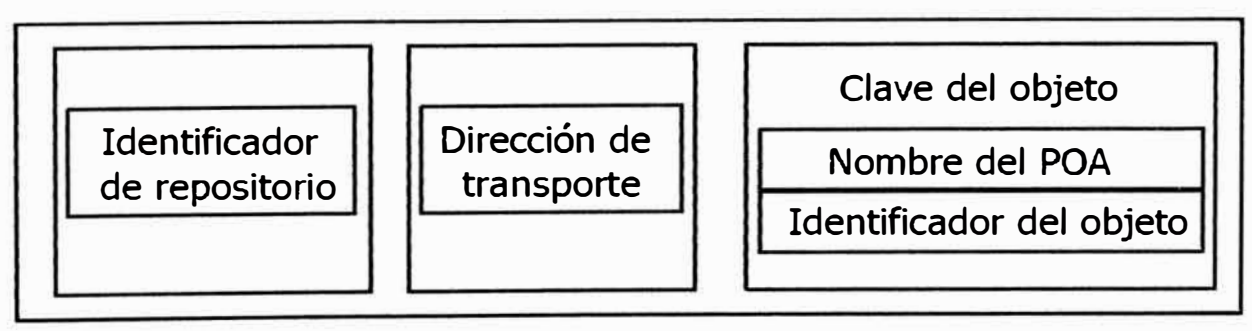



**Figura 5.3 : Estructura básica de una IOR**

- IIOP básicamente especifica el campo "dirección de transporte"
- Máquina (nombre o dirección IP) y puerto

| **Id. Repositorio** | **Datos para protocolo 1** | **Datos para protocolo 2** | **...** | **Datos para protocolo n** |
|---|---|---|---|---|

# CAPITULO VI

# APLICACIONES CON CORBA

En este capítulo se muestra la interoperabilidad que provee la arquitectura CORBA a dos componentes desarrollados en lenguajes de programación diferentes. Luego, en el apartado 6.2, se describe una aplicación que detalla el uso del Servicio de Nombres de CORBA, el cual permite un mejor manejo de las referencias a objetos remotos y facilita a los clientes la transparencia de ubicación de estos objetos.

## 6.1 INTEROPERABILIDAD ENTRE JAVA Y C++

A continuación se implementa una aplicación sencilla que muestra la interoperabilidad que CORBA es capaz de lograr entre dos componentes desarrollados en diferentes lenguajes de programación, gracias al uso del lenguaje IDL y al mapeo de tipos que realizan tanto el stub del lado del cliente como el skeleton del lado del servidor. En este caso se mostrará la interoperabilidad entre Java y C++.

En la aplicación se crea un objeto servidor al que llamaremos Hello el cual es capaz de responder a clientes remotos a través de su método sayHello, devolviendo una cadena con la frase "Saludos desde un lugar remoto!".

El objeto servidor puede ser desarrollado tanto en Java como en C++. De la misma manera, el Cliente, que invoca los métodos de este objeto, también puede ser desarrollado en dichos lenguajes. A manera de ejemplo, mostraremos la implementación del servidor y cliente en ambos lenguajes de programación.

Los pasos a seguir deben ser los siguientes:

**Escribir la interfaz IDL**

La interfaz IDL define el nexo de unión entre la parte cliente y la servidora de nuestra aplicación, especificando que atributos y operaciones son accesibles para el cliente.

Recordemos que las operaciones CORBA son el comportamiento que proporciona un servidor a un cliente que las invoca. Cada operación declarada en el interfase IDL se mapea a un método en la interfaz ya sea del Cliente o del Servidor según el lenguaje que se va a usar. En nuestro caso la interfaz la tendremos que compilarla tanto para Java como para C++ y en ambos casos el compilador (el cual viene dentro del ORB que estamos usando que es ORBacus jidl) genera todos los archivos necesarios tanto en el Cliente como en el Servidor.

**El archivo Hello.idl**

```
module HelloApp{                                    1
```

```
    interface Hello{                                    2
                void sayHello();                        3
        };
};
```

Normalmente las interfaces se declaran dentro de un modulo, ya que en un solo archivo IDL se pueden declarar varias interfaces, además por ejemplo en el caso de Java esto permite que los archivos necesarios para el mapeo (el stub y el skeleton) y otros archivos auxiliares se generen dentro de una carpeta (llamado paquete dentro de java). En nuestro caso, cuando compilemos nuestro archivo Hello.idl se generará una carpeta HelloApp como se define el módulo en la línea 1, dentro de esta carpeta o paquete Java se colocan los archivos que genera el compilador los cuales se explica luego.

En la línea 2 se define la interfaz, en este caso una sola y de nombre Hello. Asimismo, en la línea 3 se define un método sayHello el cual no devuelve ningún tipo (debido a que se le define de tipo void) ni tampoco requiere argumentos de entrada.

**Compilar el archivo Hello.idl**

Primero lo compilaremos para el lenguaje JAVA.

