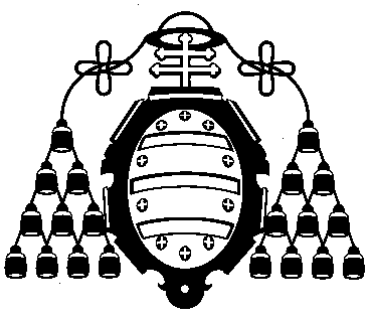
UNIVERSIDAD DE OVIEDO

Departamento de Informática



CURSO DE DOCTORADO

Objetos Distribuidos y Agentes Móviles
(Parte I)

Fernando Álvarez García

Oviedo, Mayo-Junio de 2001

ÍNDICE

CAPÍTULO 1 CONCEPTOS DE SISTEMAS DISTRIBUIDOS 1

1.1 INTRODUCCIÓN 1
1.2 ¿QUÉ ES UN SISTEMA DISTRIBUIDO? 1
1.3 CARACTERÍSTICAS CLAVE DE LOS SISTEMAS DISTRIBUIDOS 2
1.3.1 Compartición de recursos 2
1.3.2 Apertura (del inglés, openness) 3
1.3.3 Flexibilidad 4
1.3.4 Concurrencia 5
1.3.5 Rendimiento 7
1.3.6 Escalabilidad 7
1.3.7 Fiabilidad y tolerancia a fallos 8
1.3.8 Disponibilidad 10
1.3.9 Transparencia 11
1.3.10 Heterogeneidad 14
1.3.11 Seguridad 14
1.4 VENTAJAS E INCONVENIENTES DE LOS SISTEMAS DISTRIBUIDOS 15
1.4.1 Ventajas de los sistemas distribuidos frente a los centralizados 15
1.4.2 Desventajas de los sistemas distribuidos 16
1.5 DEFINICIÓN Y FUNCIONES DE UN SISTEMA OPERATIVO DISTRIBUIDO 16
1.6 ARQUITECTURA DE LAS APLICACIONES DISTRIBUIDAS 18

CAPÍTULO 2 SISTEMAS DE OBJETOS DISTRIBUIDOS 19

2.1 INTRODUCCIÓN 19
2.2 ARQUITECTURA DE SISTEMAS DISTRIBUIDOS 19
2.2.1 Modelo de procesos o cliente/servidor 19
2.2.2 Modelo basado en objetos 20
2.3 MODELO DE OBJETOS 21
2.3.1 Características del modelo de objetos de Booch 22
2.3.2 Características adicionales necesarias 25
2.4 APLICACIÓN DE LAS TECNOLOGÍAS DE OBJETOS A LA CONSTRUCCIÓN DE SISTEMAS DISTRIBUIDOS 26
2.4.1 Diseño del sistema distribuido como un marco orientado a objetos 27
2.4.2 Diseño del sistema distribuido como soporte de objetos 27
2.5 CUESTIONES DE DISEÑO DEL SOPORTE DE OBJETOS EN EL SISTEMA OPERATIVO DISTRIBUIDO 28
2.5.1 Actividad interna 28
2.5.2 Granularidad de los objetos 30
2.5.3 Visibilidad: objetos privados/objetos compartidos 31
2.5.4 Objetos volátiles/objetos persistentes 31
2.5.5 Movilidad: objetos fijos/objetos móviles 32
2.5.6 Replicación: objetos de una copia/ objetos replicados 32
2.5.7 ¿Qué es, entonces, un objeto distribuido? 33
2.6 PROBLEMAS DE LA GESTIÓN DE OBJETOS 33
2.6.1 Nombrado 33
2.6.2 Localización 34
2.6.3 Acceso 34
2.6.4 Compartición y protección 34
2.6.5 Persistencia 34
2.7 CARACTERÍSTICAS DE UN SISTEMA OPERATIVO DISTRIBUIDO ORIENTADO A OBJETOS 35

CAPÍTULO 3 PANORÁMICA DE SISTEMAS DISTRIBUIDOS ORIENTADOS A OBJETOS .. 37

3.1 INTRODUCCIÓN 37
3.2 DCE Y DC++ 37
3.2.1 Modelo de programación 37
3.2.2 RPC 37
3.2.3 Hilos 38
3.2.4 Formato de los datos 38

3.2.5 DC++ *38*
3.2.6 Crítica *39*
3.2.7 Características interesantes *39*
3.3 DCOM 40
3.3.1 Modelo de objetos *40*
3.3.2 Interoperabilidad entre objetos COM *40*
3.3.3 Modelo de programación *41*
3.3.4 Ciclo de vida *42*
3.3.5 COM+ *42*
3.3.6 Crítica *43*
3.3.7 Características interesantes *43*
3.4 GUIDE 43
3.4.1 Modelo de objetos *44*
3.4.2 Modelo de ejecución *44*
3.4.3 Invocación de objetos *44*
3.4.4 Migración de objetos *45*
3.4.5 Crítica *45*
3.4.6 Características interesantes *45*
3.5 CLOUDS 46
3.5.1 Abstracciones de Clouds *46*
3.5.2 Distribución en Clouds *46*
3.5.3 Espacio de objetos *47*
3.5.4 Crítica *47*
3.5.5 Características interesantes *47*
3.6 SPRING 47
3.6.1 Abstracciones *48*
3.6.2 Invocación de objetos *48*
3.6.3 Nombrado *49*
3.6.4 Seguridad *49*
3.6.5 Crítica *49*
3.6.6 Características interesantes *50*
3.7 SOS 50
3.7.1 Modelo de objetos *50*
3.7.2 Distribución de objetos *50*
3.7.3 Migración de objetos *51*
3.7.4 Modelo genérico *51*
3.7.5 Diseño del sistema operativo *51*
3.7.6 Crítica *52*
3.7.7 Características interesantes *52*
3.8 AMOEBA 52
3.8.1 Objetos de Amoeba *53*
3.8.2 Capacidades *54*
3.8.3 Nombrado de objetos *54*
3.8.4 Crítica *54*
3.8.5 Características interesantes *55*
3.9 EMERALD 55
3.9.1 Objetos de Emerald *55*
3.9.2 Actividad interna de los objetos *55*
3.9.3 Comunicación y localización *55*
3.9.4 Migración de objetos *56*
3.9.5 Agrupación de objetos *56*
3.9.6 Crítica *56*
3.9.7 Características interesantes *57*
3.10 COOLV2 57
3.10.1 Abstracciones base *57*
3.10.2 Soporte genérico en tiempo de ejecución *57*
3.10.3 Soportes específicos para lenguajes *58*
3.10.4 Modelo de objetos *58*
3.10.5 Invocación de objetos *58*
3.10.6 Distribución *58*

3.10.7 Crítica *59*
3.10.8 Características interesantes *59*
3.11 APERTOS 59
3.11.1 Estructuración mediante objetos y meta-objetos *59*
3.11.2 Flexibilidad *60*
3.11.3 Reflectividad *60*
3.11.4 Jerarquía de reflectores *60*
3.11.5 Invocación de objetos *61*
3.11.6 Migración de objetos *61*
3.11.7 Crítica *61*
3.11.8 Características interesantes *61*
3.12 RESUMEN DE CARACTERÍSTICAS DE LOS SISTEMAS REVISADOS 62

CAPÍTULO 4 CORBA 63

4.1 INTRODUCCIÓN 63
4.1.1 Objetos CORBA *63*
4.1.2 El ORB *63*
4.1.3 Definición de objetos CORBA *64*
4.1.4 El repositorio de interfaces *65*
4.1.5 La Interfaz de Invocación Dinámica *65*
4.1.6 Adaptadores de objetos *65*
4.1.7 Referencias *65*
4.1.8 Paso de parámetros por valor *66*
4.1.9 Servicios de objetos *66*
4.2 CONSTRUCCIÓN DE APLICACIONES CON CORBA Y JAVA 67
4.2.1 Ejemplo *67*

CAPÍTULO 5 RMI 75

5.1 INTRODUCCIÓN 75
5.2 OBJETOS RMI 75
5.3 MODELO DE PROGRAMACIÓN 75
5.4 SERVICIOS 76
5.5 CONSTRUCCIÓN DE APLICACIONES CON RMI 76
5.5.1 Ejemplo *77*

CAPÍTULO 1 CONCEPTOS DE SISTEMAS DISTRIBUIDOS

1.1 Introducción

La utilidad de un sistema informático, el alcance y la calidad de los servicios que proporciona y su facilidad de uso dependen fuertemente de su sistema operativo [Gos91]. El **sistema operativo** puede ser definido como aquella parte del sistema informático que da vida al hardware. El desarrollo de los sistemas operativos va siempre detrás del desarrollo del hardware, pero permiten mejorar el rendimiento de este, y, en el peor de los casos, ocultarán todas sus particularidades.

Hasta hace bien poco tiempo los sistemas operativos se construían para sistemas informáticos centralizados. Existe una gran cantidad de literatura sobre la teoría del diseño y construcción de los mismos [SG98, Sta98, TW97].

En la actualidad nos encontramos un paso más allá en el desarrollo de sistemas informáticos. Se está investigando, por un lado, para incrementar la capacidad de procesamiento de computadores aislados, utilizando arquitecturas multiprocesador y, por otro lado, para unir la capacidad de procesamiento de computadores independientes dispersos geográficamente. Nos encontramos, en este último caso, con el **desarrollo de los sistemas distribuidos**.

A la hora de construir un sistema distribuido, hay que pararse a estudiar, entre otras cosas, su alcance, es decir, si el sistema a construir se va a limitar a aprovechar la capacidad de procesamiento para resolver un problema concreto o si, por el contrario, se desea construir una plataforma de distribución que facilite posteriormente la construcción de aplicaciones que aprovechen, de una manera más sencilla, dicha capacidad de procesamiento. En el primer caso se habla de la construcción de **aplicaciones distribuidas**. En el segundo caso se habla de **sistemas operativos distribuidos**.

Tal como ocurría en los sistemas operativos centralizados, el objetivo principal de un sistema operativo distribuido será **proporcionar un conjunto de servicios** (los convencionales, de sistemas operativos tradicionales y otros nuevos, propios de los sistemas operativos distribuidos) a los programas de usuario de una manera sencilla, con las ventajas y los inconvenientes que lleva consigo el trabajar sobre una arquitectura distribuida. Las características, las ventajas y los inconvenientes de los sistemas distribuidos vienen descritos en los apartados siguientes.

1.2 ¿Qué es un sistema distribuido?

Un **sistema distribuido** está formado por un conjunto de computadores autónomos unidos por una red de comunicaciones y equipados con software de sistemas distribuidos. El software de sistemas distribuidos permite a los computadores coordinar sus actividades y compartir los recursos del sistema: el hardware, el software y los datos. Los usuarios de un sistema distribuido bien diseñado deberían percibir una única

facilidad de computación, aun cuando dicha facilidad podría estar formada por un conjunto de computadores localizados de manera dispersa.

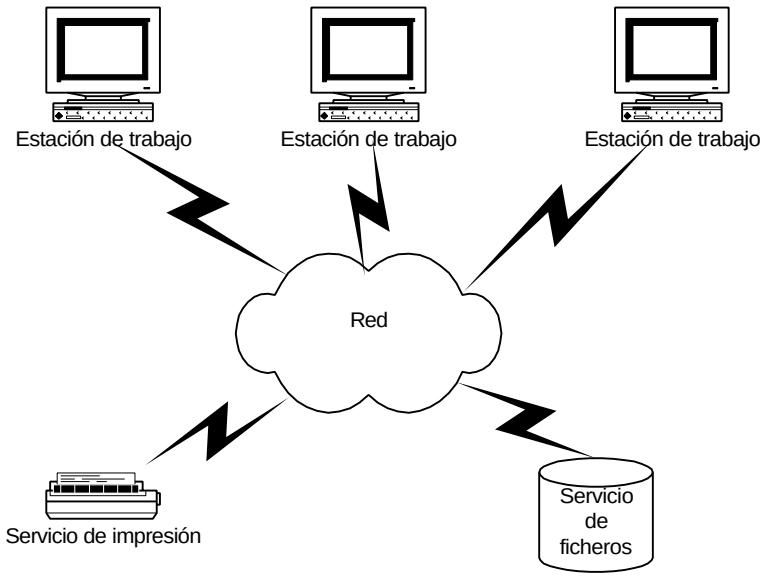


Figura 2.1. Estructura de un sistema distribuido.

El desarrollo de sistemas distribuidos ha seguido la emergencia de las redes de área local de principios de los 70. Más recientemente, la disponibilidad de computadores personales, estaciones de trabajo y servidores de alto rendimiento con un coste relativamente bajo han producido un cambio significativo hacia los sistemas distribuidos, dejando a un lado los grandes sistemas centralizados multiusuario. Esta tendencia se aceleró con el desarrollo de software de sistemas distribuidos, diseñado para permitir el desarrollo de aplicaciones distribuidas.

1.3 Características clave de los sistemas distribuidos

La utilidad de un sistema distribuido viene determinada por la aparición de un **conjunto específico de características** en el mismo: compartición de recursos, apertura (*openness*), flexibilidad, concurrencia, rendimiento, escalabilidad, fiabilidad y tolerancia a fallos, disponibilidad, transparencia, heterogeneidad y seguridad. No se puede asumir que estas características vayan a ser consecuencias propiamente dichas de la distribución, sino todo lo contrario: el software de sistemas y de aplicación habrán de ser diseñados con cuidado con el fin de asegurar que se consiguen todas ellas.

1.3.1 Compartición de recursos

Se utiliza el concepto de **recurso** para referirse a todo aquello que puede ser compartido con éxito en un sistema distribuido. Los beneficios del uso de recursos compartidos ya son bien conocidos de los sistemas operativos centralizados y multiusuario.

- Compartir dispositivos hardware (discos, impresoras y otros periféricos) reduce el coste.
- Compartir datos es esencial en muchas aplicaciones:

- Los equipos de desarrollo de software pueden necesitar acceder al trabajo de los otros y compartir las mismas herramientas de desarrollo con una única copia de compiladores, librerías, editores, depuradores, etc. Cuando se instale una herramienta, todos los usuarios obtienen acceso a ella inmediatamente.
- Muchas aplicaciones comerciales ofrecen a los usuarios la posibilidad de acceder a datos compartidos en una única base de datos activa.
- Un área en rápida expansión para las redes y los sistemas distribuidos es el uso de computadores para soporte de grupos de usuario trabajando de forma cooperativa. Es lo que se conoce como Computer Supported Cooperative Working (CSCW) o groupware y depende en gran medida de los datos que se comparten entre programadores trabajando en distintas estaciones de trabajo.

Los recursos en un sistema distribuido están encapsulados dentro de alguno de los computadores y sólo pueden ser accedidos desde otros computadores utilizando los **mecanismos de comunicación** que se proporcionen para ello. Para que la compartición sea eficiente, cada recurso será gestionado por un programa que ofrecerá una interfaz de comunicación, permitiendo que el recurso sea accedido, manipulado y actualizado de una manera fiable y consistente.

En muchas ocasiones se utiliza el término **gestor de recursos** para nombrar un módulo software que gestiona un conjunto de recursos del mismo tipo. Cada tipo de recurso necesita determinadas políticas y mecanismos de gestión, pero, en general, todos los tipos de recursos tienen algunas necesidades comunes:

- Un **esquema de nombrado** para los recursos que permita acceder a los recursos individuales desde cualquier ubicación.
- La **conversión de nombres** de recursos a direcciones de comunicación.
- La **coordinación de los accesos concurrentes** que cambian el estado de los recursos compartidos, con el fin de asegurar su consistencia.

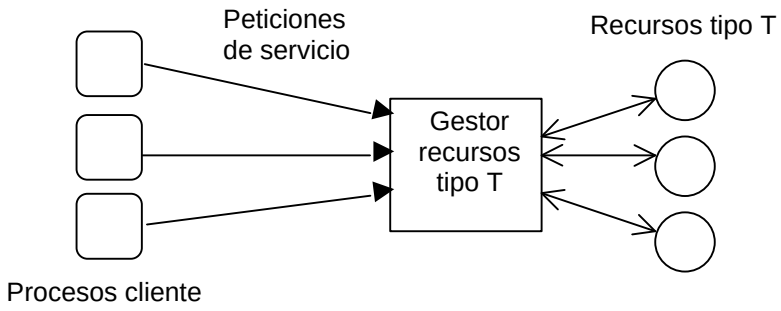


Figura 2.2. Gestor de recursos.

1.3.2 Apertura (del inglés, *openness*)

La apertura de un sistema es una característica que determina si el mismo puede ser extendido de diferentes maneras: hardware y software. Aplicada a un sistema distribuido, esta propiedad viene determinada principalmente por el **grado de permisividad del sistema para añadir servicios** sin duplicar o interrumpir los ya existentes, y se consigue haciendo pública la documentación relativa a las interfaces software consideradas clave en el sistema. Este proceso es semejante a la

estandarización de interfaces, pero no sigue habitualmente los procedimientos oficiales de estandarización, que son lentos y engorrosos.

Históricamente, los sistemas informáticos han sido cerrados. Ejecutaban programas realizados en una serie de lenguajes de programación, pero no permitían a los programadores extender la semántica de los lenguajes para explotar nuevas características del hardware o del sistema operativo.

UNIX [RT74] es el primer ejemplo de un diseño más abierto. Incluye un lenguaje de programación, el C [KR88], que permite a los programadores acceder a todas las facilidades (recursos) gestionadas por el sistema operativo. Dicho acceso se realiza a través de un conjunto de procedimientos, denominados llamadas al sistema, totalmente documentados y disponibles para todos los lenguajes de programación que soporten facilidades de llamada a procedimiento convencionales. Cuando se desea instalar un nuevo periférico en un sistema UNIX, el sistema operativo puede ser extendido para permitir a las aplicaciones acceder a él, añadiendo habitualmente nuevos valores para los parámetros de las llamadas al sistema existentes. Finalmente, al escribirse las aplicaciones en un lenguaje de alto nivel como el C, los programadores pueden realizar programas que se ejecuten sin modificaciones en una amplia variedad de computadores.

Un **sistema distribuido abierto debe ser extensible**, en el sentido de que el conjunto de recursos y facilidades del sistema disponibles para las aplicaciones no esté restringido por las llamadas al sistema del sistema operativo. La **extensibilidad se debe producir a varios niveles**: a nivel hardware, con la adición de computadores a la red, y, a nivel software, con la introducción de nuevos servicios que permitan a los programas de aplicación compartir recursos. Adicionalmente, se cita como beneficio adicional de los sistemas abiertos su **independencia de vendedores individuales**.

1.3.3 Flexibilidad

Otra importante cuestión en el diseño de los sistemas distribuidos es la flexibilidad, que es la característica más importante para el diseño de sistemas abiertos.

La necesidad de flexibilidad viene dada a partir de las siguientes razones:

- **Fácil modificación**. Algunas partes del sistema diseñado van a necesitar ser reemplazadas o modificadas debido a la detección de algún error (*bug*) detectado o porque el diseño ya no es adecuado para el entorno cambiante o los nuevos requerimientos del usuario. Debería ser fácil incorporar cambios en el sistema de forma transparente al usuario.
- **Fácil mejora**. En cualquier sistema, la nueva funcionalidad debe ser añadida según se va solicitando para hacer el sistema más poderoso o fácil de usar. Debería ser fácil añadir nuevos servicios al sistema. Es más, si a un grupo de usuarios no le gusta la forma en que el sistema proporciona un servicio particular, deberían tener la posibilidad de añadir y usar su propio servicio.

El factor de diseño más importante que influye en la flexibilidad de un sistema distribuido es el **modelo usado para diseñar su núcleo** (*kernel*). El núcleo de un sistema opera en un espacio de direcciones inaccesible para los procesos de usuario y es la única parte del sistema que el usuario no puede reemplazar o modificar.

1.3.3.1 Flexibilidad en los sistemas operativos distribuidos

Los dos modelos más comúnmente usados para el diseño del núcleo de los sistemas operativos distribuidos son el núcleo monolítico y el micronúcleo (*microkernel*).

- **Núcleo monolítico**: Mantiene que cada máquina debe ejecutar un núcleo tradicional que proporcione la mayoría de los servicios. Muchos sistemas operativos distribuidos que son extensiones o imitaciones del UNIX usan el modelo monolítico.
- **Micronúcleo**: Sostiene que el núcleo debe proporcionar la menor funcionalidad posible y que el grueso de los servicios debe obtenerse a través de los servidores a nivel usuario.

La segunda corriente es más flexible ya que el sistema operativo sólo proporciona cuatro servicios mínimos: mecanismos de comunicación entre procesos, administración de memoria a bajo nivel, administración de procesos a bajo nivel y gestión de la entrada/salida.

Comparando ambos modelos, el modelo **micronúcleo tiene varias ventajas**. En primer lugar, el tamaño de los núcleos monolíticos proporciona flexibilidad y configurabilidad menores.

El modelo micronúcleo, modular por naturaleza, es fácil de diseñar, implementar e instalar y, dado que muchos servicios se implementan a nivel usuario, es fácil de modificar el diseño de un servicio o añadir un servicio nuevo para lo que no es necesario parar el sistema y arrancar un nuevo núcleo.

La única ventaja potencial de la primera opción es el rendimiento, ya que las llamadas al núcleo y la realización por parte de éste de todo el trabajo puede ser más rápido que el envío de mensajes a los servidores remotos.

A pesar de la potencial pérdida de rendimiento, el modelo **micronúcleo es el preferido para el diseño de sistemas operativos modernos** por las siguientes razones:

- Las ventajas de flexibilidad del modelo micronúcleo compensan la penalización del rendimiento
- Algunos resultados experimentales han mostrado que, aunque en la teoría el modelo micronúcleo parece tener un rendimiento más pobre que el modelo monolítico, en la práctica otros factores tienden a dominar.

1.3.4 Concurrencia

Los sistemas distribuidos **amplían la posibilidad de ejecución concurrente** de los sistemas centralizados (ejecución paralela, en el caso de que dispongan de varios procesadores) por el hecho de existir varios computadores, cada uno de los cuales con uno o varios procesadores.

Si existe un único procesador, la concurrencia entre varios procesos de una máquina se consigue intercalando la ejecución de trozos de los procesos. Si hay N procesadores, se pueden ejecutar en paralelo hasta N procesos.

En un sistema distribuido existen muchos computadores, cada una con uno o más procesadores. Si hay M computadores con un procesador cada uno, entonces pueden ejecutarse en paralelo hasta M procesos, suponiendo que los procesos estén situados en máquinas distintas.