El comando idltojava lee y compila los archivos OMG IDL y crea todos los archivos Java necesarios para implementar ya sea un cliente que necesite invocar los métodos

del objeto Hello o para un servidor que quiera implementar los métodos definidos en la interfaz del objeto Hello, ambos hechos en Java.
En nuestro caso usaremos el siguiente comando:

***idltojava Hello.idl***

Este compilador viene como parte del ORB que estamos usando (ORBacus jidl). Al compilar la IDL se generarán una serie de ficheros, basados en las opciones pasadas por la línea de comandos. Los cinco ficheros generados son los siguientes:

**HelloPOA.java** : Esta clase abstracta es el skeleton del servidor que es el que mapea los datos que vienen en la petición del cliente que son tipos generales del lenguaje IDL al lenguaje JAVA y mapea los datos de la respuesta del JAVA al IDL. Proporciona la funcionalidad CORBA básica para el servidor. Además, esta clase se extiende de Servant (el Sirviente) e implementa la interfaz Operations. (Servidor).

**_HelloStub.java** : Esta clase es el stub del cliente, proporciona la funcionalidad CORBA para el cliente, es decir, mapea los datos del mensaje de invocación del JAVA al IDL y viceversa para los datos que viene en la respuesta del servidor . Implementa la interface Hello.java. (Cliente)

**Hello.java** : Esta interface contiene la versión Java de nuestra interface IDL. Contiene un único método sayHello. La interface Hello.java se deriva de org.omg.CORBA.Object. (Cliente/Servidor). Los lenguajes de programación

no pueden usar directamente la interfaz IDL, el compilador les genera una interfaz equivalente para dicho lenguaje.

**HelloHelper.java** : Esta clase final proporciona funcionalidad auxiliar, el método narrow necesario para hacer el casting entre la referencia de objeto CORBA y su tipo correcto. (Cliente/Servidor)

**HelloHolder.java** : Esta clase final proporciona operaciones para parámetros de tipo out e inout, que posee CORBA, pero que no son fáciles de mapear a Java. (Cliente/Servidor)

**HelloOperations.java** : Interface que permite utilizar la implementación por medio de TIE. (Servidor)

**LUEGO LO COMPILAMOS PARA C++**

El comando para realizar esto sería usando el mismo ORB:

**Idl Hello.idl**

Dicho compilador genera los ficheros necesarios para implementar ya sea un cliente que necesite invocar los métodos del objeto Hello o para un servidor que quiera implementar los métodos definidos en la interfaz del objeto Hello, ambos hechos en C++, basados en las opciones pasadas por la línea de comandos. Los ficheros generados son los siguientes:

**Hello.h y Hello.cpp:** Ficheros usados por el cliente, en realidad son los Stubs (Cliente).

**Hello_skel.h y Hello_skel.cpp:** Ficheros necesarios para implementar el objeto en el lado servidor, representan el Skeleton (Servidor).

Se puede apreciar que para el caso de C++ se generan menos archivos, debido a que el mapeo entre C++ y IDL es casi directo, por este motivo en C++ se puede usar la interfaz IDL directamente tanto para el stub como para el Skeleton.

Una vez que hemos compilado el mismo archivo IDL para ambos lenguajes ya estamos en la libertad de decidir si implementamos el Servidor en JAVA y el Cliente en C++ o viceversa y según esa decisión se usarán los archivos necesarios para cada lado de la comunicación Cliente-Servidor. A fin de que esta aplicación muestre todos los casos, se lista a continuación la implementación tanto del Cliente como del Servidor en los dos lenguajes.

## DESARROLLO DE LA APLICACIÓN EN JAVA

### CLIENTE JAVA

**Archivo : HelloClient.java**

```
//comenzamos importando todos los archivos generados por el compilador para Java
//y el cual mencionamos están en el paquete o carpeta HelloApp
import HelloApp.*;
```

```
//importamos los paquetes CORBA que vienen incorporados en la versión actual de
//JAVA, en el cual encontramos la clase que nos permite instanciar al orb y al
//adaptador de objetos necesario para poner a nuestro objeto servidor disponible para
el orb.
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

//Definimos nuestra clase  Cliente
public class HelloClient{
        public static void main (String args[]){

        //usando las librerías incorporadas en el paquete de Java 2 el
        //org.omg.CORBA instanciamos la interfaz del orb para acceder a sus
        //servicios que nos permita ubicar a nuestro objeto remoto Hello del cual
        //queremos hacer uso de sus métodos.

        ORB orb = org.omg.CORBA.ORB.init (args, null);

        //le entregamos nuestra referencia a objeto remoto al ORB que en este caso
        //la tenemos gravada en disco pero en forma de string, por lo cual no
        //necesitamos usar el servicio de nombres, aunque lo común es no tener la
        //referencia a objetoy solicitarla al servicio de nombres del cual hablare más
        //adelante.
```

//Con esta referencia entregada al orb en este caso el archivo Hello.ref, el orb //a través de su operación string_to_object desempaqueta la cadena y la //convierte en una referencia a objeto pero de tipo general o sea un Objeto CORBA.

**org.omg.CORBA.Object obj =orb.string_to_object("relfile:/Hello.ref");**

//aquí se prevé en caso de que hubiese algún error en la referencia //entregada al orb o que este no pueda encontrar dicha referencia, en //tal caso string_to_object nos devolvería un objeto nulo por lo tanto //habría que mostrar el mensaje de error respectivo.

**if(obj == null)**

**{**

**System.err.println("hello.Client: cannot read IOR from Hello.ref");**

**return 1;**

**}**

//Si no ha ocurrido ningún problema hasta ahora ya el cliente tiene la //referencia al objeto remoto Hello, pero esta referencia esta bajo el //tipo de Objeto general de CORBA y por lo tanto para poder //finalmente instanciar una referencia al Objeto Hello propiamente //debemos estrechar esta referencia obj al tipo Hello, aquí es donde //utilizamos la clase HelloHelper generada por el compilador y que //está dentro del paquete HelloApp; usamos su método narrow, el cual

//tomando la referencia a objeto obj nos devuelve una referencia a //objeto Hello propiamente (con todos sus metodos, atributos, estructuras, etc.)