En un sistema distribuido basado en el modelo de compartición de recursos descrito, las oportunidades de ejecución paralela se basan en dos razones:

- Muchos usuarios invocan simultáneamente comandos o interaccionan con programas de aplicación. De esta forma, uno o más procesos de aplicación se ejecutan en nombre de cada usuario activo. En la mayoría de los sistemas distribuidos, los procesos de aplicación se ejecutan en la estación de trabajo del usuario y no entran en conflicto por los recursos de procesamiento con los procesos de aplicación de otros usuarios. Si la estación de trabajo tiene un único procesador y hay más de un proceso de aplicación ejecutándose en ella, se ejecutan de forma intercalada. Las estaciones de trabajo con múltiples procesadores hacen posible ejecutar varias aplicaciones en paralelo o ejecutar aplicaciones que explotan más de un procesador.
- Muchos procesos servidores se ejecutan concurrentemente y cada uno responde a diferentes peticiones de distintos clientes. Puede existir más de un proceso servidor para cada tipo de recurso. Normalmente se ejecutan en computadores distintos, haciendo posible que cada servidor se ejecute en paralelo con los otros y con los procesos que se ejecutan en estaciones de trabajo. Las peticiones para acceder a los recursos en un servidor determinado se encolan al servidor y se procesan en secuencia. Incluso varias peticiones pueden ser procesadas concurrentemente por alguna de las múltiples instancias de los procesos gestores de recursos. Cuando varios procesos acceden al mismo recurso concurrentemente, el servidor debe sincronizar las acciones para asegurarse de que no hay conflicto. La sincronización debe ser planeada cuidadosamente para asegurar que los beneficios de la concurrencia no se pierden.

Las peticiones que llegan a un gestor de recursos deben ser procesadas adecuadamente para garantizar un acceso consistente a los mismos. Una posibilidad es ir encolando las peticiones y tratándolas de manera secuencial. Otra posibilidad es crear una instancia del gestor de recursos para cada petición, instancias que tendrán que sincronizar sus acciones para no entrar en conflicto.

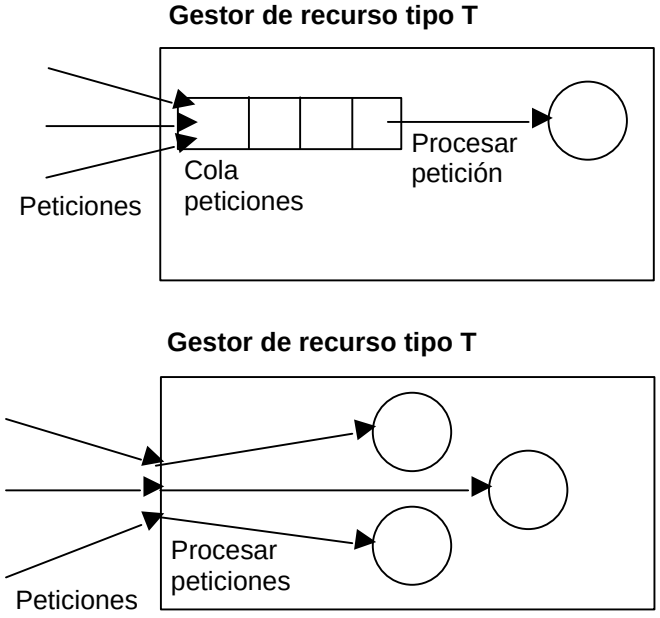


Figura 2.3. Distintas formas de gestión de peticiones sobre recursos.

1.3.5 Rendimiento

La ejecución de una aplicación en un sistema distribuido debe ser mejor (o al menos, no debe ser peor) que su ejecución en un único procesador.

El problema del rendimiento se complica por el hecho de que la comunicación, factor esencial en un sistema distribuido (y ausente en un sistema centralizado) es lenta, en general. El envío de un mensaje y la obtención de una respuesta puede tardar un milisegundo en una red de área local (LAN, *Local Area Network*). Para mejorar el rendimiento, parece que hay que minimizar el número de mensajes. El problema de esta estrategia es que la mejor forma de mejorar el rendimiento es tener muchas actividades ejecutándose de manera paralela en distintos procesadores, lo que implica el envío de muchos mensajes.

Una posible solución es dividir los trabajos utilizando **paralelismo de grano fino** (muchas tareas con poco trabajo) o **paralelismo de grano grueso** (pocas tareas con mucho trabajo). Dado que las tareas del primer modelo se comunicarán con más frecuencia, para que el paralelismo de grano grueso se ajusta mejor al modelo comentado.

Otra fuente de problemas es la tolerancia a fallos (que se describe más adelante en este capítulo), ya que la redundancia de servidores implica también una redundancia de mensajes.

1.3.6 Escalabilidad

Los sistemas distribuidos deben operar de una manera efectiva a **diferentes escalas**: desde el sistema distribuido mínimo, consistente en dos computadores y un servidor de ficheros, pasando por redes de área local que contienen un conjunto apreciable de computadores y servidores especializados, hasta conjuntos de redes locales interconectadas que contienen cientos de computadores. En todos estos casos, el conjunto de computadores forma un único sistema distribuido que permite la compartición de los recursos entre todos ellos.

El **software de sistemas y de aplicación no deben cambiar** cuando la escala del sistema cambia.

La necesidad de escalabilidad no es sólo un problema hardware o de rendimiento de red. Afecta al diseño de casi cualquier aspecto de un sistema distribuido. En un sistema centralizado ciertos recursos compartidos (memoria, procesadores, canales de entrada/salida) se suministran de forma limitada y no pueden ser replicados infinitamente. En los sistemas distribuidos, la limitación en el suministro de algunos de esos recursos se elimina, ya que es posible añadir, por ejemplo, más computadores o más memoria. Pero permanecen otras limitaciones si el diseño del sistema no reconoce la necesidad de escalabilidad. Dichas limitaciones se producen más bien en el ámbito del software de sistemas distribuidos.

La demanda de escalabilidad en los sistemas distribuidos ha llevado a una política de diseño en la que se asume que **ningún recurso hardware o software puede ser suministrado de forma restrictiva**. En lugar de eso, como la demanda de recursos crece, debería ser posible extender el sistema para cubrirla.

A medida que los sistemas distribuidos se hacen más grandes, se perfilan los siguientes principios de diseño: evitar componentes y algoritmos centralizados y ejecutar el máximo de operaciones en las estaciones de trabajo de los usuarios.

Evitar componente centralizados

En el diseño de un sistema operativo distribuido, el uso de componentes centralizados como un servidor de ficheros o una base de datos para el sistema completo hace que el sistema no sea escalable debido a las siguientes razones:

- Normalmente, el fallo de una entidad centralizada hace que todo el sistema caiga.
- El rendimiento de la entidad centralizada se convierte normalmente en un cuello de botella.
- Incluso en el caso de que una entidad centralizada tenga suficiente capacidad de procesamiento y almacenamiento, la capacidad de la red que conecta la entidad centralizada con otros nodos del sistema se satura frecuentemente cuando la contención por la entidad se incrementa más allá de un nivel determinado.
- En una gran red formada por varias redes de área local interconectadas, es obviamente ineficiente que un servicio particular esté siempre servido por un servidor que está a varias redes de distancia.

La replicación de recursos y los algoritmos de control distribuido son técnicas de uso frecuente para conseguir este objetivo. De hecho, para mejorar la escalabilidad debería utilizarse una configuración simétrica en la que todos los nodos del sistema sean aproximadamente iguales.

Evitar algoritmos centralizados

Un algoritmo centralizado es aquel que opera recopilando información de todos los nodos, procesándola en un único nodo y distribuyendo los resultados a los otros. El uso de tales algoritmos en el diseño de un sistema operativo distribuido no es aceptable desde el punto de vista de la escalabilidad. Las razones para ello, son muy similares a las vistas para los componentes.

Ejecutar el máximo de operaciones en las estaciones de trabajo de los clientes

Cuando sea posible, una operación debería ejecutarse en la propia estación de trabajo del cliente en lugar de una máquina servidora. Esto es debido a que el servidor es un recurso para varios clientes. Este principio mejora la escalabilidad y permite minimizar la degradación del rendimiento del sistema a medida que este crece reduciendo la contención por los recursos compartidos.

1.3.7 Fiabilidad y tolerancia a fallos

Este es uno de los objetivos originales de la construcción de sistemas distribuidos, el hacerlos más fiables que los sistemas con un único procesador, debido precisamente a la existencia de múltiples instancias de los recursos. Sin embargo, el sistema operativo debe ser diseñado de forma apropiada para sacar provecho de esta característica de un sistema distribuido.

Un **fallo** es un defecto algorítmico o mecánico que puede generar un error y que causa un fallo del sistema. Dependiendo del modo en que se comporta el sistema en caso de fallo, los fallos del sistema son de dos tipos:

- ***Fail stop***: El sistema detiene su funcionamiento después de cambiar a un estado en el que el fallo puede ser detectado.
- **Bizantino**: El sistema sigue funcionando pero producirá resultados erróneos.

Para que un sistema sea fiable, los mecanismos de gestión de fallos del sistema operativo deben diseñarse de tal forma que evite, tolere y detecte y recupere los fallos.

1.3.7.1 Evitar fallos

Implica el diseño del sistema de tal forma que la ocurrencia de fallos se minimice.

1.3.7.2 Tolerar fallos

La tolerancia a fallos es la habilidad de un sistema de continuar funcionando en caso de fallos parciales del sistema. El rendimiento del sistema podría degradarse pero el sistema seguiría funcionando de forma adecuada. Algunos conceptos importantes que pueden usarse para mejorar la tolerancia a fallos de un sistema operativo distribuido son:

Técnicas de redundancia

Se trata de evitar puntos de fallo en el sistema replicando el hardware y el software críticos de tal forma que, si una instancia falla, las otras pueden seguir siendo utilizadas. Con estas técnicas, se introduce cierta sobrecarga al sistema para mantener consistentes dos o más copias del recurso. Esto plantea una cuestión importante acerca de qué cantidad de réplicas es suficiente y no demasiada.

Control distribuido

Para una mejor fiabilidad, muchos de los algoritmos y protocolos utilizados en un sistema operativo distribuido deben emplear un mecanismo de control distribuido para evitar puntos únicos de fallo. Es importante recalcar que, cuando se utilizan múltiples servidores distribuidos en un sistema para proporcionar un tipo de servicio, los servidores deben ser independientes. Es decir, el diseño no debe requerir el funcionamiento simultáneo de los servidores.

1.3.7.3 Detección y recuperación de fallos

La detección y recuperación de fallos para mejorar la fiabilidad implican el uso de mecanismos hardware y software para determinar la existencia de un fallo y corregirlo, llevando al sistema a un estado aceptable para continuar su ejecución. Algunas de las técnicas más usadas son:

Transacciones atómicas

Una transacción atómica es una computación que consiste en una serie de operaciones que tienen lugar de forma indivisible en presencia de fallos y de otras computaciones concurrentes. Es decir, o todas las operaciones se ejecutan con éxito o ninguna tiene efecto y ningún otro proceso en ejecución concurrente puede modificar o siquiera observar los estados intermedios de las operaciones. Las transacciones ayudan a preservar la consistencia de un conjunto de objetos compartidos in presencia de fallos y de accesos concurrentes. Hacen la recuperación mucho más facil debido a que una transacción finaliza sólo en uno de dos estados: o todo ha ido bien o ninguna de las operaciones se ha ejecutado.

En un sistema con transacciones, si un proceso se para de forma inesperada debido a un fallo hardware o a un error software antes de que la transacción termine, el sistema restaura los objetos que se estuviesen ejecutando a sus estados originales. Si el sistema no soporta transacciones, fallos inesperados de un proceso durante el procesamiento de una operación pueden dejar los objetos bajo modificación en un estado inconsistente. Es

más, sin transacciones puede ser difícil o imposible recuperar los objetos a sus estados originales.

Servidores sin estado

En el modelo cliente-servidor, utilizado frecuentemente en sistemas distribuidos para servir las solicitudes de usuario, un servidor puede implementarse usando uno de los siguientes paradigmas de servicio: con o sin estado. Estos paradigmas se distinguen por un aspecto de la relación cliente-servidor: si la historia de la solicitud entre cliente y servidor afecta o no a la ejecución del siguiente servicio solicitado.

La aproximación con estado depende de la historia de las peticiones servidas, mientras la aproximación sin estado no. Los servidores sin estado tienen una ventaja sobre los servidores con estado en caso de fallo. El paradigma de servicio sin estado hace que la recuperación ante fallos sea mucho más sencilla, dado que no se mantiene ninguna información en el servidor acerca del estado del cliente. Por otro lado, el paradigma del servidor con estado requiere un procedimiento completo de recuperación ante fallos. Tanto el cliente como el servidor necesitan detectar fallos. El servidor necesita detectar las caídas del cliente para poder descartar cualquier estado que se esté manteniendo sobre el cliente y el cliente debe detectar las caídas del servidor para poder realizar las funciones de manejo de errores necesarias.

Retransmisión de mensajes basada en temporizadores y reconocimiento

En un sistema distribuido, eventos como el fallo de un nodo o la caída de una línea de comunicación pueden interrumpir una comunicación en marcha entre dos procesos provocando pérdidas de mensajes. Un mecanismo de comunicación fiable debe tener mecanismos de detección de pérdida de mensajes para poder retransmitirlos. La gestión de los mensajes perdidos implica, habitualmente, el retorno de mensajes de reconocimiento (*acknowledgement*) y la retransmisión en caso de vencimiento de algún temporizador. Es decir, el receptor debe retornar un mensaje de reconocimiento en un intervalo de tiempo y, en caso de que en ese tiempo el remitente no reciba el reconocimiento, asume que el mensaje se ha perdido y lo retransmite. Uno de los mayores problemas asociados con esta aproximación es el de la duplicación de mensajes que puede producirse al retransmitir un mensaje en caso de vencimiento del temporizador. Un mecanismo de comunicación fiable debería ser capaz de detectar y gestionar mensajes duplicados. La gestión de mensajes duplicados implica normalmente un mecanismo para generar y asignar automáticamente números a los mensajes.

Cualquier mecanismo utilizado para incrementar la fiabilidad del sistema distribuido provoca una potencial pérdida de eficiencia debido a la sobrecarga que introducen estas técnicas. Uno de los mayores objetivos en el diseño de sistemas operativos distribuidos consiste en integrar estos mecanismos de forma que el coste que introduzcan sea aceptable.

1.3.8 Disponibilidad

Los sistemas distribuidos también proporcionan un alto grado de disponibilidad en el caso de fallos hardware. La disponibilidad de un sistema es una medida de la **proporción de tiempo que está disponible** para su uso. Un fallo en un computador multi-usuario casi siempre provoca que deje de estar disponible para todos sus usuarios. Cuando uno de los componentes en un sistema distribuido falla, sólo el trabajo que está usando el componente que falla se ve afectado. Por ejemplo, un usuario puede moverse

a otra estación de trabajo si la que está usando falla. De una forma similar, un proceso servidor puede ser rearrancado en otra máquina.

1.3.9 Transparencia

La transparencia se define como el **ocultamiento** al usuario del hecho de que los componentes de un sistema distribuido están dispersos y separados, de tal forma que el sistema se perciba como un todo único en lugar de una colección independiente de componentes.

La separación de componentes es inherente a los sistemas distribuidos. Esta separación posibilita la ejecución paralela de programas, la contención y recuperación de fallos en los componentes sin que afecte al resto del sistema, el uso de canales de comunicación controlados y aislados como un método para reforzar la seguridad y la protección y el crecimiento/disminución incremental del sistema a través de la adición/eliminación de componentes.

Las consecuencias de la **transparencia tienen una gran influencia en el diseño** del sistema.

La **transparencia es la clave y el objetivo de los sistemas operativos distribuidos**. Un sistema operativo distribuido debe estar diseñado de tal forma que una colección de máquinas distintas conectadas por un sistema de comunicación, se presente ante los usuarios como un uniprocesador. El usuario no debe tener conocimiento de dónde están localizados sus ficheros o dónde se están ejecutando sus trabajos, es decir, se trata de engañar al usuario de forma que para éste todo el sistema parezca una única máquina con un sólo procesador en tiempo compartido.

El *ANSA Reference Manual* [ANS89] y el *International Standars Organization´s Reference Model for Open Distributed Processing (RM-ODP)* [ISO92] identifican ocho formas de transparencia, que proporcionan un resumen de las motivaciones y objetivos de los sistemas distribuidos. En estas definiciones *ANSA* utiliza el término **objeto de información** para denotar las entidades a las que se aplica la transparencia de distribución.

Transparencia de acceso

Permite el acceso a los objetos de información usando un conjunto de operaciones común, independientemente de que éstos sean locales o remotos.

Los usuarios no necesitan saber si un recurso es local o remoto. Esto significa que el sistema operativo distribuido debería permitir a los usuarios el acceso a recursos remotos del mismo modo que a recursos locales. La interfaz de usuario o llamadas al sistema no debería distinguir entre recursos locales o remotos y debe ser responsabilidad del sistema operativo localizar los recursos y arreglarlo todo para dar servicio a las peticiones de usuario. Un claro ejemplo de falta de transparencia de acceso es el presentado por un conjunto de máquinas UNIX unidas mediante una red en las que el acceso a ficheros locales se realiza mediante una llamada al sistema operativo local (*open, read,* …) mientras que los accesos a ficheros remotos necesitan una serie de operaciones especiales (*ftp,* …) para traer una copia al disco duro local.

Transparencia de localización

Permite acceder a objetos de información sin conocimiento de su localización. Los usuarios no deben ni pueden indicar en sus operaciones la localización de los recursos

del sistema, tanto software como hardware (CPU, ficheros, etc.). Por tanto, la dirección del recurso no está codificada en el nombre del recurso.

Los dos aspectos principales en la transparencia de localización son:

- **Transparencia de nombres**: Se refiere al hecho de que un recurso no revela su localización física, es decir, el nombre de un recurso es independiente de la topología del sistema, la conexión o la localización actual del recurso. Es más, para todos aquellos recursos que puedan moverse por el sistema (como ficheros) debe estar permitido moverlos sin que sean necesarios cambios en los nombres. Los nombres deben ser únicos en el sistema.
- **Movilidad de usuario**: Se refiere al hecho de que independientemente de la máquina en que se conecta el usuario, debe tener acceso a cualquier recurso con el mismo nombre. Es decir, el usuario no debería tener que usar distintos nombres para acceder a un mismo recurso desde dos nodos distintos.

Los dos tipos de transparencia más importantes son las de acceso y localización. Su presencia o ausencia afecta de una manera muy importante al uso de los recursos distribuidos. Se utiliza habitualmente el término **transparencia de red** para referirse a ambas simultáneamente. La transparencia de red proporciona un grado de anonimato para los recursos similar al encontrado en los sistemas operativos centralizados.

Transparencia de concurrencia

Permite que varios procesos operen concurrentemente utilizando objetos de información compartidos, sin que por ello haya interferencias entre ellos.

En un sistema distribuido, existen múltiples usuarios separados en el espacio que usan el sistema concurrentemente. En tal situación, es necesario permitir que se compartan recursos entre los procesos en ejecución. Sin embargo, y debido a que el número de recursos en el sistema está restringido, un proceso influirá en la ejecución de otro que se ejecuta concurrentemente al necesitar competir por los recursos. La transparencia de concurrencia significa que cada usuario tiene la impresión de que es el único usuario del sistema. Para proporcionar transparencia de concurrencia, los mecanismos de compartición de recursos deben tener las siguientes propiedades:

- **Propiedad de ordenación de eventos**: asegura que todas las peticiones de acceso a los recursos del sistema se ordenan de forma apropiada para proporcionar una visión consistente a los usuarios.
- **Propiedad de exclusión mutua**: asegura que, en cualquier momento, como mucho un proceso accede a un recurso compartido que no debe ser usado simultáneamente por varios procesos si la operación del programa debe ser correcta.
- **Propiedad de no-inanición**: asegura que en caso de que un recurso no pueda ser utilizado simultáneamente por varios procesos, cualquier petición lo liberará en algún momento y se le concederá a otro.
- **Propiedad de no interbloqueo**.

Transparencia de replicación

Permite el uso de múltiples instancias de objetos de información para aumentar la fiabilidad y el rendimiento, sin que los usuarios o los programas de aplicación tengan conocimiento de la existencia de las réplicas.

Para un mejor rendimiento y fiabilidad, casi todos los sistemas operativos distribuidos permiten la creación de réplicas de recursos en distintos nodos del sistema. En estos sistemas, la existencia de múltiples copias de un recurso y la actividad de replicación deberían ser transparentes al usuario. Dos cuestiones importantes relacionadas con la transparencia de replicación son el nombrado de las réplicas y el control de la replicación. Es responsabilidad del sistema operativo dar nombre a las copias del recurso y establecer la correspondencia entre el nombre proporcionado por el usuario y una réplica apropiada del recurso. Es más, las decisiones acerca del control de la replicación tales como cuántas copias de cada recurso deberían crearse, dónde debería estar colocada cada réplica y cuándo debería crearse o borrarse cada una deben estar automatizadas en el sistema.

Transparencia de fallo

Oculta los fallos, permitiendo a los usuarios y programas de aplicación completar las tareas a pesar de fallos en componentes hardware o software.

El sistema operativo debe enmascarar al usuario los fallos parciales del sistema como un fallo en la comunicación, el fallo de una máquina o de un dispositivo de almacenamiento. Un sistema operativo distribuido que soporte esta propiedad continuará funcionando en presencia de fallos. La transparencia completa frente a fallos no es alcanzable hoy día con el estado del arte actual en los sistemas operativos distribuidos dado que no todos los tipos de fallos pueden ser gestionados de forma transparente al usuario. Es más, cualquier intento de construir un sistema operativo distribuido completamente transparente a fallos resultaría muy lento y costoso debido a la gran cantidad de información redundante requerida para gestionar cualquier tipo de fallos.

Transparencia de migración

Permite el movimiento de objetos de información dentro del sistema sin que esto afecte a las operaciones de los usuarios o programas de aplicación.

Por razones de rendimiento, fiabilidad y seguridad, aquellos objetos que tienen capacidad de moverse, son migrados frecuentemente de un nodo a otro del sistema distribuido. El objetivo de la transparencia de migración es asegurar que el movimiento del objeto es gestionado automáticamente por el sistema de forma transparente al usuario. Hay tres cuestiones importantes para conseguir este objetivo:

- Las decisiones de migración tales como qué objeto mover desde dónde hasta dónde, debe llevarlas a cabo el sistema automáticamente.
- La migración de un objeto de un nodo a otro no debería requerir ningún cambio en su nombre.
- Cuando el objeto que migra es un proceso, el mecanismo de comunicación entre procesos debería asegurar que cualquier mensaje enviado a ese proceso llegará a su destino sin necesidad de que el proceso remitente lo reenvíe si el receptor se mueve mientras el mensaje está siendo enviado.

Transparencia de rendimiento

Permite que el sistema sea reconfigurado para mejorar el rendimiento cuando la carga varía.

Aquella situación en la que un procesador del sistema esté sobrecargado mientras otro está ocioso no debe ocurrir, es decir, la capacidad de procesamiento del sistema debería distribuirse uniformemente entre todos los trabajos disponibles en el sistema.

Transparencia de escalabilidad

Permite que el sistema y las aplicaciones se expandan sin tener que cambiar la estructura del sistema o los algoritmos de la aplicación. Esto implica que la arquitectura debe ser diseñada siguiendo criterios de sistemas abiertos y los algoritmos deben ser escalables.

1.3.10 Heterogeneidad

Un sistema distribuido heterogéneo consiste en conjuntos interconectados de hardware o software distintos. Dada esta diversidad, el diseño de sistemas distribuidos heterogéneos es más difícil que en el caso homogéneo donde cada sistema está basado en el mismo o similar hardware y software.

En un sistema heterogéneo es necesario algún tipo de transformación de datos para interaccionar entre dos nodos incompatibles. Algunos de los primeros sistemas dejaban esta transformación a los usuarios, hecho inaceptable. Esta transformación puede llevarse a cabo tanto en el lado del cliente como del servidor. Tanto uno como otro tienen serios inconvenientes.

La complejidad de este proceso de transformación puede reducirse utilizando un **formato intermedio estándar** para los datos . Con esta solución se declara un formato intermedio y cada nodo sólo requiere software que le permita transformar su propio formato al estándar y viceversa.

1.3.11 Seguridad

Con el fin de que los usuarios puedan confiar en el sistema, es necesario proteger los distintos recursos del mismo frente a accesos no autorizados. Asegurar la seguridad en un sistema distribuido es más difícil que en uno centralizado dada la inexistencia de un punto de control único y el uso de redes inseguras para el tráfico de datos.

En un sistema centralizado todos los usuarios son autenticados por el sistema en el proceso de conexión (*login*) y es sencillo comprobar si el usuario está autorizado o no.

En un sistema distribuido, sin embargo, dado que el modelo cliente/servidor se usa frecuentemente para solicitar y servir peticiones, cuando un cliente envía un mensaje al servidor, el servidor debe saber quién es el cliente. Esto no es tan simple como enviar con el mensaje la identificación del remitente, porque algún intruso podría pretender pasar por un cliente autorizado o podría cambiar el mensaje durante la transmisión. La seguridad en un sistema distribuido tiene los siguientes requerimientos adicionales:

- Debe ser posible que el remitente de un mensaje sepa si el mensaje fue recibido por el receptor deseado.
- Debe ser posible que el receptor de un mensaje sepa si el mensaje fue enviado por el remitente indicado.
- Debe ser posible que el remitente y el receptor estén seguros de que el contenido de un mensaje no fue cambiado durante la transferencia de datos.

1.4 Ventajas e inconvenientes de los sistemas distribuidos

Los sistemas operativos distribuidos son mucho más complejos de construir que los centralizados debido al hecho de que el sistema, además de tener que usar y gestionar los recursos distribuidos, también debe ser capaz de gestionar la comunicación y la seguridad. A pesar de esto, el uso de los sistemas distribuidos tiene un ritmo de crecimiento rápido.

1.4.1 Ventajas de los sistemas distribuidos frente a los centralizados

1.4.1.1 Distribución inherente de las aplicaciones

Un elemento a favor de los sistemas distribuidos es la naturaleza inherentemente distribuida de algunas aplicaciones, como las de los bancos en las que se trabaja sobre bases de datos comunes en diferentes sucursales, y en general aplicaciones que utilizan máquinas que están separadas una cierta distancia, reservas de líneas aéreas, etc.

1.4.1.2 Compartición de datos y dispositivos

La necesidad de compartir datos entre máquinas y usuarios diferentes, pudiendo utilizar ficheros compartidos es también uno de los principales motivos de la aparición de los sistemas distribuidos. Por otro lado la compartición de dispositivos físicos, como las impresoras, aumenta considerablemente las prestaciones.

1.4.1.3 Economía

En primer lugar los sistemas distribuidos tienen una razón precio/rendimiento mucho mejor que la de un único sistema centralizado. Este hecho se empezó a notar con la aparición de la tecnología de los microprocesadores, que dejan a un lado los grandes computadores de los años ochenta. Así, si se desea una gran capacidad de cómputo resulta mucho más económico comprar un conjunto de procesadores reunidos en un mismo sistema que un único procesador muy potente. Esta es la principal razón de la tendencia hacia los sistemas distribuidos.

1.4.1.4 Velocidad

La velocidad de una máquina *mainframe* está restringida a un cierto límite. Sin embargo, la utilización en paralelo de múltiples procesadores hacen que la velocidad que se pueda alcanzar sea mucho mayor. Es más, si es posible partir una computación en subcomputaciones que se ejecuten concurrentemente en los distintos procesadores, la mejora sería considerable.

1.4.1.5 Fiabilidad

Los sistemas distribuidos aportan una mayor fiabilidad que los centralizados al distribuir la carga entre muchos procesadores; el fallo de uno de ellos no tiene por qué, provocar fallos en el resto. Con un software adecuado el resto de las máquinas podrían encargarse de su trabajo.

Un aspecto muy importante de la fiabilidad es la disponibilidad que se refiere a la fracción de tiempo durante la cual el sistema está disponible para su uso. En comparación con un sistema centralizado, un sistema distribuido tiene un grado de disponibilidad mayor, lo que es importante en la computación de aquellas aplicaciones críticas cuyo fallo puede ser desastroso.

Sin embargo, la fiabilidad suele implicar una penalización en el rendimiento. Es necesario mantener una solución de compromiso entre ambos.

1.4.1.6 Extensibilidad y crecimiento por incrementos

Los sistemas distribuidos son más receptivos al crecimiento ya que es fácil añadir nuevos procesadores con poco coste si el sistema actual se queda pequeño.

1.4.1.7 Flexibilidad con respeto a la utilización de una máquina aislada

El usuario puede tener a su disposición un conjunto de ordenadores que podrán repartir la carga de trabajo de manera adecuada para conseguir una computación más eficaz. Además, al tener disponible un conjunto de máquinas distintas entre las que se puede seleccionar la más adecuada para procesar el trabajo del usuario dependiendo de la naturaleza del trabajo.

1.4.2 Desventajas de los sistemas distribuidos

1.4.2.1 Software

El problema fundamental de los sistemas distribuidos es el software. Aún no existe mucha experiencia en el diseño, implantación y uso de software distribuido. Precisamente, este es un campo de investigación actual.

1.4.2.2 Redes

Las redes son indispensables para la comunicación entre máquinas, sin embargo, pueden plantear problemas de saturación o pérdidas de mensajes.

1.4.2.3 Seguridad

El posible acceso a todo el sistema por parte de los usuarios plantea el inconveniente de la necesidad de un sistema de seguridad adecuado.

A pesar de todos estos problemas, la tendencia hacia la distribución es indiscutible.

1.5 Definición y funciones de un sistema operativo distribuido

En los apartados anteriores se describieron con detalle las características, ventajas e inconvenientes de la utilización de sistemas distribuidos. Llega el momento de definir lo que se entiende por **sistema operativo distribuido**.

En primer lugar, un sistema operativo distribuido es un tipo de sistema operativo. En este sentido, un sistema operativo distribuido proporciona un entorno para la ejecución de los programas de usuario, controla su ejecución para prevenir errores y gestiona los recursos disponibles.

En segundo lugar, un sistema operativo distribuido es un tipo de sistema distribuido. Por tanto, todas las características anteriormente descritas le son aplicables. Evidentemente, no se exige a todo sistema operativo distribuido todas las características anteriores con un grado de exigencia máximo, aunque sí hay una característica en la que la mayor parte de los autores suelen hacer hincapié: la transparencia.

Teniendo en cuenta todo lo dicho anteriormente, es posible dar una definición informal de un sistema operativo distribuido [Hut87]:

> *"Un sistema operativo distribuido es aquel que proporciona a sus usuarios la misma visión que un sistema operativo centralizado ordinario, pero que se ejecuta en unidades de procesamiento múltiples e independientes. El concepto clave es la transparencia. En otras palabras, el uso de múltiples procesadores debería ser invisible (transparente) al usuario"*

En la literatura de sistemas operativos se tiende a definir el concepto de sistema operativo ofreciendo una descripción de su funcionalidad. Siguiendo dicha tendencia, y con el fin de completar la definición de sistema operativo distribuido, se puede decir que un **sistema operativo distribuido** debe:

- Controlar la asignación de recursos para posibilitar su uso de la manera más eficiente. Debido al carácter distribuido de los recursos, el sistema operativo podrá optar por la utilización de técnicas de replicación, que permitan acercar al máximo los recursos a las aplicaciones que los utilizan.
- Proporcionar al usuario un computador virtual que le sirva como entorno de programación de alto nivel. No es conveniente que el usuario, al menos el poco experimentado, tenga que lidiar con la complejidad del sistema distribuido. Al igual que ocurrió en su momento con la introducción de los sistemas de tiempo compartido, se proporcionará al usuario un computador virtual donde escribir y ejecutar sus programas.
- Ocultar la distribución de los recursos. El computador virtual que se proporciona a los usuarios estará formado por un conjunto de recursos virtuales, que el usuario podrá manejar como si fueran locales. El sistema operativo se encargará de la asociación de los recursos virtuales con recursos reales, con la dificultad añadida de que pueden encontrarse en cualquier lugar del sistema distribuido.
- Proporcionar mecanismos para proteger los recursos del sistema contra accesos de usuarios no autorizados. Los recursos del sistema operativo, tomen la forma que tomen, deberán ser protegidos de la misma forma que ocurría en los sistemas centralizados. El sistema operativo tendrá en cuenta que las peticiones de utilización de dichos recursos pueden llegar de cualquier parte del sistema distribuido.
- Proporcionar una comunicación segura. En un sistema distribuido las comunicaciones son una parte imprescindible. A través de las líneas de comunicación que unen los diferentes nodos del sistema distribuido circula información crítica para el buen funcionamiento del sistema. Es imprescindible que dicha información está a salvo de lecturas o modificaciones por parte de usuarios no autorizados que podrían traer consigo un incorrecto funcionamiento del sistema o la obtención por parte los usuarios de privilegios para su utilización.

El diseño de un sistema operativo distribuido es más complejo que el diseño de un sistema operativo centralizado por varias razones. En el diseño de un sistema operativo centralizado se asume que el sistema operativo tiene completo acceso a información acerca del sistema y su funcionamiento. Sin embargo, un sistema operativo distribuido, debe diseñarse con la suposición de que tal información completa y centralizada nunca estará disponible. En un sistema distribuido los recursos están físicamente distribuidos, no existe un reloj centralizado en el sistema, el envío de mensajes supone un retraso en el sistema e incluso, pueden perderse. Debido a todas estas razones, un sistema operativo distribuido no puede tener una información global, actualizada y consistente acerca del estado de los distintos componentes del sistema.

1.6 Arquitectura de las aplicaciones distribuidas

Como ya se ha mencionado, los sistemas distribuidos posibilitan la compartición de recursos entre diferentes clientes ubicados, posiblemente, en diferentes computadores de una red. Para construir una aplicación distribuida es necesario decidir cómo se va a dividir en unidades funcionales y cómo se van a asignar éstas a la parte cliente o a uno o más servidores. Las unidades funcionales más típicas son: la interfaz de usuario, la lógica de la aplicación (denominada también lógica de negocio) y los datos compartidos.

En aplicaciones (arquitectura) de **2 capas** (*2-tier*) la lógica de la aplicación está encapsulada en la interfaz de usuario en el cliente, en la base de datos en el servidor o en ambas. Un servidor de base de datos con procedimientos almacenados es un ejemplo de sistema de 2 capas.

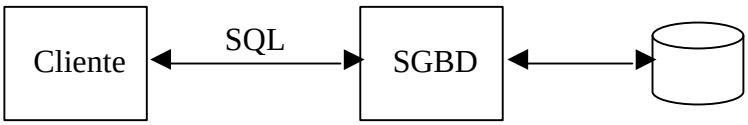


Figura 2.4. Aplicación de 2 capas.

Su simplicidad es el factor fundamental que ha guiado la popularidad de los sistemas de 2 capas. Las herramientas visuales (Delphi, C++ Builder, etc.) permiten construir aplicaciones de este tipo con gran facilidad.

Su mayor problema es el tamaño de la parte cliente, dado que deben contener tanto la interfaz con el usuario como buena parte de la lógica de la aplicación.

En las aplicaciones de **3 capas** (*3-tier*), la lógica de la aplicación reside en la capa intermedia: está separada tanto de la interfaz con el usuario como de los datos. Un servidor de aplicaciones Web es un ejemplo típico de sistema de 3 capas.

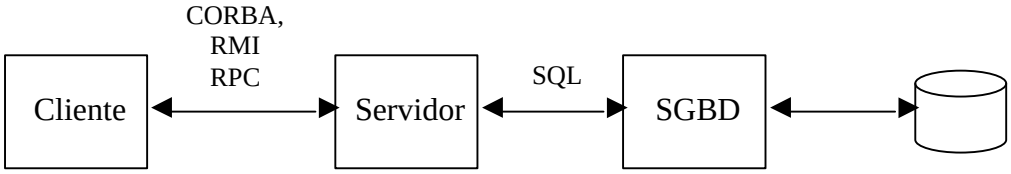


Figura 2.5. Aplicación de 3 capas.

En teoría, los sistemas de 3 capas son más escalables, robustos y flexibles. Además, son más fáciles de desplegar en la red, dado que la mayor parte del código se ejecuta en los servidores, sobre todo con tecnologías como los *applets* de Java. Los clientes, en lugar de interactuar directamente con la base de datos, invocan a la lógica de negocio de la capa intermedia. La capa intermedia accede a la base de datos de parte del cliente. Finalmente, los sistemas de 3 capas proporcionan mejor seguridad, al no exponer el esquema de la base de datos al cliente, y permitiendo un control de accesos de grano más fino a la información en ella almacenada.

En general, la capa intermedia de las aplicaciones de 3 capas no está implementada como un programa monolítico. Está implementada, más bien, como una colección de componentes que se utilizan en una gran variedad de operaciones solicitadas por los clientes. Los componentes se llaman unos a otros para resolver la petición del cliente. Se pasa, entonces, a la arquitectura de **N capas** (*N-tier*).

CAPÍTULO 2 SISTEMAS DE OBJETOS DISTRIBUIDOS

2.1 Introducción

Una vez que se han definido las características de los sistemas distribuidos es necesario plantearse la forma en que se va a afrontar su diseño y construcción. Habitualmente, los sistemas distribuidos se han venido construyendo utilizando conceptos utilizados en los sistemas operativos tradicionales. El concepto de proceso y los mecanismos de comunicación entre procesos son los más destacables.

En los últimos tiempos, el auge de las tecnologías orientadas a objetos ha llevado a los diseñadores a plantearse la forma en que pueden aplicarlas para la construcción de sistemas distribuidos. La aplicación de dichas tecnologías no se ha detenido en la construcción en sí del sistema distribuido, sino en la forma que éste proporciona sus servicios y en la manera en que se construyen las aplicaciones de usuario.

2.2 Arquitectura de sistemas distribuidos

En el capítulo anterior se describieron las características esenciales de todo sistema distribuido. El problema que surge inmediatamente es cómo construir el sistema distribuido para que cubra dichas expectativas. La construcción se deberá realizar basándose en uno de los dos modelos siguientes: el modelo de procesos (o cliente/servidor) o el modelo de objetos.

2.2.1 Modelo de procesos o cliente/servidor

Es el más ampliamente conocido y aceptado para los sistemas distribuidos tradicionales. Existe un conjunto de **procesos servidores**, cada uno de los cuales actúa como **gestor de recursos** para una colección de recursos de un tipo dado y una colección de **procesos clientes**, cada uno de los cuales ejecuta una tarea que requiere acceder a algunos de los recursos hardware y/o software. Incluso los propios gestores pueden necesitar acceder a los recursos compartidos gestionados por otros procesos de tal forma que algunos procesos son clientes y servidores simultáneamente.

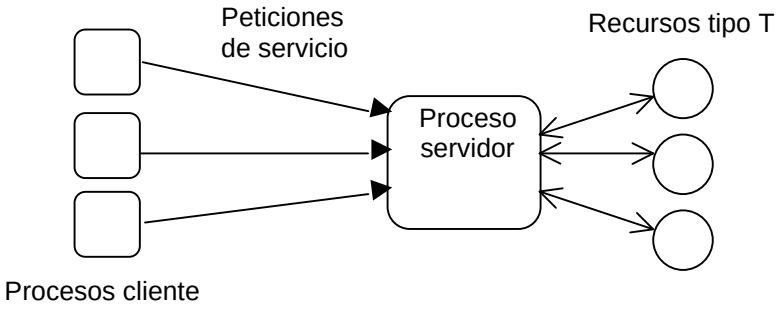


Figura 3.1. Modelo de procesos.

Los servidores se encargan de organizar eficientemente el acceso a los recursos que gestionan. Cada servidor proporciona un conjunto de operaciones, que deberán ser utilizadas por los clientes para actuar sobre los recursos.

Los clientes emiten peticiones a los servidores cuando necesiten acceder a alguno de sus recursos. Si la petición es válida, el servidor efectúa la operación solicitada y envía el resultado de su realización al cliente.

Este modelo proporciona una aproximación de propósito general a la compartición de recursos e información en sistemas distribuidos. Se puede implantar en una amplia variedad de entornos hardware y software. Los computadores usados pueden ser de muchos tipos y tanto clientes como servidores pueden estar ejecutándose en la misma o distintas máquinas.

En esta visión tan simple los procesos servidores podrían verse como suministradores centralizados de los recursos que manejan, hecho no deseable en un sistema distribuido (problema de falta de escalabilidad). Por esta razón, es necesario desligar los conceptos de servicio proporcionado a un cliente y servidor que lo proporciona. Un **servicio** es una entidad abstracta que puede ser proporcionado por varios procesos servidores que se están ejecutando en distintas máquinas y que cooperan vía una red de comunicación.

Sin embargo, por motivos de eficiencia, hay algunos recursos que deben permanecer siempre locales a cada máquina: memoria RAM, procesador central y la interfaz de red local.

Este modelo utiliza, habitualmente, como herramientas para comunicar clientes y servidores **llamadas a procedimientos remotos** (RPC, *Remote Procedure Call*) o ***sockets***.

2.2.2 Modelo basado en objetos

El modelo basado en objetos afronta la construcción del sistema identificando las **abstracciones clave** del dominio del problema. Dichas entidades son vistas como agentes autónomos que colaboran para **llevar a cabo un comportamiento de nivel superior**.

El modelo basado en objetos pone énfasis en la identificación y caracterización de los componentes del sistema, que serán modelados con objetos. Informalmente, se puede definir un **objeto** como una colección formada por una serie de datos y un

conjunto de operaciones definidas sobre ellos, de tal manera que se garantiza la ocultación de los detalles internos.

Para alterar el estado de un objeto, se tendrá que invocar la operación apropiada, de tal forma que el conjunto de las operaciones definidas para un objeto definirá colectivamente su comportamiento. El mecanismo de encapsulación de los objetos se encarga de que esto sea así.

También se pueden utilizar los **mecanismos de herencia y polimorfismo** como herramientas adicionales de estructuración para el diseño y construcción de sistemas distribuidos. La aplicación de conceptos de orientación a objetos tal cual para la construcción de sistemas distribuidos es denominada en [BG96] **aproximación aplicativa**. Un ejemplo clásico del uso de estos mecanismos es el sistema operativo Choices [CIR+93].

El modelo basado en objetos no difiere del tradicional paradigma de OO en el que cada entidad es un objeto con una interfaz de mensajes que proporcionan sus operaciones. En el modelo basado en objetos para los sistemas distribuidos, cada recurso compartido se representa como un objeto. Los objetos son identificados de forma unívoca y se pueden mover por la red sin cambiar su identidad. Cuando un proceso requiere el acceso a un recurso, envía un mensaje al objeto correspondiente. Este mensaje es despachado al método adecuado que resuelve la petición y envía un mensaje de respuesta al proceso adecuado.

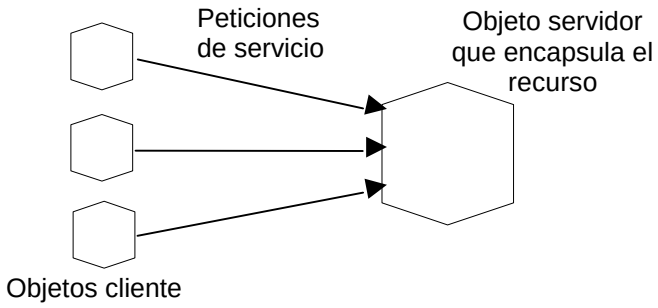


Figura 3.2. Modelo basado en objetos.

El modelo basado en objetos es **simple y flexible**. Ofrece una **visión uniforme** de todos los recursos. Como en los modelos descritos anteriormente, los objetos pueden actuar como clientes y servidores. Sin embargo, en el modelo cliente/servidor, el esquema de nombrado de los recursos dependía del gestor mientras que aquí todos los recursos se referencian de manera uniforme.

2.3 Modelo de objetos

Las características del modelo de objetos a implantar en un sistema de distribución de objetos no deben diferir de las que son consideradas de manera tácita como las más representativas de la OO, y que fundamentalmente se encuentran recogidas en la metodología de Booch [Boo94].

2.3.1 Características del modelo de objetos de Booch

Propiedades fundamentales:

- Abstracción
- Encapsulamiento
- Modularidad
- Jerarquía de herencia y de agregación

Propiedades secundarias:

- Tipos
- Concurrencia
- Persistencia
- Distribución

2.3.1.1 Abstracción y encapsulamiento. Clases

Estas propiedades suelen describirse como un conjunto. La **abstracción** se define en [Boo94] como *"Una abstracción denota las características coincidentes de un objeto*[1] *que lo distinguen de todos los demás tipos de objeto y proporciona así fronteras conceptuales nítidamente definidas respecto a la perspectiva del observador"*. Podría resumirse indicando que un objeto presenta una determinada interfaz que muestra su comportamiento esencial. Este comportamiento se presenta mediante una serie de métodos u operaciones que se pueden invocar sobre el objeto.