**Hello helloRef = HelloHelper.narrow(obj);**

//Ahora helloRef representa para el Cliente una instancia del objeto //Hello y ya puede hacer uso de sus métodos como si se tratase de un //objeto local, en este caso esta invocando el único método de este //objeto que es sayHello sin ningún argumento

**helloRef.sayHello();**

//Para el Cliente a partir de ahora cada vez que haga uso de los //métodos del objeto Hello a través de su referencia helloRef no //necesita preocuparse de todo el proceso del mapeo de los datos que //entran y salen, ni de los mecanismos y protocolos de red que están //soportando su comunicación con el Servidor Remoto que es quien //realmente procesa y responde sus invocaciones desde algún lugar //dentro del Sistema Distribuido, soportado por algún sistema //operativo local, de red, implementado en algún lenguaje de //programación, etc. Esto es lo que permite CORBA a los Clientes, //transparencia en todos los aspectos ya explicados anteriormente.

```
        }
}
```

**EN EL LADO DEL SERVIDOR PARA JAVA**

En esta parte veremos como implementar el objeto Hello si elegimos como lenguaje Java y además describiremos como activarlo a través de un programa Servidor para que pueda recibir las invocaciones remotas a sus métodos, en este caso sayHello, para esto se requiere que este objeto tenga una referencia que lo identifique de manera única dentro de todo el Sistema Distribuido a través de la cual el orb sepa que invocaciones van dirigidas al objeto servidor Hello.

La Implementación en JAVA debe heredar el Adaptador de Objeto HelloPOA generado por el compilador.

**Archivo : Hello_impl.java**

```
//al igual que para el caso del Cliente importamos todos los archivos generados por el
//compilador y que están en el paquete HelloApp, en especial necesitamos el archivo
//HelloPOA (el adaptador de Objeto) quien contiene todos los métodos que permiten
//que nuestro objeto Hello pueda registrarse ante el orb, ser activado, desactivado,
//localizado,etc.
package HelloApp;
```

```
//Creamos la clase Hello_impl que implementa todos los métodos que hemos
//definido públicamente para nuestro objeto Hello, en este caso el único método es
//sayHello.
public class Hello_impl extends HelloPOA{

        //implementamos el método sayHello de nuestro objeto Hello y como
        //mencionamos al inicio lo que este método hace es mostrar en pantalla un
        //saludo desde el servidor hacia los clientes que lo invocan
        public void say_hello(){
                System.out.println("Saludos desde un lugar remoto!");
        }
}
```

Este archivo debe estar en HelloApp, es decir, al alcance del programa Servidor que cargará dicho paquete para poder trabajar.

Ahora nos toca implementar el programa Servidor que se encarga de instanciar al objeto Hello, registrar su referencia de objeto remoto al orb, activarlo y ponerlo a la espera de las invocaciones de los clientes, en este caso como no hacemos uso del servicio de nombres la referencia de objeto generada para nuestro objeto Hello es una cadena que se grava en un archivo llamado en este caso "Hello.ref" el cual se debe hacer llegar a los que implementan a los Clientes, ya sea vía e-mail, un medio de almacenamiento portátil, web service, etc.

**Archivo : Servidor.java**

```
package HelloApp;
public class Server {
        public static void main(String args[]) {
        //establecemos las propiedades iniciales del entorno necesarias para iniciar
        //el ORB
        java.util.Properties props = System.getProperties();
        props.put("org.omg.CORBA.ORBClass", "com.ooc.CORBA.ORB");
        props.put("org.omg.CORBA.ORBSingletonClass",
        "com.ooc.CORBA.ORBSingleton");

        int status = 0;
        //instanciamos la interfaz del ORB
        org.omg.CORBA.ORB orb = null;

        try {
        //inicializamos el ORB con las propiedades de Sistema configuradas
        //anteriormente, luego usando una de las operaciones del orb obtenemos una
        //referencia hacia el adaptador de objetos principal que luego nos servirá para
        //enlazar el objeto que implementa a la interfaz del objeto Hello Hello_impl
        //con la referencia de objeto remoto generada por el orb.
                orb = org.omg.CORBA.ORB.init(args, props);
```

```
org.omg.PortableServer.POA rootPOA =
org.omg.PortableServer.POAHelper.narrow(
orb.resolve_initial_references("RootPOA"));
org.omg.PortableServer.POAManager manager =
        rootPOA.the_POAManager();
```

```
//Aquí instanciamos la implementación del objeto Hello junto con
//su adaptador de objetos, esto es posible pues el archivo Hello_impl
//esta incluido en el paquete HelloApp.
Hello_impl helloImpl = new Hello_impl();
```

```
//Aquí se registra para el ORB el adaptador de objeto del objeto
//Hello enlazado a su implementación Hello_impl, como respuesta el
//orb nos devuelve la referencia de objeto remoto para Hello pero
//como un Objeto CORBA general.
Hello hello = helloImpl._this(orb);
```

```
try {
//la referencia a objeto la convertimos en cadena a fin de poder
//almacenarla en este caso en un archivo, ya que no estamos usando
//el servidor de nombres, se sobreentiende que esta referencia luego es
//la que se le debe dar al que implementa el cliente en forma de
//archivo como se vio anteriormente en la implementación del
```

```
        //Cliente Java para que este pueda usar dicha referencia e invocar al
        //objeto Hello.
        String ref = orb.object_to_string(hello);
        String refFile = "Hello.ref";
        java.io.PrintWriter out = new java.io.PrintWriter(
        new java.io.FileOutputStream(refFile));
        out.println(ref);
        out.close();

        }
    catch(java.io.IOException ex) {
        ex.printStackTrace();
        return 1;
    }

    //aquí se activa el adaptador de objetos para que esté listo a escuchar las
    //peticiones y también se ejecuta el orb.
    manager.activate();
    orb.run();
    return 0;
}
```

```
//se anticipan los casos en los que podrían fallar alguno de los pasos anteriores por
//cuestiones de red, hardware u otros motivos a través del lanzamiento de las
//respectivas excepciones de sistema, pero se podría implementar excepciones más
//específicas las cuales deben declararse en la interfaz IDL y luego implementarse
//dentro de Hello_impl.
catch(Exception ex)
    {
   ex.printStackTrace();
   status = 1;
    }

if(orb != null) {
        try {
                ((com.ooc.CORBA.ORB)orb).destroy();
        }
        catch(Exception ex) {
                ex.printStackTrace();
                status = 1;
        }
 }
System.exit(status);
}
```