Esta propiedad se suele combinar con la de **encapsulamiento**, que establece el principio de ocultación de información: *"el encapsulamiento es el proceso de almacenar en un mismo compartimento los elementos de una abstracción que constituyen su estructura y su comportamiento, sirve para separar la interfaz contractual de una abstracción de su implementación"*. La implementación de un objeto (la implementación de sus métodos y sus estructuras internas o estado interno) debe estar totalmente encapsulada en el mismo y separada de su interfaz. La única manera de utilizar un objeto es mediante los métodos de su interfaz. No se puede acceder directamente a la implementación de un objeto.

También puede considerarse que el objeto encapsula en general toda su semántica, incluyendo las propiedades implícitas que puedan existir en el modelo de objetos, como la concurrencia, que se describen más adelante.

Como unidad de abstracción y representación del encapsulamiento, la mayoría de los sistemas OO utilizan el concepto de **clase**. Todos los objetos que tienen las mismas características se agrupan en una misma clase. La clase describe las características que tendrán todos los objetos de la misma: la representación del estado interno y su interfaz. Así, cuando se crea un nuevo objeto, se crea a partir de una clase, de la que toma su estructura.

[1] Se utilizarán los nombres objeto e instancia indistintamente

2.3.1.2 Modularidad

La **modularidad**[2] permite fragmentar un problema complejo en una serie de conjuntos de objetos o módulos, que interactúan con otros módulos. En cierta manera, representa un nivel superior al nivel de encapsulamiento, aplicado a un conjunto de objetos.

Jerarquía. La relación "es-un" (herencia). La relación "es-parte-de" (agregación)

La **jerarquía** es "una clasificación u ordenación de abstracciones". Permite comprender mejor un problema agrupando de manera jerárquica las abstracciones (objetos) en función de sus propiedades o elementos comunes, descomponiendo en niveles inferiores las propiedades diferentes o componentes más elementales.

Las dos jerarquías más importantes son las representadas por las relaciones "**es-un**" y "**es-parte-de**".

Herencia. La relación "es-un"

Es una relación que se establece entre las diferentes clases de un sistema. Esta relación indica que una clase (**subclase**[3]) comparte la estructura de comportamiento (las propiedades) definidas en otra clase (**superclase**[4]). La clase "hereda[5]" las propiedades de la superclase. Se puede formar de esta manera una jerarquía de clases[6]. Un objeto de una clase también "es-un" objeto de la superclase (por ejemplo, un coche es un vehículo). En el caso en que una clase pueda heredar de más de una superclase se habla de **"herencia múltiple"** por contraposición al caso de **"herencia simple"**.

Agregación. La relación todo/parte ("es-parte-de")

Las jerarquías "es-parte-de" describen relaciones de agregación entre objetos para formar otros objetos de un nivel superior. Permiten definir un objeto en términos de agregación de sus partes, de objetos componentes más elementales (chasis, motor y carrocería son parte de un coche).

2.3.1.3 Tipos y polimorfismo

"Los tipos son la puesta en vigor de la clase de los objetos, de manera que los objetos de tipos distintos no pueden intercambiarse, o, como mucho, pueden intercambiarse sólo de formas muy restringidas". En realidad un **tipo** denota simplemente una estructura común de comportamiento de un grupo de objetos. Normalmente se identifica el concepto de tipo con el de clase[7], haciendo que la clase sea la única manera de definir el comportamiento común de los objetos. De esta manera la jerarquía de tipos se funde con la jerarquía de clases. Así un objeto de un subtipo (subclase) determinado puede utilizarse en cualquier lugar en el que se espera un objeto de sus tipos ancestros (clases ancestras).

---

[2] Un excelente tratamiento sobre la modularidad y la orientación a objetos en general se encuentra en [Mey88, Mey97].
[3] O clase hija.
[4] O clase padre.
[5] O "deriva de".
[6] Las clases situadas por encima de una clase en la línea de herencia son los ancestros, antepasados o ascendientes. Las que derivan de ella son los descendientes.
[7] En este trabajo se asociará siempre el concepto de tipo con el de clase, salvo indicación explícita de lo contrario.

La existencia de tipos permite aplicar las normas de comprobación de tipos a los programas, ayudando a la detección de un número de errores mayor en el desarrollo de un programa. La **comprobación de tipos** permite controlar que las operaciones que se invocan sobre un objeto forman parte de las que realmente tiene. En la comprobación de tipos puede haber varios niveles de comprobación: comprobación estricta en la compilación de un programa (**comprobación estática**), sólo comprobación en tiempo de ejecución (**comprobación dinámica**) o una estrategia híbrida.

Enlace estático y dinámico. Polimorfismo

El enlace hace referencia al momento en el que un nombre (o referencia) se asocia con su tipo. En el **enlace estático** esto se realiza en tiempo de compilación. De esta manera la invocación de una operación sobre un objeto siempre activa la misma implementación de la operación.

En el **enlace dinámico**, el tipo del objeto se determina en tiempo de ejecución y el sistema determinará cuál es la implementación de la operación que se invocará en función de cual sea el objeto utilizado. Esto normalmente se denomina **polimorfismo** puesto que podemos aplicar el mismo nombre de operación a un objeto y en función de cual sea este objeto concreto el sistema elegirá la implementación adecuada de la operación.

Este mecanismo de enlace dinámico permite crear programas generales en los que el tipo de objeto sobre el que operen sea determinado en tiempo de ejecución. Por ejemplo un programa gráfico que simplemente indique a un objeto que realice la operación de dibujar. En tiempo de ejecución se enlaza dinámicamente con la implementación de la operación “dibujar” correspondiente al objeto concreto en tiempo de ejecución que se trate: un círculo, un rectángulo, etc.

El polimorfismo se suele considerar como una propiedad fundamental del modelo de objetos. Combinado con la herencia ofrece un compromiso entre la comprobación de tipos en tiempo de ejecución y la flexibilidad del enlace dinámico. De esta manera se puede usar el polimorfismo en un árbol de herencia compartiendo el mismo nombre de operación para las superclase y sus subclases, pero restringiendo la utilización de las operaciones a las que se pueden aplicar sobre el objeto (las de su árbol de herencia), siempre que pertenezca a una de las clases de la jerarquía.

2.3.1.4 Concurrencia

“La concurrencia es la propiedad que distingue un objeto activo de uno que no está activo”. La mayoría de los lenguajes OO están pensados para sistemas tradicionales con conceptos como fichero y proceso. Esto hace tender a que los objetos se consideren entidades “teledirigidas” por el flujo de ejecución secuencial del programa. Sin embargo, entre las propiedades de un objeto está la de la **concurrencia**, es decir, la posibilidad de tener actividad independientemente del resto de los objetos. Esto permite describir sistemas de una manera más cercana a la realidad, en la que los objetos en muchos casos tienen funcionamiento independiente.

2.3.1.5 Persistencia

“Es la propiedad de un objeto por la que su existencia trasciende el tiempo (es decir el objeto continúa existiendo después de que su creador deja de existir) y/o el espacio (es decir, la posición del objeto varía con respecto al espacio de direcciones en el que fue creado)”.

Para que los sistemas OO sean más cercanos a la realidad, es necesaria la **persistencia**, es decir, un objeto no tiene por qué dejar de existir por el simple hecho de la finalización del programa que lo creó. El objeto representa una abstracción que debe permanecer como tal hasta que ya no sea necesaria su existencia dentro del sistema. Hay que hacer notar que no sólo los datos de un objeto deben persistir, la clase a la que pertenece también debe hacerlo para permitir la utilización del objeto con toda su semántica.

2.3.1.6 Distribución

Aunque no denominada como tal, la propiedad de la **distribución** es referida con el término de **persistencia en el espacio**. Un objeto puede moverse (en el espacio) por los distintos ordenadores que conformen el sistema de computación, manteniendo todas sus propiedades como en el caso de la persistencia. Es decir, los **objetos son distribuidos**.

2.3.2 Características adicionales necesarias

Como se verá más adelante, es necesario **completar el conjunto de características** del modelo de objetos recogidas en el modelo de Booch para un sistema de distribución de objetos: relaciones generales de asociación, identidad única de los objetos, excepciones y seguridad.

2.3.2.1 Relaciones generales de asociación

En un sistema, además de las relaciones “es-un” y “es-parte-de”, existen más relaciones entre objetos. Es conveniente conocer qué objetos están relacionados con otros. Esta información puede ser útil cuando se tienen en cuenta aspectos de distribución de los objetos. Por ejemplo, al mover un objeto a otro computador, es interesante mover también los objetos que están relacionados, puesto que es muy probable que se intercambien muchos mensajes entre sí. Con la relación “es-parte-de” se pueden conocer los objetos integrantes de otro objeto y moverlos a la vez. Sin embargo, en el caso de un libro y la estantería donde está situado, la relación que existe entre ellos no puede modelarse como “es-parte-de”. Si no existe otro tipo de relación en el modelo, el sistema no tendría conocimiento de esta relación y no podría explotarlo al igual que lo puede hacer con la relación “es-parte-de”.

De hecho, relaciones adicionales a las de agregación y herencia son utilizadas por las metodologías. La propia metodología de Booch dispone de relaciones de uso entre objetos.

Como compromiso, se añade un tercer tipo de relación al modelo de objetos: la **relación de asociación** “asociado-con”, que represente en general la existencia de otras relaciones diferentes a las de herencia o agregación entre objetos.

2.3.2.2 Identidad única de objetos

Al igual que en el mundo real, todos los objetos tienen una identidad que los diferencia del resto de los objetos. En un sistema de gestión de objetos tiene que existir un mecanismo para que el sistema distinga un objeto de otros, al objeto de trabajar con el mismo. Al usarse sólo OO lo único que el sistema hará con un objeto es invocar uno de sus métodos. La manera de hacer esto es mediante un identificador que tenga cada objeto.

En lenguajes como C++, se utiliza la posición de memoria que ocupa el objeto para acceder al mismo. Sin embargo, en un sistema global, con necesidades de persistencia,

distribución, etc. esto no es válido pues un objeto puede cambiar de computador. Es necesario utilizar un identificador global que identifique de manera unívoca cada objeto dentro del sistema hasta que el objeto desaparezca. Este identificador único será la única manera de acceder al objeto dentro del sistema. Como se verá, la utilización de un **identificador único** para los objetos facilita la realización de propiedades como la persistencia, distribución, etc. de manera transparente.

2.3.2.3 Excepciones

Las excepciones son un concepto que está orientado fundamentalmente a la realización de programas robustos mediante el manejo estructurado de los errores. Básicamente, una **excepción** es un evento que se produce cuando se cumplen ciertas condiciones (normalmente condiciones de error). En estos casos se "lanza una excepción". La excepción será "atrapada" por un manejador de excepciones que realizará el tratamiento adecuado a la misma.

Las excepciones son un concepto importante con soporte en lenguajes populares como C++ y Eiffel. Parece adecuado, pues, introducirlo en el modelo de objetos del sistema para promover el desarrollo de programas robustos.

2.3.2.4 Seguridad

Aunque no sea de gran importancia en las fases de diseño y análisis de aplicaciones, en un sistema de computación real completo la seguridad es muy importante. Debe existir un mecanismo que permita la protección de los recursos del sistema, en este caso de los objetos.

En el caso de un sistema de distribución de objetos, el objetivo de la protección son los objetos, más concretamente la utilización de los métodos de un objeto por parte de otros objetos. Se trata de posibilitar que sólo ciertos objetos puedan invocar métodos de otros objetos y en el caso de cada objeto qué métodos concretos puede invocar. Por ejemplo, un objeto estudiante podría invocar el método leerNota de un objeto acta, pero no el método ponerNota. Un objeto profesor podría invocar ambas operaciones. Es importante que al igual que en el resto de los conceptos del sistema, este mecanismo sea uniforme e igual para todos los objetos.

2.4 Aplicación de las tecnologías de objetos a la construcción de sistemas distribuidos

En la actualidad, ingeniería del software y metodologías orientadas a objetos son casi sinónimos. Los objetos están presentes en todas las fases del desarrollo de software: análisis, diseño e implementación. Existen gran cantidad de libros y artículos que hablan de los beneficios de la utilización de las tecnologías de objetos en el desarrollo de software, de los que destacan [Boo94, Mey97].

Debido a su tamaño y complejidad, los sistemas operativos han sufrido históricamente de problemas de escalabilidad, mantenibilidad y extensibilidad. Los sistemas distribuidos han venido a exacerbar dichos problemas. La utilización de tecnologías orientadas a objetos se propone como solución a estos problemas de ingeniería [MR93].

Cuando confluyen construcción de sistemas distribuidos y desarrollo de software con tecnologías de objetos, surgen inmediatamente una serie de preguntas ¿Dónde encajan los objetos en la construcción de sistemas distribuidos (entiéndase aquí sistema

distribuido en su sentido más amplio)? ¿Es posible mantener los beneficios de la encapsulación, herencia y polimorfismo en un sistema distribuido construido con tecnologías de objetos?

Básicamente, la aplicación de las tecnologías de objetos a la construcción de sistemas distribuidos se puede realizar a dos niveles distintos [Kra93, CJM+91]: estructurando e implementando el sistema como un conjunto de objetos y proporcionando soporte para los objetos.

Con el objetivo de conseguir los mayores beneficios en la utilización de las tecnologías de objetos será preciso alentar la construcción de sistemas distribuidos aplicando los dos enfoques.

2.4.1 Diseño del sistema distribuido como un marco orientado a objetos

Se trata en este caso de **aplicar los principios de diseño e implantación del paradigma de la orientación a objetos** al propio sistema distribuido, de forma que quede organizado como un marco orientado a objetos. El sistema distribuido se diseña como una jerarquía de clases, cada una de las cuales ofrece determinada funcionalidad.

Los beneficios de esta aproximación son dobles: por un lado, el sistema tiene una estructura modular, en la cual cada uno de sus componentes puede ser identificado y reutilizado; por otro lado, con el uso de mecanismos como la herencia, el sistema puede ser configurado para unas necesidades específicas o para cumplir determinadas restricciones.

Los objetos se presentan como una herramienta muy útil para estructurar los sistemas distribuidos por varias razones [Kra93]:

- Los objetos proporcionan modularidad y encapsulación, dado que separan claramente la interfaz de la implementación.
- La comunicación cuenta con un mecanismo uniforme de alto nivel, la **invocación de métodos** o **paso de mensajes** entre objetos, que es bien conocido y para el que existen técnicas para conseguir implementaciones eficientes.
- Los objetos proporcionan un medio adecuado para la compartición de información. La ventaja respecto a los ficheros compartidos es una semántica de más alto nivel y una mejor protección ante errores de programación.
- Los sistemas estructurados en términos de objetos evolucionan fácilmente (más fáciles de extender y mantener), por reemplazo de sus partes. Además los subsistemas y objetos individuales podrán ser reutilizados con facilidad y adaptados a nuevas condiciones de operación o entorno vía el mecanismo de herencia.

El sistema distribuido se estructura internamente como un conjunto de objetos que implementan la funcionalidad del sistema. Además, en tiempo de ejecución, se instanciarán objetos reales de las distintas clases en que se oferta la funcionalidad del sistema, con la semántica que ello conlleva.

2.4.2 Diseño del sistema distribuido como soporte de objetos

En este caso, el **sistema proporciona soporte para los objetos**, es decir, el objeto es el elemento básico del sistema, en claro contraste con los habituales proceso y fichero. Además, el sistema proporciona un entorno que soporta la definición dinámica de jerarquías de clases y herencia, tanto para los programas de sistemas como para las

aplicaciones. Sus beneficios son, en general (y principalmente para los desarrolladores de aplicaciones), los mismos que los que proporcionan las tecnologías de objetos. El soporte para los objetos se puede dar, a su vez, a diferentes niveles:

- Incluyendo las funciones de gestión de objetos en el propio sistema operativo.
- Proporcionando una capa de gestión de objetos por encima de un sistema operativo existente.
- Implementando la capa de objetos como una capa de soporte en tiempo de ejecución (run-time) para lenguajes de programación específicos (lenguajes orientados a objetos).
- El soporte viene dado por el hardware (o máquina abstracta) subyacente.

2.5 Cuestiones de diseño del soporte de objetos en el sistema operativo distribuido

Una vez decidido que los objetos (por extensión, las tecnologías orientadas a objetos) son adecuados para construir sistemas distribuidos (de nuevo, por extensión, sistemas operativos distribuidos) y que es necesario que estos proporcionen soporte para los objetos, el diseñador debe tomar en consideración la **estructura de los objetos** que van a ser soportados por el sistema distribuido: actividad interna, granularidad, visibilidad, volatilidad/persistencia, movilidad, replicación y composición.

Es necesario, en este momento, integrar conceptos como la relación entre procesos y objetos, el soporte de réplicas por parte del sistema, etc. Se trata, en definitiva, de unificar conceptos de paralelismo y distribución en general con conceptos de orientación a objetos. Dicha unificación recibe en [BG96] el nombre de **aproximación integradora**.

En el capítulo 10 se identifican y describen las opciones adoptadas para cada una de las cuestiones de diseño presentadas a continuación.

2.5.1 Actividad interna

La relación entre los procesos y los objetos de un sistema de distribución de objetos caracteriza la actividad interna de los objetos. Los procesos pueden estar separados y relacionados de manera temporal con los objetos que invocan, o bien, pueden estar asociados de manera permanente a los objetos en los que se ejecutan. Estas dos aproximaciones se corresponden, respectivamente, con el **modelo de objetos pasivo** y el **modelo de objetos activo**.

2.5.1.1 Modelo de objetos pasivo

El modelo de objetos pasivo se basa en ofrecer una clase de objetos especiales, representativa de la computación, equivalente al concepto de proceso en los sistemas operativos tradicionales. Los demás objetos son meros contenedores de datos y los métodos que los manipulan.

Un proceso no está restringido a un solo objeto, sino que puede ejecutarse en varios objetos durante su existencia para satisfacer una acción. Cuando un proceso realiza una invocación en otro objeto, su ejecución en el objeto actual se suspende. Conceptualmente, el proceso se asocia al espacio de direcciones del segundo objeto, donde ejecuta la operación apropiada. Cuando se completa dicha operación, el proceso vuelve al primer objeto, donde reanuda su ejecución de la operación original.

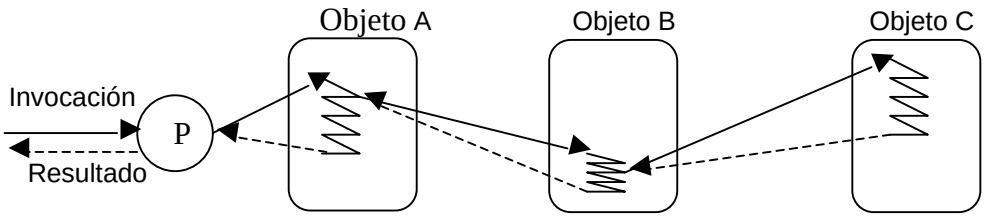


Figura 3.3. Realización de una acción en el modelo de objetos pasivo.

Una ventaja de la utilización del modelo de objetos pasivos es que virtualmente no hay restricción en el número de procesos asociados a un objeto. Como inconvenientes están las dificultades y costes de asociar y desasociar procesos a los espacios de direcciones de los objetos en los que se ejecutan.

Adicionalmente, el modelo de objetos pasivo presenta una gran dificultad para la distribución de los objetos. Sería necesario transformar los procesos en procesos distribuidos, construidos en base a estructuras de computación distribuidas, de tal forma que en todo momento sea factible conocer la secuencia de ejecución de métodos, tanto en una máquina como en varias.

2.5.1.2 Modelo de objetos activo

El modelo de objetos activo considera los objetos como entidades autónomas, que encapsulan datos, métodos y computación. De alguna manera, es como si los procesos formasen parte de los objetos. Con esta aproximación, múltiples actividades independientes pueden estar involucradas en la realización de una acción. La interacción entre objetos se realiza mediante **paso de mensajes** que contienen todos los datos necesarios para la invocación.

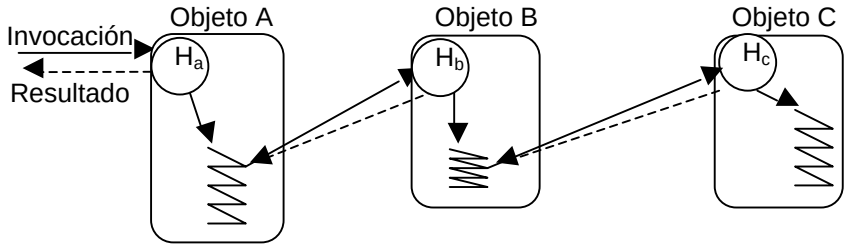


Figura 3.4. Realización de una acción en el modelo de objetos activo.

El modelo de objetos activo permite que el sistema de gestión de objetos ofrezca una sola entidad (el objeto) con un contenido semántico superior al modelo pasivo. De esta forma, ante la realización de invocaciones concurrentes a sus métodos, un objeto puede decidir aceptar o retrasar la ejecución de las mismas de acuerdo a su estado interno.

Entre otras cosas, el número máximo de procesos que un objeto puede albergar deberá ser tenido en cuenta para cada invocación que llega, de manera que todas las invocaciones a un objeto en el que ya existe el máximo de procesos deberán ser rechazadas o encoladas para un tratamiento posterior.

El principal inconveniente que se le achaca a este modelo es la eficiencia, debido a la sobrecarga que produce el incluir en la representación en tiempo de ejecución del objeto las estructuras que permiten asociar a cada objeto su computación.

Las ventajas que proporciona, son varias. En primer lugar, ofrece una única abstracción, el objeto, para modelar los sistemas. Estos objetos son autocontenidos, es decir, incluyen datos, métodos y computación y, por tanto, son quienes la gestionan. Además, dado que los objetos realizan sus acciones ante la llegada de mensajes, el propio mecanismo de paso de mensajes sirve para activar métodos y sincronizar ejecuciones.

El modelo de objetos activo es muy adecuado para la distribución de objetos. Al proporcionar una visión única del objeto, la computación se hace transparente al mecanismo de la distribución. Así, la migración de un objeto comprende también la migración de su computación. Por su parte, el mecanismo de paso de mensajes tiene que ser incrementado para que estos puedan enviarse a través de una red.

2.5.2 Granularidad de los objetos

El tamaño, sobrecarga y cantidad de procesamiento realizada por un objeto caracterizan su **granularidad**. Este aspecto es muy importante, dado que está relacionado directamente con la usabilidad, rendimiento y homogeneidad en la utilización de los objetos como tecnología de construcción de sistemas distribuidos. Se distinguen tres grados en la granularidad de los objetos: grano grueso, grano medio y grano fino.