## DESARROLLO DE LA APLICACIÓN EN C++

### CLIENTE C++

**Archivo : Cliente.cpp**

```
// Se cargan las librerías de CORBA
#include <OB/CORBA.h>
//Incluimos el archivo stub Hello.h
#include <Hello.h>

int
main(int argc, char* argv[], char*[])
{
   //instanciamos la interfaz del ORB
   CORBA::ORB_var orb;
   try {
      //se instancia e inicializa el ORB con las variables de entorno del sistema por
      //defecto
      orb = CORBA::ORB_init(argc, argv);
```

```
//cargamos la referencia al objeto remoto Hello desde el archivo guardado
//en disco Hello.ref
const char* refFile = "Hello.ref";
if stream in(refFile);
char s[2048];
in >> s;

//a través de una operación string_to_object del orb reconstruimos a partir del
//string  s la referencia al objeto remoto Hello pero esta operación nos
//devuelve una referencia al objeto remoto CORBA de tipo general  obj
CORBA::Object_var obj = orb -> string_to_object(s);

//usando uno de los archivos generados por el compilador podemos estrechar
//la referencia o objeto remoto obj de tipo general obtenida en el paso anterior
//al tipo específico de referencia remoto al objeto Hello a través de la
//operación _narrow() de su clase Hello
Hello_var hello = Hello::_narrow(obj);


// a partir de ahora el cliente ya tiene una referencia al objeto Hello a través de
//hello y ya puede invocar a todos los métodos de este; publicados en su
//interfaz,
hello -> sayHello();
} catch(const CORBA::Exception& ex)
```

```
        {
    cout << ex << endl;
    status = EXIT_FAILURE;
        }
return 0;
}
```

**EN EL LADO DEL SERVIDOR PARA C++**

Aquí vamos a definir el archivo de cabecera que nos permite declarar la clase que implementará al objeto Hello.

**Archivo :Hello_impl.h**

```
//incluimos el archivo Hello_skel.h que es el Skeleton generado por el compilador
//para el mapeo de los datos intercambiados entre las peticiones y respuestas.
#include <Hello_skel.h>

//Se declara la clase Hello_impl heredando de la  clase adaptador de objetos
//POA_Hello del archivo Hello_skel.h que nos permitirá enlazar la implementación
//del objeto Hello con el orb.
class Hello_impl : public POA_Hello,
public PortableServer::RefCountServantBase {
```

```
    //declara el orb y el adaptador de objetos para el objeto Hello
    CORBA::ORB_var orb_;
    PortableServer::POA_var poa_;

public:
        //declara el contstructor  y los métodos del objeto implementación Hello_impl
        Hello_impl(CORBA::ORB_ptr, PortableServer::POA_ptr);
        virtual PortableServer::POA_ptr _default_POA();
        virtual void sayHello() throw(CORBA::SystemException);
 };
```

A continuación implementamos los métodos declarados en la interfaz del objeto Hello.

**IMPLEMENTACION DEL SERVIDOR :**

**Archivo : Hello_impl.cpp**

```
#include <OB/CORBA.h>
#include <Hello_impl.h>

//aquí se implementa el constructor del objeto Hello_impl
Hello_impl::Hello_impl(CORBA::ORB_ptr orb, PortableServer::POA_ptr poa)
```

```
: orb_(CORBA::ORB::_duplicate(orb)),
poa_(PortableServer::POA::_duplicate(poa)){
}

//aquí se implementa el método defaultPOA() que permite que nuestra clase pueda
//establecer el adaptador de objetos que luego la enlace con el orb, hay que recordar
//que en C++ debemos hacernos cargo del manejo de memoria por lo que recurrimos
//al método _duplicate()
PortableServer::POA_ptr Hello_impl::_default_POA()
{
return PortableServer::POA::_duplicate(poa_);
}


//aquí se implementa el método sayHello() y se define la excepción que lanzan en
//caso de algún tipo de error
void Hello_impl::sayHello() throw(CORBA::SystemException)
{
cout << "Saludos desde un lugar remoto!" << endl;
}
```

Ahora se desarrolla el programa Servidor que se encarga de instanciar el objeto implementación a través de su adaptador de objetos y ponerlo al servicio del orb para así poder comenzar a recibir las peticiones desde los clientes remotos.