2.5.2.1 Objetos de grano grueso

Los objetos de **grano grueso** se caracterizan por un gran tamaño, un número relativamente grande de instrucciones a ejecutar para servir una invocación y un número relativamente reducido de interacciones con otros objetos. Este tipo de objetos se asocia inmediatamente con los procesos servidores de recursos del modelo cliente/servidor tradicional (ver apartado 3.2.1 de este mismo capítulo).

La utilización de un esquema puro de objetos de grano grueso ofrece como ventaja la simplicidad, pero existen ciertos inconvenientes cuando se desea implantar en un sistema de distribución de objetos: en primer lugar, el control y la protección de los datos están al nivel de los objetos de grano grueso, lo cual restringe la flexibilidad del sistema y la cantidad de concurrencia entre objetos que puede proporcionar. En segundo lugar, el sistema no va a poder proporcionar un modelo de datos homogéneo. Las grandes entidades de datos se representarán con objetos, pero las pequeñas entidades, como listas enlazadas o enteros, se tendrán que representar con abstracciones de datos convencionales de lenguajes de programación.

2.5.2.2 Objetos de grano medio

Para proporcionar un nivel de control más fino sobre los datos, el sistema puede soportar tanto objetos de grano grueso como objetos de **grano medio**. Los objetos de grano medio son más fáciles (baratos) de crear y mantener, dado que son más pequeños en tamaño y menos ambiciosos en cuanto a su funcionalidad, que los objetos de grano grueso. Ejemplos típicos de estos objetos son estructuras de datos como una lista enlazada y una cola. Como inconvenientes de los objetos de grano medio están la sobrecarga adicional derivada del mayor número de objetos a ser gestionados por el sistema y que su existencia no proporciona todavía un modelo de datos homogéneo (entidades como los enteros siguen siendo representados con abstracciones de los lenguajes de programación).

2.5.2.3 Objetos de grano fino

Finalmente, para proporcionar un nivel aún más fino de control sobre los datos, el sistema puede soportar objetos de grano grueso, de grano medio y de **grano fino**. Los objetos de grano fino se caracterizan por su pequeño tamaño, un reducido número de instrucciones a ejecutar y un gran número de interacciones con otros objetos. Ejemplos de este tipo de objetos se encuentran en tipos de datos proporcionados por los lenguajes de programación, como los enteros, reales, lógicos, etc. El mayor problema de los sistemas que utilizan objetos de grano fino es la pérdida de rendimiento derivada de la gestión de un gran número de objetos. Sin embargo, esta aproximación proporciona un entorno homogéneo, donde toda entidad es un objeto, independientemente de su tamaño.

La experiencia ha mostrado que los programadores tienden a utilizar objetos pequeños (de unos cientos de bytes), mientras que los mecanismos básicos del sistema (comunicación, compartición, etc.) son más eficientes para objetos de grano grueso [Kra93]. Esto último suele ser así en diferentes sistemas distribuidos que proporcionan soporte de objetos (ver capítulo 4), en los que estos juegan un papel similar al de los procesos en los sistemas operativos tradicionales.

2.5.3 Visibilidad: objetos privados/objetos compartidos

Los **objetos privados** son visibles únicamente en el ámbito de los objetos que los crean. Juegan habitualmente el papel de atributos agregados de otros objetos o bien existen de manera transitoria como objetos locales de un método.

Los **objetos compartidos** ofrecen un medio de comunicación uniforme entre objetos, aunque implican la necesidad de mecanismos de protección y sincronización, dado que van a ser invocados por varios objetos incluso de manera simultánea. Juegan el papel de los procesos servidores del modelo de procesos descrito al principio del capítulo.

Habitualmente coexistirán objetos privados y compartidos.

2.5.4 Objetos volátiles/objetos persistentes

Los **objetos volátiles** existen durante el tiempo que el sistema esté funcionando. Juegan el papel de las variables de los sistemas tradicionales.

Los sistemas tradicionales adolecen del problema de utilizar abstracciones distintas en función del tiempo que se desee que perduren los datos que manejan: para los datos en memoria volátil se utilizan variables, de distintos tipos, y para los datos en memoria persistente, se utilizan los ficheros.

Los **objetos persistentes** unifican ambas visiones, ofreciendo una abstracción única e independiente de la localización instantánea de un objeto. Como inconveniente de la introducción de objetos persistentes distribuidos destaca la dificultad a la que pueden llevar la tarea de la recolección de basura.

A la vez que un objeto persistente reside en memoria secundaria puede existir alguna copia volátil del mismo en memoria principal. Las operaciones sobre el objeto modificarán la copia volátil, hasta que se tome la decisión de actualizar la copia persistente. En el caso de que se realizase una invocación a un objeto que únicamente dispusiese de la copia persistente, se crearía automáticamente una copia volátil para atender la invocación. El sistema de persistencia decidirá cuándo prescindir de copias

volátiles de objetos que se encuentran actualizados en el almacenamiento de persistencia.

2.5.5 Movilidad: objetos fijos/objetos móviles

La migración de procesos en sistemas distribuidos tradicionales es muy útil para mejorar el reparto de carga de los diferentes computadores con el fin de mejorar el rendimiento global y para ciertas restricciones de administración o seguridad.

Si se utiliza como unidad de migración el objeto, se obtienen inmediatamente **objetos móviles**. En contraposición, los **objetos fijos** no se moverán del nodo en el que fueron creados durante toda su existencia, pudiendo, en principio, comunicarse igualmente con cualquier otro objeto del sistema distribuido.

Los objetos móviles migrarán entre los diferentes nodos del sistema distribuido por razones similares a las que se tomaban para la migración de procesos en sistemas distribuidos tradicionales [Gos91]. También se tendrán que tomar decisiones sobre los objetos susceptibles de ser migrados, si la migración se plantea individual o colectivamente, etc.

El inconveniente principal de los objetos móviles es que hacen la localización de objetos más compleja.

2.5.6 Replicación: objetos de una copia/ objetos replicados

Un esquema de **objetos de una copia** proporciona una identidad única y diferente a todos los objetos del sistema distribuido. La no-disponibilidad de un nodo del sistema distribuido podría hace inaccesibles todos los objetos de este tipo que residían en él en el momento en que se produjo el problema.

Un esquema de **replicación de objetos** permite la existencia de múltiples copias de un objeto en diferentes nodos del sistema distribuido, de manera que es tolerable el fallo de algún nodo mientras todavía se proporciona una funcionalidad casi total. El fallo de un nodo cualquiera solo resultaría en la no-disponibilidad de las réplicas que residen en el mismo, con el problema adicional de la merma en el rendimiento del sistema.

Como inconvenientes de la replicación de objetos destacan la dificultad del mantenimiento de información consistente entre réplicas del mismo objeto y la sincronización de actividades de múltiples clientes.

La forma más sencilla de introducir algún aspecto de replicación de objetos en un sistema distribuido es aplicándola únicamente a **objetos inmutables**. Los objetos inmutables son aquellos que no pueden ser modificados una vez que han sido creados. La replicación de este tipo de objetos se simplifica notablemente, dado que no es necesario mantener un estado consistente ni sincronizar las operaciones de acceso al mismo. Como inconveniente aparece su uso limitado, dado que sólo es aplicable a objetos no modificables.

En el caso de que se opte por un esquema de objetos replicados, el modelo de objetos implantado por el sistema distribuido debería ocultar este hecho tras una interfaz uniforme, de tal forma que la réplica concreta que realmente vaya a servir una petición de un cliente sea irrelevante para éste.

2.5.7 ¿Qué es, entonces, un objeto distribuido?

Como se ha comentado anteriormente, un objeto representa una unidad independiente de ejecución que encapsula datos, procedimientos y, posiblemente, recursos privados (actividad) para el procesamiento de las peticiones. Por lo tanto, una opción natural es considerar el **objeto como unidad de distribución y posible replicación**.

Adicionalmente, la autocontención de los objetos (datos + métodos + posible actividad interna) facilita la posibilidad de su migración. Además, el paso de mensajes no solo asegura la separación entre los servicios ofertados por un objeto y su representación interna, sino que también proporciona independencia de su localización física. Por tanto, el paso de mensajes aglutinará tanto la invocación local como remota (los objetos emisor y receptor residen en el mismo o diferentes nodos), así como la posible no disponibilidad de un objeto a pesar de existir.

Para considerar a un **objeto distribuido** como tal, el sistema operativo tiene que dotarlo de las propiedades siguientes:

- Que pueda ser **invocado tanto local como remotamente**. El objeto podrá recibir invocaciones por parte de otros objetos con independencia de sus respectivas ubicaciones dentro del sistema distribuido.
- Que pueda **migrar** entre diferentes nodos del sistema distribuido. De acuerdo con las políticas de migración de objetos que se establezcan, el objeto puede modificar su ubicación en el sistema distribuido sin que ello afecte a la computación estuviese en curso relativa al objeto que.

2.6 Problemas de la gestión de objetos

La gestión de los objetos de un sistema distribuido introduce un conjunto de problemas nuevos y matiza alguno de los ya existentes en el diseño y construcción de sistemas distribuidos. A continuación se describen los más relevantes.

2.6.1 Nombrado

En un sistema distribuido, los **nombres** se utilizan para hacer referencia a los diferentes recursos, como servicios, usuarios, estaciones de trabajo, etc. En un sistema distribuido orientado a objetos la única entidad que existe es el objeto, y serán los objetos los referenciados mediante los nombres.

El **nombrado de los objetos** es una función básica de todo sistema de gestión de objetos, y puede ser llevada a cabo a diferentes niveles. Nos encontramos habitualmente con nombres simbólicos, de alto nivel, y nombres internos, de bajo nivel.

Los **nombres internos**, también denominados **identificadores**, se eligen de manera que las operaciones que los manipulan se puedan realizar de manera eficiente. En algunos casos pueden venir acompañados de información de seguridad, que incluye derechos para realizar operaciones sobre objetos y que será difícil de falsificar. En otros casos, pueden venir acompañados de información que proporcione una idea de la ubicación actual del objeto. Los nombres internos pueden ser, a su vez, locales (relativos a un contexto dado) o universales, que son únicos en el espacio y el tiempo. Todo objeto tiene siempre un nombre interno.

Los **nombres simbólicos** son utilizados para hacer referencia a los objetos de manera que sean sencillos de reconocer. Como ejemplo típico de nombres simbólicos

están los nombres de ficheros que utilizan los usuarios de los sistemas operativos convencionales. Todo objeto puede tener cero o más nombres simbólicos asociados.

Los nombres simbólicos se asocian con los nombres internos utilizando lo que se denomina **servicio de nombrado**, que almacena una base de datos de relaciones entre ambos y que puede ser consultada y actualizada ante peticiones de los usuarios.

2.6.2 Localización

El problema de la localización de objetos consiste en **determinar la ubicación** actual (nodo o procesador) de un objeto a partir de su nombre. Si el nombre del objeto es puro, es decir, no contiene información de localización, el problema de la localización de un objeto de manera eficiente en un sistema grande es de gran dificultad. La difusión (*broadcast*) de mensajes para la localización debería ser utilizada como último recurso, dado que supone un gran coste en un sistema distribuido de gran tamaño. Existen técnicas que se basan en la localidad de acceso a los objetos (las referencias a objetos tienden a agruparse) y su permanencia (los objetos tienden a moverse con poca frecuencia) para optimizar la operación de localización.

2.6.3 Acceso

El acceso a un objeto (invocación de uno de sus métodos) involucra la siguiente secuencia de operaciones: localizar el objeto; comprobar la conformidad de los tipos (si no se hizo en tiempo de compilación); determinar el método a invocar (si es posible la invocación dinámica); finalmente, ejecutar la llamada. La ejecución del método puede terminar con éxito, devolviéndose un resultado, o bien fracasar, en cuyo caso sería importante que el sistema dispusiese de un modelo de excepciones.

2.6.4 Compartición y protección

La compartición de objetos se presenta como un modelo natural de comunicación entre objetos. La compartición secuencial, es decir, no concurrente, puede ser implementada con objetos persistentes. La compartición concurrente debe involucrar mecanismos para el control de la concurrencia, que podrían llegar al nivel de complejidad de las transacciones [HR83].

La protección debe ser adaptada al modo en que los objetos son utilizados, es decir, se debe definir en términos de los métodos que se permite invocar en lugar de los derechos de acceso tradicionales basados en permisos de lectura/escritura.

2.6.5 Persistencia

Uno de los problemas principales a resolver por el subsistema de persistencia es el de la diferencia de las referencias a objetos volátiles y persistentes, conocido como transformación de punteros (*pointer swizzling*), que se habrá de solucionar ofreciendo una abstracción única de referencia a objeto, independiente de su estado.

Otra cuestión importante relacionada con la persistencia es la recolección de basura. Se tiene que llegar a una solución de compromiso entre eficiencia y seguridad, lo que lleva a utilizar métodos pesimistas: algo de “basura” puede llegar a no ser eliminada nunca.

2.7 Características de un sistema operativo distribuido orientado a objetos

En la bibliografía tradicional de sistemas operativos es difícil encontrar una definición como tal de sistema operativo. Los autores definen, en general, los sistemas operativos mostrando una relación de las funciones que debe realizar. Intentar dar aquí una definición de lo que es un sistema operativo distribuido orientado a objetos sería muy osado, por lo que se va a seguir la pauta marcada por otros autores. La definición vendrá dada como un compendio de las características y funciones que vamos a exigir a todo aquel software que quiera ser catalogado como un sistema operativo distribuido orientado a objetos.

Un **sistema operativo distribuido orientado a objetos** será aquel que, además de cumplir con la definición que para un sistema operativo distribuido aparece en el capítulo anterior, tiene las características siguientes:

- Está construido como un conjunto de objetos, o lo que es lo mismo, el sistema operativo está diseñado como un marco orientado a objetos, en el que cada uno de los objetos proporciona una funcionalidad específica del mismo.
- Proporciona soporte para objetos de nivel usuario. Los objetos no se circunscriben únicamente al diseño y la construcción del sistema operativo, sino que se convierten en el elemento fundamental para la construcción de aplicaciones. La unión de esta característica y la anterior concluyen que todo en el sistema operativo es un objeto.
- Proporciona un mecanismo uniforme para la comunicación entre objetos de aplicación y objetos del sistema y entre los propios objetos de aplicación, que se concreta en la invocación de métodos. La tradicional solicitud de servicios al sistema operativo se realiza por invocación de métodos de objetos que proporcionan su funcionalidad. De la misma manera, la interacción entre las aplicaciones se concreta en invocaciones de métodos.
- La invocación de métodos debe ser proporcionada de manera transparente e independiente de la ubicación de los objetos (local o remota), el estado de los objetos (volátil o persistente) y la función del objeto (sistema o aplicación). La transparencia en todos estos aspectos es fundamental, dado que se consigue el resultado de que el programador únicamente se tiene que preocupar del dominio del problema a resolver. En tiempo de ejecución, el sistema operativo se encargará de solventar todos los problemas que surgen debidos a las diferentes situaciones en que se pueden encontrar los objetos, sin necesidad de que el programador haya tenido que escribir código para su tratamiento.
- Proporciona mecanismos para la migración de objetos entre los distintos nodos del sistema distribuido. Todo sistema operativo debe hacer un uso racional y eficiente de los recursos. Las unidades de cómputo (procesadores o computadores) son unos de los recursos más críticos, de tal forma que no es admisible tener algunas ociosas mientras otras soportan una gran cantidad de tareas. El sistema operativo deberá equilibrar la carga de las diferentes unidades de cómputo con el fin de maximizar el rendimiento global. Así mismo, las operaciones a realizar que requieren el envío de mensajes a través de la red sufren de un considerable retraso con respecto a las que se pueden resolver dentro de una misma unidad de cómputo. Es importante, entonces, que todas

aquellas entidades involucradas en la realización de una operación estén lo más próximas entre sí que sea posible. En un sistema operativo distribuido orientado a objetos, la computación a realizar está asociada a los propios objetos, de tal manera que mover carga computacional de una unidad de cómputo a otra se traduce en mover (migrar) objetos.

CAPÍTULO 3 PANORÁMICA DE SISTEMAS DISTRIBUIDOS ORIENTADOS A OBJETOS

3.1 Introducción

En este capítulo se revisan diferentes sistemas operativos y "*middleware*" que comparten algunas de las características deseables para un sistema distribuido orientado a objetos. Se trata de detectar características comunes y estrategias que sean de utilidad para el diseño de una arquitectura para el sistema de distribución de objetos de un sistema operativo.

Los sistemas revisados intentan ser representativos de las diferentes tendencias actuales y son una selección de los sistemas examinados. Los aspectos relevantes de estos y otros sistemas para apartados específicos del sistema de distribución se examinarán posteriormente al tratar estos apartados.

3.2 DCE y DC++

DCE (*Distributed Computing Environment*) de *Open Software Foundation* [OSF92], ahora denominada *Open Group*, es un conjunto integrado de herramientas y servicios que soportan el desarrollo de aplicaciones distribuidas, de entre los que destacan las **llamadas a procedimientos remotos** (RPC, *Remote Procedure Call*), el **servicio de directorio de celda** (CDS, *Cell Directory Service*), los **servicios globales de directorio** (GDS, *Global Directory Services*), el **servicio de seguridad**, los **hilos** DCE, y el **servicio de ficheros distribuidos** (DFS, *Distributed File Service*).

DCE es una tecnología de tipo "*middleware*" o "habilitadora" que no tiene sentido de existir por sí misma, sino más bien como un extra de un sistema operativo. Funciona en diferentes tipos de computadores, sistemas operativos y redes, facilitando la portabilidad del software, al ocultar las particularidades del entorno en que se ejecuta.

3.2.1 Modelo de programación

El modelo de programación de DCE es el **modelo cliente/servidor**. Las dos facilidades que ofrece DCE y que no pueden considerarse como servicios son las llamadas a procedimientos remotos (RPC) y los hilos.

3.2.2 RPC

La **llamada a procedimiento remoto** (RPC, *Remote Procedure Call*) es la facilidad que hace posible a un programa cliente acceder a un servicio remoto invocando simplemente un procedimiento local. Es responsabilidad del sistema de RPC ocultar todos los detalles a los clientes y servidores: localizar el servidor correcto, construir y transportar los mensajes en ambas direcciones y realizar todas las conversiones de tipos necesarias entre el cliente y servidor, salvando las posibles diferentes arquitecturas en las que se ejecutan ambos.

Para que un servidor especifique los servicios que oferta a sus clientes, debe construir una interfaz con el **lenguaje de definición de interfaces** (IDL, *Interface Definition Language*). Toda interfaz tiene asignado un **identificador único** (UUID, *Universal Unique Identifier*), que se obtiene realizando una llamada al programa (suministrado como herramienta) *uuidgen*. La unicidad de este valor está asegurada dado que dicho identificador incluye el instante de creación y la ubicación (el computador) en la que se realizó. El compilador de IDL genera sustitutos (*stubs*) en C a partir de las interfaces, para ser ensamblados con las aplicaciones cliente y servidor.

La característica más potente de la RPC es que puede ser integrada con los servicios de seguridad y nombrado de DCE. Esta integración hace posible autenticar toda llamada a procedimiento y localizar los servidores dinámicamente en tiempo de ejecución.

3.2.3 Hilos

Los servidores pueden servir RPC concurrentes utilizando varios **hilos**. De la misma manera, un cliente puede necesitar varios hilos para atender una interfaz de usuario a la vez que realizan una RPC de un servidor.

3.2.4 Formato de los datos

La representación de datos en la red (NDR, *Network Data Representation*) de DCE especifica un estándar de formato de datos independiente de la arquitectura, con el fin de facilitar la transferencia de datos entre arquitecturas heterogéneas. Este esquema de codificación/decodificación aísla el código de las aplicaciones de las diferencias en los tipos de datos, facilitando la portabilidad e interoperabilidad de las aplicaciones.

3.2.5 DC++

DC++ [SM93] es un **entorno distribuido orientado a objetos** construido sobre DCE. En oposición a la visión procedimental de DCE (llamada a procedimiento remoto), soporta un modelo de objetos uniforme, invocación de objetos de grano fino independiente de la localización, paso de parámetros usando referencias remotas, migración dinámica de objetos e integración con el lenguaje C++. Los servicios fundamentales de DCE que utiliza DC++ son los hilos, las RPC y el CDS.

Los **objetos distribuidos** pueden ubicarse en diferentes nodos del sistema distribuido y manipulan referencias locales y remotas.

Las referencias remotas se implementan con una indirección proporcionada por un **representante** (*proxy*). Un representante contiene una pista de la localización del objeto referenciado y le reenvía las invocaciones de manera transparente utilizando RPC. Se instalará un representante en todo nodo que conozca la existencia de un objeto remoto. Este ocurre cuando se pasa una referencia a un objeto remoto como parámetro de una invocación. También ocurre cuando un objeto migra y tiene referencias a objetos remotos: en el nodo destino se tendrán que instalar representantes para todas las referencias.

Cada nodo mantiene una tabla de correspondencias entre **identificadores globales de objetos** (se utilizan los UUID de DCE) que vienen en las invocaciones entrantes y direcciones de almacenamiento reales de objetos C++.

Forman también parte del entorno uno o más servidores CDS, encargados de almacenar representantes de objetos que tienen registrado un nombre simbólico. De esta

manera, un objeto puede adquirir un representante de un objeto remoto proporcionando su nombre a un CDS.

La migración de objetos se solicita invocando un método generado automáticamente para todos ellos. Una vez movido el objeto, en el nodo original se dejará un representante del mismo. Como prerrequisito para la migración de un objeto, se asume que la clase a la que pertenece el objeto está disponible en el nodo destino basándose en un servicio de replicación de clases.

3.2.6 Crítica

Uso de paradigma procedimental

A pesar de que el uso de DCE está muy extendido, su enfoque procedimental excluye un funcionamiento orientado a objetos. Incluso aunque cliente y servidor estén construidos como conjuntos de objetos, su interacción pasa necesariamente por las llamadas a procedimientos.

De alguna manera, DC++ viene a solucionar dicho problema, proporcionando un entorno orientado a objetos. Además, aunque DC++ proporciona un entorno orientados a objetos, su construcción está basada en el carácter procedimental de DCE.

Descripción de servicios con IDL

Los servicios que proporcionan los servidores, en DCE, y los objetos distribuidos, en DC++, se tienen que describir utilizando el lenguaje de definición de interfaces IDL. El programador tiene, por tanto, que realizar una descripción de los servicios en un lenguaje (IDL) y la implantación de los mismos en otro (C++, en el caso de DC++).

Programación únicamente en C++

La construcción de aplicaciones, en el caso de DC++, está restringida al uso del lenguaje C++.

Capa de software

Cualquiera de las dos variantes (DCE y DC++) no deja de ser una capa de software o *middleware* que se ejecuta sobre un sistema operativo tradicional.

3.2.7 Características interesantes

Generación de identificadores únicos

Los identificadores globales y únicos de objetos permite establecer una correspondencia biunívoca entre ambos, de tal manera que los identificadores no pueden repetirse en el espacio (diferentes nodos) ni en el tiempo.

Pistas de localización en representantes en DC++

La utilización de pistas de localización almacenadas en los representantes de los objetos permite evitar búsquedas de objetos por difusión (*broadcast*) en muchos casos. Dado que se supone que los objetos se mueven con menos frecuencia que con la que son accedidos, en la mayor parte de los casos la pista de localización va a ser una indicación real de la ubicación del objeto.

Invocación independiente de la localización

Los objetos se comunican de manera uniforme a través de la invocación de métodos, con independencia de su ubicación. La tarea de localización de los objetos es realizada de manera transparente por DC++.

Migración de objetos de grano fino

Cualquier objeto, con independencia de su granularidad, puede migrar. La migración de objetos se realiza de manera transparente, de modo que el objeto puede reanudar su actividad en el momento en que es activado en el nodo destino.

3.3 DCOM

COM (*Component Object Model*) [Mic95] es la tecnología de definición y manipulación de componentes de Microsoft que proporciona un modelo de programación y un estándar binario para los mismos. DCOM [Mic98] (de *Distributed COM*) es la tecnología que extiende COM para permitir a los objetos componentes residir en máquinas remotas y está disponible desde la aparición de Windows NT 4.0. A partir de ahora se utilizarán los términos COM y DCOM indistintamente.

3.3.1 Modelo de objetos

Un **objeto COM** se define en términos de las interfaces individuales que soporta (una o más) y que están definidas en la clase (objeto clase) a la que pertenece. Cada interfaz está identificada por un **identificador único** (IID, *Interface Identifier*), que es un caso particular de **identificador global y único** (GUID, *Global Unique Identifier*). Los GUID son valores de 128 bits que se garantizan únicos estadísticamente en el espacio y en el tiempo.

Las interfaces son el único medio de interactuar con un objeto COM. Un cliente que desea utilizar los servicios de un objeto habrá de obtener primero un puntero a una de sus interfaces.

Los **objetos clase** implementan una o más interfaces y se identifican por un **identificador de clase** (CLSID, *Class Identifier*). Los CLSID son también un caso particular de GUID. Con el fin de que un cliente pueda crear un objeto COM, es necesario describir su clase utilizando el **lenguaje de definición de interfaces** (IDL, *Interface Definition Language*). La compilación de dicha descripción genera un representante (*proxy*) para los clientes y un sustituto (*stub*) para el servidor.

COM no proporciona la herencia como instrumento para lograr la reutilización. En su lugar proporciona los **mecanismos de contención y agregación**. El polimorfismo es conseguido cuando diferentes clases soportan la misma interfaz, permitiendo a una aplicación utilizar el mismo código para comunicarse con cualquiera de ellas.

3.3.2 Interoperabilidad entre objetos COM

El objetivo principal de COM es proporcionar un medio por el que los clientes pueden hacer uso de los objetos servidores, sin tener en cuenta que pueden haber sido desarrollados por diferentes compañías utilizando diferentes lenguajes de programación. Con el fin de lograr este nivel de interoperabilidad, COM define un **estándar binario**, que especifica cómo se dispone un objeto en memoria principal en tiempo de ejecución. Cualquier lenguaje que pueda reproducir dicha disposición en memoria podrá crear objetos COM.

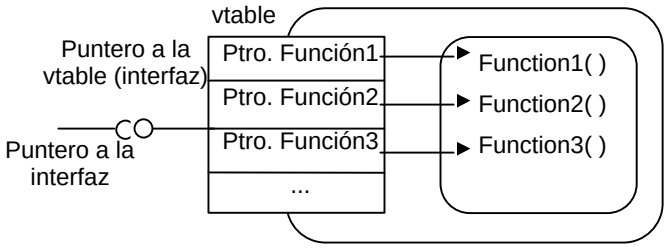


Figura 4.1. Disposición de un objeto COM en memoria.

Además del objetivo de la interoperabilidad, COM tiene otros objetivos:

- Proporcionar una solución para los problemas de versiones y evolución.
- Proporcionar una visión del sistema de los objetos.
- Proporcionar un modelo de programación singular.
- Proporcionar soporte para capacidades de distribución.

3.3.3 Modelo de programación

En el modelo de programación COM, los clientes COM se conectan a uno o más objetos COM. Cada objeto COM expone sus servicios a través de una o más interfaces, que no son más que agrupaciones de funciones relacionadas semánticamente.

La implementación compilada de cada objeto COM está contenida en un módulo binario (`exe` o `dll`) denominado **servidor COM**. Un único servidor COM es capaz de contener la implementación compilada de varios objetos COM.

Un servidor COM puede estar enlazado al proceso cliente (*in-process server*), puede ejecutarse en un proceso distinto del cliente pero en la misma máquina (*local server*) o bien, puede ejecutarse en un proceso separado en una máquina distinta, incluso en un sistema operativo distinto (*remote server*). Para la comunicación con objetos situados en espacios de direcciones distintos del espacio de direcciones del cliente, se utilizan intermediarios en la forma de representantes y sustitutos.

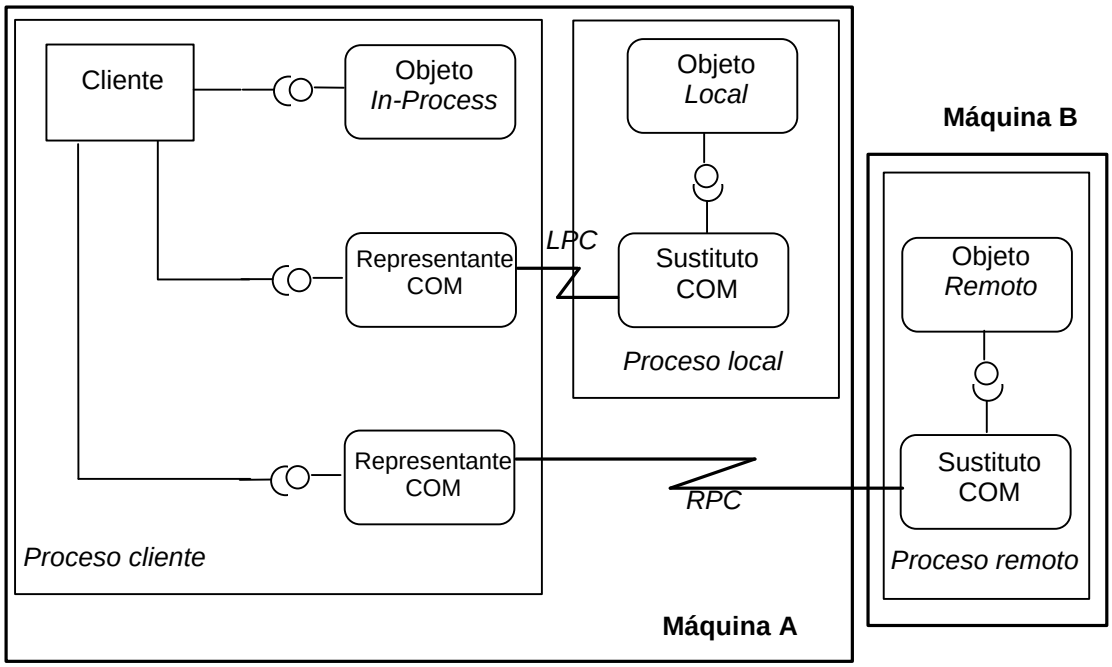


Figura 4.2. Diferentes disposiciones de un objeto COM respecto a un cliente.

El modelo de programación COM define que un servidor COM debe exponer objetos COM, un objeto COM debe exponer sus servicios y un cliente COM debe usar los servicios de objetos COM.

Para comunicarse con un objeto que no es local, COM emplea un mecanismo de comunicación entre procesos que es transparente a la aplicación, incluso en lo relativo a la localización del objeto.

3.3.4 Ciclo de vida

Todos los objetos COM tienen que implementar una interfaz particular, denominada **IUnknown**. Esta interfaz es el corazón de COM y es utilizada para negociación de interfaces en tiempo de ejecución (preguntar al objeto qué interfaces soporta y obtener punteros a las mismas), gestión del ciclo de vida del objeto y agregación.

Cada objeto mantiene una **cuenta de referencia** que es incrementada cada vez que un cliente solicita un puntero para una interfaz o pasa una referencia al mismo. Cuando la interfaz deja de ser utilizada por el cliente, la cuenta de referencia se decrementa. Las operaciones de incremento y decremento son realizadas de manera explícita por el cliente invocando funciones pertenecientes a la interfaz *IUnknown*.

3.3.5 COM+

COM+ [Kir97] es la nueva generación de COM y se presenta como la base del nuevo sistema operativo de Microsoft, Windows 2000.

COM+ viene a ocultar la mayor parte de las tareas que en COM resultaban tediosas y dificultosas para los programadores, como el control del ciclo de vida, la negociación de interfaces, etc. A pesar de todo, el modelo básico de objetos sigue siendo el de COM.

El conjunto de servicios que introduce COM+ está orientado a la construcción de aplicaciones empresariales, de tal forma que los programadores se concentran en la

escritura de la lógica de negocio y no tienen que perder tiempo escribiendo infraestructura u otros servicios.

Las características principales que se pueden encontrar en COM+ son: servidores, transacciones, seguridad, administración, equilibrado de carga, componentes encolados (*queued components*) y eventos.

3.3.6 Crítica

Estándar binario

COM proporciona un concepto de objeto muy vago y alejado del utilizado habitualmente y descrito en el capítulo anterior. En primer lugar, la herencia no está contemplada en el modelo de objetos para la reusabilidad, aunque se proporcionan otros mecanismos. En segundo lugar, los objetos no tienen una identidad propiamente dicha, sino que solamente existen para los clientes en tanto en cuanto se disponga de algún puntero a alguno de sus interfaces.

Los objetos necesitan procesos

Los objetos no son entidades autónomas, sino que necesitan de los procesos, que les proporcionan un entorno en el que ejecutarse.

Dificultad en la construcción de programas

A pesar de que COM+ viene a solucionar, en parte, las dificultades inherentes a la programación con objetos COM, la programación sigue siendo compleja. Los programadores de objetos COM (servidores) deben realizar tareas adicionales a las propias del dominio del problema, con el fin de que su objeto pueda ser utilizado por los clientes. Los clientes, por su parte, sólo pueden utilizar punteros para referenciar los objetos a través de alguna de sus interfaces.

3.3.7 Características interesantes

Transparencia de localización y acceso

COM se encarga de proporcionar a los clientes y objetos servidores la transparencia necesaria para las invocaciones locales y remotas. Una vez que un cliente obtiene un puntero a una interfaz de un objeto, puede invocar sus funciones con independencia de su ubicación. De la misma forma, no es necesario que el programador realice ninguna tarea para el tratamiento de las invocaciones locales y/o remotas.

Integración del modelo de objetos en el sistema operativo

COM es utilizado no solamente para proporcionar un modelo de objetos distribuidos al programador, sino también para que el sistema operativo proporcione sus servicios a los programadores en forma de objetos. Evidentemente, dicha integración se está llevando a cabo únicamente en los sistemas operativos de Microsoft.

3.4 Guide

Guide [BBD+91], nacido en el instituto INRIA, en Francia, constituye un sistema que pretende dar soporte a aplicaciones distribuidas cooperativas. Se intenta ofrecer un universo distribuido compartido, organizado como un conjunto de objetos compartidos por actividades concurrentes. Las diferentes actividades o procesos se comunican entre sí a través del uso de operaciones de objetos situados en otras actividades. Se trata, por tanto de un sistema operativo que pretende dar solución al problema de la

interoperabilidad entre objetos, surgido recientemente con la expansión de la tecnología de la orientación a objetos y la distribución.

El sistema constituye una plataforma para lenguajes de programación orientados a objetos como el C++ y el lenguaje Guide. Se pretende dar soporte para que aplicaciones orientadas a objetos implementadas en estos lenguajes, puedan establecer comunicación entre sí de manera segura, sin confiar en el código de cada uno.

Guide puede verse con una máquina virtual distribuida y multiprocesador. La distribución no es visible a los usuarios aunque sí lo es el paralelismo. Las aplicaciones se estructuran en objetos, que se almacenan de manera transparente en almacenamiento secundario y serán cargados bajo demanda en almacenamiento volátil para su ejecución.

3.4.1 Modelo de objetos

Guide soporta un modelo de **objetos pasivo**, de tal forma que un objeto encapsula únicamente datos y operaciones. Todos los objetos son persistentes y tienen un nombre único, global e independiente de su localización denominado **referencia del sistema**. Las referencias del sistema no son visibles a los usuarios y son utilizadas internamente para la invocación de objetos.

La distribución de los objetos no es visible para el programador, que no necesitará saber dónde está localizado un objeto particular. Así mismo, los objetos pueden migrar entre diferentes nodos.

3.4.2 Modelo de ejecución

La **tarea** (*job*) es la unidad de ejecución. Una tarea es un espacio virtual potencialmente distribuido y está compuesta de un conjunto de hilos de control concurrentes denominados **actividades**, que operan en los objetos pasivos. Para arrancar una aplicación, el usuario tiene que especificar un objeto inicial y un método inicial. Se crea entonces una tarea, asociando el objeto inicial a la misma y creándose una actividad para la invocación al método inicial.

Una tarea puede extenderse sobre varios nodos, creciendo o decreciendo el número de estos de manera dinámica, de acuerdo al patrón de invocaciones a objetos. La ejecución de una actividad consiste en invocaciones sucesivas a métodos de objetos, pudiendo tener lugar dicha invocación en cualquier nodo del sistema. Durante cada invocación, el objeto referenciado es localizado y asociado dinámicamente a la memoria virtual de la tarea. El conjunto de todos los objetos asociados a una tarea se denomina **contexto** de la tarea.

No existe una comunicación explícita entre actividades. La única forma que tienen de conseguirlo es a través de invocaciones a objetos compartidos.

3.4.3 Invocación de objetos

La operación de **invocación de objetos** debe especificar la referencia del sistema del objeto invocado (su nombre interno), el nombre del método y los parámetros de la invocación. La referencia contiene un **identificador único** de objeto (*oid*) y alguna **pista de su ubicación**.

Una **invocación remota** es aquella que tiene lugar en un nodo distinto de aquel en el que se realizó la invocación. En ese caso, la tarea y la actividad se difunden al nodo remoto y el objeto es invocado allí. En el caso de que el objeto invocado se encuentre únicamente en el almacenamiento persistente, deberá ser cargado en el almacenamiento volátil, eligiéndose el correspondiente al nodo desde el que se realizó la invocación (Guide no

incluye equilibrado de carga). El objeto será cargado en el nodo en el que fue creado únicamente si fue especificado como inmóvil (*unmovable*).

3.4.4 Migración de objetos

Los mecanismos de migración que se proporcionan en Guide están diseñados para ser utilizados en tiempo de administración, es decir, cuando la aplicación no está ejecutándose. No es posible, por tanto, que los objetos migren dinámicamente, mientras están siendo utilizados.

3.4.5 Crítica

Objetos pasivos

El sistema operativo Guide tiene un modelo de objetos pasivo, de tal forma que se diferencian las entidades activas, los procesos, de las entidades pasivas, los objetos, que son meros contenedores de datos y métodos. Como ya se comentó en el capítulo anterior, el modelo de objetos pasivo rompe la uniformidad del sistema, al tener que gestionarse dos tipos de entidades claramente diferentes, y dificulta la migración de objetos notablemente.

Lenguajes de programación

Los únicos lenguajes de programación disponibles son el C++ y el Guide. La interoperabilidad entre objetos se restringe, por tanto, a objetos creados con estos lenguajes.

Migración estática de objetos

El mecanismo de migración está muy limitado, dado que no permite que los objetos se muevan dinámicamente a la vez que están siendo utilizados por las aplicaciones. Una de las consecuencias principales de la falta de migración dinámica es la ausencia de equilibrado de carga en el sistema.

Ausencia de equilibrado de carga

Un sistema operativo distribuido que permite la migración de objetos debe proporcionar algún mecanismo para el equilibrado de carga. Su ausencia es un handicap para conseguir un buen rendimiento en situaciones en las que hay un gran desequilibrio en la carga de computación de los diferentes nodos.

3.4.6 Características interesantes

Pistas en las referencias

La utilización en las referencias de pistas sobre la ubicación del objeto referenciado es muy útil en la invocación remota, dado que en muchos casos va a ser una información válida. Dado que se espera que los objetos migren con menos frecuencia que con la que son invocados, las pistas de ubicación podrán ser utilizadas en lugar de operaciones de localización más costosas como la difusión.

Distribución no visible al programador

El programador puede concentrarse en el código correspondiente al dominio del problema, dado que va a ser totalmente ajeno a la ubicación de los objetos involucrados en el mismo.

Activación transparente de los objetos persistidos

Aquellos objetos que son invocados y se encuentran en el almacenamiento de persistencia, se activan automáticamente. En el caso de que la activación se realizase en un nodo distinto al nodo desde el que se realizó la invocación, la invocación remota resultante se realizaría también de manera transparente.

3.5 Clouds

Clouds es un sistema operativo pensado para dar soporte a objetos distribuidos [DAM+90, DLA+91] desarrollado en el Instituto de Tecnología de Georgia. Está implementado sobre el micronúcleo Ra, que funciona sobre máquinas Sun. Puede ser utilizado para aplicaciones centralizadas y distribuidas que pueden ser escritas con lenguajes orientados y no orientados a objetos.

3.5.1 Abstracciones de Clouds

Clouds utiliza la abstracción de **hilo** para la computación y la de **objeto** para representar el espacio de almacenamiento.

Un objeto Clouds es un **objeto pasivo de grano grueso**, que equivale a un espacio de direcciones virtuales de un proceso convencional. En este espacio se almacenan los datos y el código del objeto.

La estructura interna de un objeto no es conocida por el sistema. Puede ser internamente una colección de objetos programados en C++, pero estos objetos internos no pueden ser utilizados desde fuera del objeto Clouds. A pesar de que el usuario puede crear estos objetos Clouds, **no existe el concepto de clases ni de herencia** en este sistema.

Los objetos son globales y como tales tienen un **nombre único** (*sysname*) siempre válido dentro del sistema distribuido. La utilización de este identificador único permite la invocación transparente de un objeto independientemente de su localización. Los usuarios pueden definir nombres de más alto nivel para los objetos, que serán traducidos a los nombres de bajo nivel con un **servidor de nombres**.

Los **hilos** no están asociados a ningún objeto concreto, y van ejecutándose en los espacios de direcciones de los objetos, viajando de uno a otro a medida que se van invocando métodos de los distintos objetos. Los hilos pueden tener unas etiquetas especiales que permiten mantener diferentes tipos de atomicidad y consistencia en la invocación a operaciones de los objetos.

3.5.2 Distribución en Clouds

El sistema operativo Clouds se ejecuta sobre un conjunto de nodos que juegan uno o varios de los siguientes papeles: servidor de cómputo, servidor de datos o estación de usuarios.

Los **servidores de cómputo** proporcionan la potencia de computación. Los **servidores de datos** funcionan como un repositorio para los datos (objetos) persistentes. Las **estaciones de usuarios** proporcionan un entorno de programación a los usuarios y sirven de interfaz con los servidores de cómputo y de datos. Todos los objetos se almacenan en servidores de datos, pero son accesibles desde todos los servidores de cómputo del sistema de una manera transparente.

El **núcleo** del sistema está formado por un conjunto de máquinas homogéneas del tipo servidor de cómputo, que no tienen almacenamiento secundario y que proporcionan el servicio de ejecución para los hilos.

3.5.3 Espacio de objetos

Los objetos Clouds forman un **espacio de objetos disponible y accesible desde cualquier máquina** del sistema, a la manera de una memoria global compartida. La compartición de esta memoria global se proporciona con un mecanismo denominado **Memoria Compartida Distribuida** (DSM, *Distributed Shared Memory*).

Si un hilo en un nodo X invoca un método de un objeto O, la ejecución del método invocado tendrá lugar en el nodo X. Si O no está ubicado en el nodo X, se producirá una serie de faltas de página que serán servidas leyendo páginas de O desde el servidor de datos en el que reside actualmente. Únicamente las partes necesarias del código y los datos de O serán llevados a X. Este esquema permite que todos los objetos sean accesibles desde todos los servidores de cómputo. Se utiliza adicionalmente un protocolo de coherencia de la DSM para que varios hilos que se estén ejecutando en el mismo objeto utilicen de manera consistente los datos del mismo, incluso si se están ejecutando en diferentes servidores de cómputo.

La conclusión del uso del mecanismo de memoria compartida distribuida es que todo objeto del sistema reside lógicamente en cada uno de sus nodos. Este concepto permite separar el almacenamiento de los objetos de su uso.

3.5.4 Crítica

Modelo de objetos restringido

El modelo de objetos que soporta el sistema está muy alejado del modelo mencionado en el capítulo anterior. No soporta el concepto de clases, ni el de herencia. Los objetos son de un grano muy grueso, muy alejado del grano fino de los objetos de las aplicaciones. La estructura interna de objetos dentro de un objeto Clouds es totalmente desconocida para el sistema. El objeto Clouds es una pequeña evolución de un espacio de direcciones (objeto) sobre los que pueden funcionar los proceso (hilos que pasan por el objeto).

3.5.5 Características interesantes

Replicación de objetos soportada por el sistema operativo

El propio sistema operativo ofrece, de manera transparente, la replicación de aquellos objetos que están siendo invocados por parte de diferentes hilos, e incluso en diferentes nodos. Se asegura, además, la coherencia de los datos replicados.

Identificador global de objetos para transparencia de localización

Otro aspecto interesante es el uso de un identificador global de objetos, que permite la invocación transparente de un objeto independientemente de su localización.

3.6 Spring

Spring [MGH+94] es un sistema operativo distribuido, orientado a objetos y muy modular desarrollado por Sun, centrado en el desarrollo de interfaces fuertes entre los distintos componentes del sistema operativo, y que trata dichos componentes como partes reemplazables.