**Archivo : Server.cpp**

```
#include <OB/CORBA.h>
#include <Hello_impl.h>
int main(int argc, char* argv[], char*[])
{
    CORBA::ORB_var orb;
    try
    {
        //instancia la interfaz hacia el orb e inicializa los parámetros del
        //entorno del sistema en el que se ejecuta
        orb = CORBA::ORB_init(argc, argv);

        //instancia el manejador del adaptador de objeto y lo enlaza al orb
        PortableServer::POAManager_var manager =
        OBBiDir::IIOP::createPOAManager("RootPOAManager", orb,
        argc, argv, props);

        //a través de la operación resolve_initial_references del orb se obtiene
        //una referencia al Adaptador de Objetos raíz en la variable poaObj
        //pero  una vez más esta referencia es de tipo general, por lo que luego
        //con el método narrow del POA se obtiene la referencia exacta al
        //adaptador de objetos a través de la variable root
```

```
CORBA::Object_var poaObj = orb ->
resolve_initial_references("RootPOA");
PortableServer::POA_var  root =
PortableServer::POA::_narrow(poaObj);

//instancia el objeto implementación a través de su constructor y como
//se puede observar le entrega como argumentos tanto la referencia al
//orb como al adaptador de objetos que lo van a manejar
Hello_impl* helloImpl = new Hello_impl(orb, root);
PortableServer::ServantBase_var servant = helloImpl;

//genera la referencia de objeto remoto que entregara al ORB para que
//sepa donde ubicar la implementación del objeto Hello
Hello_var hello = helloImpl -> _this();


//convierte la referencia a objeto en un string y lo guarda en forma
//de archivo el cual luego se le debe de hacer llegar al Cliente para
//que pueda hacer la invocación sin usar el Servicio de Nombres
CORBA::String_var s = orb -> object_to_string(hello);
const char* refFile = "Hello.ref";
ofstream out(refFile);
if(out.fail())
        {
```

```
                cerr << argv[0] << ": can't open '" << refFile << "': "
                << strerror(errno) << endl;
                return EXIT_FAILURE;
                }
        out << s << endl;
        out.close();

        //activa el orb y el objeto esta listo para recibir las peticiones que
        //vienen del orb
        manager -> activate();
        orb -> run();
        return EXIT_SUCCESS;

}
catch(const CORBA::Exception& ex)
 {
        cerr << ex << endl;
        status = EXIT_FAILURE;
 }

if(!CORBA::is_nil(orb))
{
        try
        {
```

```
            orb -> destroy() ;
        }
        catch(const CORBA::Exception& ex)
        {
            cerr << ex << endl;
            status = EXIT_FAILURE;
              }
    }

    return status;
}
```

Se puede ver con que facilidad se pueden ampliar las interfaces IDL, y vemos que estos cambios no afectan para nada al servidor de CORBA. El cliente CORBA solo se ve afectado en el caso de que se elimine alguna operación de la interfaz que el utilice.

La facilidad para ampliar las interfaces y el hecho de que los desarrolladores de las interfaces de IDL no tengan porque saber nada de la arquitectura son dos características fundamentales de CORBA.

## 6.2 IMPLEMENTACIÓN DE APLICACIONES USANDO EL SERVICIO DE NOMBRES DE CORBA

En esta aplicación el objetivo principal es mostrar los beneficios del uso del Servicio de Nombres de CORBA, sin embargo se hará los comentarios necesarios durante cada etapa para no perder de vista el objetivo de la aplicación en su conjunto.

La aplicación la mostraremos implementada en JAVA para destacar el hecho de que la plataforma Java 2 incorpora como parte de su núcleo un ORB compatible con CORBA 2.0 que implementa el Servicio de Nombres.

Vamos a crear un servidor cuya finalidad será facilitar a los clientes la ejecución de consultas contra una base de datos remota, con la consiguiente obtención de resultados.

El Código en IDL CORBA sería:

**Consulta.Idl**

```
module SvrConsultas {
        interface IConsulta {
                string Consulta(in string Parametros);
        };
};
```

Compilamos el Archivo Consulta.Idl desde la línea de comandos :

**Idltojava –fno –cpp Consulta.Idl**

Esto generará una carpeta SvrConsultas (visto por Java como un paquete) con cinco archivos:

**LA INTERFAZ JAVA EQUIVALENTE :**

**Archivo : IConsulta .Java**

```
package SvrConsultas;
public interface IConsulta extends org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity {
        String Consulta(String Parametros);
}
```

La interfaz IDL CORBA no puede ser usada directamente desde un lenguaje de programación por lo cual se genera una interfaz equivalente para Java.

**EL SKELETON DEL SERVIDOR :**

**Archivo generado por el compilador :_IConsultaImplBase.Java**

```
package SvrConsultas;
public abstract class _IConsultaImplBase extends
org.omg.CORBA.DynamicImplementation
implements SvrConsultas.IConsulta {

        // Constructor
        public _IConsultaImplBase() {
        super();
        }

        private static final String _type_ids[] = {
        "IDL:SvrConsultas/IConsulta:1.0"
        };

        public String[] _ids() { return (String[]) _type_ids.clone(); }

        private static java.util.Dictionary _methods =
        new java.util.Hashtable();
        static {
        _methods.put("Consulta", new java.lang.Integer(0));
        }
```

```
//El método invoke recibe la petición como un objeto ServerRequest, se
//definen los métodos que están declarados en la interfaz IDL, en este caso
//solo uno que es Consulta el cual recibe un solo argumento tipo String
public void invoke(org.omg.CORBA.ServerRequest r) {
        switch (((java.lang.Integer) _methods.get(
        r.op_name())).intValue()) {
        case 0: // SvrConsultas.IConsulta.Consulta
        {
//Se hace el mapeado de IDL a Java de los tipos que viene en los argumentos de la
//petición
        org.omg.CORBA.NVList _list = _orb().create_list(0);
        org.omg.CORBA.Any _Parametros = _orb().create_any();


  Parametros.type(org.omg.CORBA.ORB.init().get_primitive_tc(
        org.omg.CORBA.TCKind.tk_string));

        _list.add_value("Parametros", _Parametros,
        org.omg.CORBA.ARG_IN.value);

        r.params(_list);
        String Parametros;
        Parametros = _Parametros.extract_string();
```

```
            //Se hace el mapeado de Java a IDL de el resultado que se enviará en
            //el mensaje de respuesta del servidor
            String ___result;
            ___result = this.Consulta(Parametros);
            org.omg.CORBA.Any __result = _orb().create_any();
            __result.insert_string(___result);
            r.result(__result);
            }

            break;
            default:
            throw new org.omg.CORBA.BAD_OPERATION(0,
            org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE);
            }
        }
}
```

Como se puede ver es el skeleton quien hace el mapeado de los tipos entre Java e IDL para el servidor.

**EL STUB DEL CLLIENTE :**

**Archivo generado por el compilador : IConsultaStub.Java**

```
package SvrConsultas;

public class _IConsultaStub extends org.omg.CORBA.portable.ObjectImpl
implements SvrConsultas.IConsulta {
        public _IConsultaStub(org.omg.CORBA.portable.Delegate d) {
        super();
        _set_delegate(d);
        }

private static final String _type_ids[] = {"IDL:SvrConsultas/IConsulta:1.0"
};
public String[] _ids() { return (String[]) _type_ids.clone(); }

//Declarados el método llamado Consulta
public String Consulta(String Parametros)
{

//Se construye el objeto Request que es el que se enviará en la petición al
//Servidor, asimismo se mapea los tipos de datos de los argumentos de Java a IDL
org.omg.CORBA.Request r = _request("Consulta");
r.set_return_type(org.omg.CORBA.ORB.init().get_primitive_tc(
org.omg.CORBA.TCKind.tk_string));
org.omg.CORBA.Any _Parametros = r.add_in_arg();
_Parametros.insert_string(Parametros);
```

```
r.invoke();

//Se mapea los tipos de valor de los datos que viene en la respuesta
String __result;
__result = r.return_value().extract_string();
return __result;
}
};
```

Como se puede ver es el stub quien hace el mapeado de los tipos entre Java e IDL para el Cliente.

Además de estos tres archivos se generan dos archivos auxiliares uno de los cuales es el archivo Helper que contiene el método narrow() cuya función es explicó en el ejemplo anterior.

**IMPLEMENTACION DEL SERVIDOR :**

Este módulo contiene el servidor, con el correspondiente método main(), y la clase que servirá a las peticiones de los clientes CORBA importamos el esqueleto y clases auxiliares

```
import SvrConsultas.*;

// Paquetes para acceder al ORB y al servicio de nombres
```

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

//Esta clase implementa la interfaz IConsulta derivando del esqueleto generado por
//idltojava ya que necesitamos sus métodos para poder enlazarse al orb y registrar su
//referencia de objeto remoto.
class ConsultaServant extends _IConsultaImplBase {

//implementamos el método del objeto declarado como Consultas
public String Consulta(String Parametros) {

// a modo de ejemplo el método solo se limitará a mostrar nuevamente los parámetros
//recibidos
System.out.println(Parametros);

// y devolverá una cadena como resultado
return "Ejecutada la consulta\n" + Parametros;
}
}

// Aquí en donde implementamos el programa servidor que habilita a la
//implementación de nuestro objeto a recibir las invocaciones a sus métodos.
public class ServidorConsulta {
        public static void main(String args[]) {
```

```
try {

        // Obtenemos una referencia al ORB, inicializándolo
        ORB Orb = ORB.init(args, null);

        // Creamos el objeto que implementa la interfaz IConsulta
        ConsultaServant Servidor = new ConsultaServant();

        // Conectamos el objeto CORBA con el ORB
        Orb.connect(Servidor);

        //Obtenemos la referencia al servicio de nombres a través del
        //método resolve_initial_references del orb que sirve para
        //cualquier servicio CORBA, luego esta referencia obtenida de
        //tipo general la estrechamos al tipo específico para el servicio
        //de nombres a través del método narrow de su clase Helper.
        NamingContext ServNombres =
                NamingContextHelper.narrow(
        Orb.resolve_initial_references("NameService"));


        // Aquí construimos el nombre de la interfaz y de su método
        //para  que el servicio de nombres pueda ubicarlos, recordar
```