Los recursos del sistema se representan como objetos y todas sus interfaces se definen con un **lenguaje de definición de interfaces** (IDL, *Interface Definition Language*). Las interfaces únicamente especifican qué hacen los objetos, pero no cómo están implementadas sus operaciones.

Adicionalmente, el sistema operativo está estructurado sobre un micronúcleo (denominado **nucleus**), que implementa la mayor parte de la funcionalidad del sistema (sistemas de ficheros, paginadores, software de red, etc.) como servicios de nivel aplicación que se ejecutan sobre dicho micronúcleo, excepto el gestor de memoria virtual, que se ejecuta en modo privilegiado (*kernel*).

Spring no especifica cómo se deben implementar los objetos. Cada aplicación puede implementar los objetos de una manera diferente, llegando incluso a poder optar por diferentes maneras de invocar objetos.

3.6.1 Abstracciones

El micronúcleo soporta tres abstracciones básicas:

- **Dominios** (*domain*): análogos a los procesos de Unix, proporcionan un espacio de direcciones para la ejecución de aplicaciones y funcionan como contenedores de recursos, como los hilos (*thread*) y las puertas (*door*).
- **Hilos** (*thread*): se ejecutan dentro de dominios. En general, todo dominio es multihilo. Cada hilo individual se encarga de realizar una parte del trabajo global de la aplicación.
- **Puertas** (*door*): soportan las llamadas orientadas a objetos entre diferentes dominios. Una puerta describe un punto de entrada particular a un dominio. De alguna manera, recuerda los *sockets* de Unix BSD [LMK+89] o los puertos de Mach [ABB+86].

3.6.2 Invocación de objetos

Spring solamente proporciona mecanismos para la invocación de objetos en diferentes dominios. La forma en que se resuelve la invocación de objetos dentro de un mismo dominio es responsabilidad de lenguaje de programación con el que se ha implementado la aplicación.

Cuando un hilo en un dominio desea realizar una invocación de un objeto en otro dominio, tendrá que hacerlo a través de una puerta del dominio invocado, que permite identificar el objeto a invocar. En este caso, el micronúcleo crea un hilo en el dominio invocado, le transfiere el control y le pasa toda la información que necesita para atender la invocación. Cuando finaliza la ejecución del hilo del objeto invocado, el micronúcleo desactiva dicho hilo y reactiva el hilo que realizó la invocación, pasándole los datos devueltos por el hilo invocado.

Para proporcionar invocación de objetos localizados en diferentes nodos del sistema, el mecanismo de invocación se extiende con el uso de **representantes de red** (*network proxy*), que conectan los micronúcleos de diferentes máquinas de una manera transparente. Estos representantes no son más que dominios normales, no recibiendo soporte especial del micronúcleo y hablarán un protocolo de red determinado. Con el fin de poder trabajar con diferentes protocolos, una máquina Spring podrá disponer de varios representantes.

Los representantes se encargan de reenviar invocaciones a puertas entre dominios de diferentes máquinas, haciendo transparente a los dominios cliente y servidor su existencia.

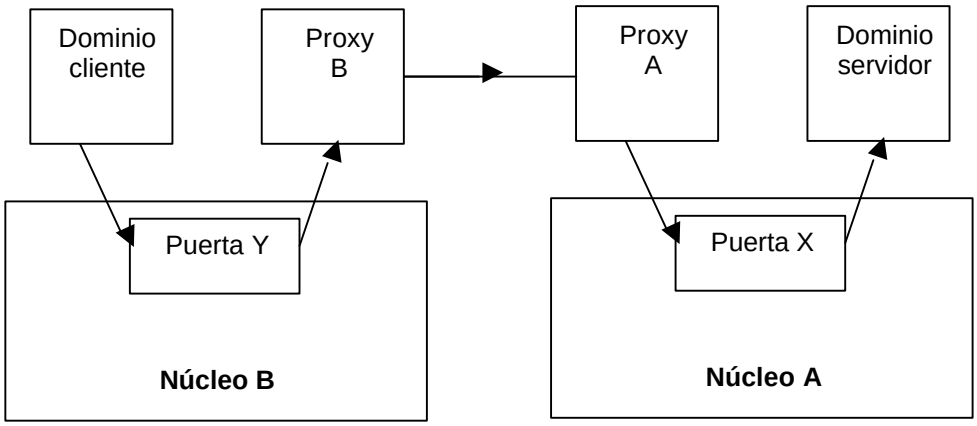


Figura 4.3. Uso de representantes para posibilitar las invocaciones remotas.

3.6.3 Nombrado

Spring proporciona un servicio de nombres uniforme. Todo objeto puede tener un nombre. Los nombres pueden pertenecer a contextos y los contextos a su vez pueden contener otros contextos, de tal forma que se puede construir un espacio de nombres jerárquico. Los contextos y los nombres son a su vez objetos Spring.

3.6.4 Seguridad

El acceso a los recursos está regulado por un mecanismo de seguridad. En Spring el acceso seguro a objetos se proporciona con un modelo de capacidades software y un modelo de listas de control de acceso.

3.6.5 Crítica

Modelo de objetos elemental

El modelo de objetos de Spring no soporta mecanismos de reutilización como herencia, polimorfismo, etc., mínimamente exigibles a un sistema de gestión de objetos. Los objetos Spring parecen, más bien, tipos abstractos de datos (TAD).

Falta de uniformidad en la orientación a objetos

Los objetos Spring residen en espacios de direcciones (dominio) en los que está implementada su funcionalidad. Spring es ajeno a la forma en que los objetos se disponen en los dominios y la manera en que se proporciona dicha funcionalidad, por lo que coexistirán (posiblemente) varios modelos de objetos: el de Spring, basado en dominios, hilos y puertas, y el de la implementación de los objetos (si se utiliza algún lenguaje orientado a objetos).

La falta de uniformidad afecta también a la forma en que se invocan los objetos. La invocación de objetos en diferentes dominios se realiza utilizando los mecanismos que proporciona Spring. La invocación dentro del mismo dominio no está especificada, y queda determinada por el lenguaje de programación utilizado. El programador tiene que estar al tanto de estas dos maneras de utilizar los objetos en función de su ubicación.

Falta de soporte para la migración

Spring no proporciona ningún tipo de soporte para la migración de objetos o dominios, lo que impide tareas como el equilibrado dinámico de carga entre los diferentes nodos del sistema.

3.6.6 Características interesantes

Uso de representantes de red

Los representantes de red posibilitan la invocación transparente de objetos que residen en dominios remotos. Su independencia del núcleo es muy importante, dado que permite su sustitución o modificación sin afectar al sistema operativo.

Relación entre hilos y dominios

Los hilos se crean dentro de los dominios cuando un objeto que reside en uno de ellos recibe una invocación externa. Este tipo de relación entre objetos e hilos recuerda el modelo de objetos activo, mencionado en el capítulo anterior y sobre el que ya se han comentado una serie de ventajas para la distribución.

3.7 SOS

El sistema operativo SOS [SGH+89] es un sistema operativo distribuido que da soporte de objetos, desarrollado por el INRIA dentro del proyecto Esprit SOMIW (*Secure Open Multimedia Integrated Workstation*).

3.7.1 Modelo de objetos

El modelo de objetos soportado por el sistema operativo es a la vez sencillo y potente. Un **objeto elemental** es un conjunto de datos y código definido por el usuario. Los objetos elementales no son conocidos por el sistema, siempre y cuando no se desee migrarlos, almacenarlos o que sean accesibles de manera remota. Los objetos SOS sí son conocidos por el sistema y a ellos se referirá el resto de la exposición.

Todos los objetos pertenecen a un **contexto**, que es un espacio de direcciones. Un objeto SOS puede migrar entre contextos, y, en un momento dado, o bien está activo en un contexto, o está almacenado en disco.

Todo objeto tiene un **identificador único**, denominado **OID**. Desde dentro del contexto, un objeto es referenciado por su dirección, y, globalmente, por una referencia, que contiene un OID y una pista de su ubicación.

3.7.2 Distribución de objetos

SOS está diseñado para favorecer el uso del concepto de **representante** (*proxy*) [Sha86] para la estructuración de aplicaciones distribuidas. Los representantes permiten extender el concepto de objeto al de objeto distribuido u **objeto fragmentado**. Externamente, un objeto fragmentado parece un objeto simple. Sus fragmentos locales o representantes, que son objetos elementales, proporcionan su interfaz. Internamente, todos los posibles fragmentos estarán distribuidos.

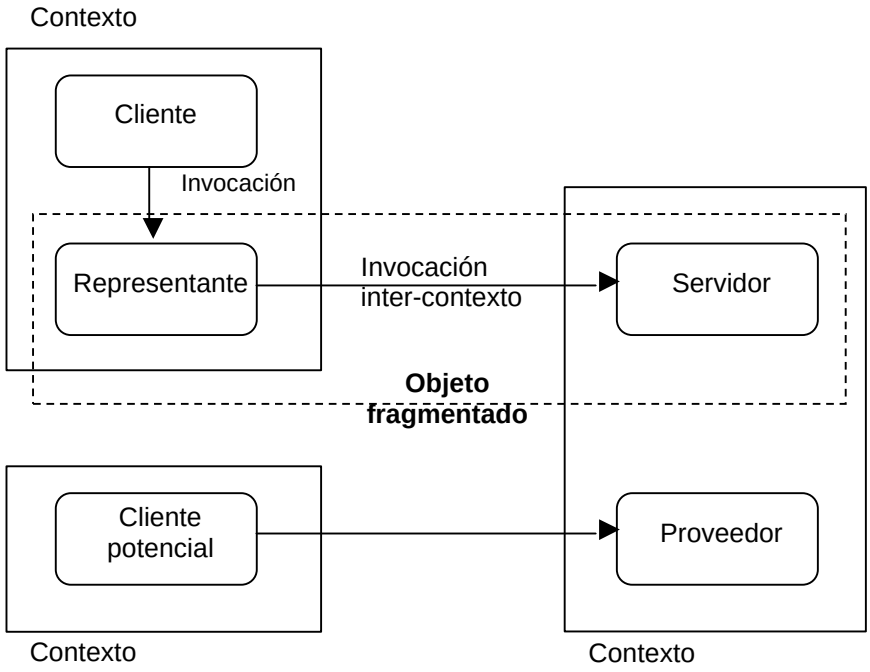


Figura 4.4. Concepto de representante: un servicio implementado como un objeto fragmentado.

Un servicio distribuido se implementa como un objeto fragmentado. Un cliente puede acceder al servicio invocando un fragmento local. Para que un cliente pueda conseguir acceso a un nuevo servicio, deberá solicitar al **gestor de objetos fragmentados** (vía su representante local) un representante para dicho servicio.

3.7.3 Migración de objetos

Un mecanismo básico de SOS es la migración de objetos. Un objeto que migra lleva consigo una lista de **prerrequisitos**, en la cual consta el conjunto de objetos que deben estar presentes siempre en su entorno. En caso de que sea necesario, el sistema migrará a su vez los representantes necesarios de dicho conjunto.

SOS utiliza una **interfaz de retrollamadas** (*upcall*) que permite a los objetos imponer su propia política sobre el mecanismo de migración del sistema. El método del objeto que se invoca con la retrollamada tomará decisiones como por ejemplo si la migración se realizará por movimiento o por copia, si le afectará sólo a él o a algún otro objeto, etc. Finalizada la migración, una retrollamada permitirá al objeto restablecer su entorno.

3.7.4 Modelo genérico

Todos los mecanismos mencionados son genéricos e independientes del lenguaje de programación. Para que SOS soporte objetos de un lenguaje de programación cualquiera, basta con asociar los prerrequisitos apropiados y los métodos invocados por las retrollamadas.

3.7.5 Diseño del sistema operativo

SOS está construido utilizando sus propios mecanismos: todos los servicios del sistema SOS están implementados como objetos fragmentados con representantes

locales. Entre ellos destacan el gestor de objetos, un servicio de nombrado flexible y un servicio de comunicaciones orientado a objetos.

3.7.6 Crítica

Coexistencia de varios modelos de objetos

Como ocurre en otros sistemas operativos ya revisados, en SOS coexisten dos modelos de objetos: el del propio sistema SOS u objeto fragmentado, que es conocido por el resto del sistema, y el de los objetos elementales, opacos a SOS y al resto de los objetos del sistema que no residan en su mismo contexto.

Falta de transparencia de localización

Los objetos se referencian de manera distinta según su ubicación respecto al objeto que los utiliza. Para referirse a un objeto que está en el mismo contexto, se utiliza su dirección; si el objeto está en otro contexto, se utiliza una referencia, que contiene su identificador de objeto y una pista de su ubicación.

3.7.7 Características interesantes

Uniformidad en la construcción del sistema

Tanto el propio sistema operativo como las aplicaciones de los usuarios se basan en la utilización de los objetos fragmentados con representantes. Dicha uniformidad permite a los objetos de usuario interactuar entre ellos y con los del sistema operativo de una manera común.

Utilización de pistas de localización de los objetos

Como ocurre en el sistema operativo Guide, la utilización de pistas en las referencias permite optimizar las operaciones de búsqueda de los objetos, ya que, en muchas ocasiones, las pistas proporcionarán información totalmente válida sobre la ubicación del objeto.

Migración conjunta de objetos

La lista de prerrequisitos para la migración permite identificar qué grupos de objetos tienen que moverse de manera solidaria, dado que se les supone una fuerte interrelación. La migración solidaria permite maximizar el número de invocaciones a objetos que se resuelven de manera local.

Especificación por el usuario de la política de migración

La separación de políticas y mecanismos de un sistema distribuido es muy importante para facilitar al máximo su adaptabilidad. La interfaz de retrollamadas permite modificar la política de migración que por defecto proporciona SOS. Esta posibilidad es muy útil para poder diseñar políticas de migración personalizadas.

3.8 Amoeba

Amoeba [TRS+90, MRT+90] es un sistema operativo distribuido basado en un micronúcleo [Gie90], desarrollado en la Universidad de Vrije y el Center for Mathematics and Computer Science de Amsterdam.

El diseño de Amoeba estuvo condicionado por los siguientes objetivos principales: transparencia, soporte para programación paralela, uso de la orientación a objetos y aproximación micronúcleo.

- **Transparencia**: se trata de proporcionar una imagen de sistema único. Para conseguir este objetivo, la decisión de diseño más importante fue la utilización del modelo del **conjunto** (*pool*) **de procesadores**, en el que no existe el concepto de máquina del usuario, y todos los recursos pertenecen al sistema como un todo.
- **Soporte para programación paralela**: aunque la transparencia es muy útil para casi todos los usuarios de un sistema distribuido, algunos de ellos están interesados en utilizarlo como plataforma de pruebas para experimentar con algoritmos, lenguajes, herramientas y aplicaciones distribuidas y paralelas. Amoeba proporciona soporte para este tipo de usuarios haciendo accesible para ellos el paralelismo subyacente.
- **Aproximación orientada a objetos, basada en capacidades**: otro de los objetivos fundamentales de Amoeba fue investigar la posibilidad de utilizar una aproximación orientada a objetos basada en capacidades para la construcción de un sistema distribuido. Con este motivo, los objetos y las capacidades se utilizan de una manera uniforme en el diseño del sistema operativo. En particular, el software del sistema operativo está basado en objetos, los cuales tienen nombre y están protegidos con el uso de capacidades.
- **Aproximación micronúcleo**: con este objetivo se intenta minimizar el tamaño del núcleo con el fin de incrementar la flexibilidad. De esta manera, muchos de los servicios estándar del sistema (por ejemplo, el servicio de ficheros) están construidos en el espacio de usuario, de tal forma que pueden ser fácilmente modificados.

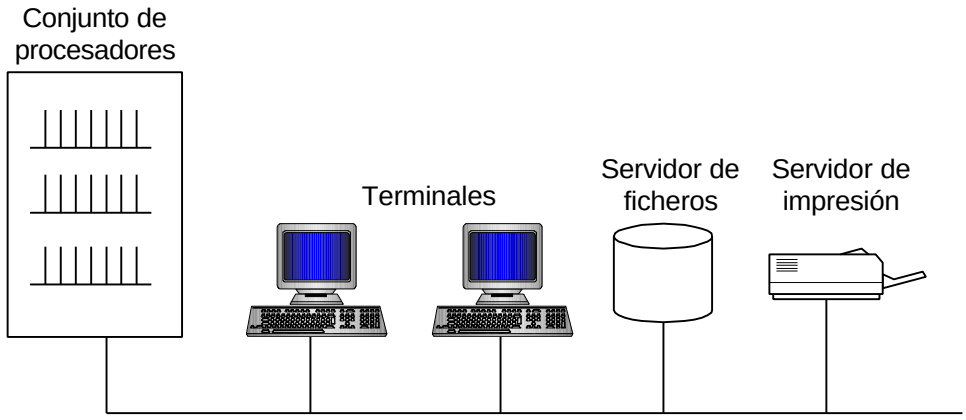


Figura 4.5. Arquitectura del sistema operativo Amoeba.

3.8.1 Objetos de Amoeba

Un objeto de Amoeba es similar a un tipo abstracto de datos, formado por un conjunto de datos encapsulados y un conjunto de operaciones que actúan sobre ellos. Los **objetos son pasivos** por naturaleza, de manera que no pueden hacer nada por sí mismos. Todo objeto es gestionado por un proceso servidor, que realiza las operaciones definidas por el objeto que gestiona y al que se pueden enviar solicitudes vía RPCs. Toda RPC especifica el objeto a utilizar (realmente se utiliza una capacidad, que se describe más adelante), la operación a realizar y los parámetros pasados y se ejecuta de manera síncrona. Ejemplos típicos de objetos Amoeba son los ficheros, directorios, segmentos de memoria, procesadores, discos, etc.

3.8.2 Capacidades

Todos los objetos están identificados y protegidos por una **capacidad**, que está compuesta por los siguientes campos:

- **Número de puerto del servidor**: identifica el servidor que gestiona el objeto referenciado por la capacidad.
- **Número de objeto**: dado que un servidor puede gestionar varios objetos del mismo tipo, el número de objeto permite al servidor identificar el objeto específico de entre todos los objetos que gestiona. La combinación del número de puerto del servidor y el número de objeto permite identificar unívocamente un objeto en el sistema.
- **Derechos**: denota qué operaciones puede realizar el poseedor de la capacidad sobre el objeto referenciado.
- **Control**: sirve para validar la capacidad, de tal manera que se evite la posibilidad de que se falsifiquen capacidades.

Las capacidades son gestionadas íntegramente por los procesos de usuario. Para crear un objeto, un proceso cliente envía un mensaje al servidor apropiado. El servidor crea el objeto y devuelve la capacidad al proceso cliente. Esta capacidad se denomina **capacidad del propietario**, y proporciona todos los derechos sobre el objeto. Si el propietario del objeto desea pasar derechos de acceso restringidos sobre el mismo a otros procesos, envía un mensaje al servidor solicitando la creación de una **capacidad restringida**, la cual, una vez creada, es enviada finalmente al proceso destinatario.

3.8.3 Nombrado de objetos

El **esquema de nombrado** basado en capacidades es transparente respecto a la localización del objeto y del servidor, dado que para realizar una operación sobre un objeto, no es necesario conocer la localización del objeto ni del servidor que lo gestiona. Para proporcionar dicha transparencia a los procesos de usuario, el núcleo es el encargado de resolver la localización de un objeto o servidor. El mecanismo de localización consiste en una serie de **preguntas difundidas** (*broadcast queries*) en toda la red, que incluyen el número del puerto del servidor que gestiona el objeto buscado. El mensaje de respuesta contiene la dirección de red en la que reside el servidor, que es almacenada en una cache para su futuro uso.

3.8.4 Crítica

Modelo de objetos como tipo abstracto de datos

A pesar de que en la terminología se hable de objetos, las entidades que utiliza Amoeba son más bien tipos abstractos de datos. No se soportan, por tanto, herencia, polimorfismo, etc.

Objetos pasivos

Los objetos se convierten en meros contenedores de datos y operaciones, recayendo la capacidad de computación en los procesos.

Implantación en el núcleo de mecanismos relacionados con la distribución

La introducción en el núcleo del mecanismo de localización de objetos dificulta notablemente su adaptación. No es posible cambiar el mecanismo o la política que lo rige sin modificar el propio núcleo del sistema operativo.

3.8.5 Características interesantes

Construcción del sistema operativo como conjunto de objetos

En Amoeba se consigue cierto grado de uniformidad, al estar construido el propio sistema operativo como conjunto de objetos. De esta manera, los objetos de usuario pueden invocar a otros objetos de usuario o al sistema operativo de una manera uniforme.

Transparencia de localización y acceso

Todos los objetos pueden ser accedidos de manera transparente a partir de una capacidad. El núcleo del sistema operativo se encargará de realizar las operaciones necesarias para proporcionar dicha transparencia.

Cache de localizaciones remotas de objetos

En otros sistemas operativos las pistas de localización eran almacenadas en las referencias a los objetos. En Amoeba se agrupan todas las pistas en una cache mantenida por el núcleo del sistema operativo.

3.9 Emerald

Emerald es un lenguaje de programación orientado a objetos para la construcción de aplicaciones distribuidas [Hut87, Jul88, RTL+91]. Aunque se desarrolló originalmente para simplificar la construcción de aplicaciones distribuidas eficientes, Emerald proporciona un modelo de programación de propósito general.

3.9.1 Objetos de Emerald

Un objeto de Emerald es el único mecanismo de abstracción del lenguaje. Consiste en:

- Un **nombre único** en todo el sistema distribuido.
- Su **representación**, es decir, los datos locales del objeto.
- Un **conjunto de operaciones** que pueden ser invocadas en el objeto.
- Un **proceso opcional**, que se arranca cuando el objeto es inicializado y se ejecutará en paralelo con las invocaciones a las operaciones del objeto.

Adicionalmente, un objeto puede tener varios atributos. Todo objeto tiene una **ubicación**, que especifica el nodo en el que reside actualmente el objeto. Un objeto puede ser definido como **inmutable**, definición que es dada por el programador y que indica que su estado no puede cambiar.

3.9.2 Actividad interna de los objetos

Emerald soporta concurrencia entre objetos y dentro de los objetos. Dentro de los objetos pueden estar progresando varias invocaciones en paralelo con el proceso interno del objeto.

3.9.3 Comunicación y localización

El único **mecanismo de comunicación** en Emerald es la invocación, siempre **síncrona**. En principio todos los argumentos y el resultado de una invocación se pasan por referencia. Existen, sin embargo, optimizaciones para permitir el paso de determinado tipo de argumentos por valor.