```
                //que los nombres dentro del servicio de nombres se
                //implementan con la clase NameComponent
                NameComponent NombreInterfaz =
                new NameComponent("IConsulta", "");
                NameComponent CaminoObjeto[] = {NombreInterfaz};

                // y lo registramos en el espacio de nombres
                // facilitando el camino y la referencia al objeto
                ServNombres.rebind(CaminoObjeto, Servidor);
                System.out.println(
                "Servidor esperando solicitudes de consultas");

                //Esperamos por peticiones de clientes
                java.lang.Object RS = new java.lang.Object();
                synchronized(RS) {RS.wait();          }
            } catch (Exception X) { // en caso de fallo
                //mostrar toda la información de la excepción ocurrida
                System.err.println(X);
                X.printStackTrace(System.out); // disponible
        }
}
}
```

**IMPLEMENTACIÓN DEL CLIENTE:**

Este es el programa cliente, que obtendrá una referencia al objeto sirviéndose del servicio de nombres, tras lo cual ejecutará la consulta que se haya introducido como parámetro de entrada

```
//Importamos el stub y las clases auxiliares
import SvrConsultas.*;

// Paquetes para acceder al ORB y al servicio de nombres
import org.omg.CosNaming.*;
import org.omg.CORBA.*;
public class Cliente {

// Punto de entrada al cliente
public static void main(String args[]) {
try {
        // Inicializamos el ORB
        ORB Orb = ORB.init(args, null);
        // Obtenemos una referencia al servicio de nombres, una vez más como un
        //tipo objeto CORBA general pero con la clase Helper de su Interface y su
        //método narrow la convertimos en una referencia específica hacia el Servicio
        //de Nombres, notar que esta parte es la misma que para el Servidor
        NamingContext ServNombres = NamingContextHelper.narrow(
```

```
Orb.resolve_initial_references("NameService"));

// Preparamos el camino con el nombre del objeto de manera similar como en
//el caso del servidor usando la clase NameComponent
NameComponent NombreInterfaz =
new NameComponent("IConsulta", "");
NameComponent CaminoObjeto[] = {NombreInterfaz};

//Aquí con Servnombres.resolve(CaminoObjeto) hacemos que el servicio de
//nombre ServNombres, a través de su método resolve, nos devuelva la
//referencia al objeto remoto registrado con el nombre IConsulta contenido
//dentro de la  variable CaminoObjeto, luego dicha referencia con la clase
//Helper, generada por el compilador del IDL, la estrechasmo a una referencia
//específica al objeto remoto que queremos  IConsulta
IConsulta RefServ = IConsultaHelper.narrow(
ServNombres.resolve(CaminoObjeto));

//una vez que tenemos la referencia al objeto remoto IConsulta a través de la
//variable RefServ podemos hacer uso de ella como si se tratase de un objeto
//local, en este caso invocando su método Consulta con el argumento dado al
//iniciar el programa cliente
String Resultado = RefServ.Consulta(args[0]);
System.out.println(Resultado); // y mostrarla
} catch (Exception X) { // si hay algún fallo
```

```
            System.out.println(X); // mostrar toda la información
            X.printStackTrace(System.out); // disponible
    }
}
}
```

# CONCLUSIONES

1. Actualmente el desarrollo de sistemas distribuidos tiene entre sus principales áreas de aplicación aquellas donde la distribución es fundamentalmente un medio para conseguir un fin y donde el uso de soluciones distribuidas sirve para acercarse a las siguientes metas:

- **Computación masivamente paralela**, de propósito general y de alta velocidad.
- **Tolerancia a fallos** (confianza y disponibilidad).
- Respuesta a demandas con requisitos de **tiempo real**.

2. Existen aquellas áreas donde la distribución es un problema en sí mismo, es decir, donde son los propios requisitos de la aplicación los que fuerzan a evolucionar hacia soluciones distribuidas, debido a que una solución de proceso centralizado tendría serios problemas de desempeño o donde los costos de implementación se vuelven prohibitivos:

- **Bases de datos distribuidas**. Es necesario acceder a los datos desde lugares geográficamente dispersos, y además puede ser conveniente (e incluso imprescindible) almacenarlos también en varios lugares diferentes.

- **Fabricación automatizada**. Es necesaria la colaboración de muchos procesadores para coordinar las tareas en una fábrica.
- **Supervisión remota y control**. Los sensores, los actuadores y los nodos donde se toman las decisiones de control pueden estar en diferentes partes de un sistema distribuido.
- **Toma de decisiones coordinada**. Hay muchas aplicaciones donde es necesario que varios procesadores participen en la toma de decisiones, por ejemplo, porque cada uno de ellos tiene una parte de los datos relevantes.

3. Estamos entrando a una nueva era en la tecnología digital denominada Computación Ubicua, conocida también como Inteligencia Ambiental. Se va terminando el ciclo del Ordenador Personal, que siguió a la era ya superada, de los grandes ordenadores o "Mainframes".

Esta nueva era, la de la Computación Ubicua, estará caracterizada porque cada persona actuará sobre una multitud de diferentes dispositivos programables dotados de lo que se hace llamar "sensibilidad", (*smartness*), o la propiedad, de cambiar su comportamiento de acuerdo con las circunstancias ambientales que le rodean.

4. Para que la Computación Ubicua haga posible la interacción remota entre objetos distribuidos, se necesita que estos pequeños y baratos procesadores estén conectados a sensores y actuadores colocados en objetos del entorno del usuario y que se encuentren formando una red que soporte al que entra en juego mediante **procesos distribuidos**, donde cada procesador, aparte de modificar el comportamiento de un objeto, interactúa e intercambia información con los demás objetos para conseguir un objetivo a través del trabajo coordinado de todos ellos.

5. La mayor dificultad que plantea la Computación Ubicua está en establecer los estándares que permitan, que equipos de todo tipo procedentes de multitud de fabricantes (heterogéneos en hardware, lenguaje de programación, protocolo de red, etc.) lleguen a interoperar entre si armónicamente, pese a que ni siquiera se puede inicialmente definir con exactitud que dispositivos van a entrar a formar parte de la red de este sistema distribuido, pues en la mayoría de los casos la red se formará con el paso del tiempo por agregación de dispositivos con capacidad de Computación Ubicua, sin un orden preestablecido, adquiridos en muchos casos como substitutos de equipos preexistentes sin esa capacidad, rotos y obsoletos.

6. El desarrollo de Aplicaciones bajo entornos distribuidos será cada vez mayor y sus áreas de aplicación cada vez más cercanas a la vida cotidiana si acaso ya no lo están, en consecuencia, surgen nuevas exigencias y requerimientos que considerar en el desarrollo de sistemas distribuidos, uno de los mayores desafíos es la capacidad de integración total de componentes heterogéneos a través de la interoperabilidad directa.

7. Existen varias alternativas para el desarrollo de Sistemas Distribuidos como es MICROSOFT con su tecnología DCOM, SUN con JAVA RMI pero que están hechas para trabajar sólo bajo una plataforma, limitando mucho la interoperabilidad con componentes desarrollados en otras tecnologías, ya sean: lenguajes de programación, protocolos de red, hardware, sistema operativo, etc.

8. La OMG a través de su arquitectura CORBA presenta como una de las mejores, sino la mejor, alternativa para poder cubrir con todos los requerimientos que el diseño y la implementación de sistemas de objetos distribuidos exige actualmente, al estar basado en una arquitectura basada en objetos, como es OMA, permite

aprovechar implementaciones anteriores a través de la construcción de componentes y su integración en base a interfaces.

9. CORBA consigue de manera muy amplia vencer el gran desafío en la construcción de sistemas distribuidos como es la heterogeneidad de sus componentes, gracias al uso de un lenguaje de definición de interfaces como es CORBA IDL independiente de cualquier lenguaje de programación, el cual permite la interoperabilidad entre componentes implementados en lenguajes de programación diferentes.

10. El protocolo GIOP que es el medio de comunicación entre los ORBs permite conseguir la transparencia en cuanto a sistemas operativos, hardware, protocolo de red y demás mecanismos de bajo nivel que participan en la comunicación remota de los componentes teniendo como condición estar soportado por un protocolo de transporte orientado a la conexión. El caso concreto de GIOP sobre TCP/IP es el denominado Protocolo IIOP que permite la comunicación entre ORBs sobre la plataforma de Internet y que apunta a ser el más usado.

11. Además de su framework base, la arquitectura CORBA ofrece la implementación de una serie de servicios (servicios CORBA) que facilitan aun más la administración de los sistemas distribuidos basados en objetos tales como el servicio de nombres (*Namig Service*) que de manera muy análoga al servicio DNS de Internet permite la localización de recursos u objetos servidores a través de nombres sencillos, así como otros servicios que permiten el manejo de invocaciones dinámicas entre objetos, seguridad de acceso, manejo de acceso concurrente, etc.

12. Una de las desventajas de CORBA es el retraso existente entre las especificaciones publicadas por el OMG y su incorporación a los ORBs desarrollados, que representan el núcleo para el desarrollo de estas aplicaciones. Esto es especialmente grave en el caso de los nuevos servicios CORBA que van surgiendo y donde se depende completamente de que el ORB elegido tenga implementado dichos servicios.

13. CORBA es dentro de los estándares abiertos, una de las mejores opciones para el desarrollo de sistemas distribuidos basados en objetos que cumple con todos los requerimientos de las aplicaciones distribuidas ya conocidas y de las que están apareciendo. Asimismo, permite con facilidad ser extendido a través de nuevos servicios.

## BIBLIOGRAFÍA


1. George Coulouris - Jean Dollimore – Tim Kindberg, "SISTEMAS DISTRIBUIDOS CONCEPTOS Y DISEÑO", Addison Wesley – España, 2001
2. William Stallings, "COMUNCIACIONES Y REDES DE COMPUTADORES", Prentice Hall – España, 2001
3. Andrew S. Tanenbaum, "REDES DE COMPUTADORAS", Pearson Education – México, 2003
4. José Manuel Huidoro Montoya, "REDES Y SERVICIOS DE TELECOMUNICACIONES", Editorial Paraninfo – España, 2000
5. Reaz Hoque, "CORBA 3 - Developing Industrial-Strength Cliente/Server and Web Applications", IDG Books Worldwide – USA, 2000
6. Andreas Vogel – Bhaskar Vasudevan – Maira Benjamin – Ted Villalba , "C++ PROGRAMMING WITH CORBA",Wiley Computer Publishing – Canada, 1999
7. Common Object Request Broker Architecture: Core Specification

http://www.omg.org/technology/documents/corba_spec_catalog.htm

8. Alberto Caramazana, "Estándar CORBA - C++/Java - (ORBacus)"

Universidad Pontificia de Salamanca – Madrid, 2001