En general, los objetos en Emerald no tienen por qué estar al tanto de su **ubicación** para ejecutarse de manera adecuada, proporcionándose la **distribución de una manera transparente** al programador. Pero existe cierto tipo de aplicaciones (bases de datos distribuidas, servidores de nombres distribuidos, etc.) que existen porque existen entornos distribuidos. Para este tipo de aplicaciones el lenguaje proporciona primitivas que le permiten controlar la ubicación y el movimiento de los objetos.

3.9.4 Migración de objetos

Emerald proporciona un conjunto de primitivas para la distribución que permiten:

- Obtener la información de ubicación de un objeto.
- Mover un objeto a otro nodo.
- Fijar (atar) un objeto en un nodo y liberarlo de su atadura.

Adicionalmente, un objeto puede sufrir una modificación en su ubicación si se utiliza como parámetro en invocaciones remotas. Emerald proporciona tres variantes para el paso de parámetros por referencia, que es el modo por defecto y el más habitual:

- *Call-by-move*: el objeto representado por el parámetro se mueve al nodo en el que se encuentra el objeto invocado.
- *Call-by-visit*: muy parecido al anterior. Únicamente hay que añadir que el objeto parámetro vuelve a su nodo de origen una vez haya finalizado la invocación.
- *Call-by-move-return*: en este caso, el objeto resultado se mueve al nodo del objeto que realizó la invocación.

Es importante señalar que estas variantes en el paso de parámetros son conseguidas con la utilización de palabras clave del lenguaje que acompañan a los diferentes parámetros de una invocación.

En cualquiera de los casos anteriores, los atributos (estado) del objeto migran solidariamente con él.

3.9.5 Agrupación de objetos

Con el fin de optimizar el rendimiento en las operaciones de migración, Emerald proporciona primitivas para crear grupos de objetos, que se tienen en cuenta en las operaciones de migración. Se identifica un objeto como el principal, de tal forma que el resto de los objetos migrarán cuando lo haga el objeto principal.

3.9.6 Crítica

Uso restringido

El hecho de que Emerald sea un lenguaje de programación con características distribuidas limita su uso a aquellas plataformas para las cuales existe soporte.

Además, algunas primitivas de distribución (por ejemplo, migración en el paso de parámetros) son proporcionadas a nivel sintáctico, y no es posible modificarlas o adaptarlas.

Modelo de objetos incompleto

Una característica que se ha descrito como imprescindible en un modelo de objetos, como es la herencia, no está presente en Emerald. De nuevo los objetos se convierten en

tipos abstractos de datos. La inexistencia de herencia en el modelo simplifica notablemente los distintos mecanismos de la distribución.

3.9.7 Características interesantes

Soporte de objetos de cualquier granularidad

Los objetos soportados por Emerald pueden ser de cualquier granularidad. El objeto se convierte en la única entidad que maneja el sistema, proporcionándose la deseada uniformidad conceptual.

Transparencia de localización y acceso

Los objetos son accedidos de manera transparente con respecto a su ubicación. En este sentido, el nombre único asociado a todos los objetos permite localizar un objeto de manera unívoca dentro del sistema distribuido.

Control de la distribución

El programador puede ejercer el control sobre la distribución de los objetos haciendo uso de un conjunto de primitivas al efecto. Esto es muy útil para determinadas aplicaciones que pueden necesitar sacar partido de la arquitectura distribuida del sistema.

Agrupaciones de objetos

La agrupación de objetos permite minimizar el número de invocaciones remotas generadas por la migración de un objeto. El programador será el responsable de determinar qué objetos tienen que moverse de manera solidaria, aunque determinadas agrupaciones pueden ser establecidas por el propio sistema (por ejemplo, los atributos se mueven con el propio objeto).

3.10 COOLv2

COOLv2 [LJP93] es una capa de soporte para objetos distribuidos construida sobre el micronúcleo CHORUS [RAA+92]. El proyecto se proponía reducir la desadaptación de impedancias entre las abstracciones de los lenguajes y las abstracciones proporcionadas por el sistema. Para ello extiende el micronúcleo CHORUS con abstracciones más adecuadas para sistemas orientados a objeto, con la idea de reducir la ineficiencia de capas software añadidas.

3.10.1 Abstracciones base

Las abstracciones incluidas en el sistema base son los **agrupamientos** (***cluster***) y los **espacios de contexto** que abstraen los micronúcleos distribuidos y el almacenamiento persistente.

Un agrupamiento es un conjunto de regiones de memoria respaldadas en disco que serán usadas para colocar sobre ellas los objetos. Los agrupamientos se hacen corresponder con espacios de direcciones virtuales distribuidos que forman un espacio de contexto.

3.10.2 Soporte genérico en tiempo de ejecución

Sobre las abstracciones base se coloca una capa de software denominada **GRT** (*Generic Run Time*, **soporte genérico en tiempo de ejecución**). El GRT implementa la noción de objetos usando un modelo de objetos básicos organizados en clases. También

proporciona invocación de métodos y actividades (hilos) e interactúa con los niveles inferiores para valerse del soporte que proporcionan para la persistencia y la memoria compartida distribuida.

3.10.3 Soportes específicos para lenguajes

Este GRT se complementa con **soportes en tiempo de ejecución específicos** para diferentes lenguajes (C++, Eiffel, etc.). La combinación del GRT con el soporte específico de un lenguaje proporciona el soporte para el modelo de objetos de un lenguaje determinado.

3.10.4 Modelo de objetos

Los objetos se tratan como objetos pasivos. Los hilos de ejecución (**actividades**) viajan, por tanto, de objeto en objeto mediante las invocaciones de métodos.

Todo objeto existe dentro de un **contexto**, que le sirve como entorno de ejecución.

Un conjunto de atributos asociados a un objeto determina si es conocido globalmente y si es persistente. Un objeto tiene que solicitar la recepción de mensajes enviados por objetos remotos, convirtiéndose en un objeto global.

3.10.5 Invocación de objetos

Existen dos maneras de invocar los objetos. Dentro de un agrupamiento se accede a los objetos locales utilizando las referencias propias de cada lenguaje, que serán direcciones de memoria virtual (punteros). Para invocar a objetos que no están en el agrupamiento, se utiliza un **objeto de interfaz** (*proxy* o representante) que representa al objeto remoto y al que se accede usando un identificador global persistente. El código para estos representantes es generado por compiladores especiales modificados para ello.

3.10.6 Distribución

Los objetos se identifican de manera transparente con respecto a su localización, lo que hace a su vez transparente su migración. De la misma forma pueden migrar del almacenamiento secundario a un contexto cuando no se encuentran en el almacenamiento volátil.

COOL está organizado como un **conjunto de servidores**, en cada uno de los cuales existe un gestor (*manager*) que proporciona la gestión básica de objetos y contextos del servidor. Los servidores COOL se comunican utilizando los mecanismos IPC del micronúcleo Chorus subyacente.

Los objetos creados con el **atributo global** pueden ser el objetivo de invocaciones remotas, pero tienen que solicitarlo de manera explícita. Las invocaciones no son tales en el sentido estricto de la palabra, sino que son, más bien, operaciones de paso de mensajes.

La **migración de objetos** se produce en las operaciones de paso de mensaje, de tal forma que el objeto emisor especifica los objetos a ser movidos o copiados en el contexto destino. El soporte en tiempo de ejecución para cada lenguaje pueden construir sobre el mecanismo elemental primitivas de migración o semánticas de paso de parámetros como las descritas para Emerald. En cualquier caso, lo que realmente migran son los identificadores de los objetos. Los objetos propiamente dichos serán reclamados en el destino a través de los mecanismos de memoria virtual subyacentes.

3.10.7 Crítica

Falta de uniformidad en la orientación a objetos

El uso de objetos no tiene una uniformidad total. Por ejemplo se usan dos maneras de referenciar a los objetos, en función de su situación. Dentro del mismo módulo se usan las referencias de cada lenguaje, y para acceder a un objeto remoto se utiliza un mecanismo de acceso diferente basado en un identificador global.

Los objetos deben solicitar explícitamente poder ser conocidos globalmente

Por defecto, los objetos no pueden ser invocados por otros objetos. Deben solicitarlo explícitamente. Este hecho resta cierta transparencia al mecanismo de invocación.

Paso de mensajes en lugar de invocación

La invocación de objetos no existe como tal. Es remplazada por el mecanismo de paso de mensajes proporcionado por el micronúcleo Chorus.

Orientación a objetos sólo en el espacio del usuario: pérdida de flexibilidad en el sistema

El soporte para objetos sólo existe en las aplicaciones de usuario. El resto del sistema es convencional y por tanto hay que acceder mediante las interfaces no orientadas a objetos del mismo. Las ventajas de la orientación a objetos para la extensibilidad, etc. no se pueden aplicar al sistema base. Se pierde flexibilidad.

3.10.8 Características interesantes

Mecanismo mínimo de migración

El establecimiento de un mecanismo mínimo de migración de objetos no sujeto a ningún tipo de política permite que el sistema sea flexible, en el sentido que otras entidades de más alto nivel serán las que tengan que determinar de qué manera va a tener lugar la migración. Es posible, por tanto, que diferentes lenguajes de programación utilicen políticas o semánticas distintas sobre el mismo mecanismo básico.

3.11 APERTOS

Apertos [Yok92], una evolución de su predecesor Muse [YTY+89] suele considerarse como el pionero en la aplicación de la **reflectividad** en los sistemas operativos orientados a objetos.

Es un sistema operativo para dar soporte a objetos y estructurado uniformemente en términos de objetos. La motivación inicial de aplicación es para su uso en un entorno de computación móvil, compuesto por ordenadores que pueden estar conectados a la red y también desconectarse para trabajar independientemente o trabajar remotamente mediante enlaces sin hilos, etc. La gran novedad de este sistema consiste en utilizar una separación de los objetos en dos niveles: meta-objetos y objetos base.

3.11.1 Estructuración mediante objetos y meta-objetos

Los **meta-objetos** proporcionan el entorno de ejecución y dan soporte (definen) a los **objetos base**. Cada objeto base está soportado por un **meta-espacio**, que está formado por un grupo de meta-objetos. Cada meta-objeto proporciona una determinada funcionalidad a los objetos del meta-espacio en que se encuentra. Por ejemplo, un determinado objeto puede estar en un meta-espacio con meta-objetos que le

proporcionen una determinada política de planificación, un mecanismo de sincronización determinado, el uso de un protocolo de comunicación, etc.

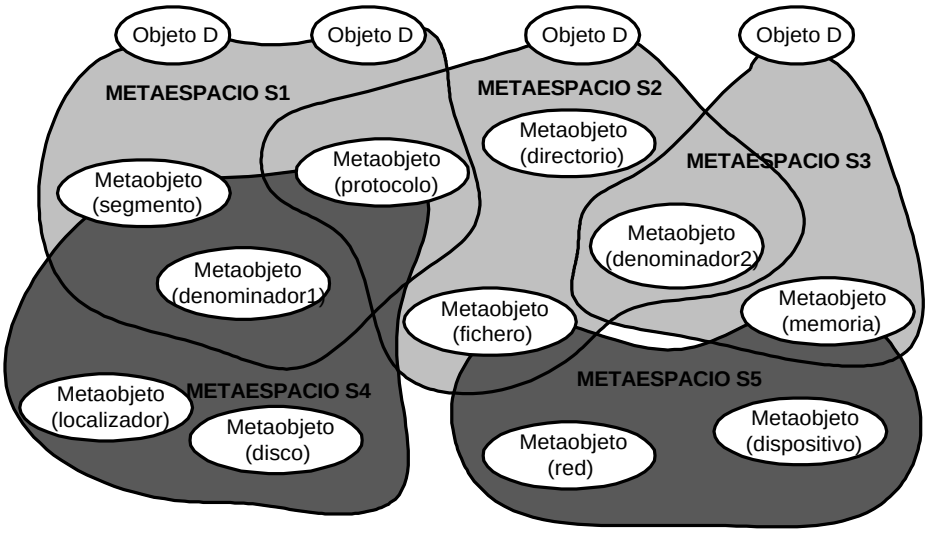


Figura 4.6. Separación entre los espacios de objetos y de meta-objetos en el sistema Apertos.

3.11.2 Flexibilidad

La flexibilidad del sistema se consigue mediante la posibilidad de que un objeto cambie de meta-espacio, o la adición de nuevos meta-objetos que proporcionen funcionalidad adicional. Por ejemplo, un meta-objeto podría migrar de un meta-espacio dado a otro meta-espacio que use un protocolo de comunicaciones para una red sin hilos cuando el ordenador en el que se encuentre se desconecte físicamente de la red.

3.11.3 Reflectividad

La reflectividad se produce al existir una interfaz bidireccional entre los objetos base y los meta-objetos que los soportan. Los objetos pueden dialogar con sus meta-objetos, usando un punto de entrada al meta-espacio denominado **reflector**. A su vez, los meta-objetos influyen en el funcionamiento de los objetos base. Por otro lado ambos, objetos y meta-objetos, comparten el mismo marco conceptual al estar descritos en los mismos términos de objetos. Los meta-objetos también pueden considerarse como objetos, por lo que tendrían su propio meta-meta-espacio y así sucesivamente. Esta regresión termina en un meta-objeto primitivo que no tiene meta-meta-espacio (se describe a sí mismo).

Los objetos y meta-objetos se organizan en una jerarquía de clases, al igual que los reflectores. Apertos se estructura usando una serie de jerarquías predefinidas de reflectores y meta-objetos.

3.11.4 Jerarquía de reflectores

La jerarquía de reflectores define la estructura común de programación de meta-objetos del sistema. Un reflector, como punto de entrada a un meta-espacio, ofrece a los objetos base una serie de operaciones que pueden usar de sus meta-objetos. La jerarquía base define las operaciones comunes del sistema.

MetaCore es un meta-objeto terminal en el que finaliza la regresión de meta-meta-objetos. Puede considerarse como el equivalente de un micronúcleo de otros sistemas. Proporciona a todos los demás objetos del sistema con los elementos básicos del sistema, es decir, las primitivas comunes de separación entre objetos y meta-objetos y de migración de objetos. Proporciona el concepto de contexto de ejecución para virtualizar la CPU y las primitivas básicas de la computación reflectiva.

3.11.5 Invocación de objetos

Los objetos del nivel base pueden comunicarse entre sí de manera uniforme sin necesidad de tener en cuenta si el objeto invocado reside o no en la misma máquina. La naturaleza de la comunicación (local o remota) es determinada por un meta-objeto, que llevará a cabo las acciones oportunas para que tenga lugar la invocación.

3.11.6 Migración de objetos

El concepto de migración de objetos en Apertos es ligeramente distinto del utilizado hasta ahora. En Apertos, un objeto migra cuando cambia su metaespacio, es decir, el objeto atraviesa una metajerarquía.

La migración es realiza por metaobjetos, tanto en el metaespacio origen como en el metaespacio destino. Dado que un objeto es representado internamente como un conjunto de metaobjetos, la migración de un objeto consiste en la migración de metaobjetos al metaespacio destino.

3.11.7 Crítica

Complejidad de estructura

La separación de la estructura del sistema en múltiples meta-espacios recursivos, aunque da mucha flexibilidad al sistema, también complica la comprensión del mismo de manera sencilla por el usuario.

Falta de uniformidad por la separación espacio/meta-espacio de objetos

La separación completa de ambos espacios introduce una cierta falta de uniformidad en el sistema. La jerarquía de meta-objetos está totalmente separada de la de los objetos de usuario. La programación de meta-objetos se realiza por tanto de una manera diferente a la de los objetos normales.

3.11.8 Características interesantes

Reflectividad para la flexibilidad

La adopción de la reflectividad en el sistema es muy importante para lograr la flexibilidad en el sistema de manera uniforme. Esta propiedad permite describir mediante objetos los propios elementos de la máquina. De esta manera se unifican dentro del paradigma de la OO los elementos del usuario y los elementos del sistema que los soportan. Esto permite la extensibilidad del sistema de una manera uniforme, al permitir que los objetos de usuario puedan acceder a los objetos del sistema usando el mismo paradigma de la OO.

Migración entre metaespacios

La migración de objetos entre metaespacios permite extender el concepto habitual de migración (entre diferentes nodos o computadores). Un objeto puede migrar, por

ejemplo, a un almacenamiento secundario, siempre que esté representado por el metaespacio correspondiente.

3.12 Resumen de características de los sistemas revisados

	Características modelo objetos	**Relación objetos/computación**	**Tipos de las referencias**	**Característica destacable**
DC++	El de C++	Objetos pasivos	Locales: punteros Remotas: ID	Entorno sobre DCE
CORBA	Herencia de interfaces	Objetos pasivos	Locales: lenguaje Remotas: ref. objeto CORBA	Multilenguaje, multi S.0., multiplataforma
COM	Contención y agregación	Objetos pasivos	Locales: lenguaje Remotas: puntero a interfaz	Multilenguaje, entornos Microsoft
RMI	El de Java	Objetos pasivos	Uniformes	Entornos Java
Guide	Estándar	Objetos pasivos	Uniformes	Persistencia integrada con distribución
Clouds	TAD	Objetos pasivos	Uniformes	Replicación y Memoria Compartida Distribuida
Spring	TAD	Objetos activos	Locales: lenguaje Remotas: representante	Acceso a objetos regulado por mecanismo seguridad
SOS	No uniforme	Objetos pasivos	Locales: dirección Remotas: ID	Interfaz de retrollamadas para intervención del usuario
Amoeba	TAD	Objetos pasivos	Uniformes (capacidades)	Aproximación micronúcleo
Emerald	Carece de herencia	Objetos activos	Uniformes	Soporte completo a la distribución
COOLv2	No uniforme	Objetos pasivos	Locales: lenguaje Remotas: representante	Clara separación entre políticas y mecanismos
Apertos	No uniforme objetos/metaobjetos	Objetos activos	Uniformes	Reflectividad

CAPÍTULO 4 CORBA

4.1 Introducción

CORBA (*Common Object Request Broker Architecture*) [OMG99] es una especificación definida por el OMG (*Object Management Group*) para la creación y uso de objetos remotos, cuyo objetivo en proporcionar interoperabilidad entre aplicaciones en un entorno distribuido y heterogéneo. OMG es el mayor consorcio mundial de compañías de software (alrededor de 800) entre las que destaca la ausencia de Microsoft, que tiene su propia plataforma de objetos distribuidos: DCOM (*Distributed Component Object Model*, Modelo de Objetos Componentes Distribuido).

Es conocido como un tipo de "*middleware*", ya que no realiza las funciones de bajo nivel necesarias para ser considerado un sistema operativo. A pesar de que debe funcionar sobre sistemas operativos tradicionales, realiza muchas de las operaciones que tradicionalmente se han considerado del dominio de los sistemas operativos para entornos distribuidos.

4.1.1 Objetos CORBA

Los **objetos CORBA** se diferencian de los objetos de los lenguajes habituales de programación en que:

- pueden estar localizados en cualquier lugar de la red,
- pueden ejecutarse en cualquier plataforma (hardware + sistema operativo), y
- pueden estar escritos en cualquier lenguaje.

Un cliente puede utilizar un objeto CORBA sin saber donde está ni en qué lenguaje ha sido implementado; lo único que tiene que conocer es la interfaz que publica dicho objeto.

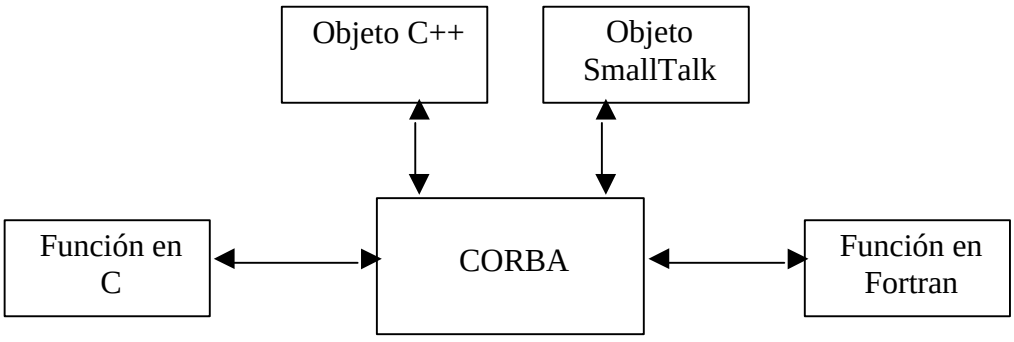


Figura 4.7. La función de CORBA.

4.1.2 El ORB

El corazón de CORBA es su ORB (*Object Request Broker*). El ORB es el responsable de:

- Buscar la implementación del objeto servidor.
- Prepararlo para recibir la petición.
- Poner en contacto el cliente con el servidor.
- Transportar los datos (parámetros y valores de retorno) entre uno y otro, transformándolos adecuadamente.

En definitiva es el responsable de que ni cliente ni servidor necesiten conocer ni la localización ni el lenguaje de implementación del otro.

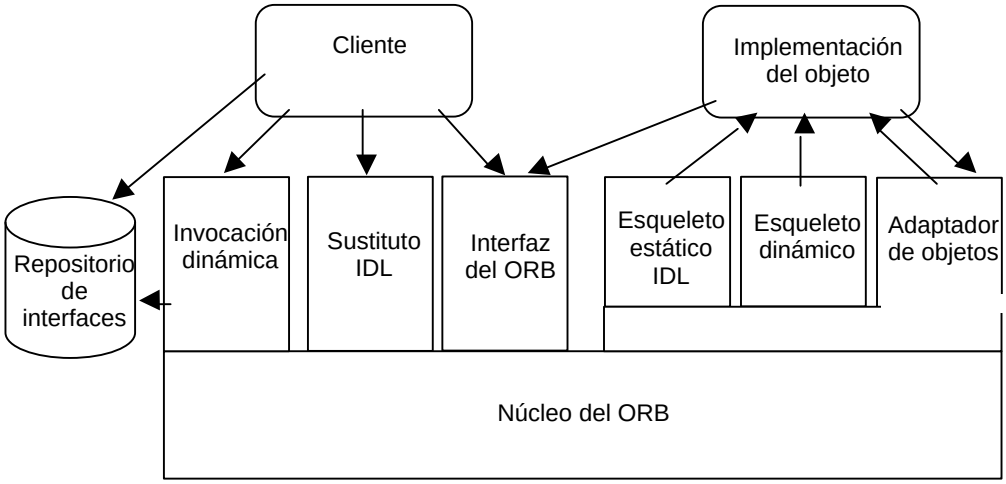


Figura 4.8. Estructura del ORB de CORBA.

4.1.3 Definición de objetos CORBA

Los objetos CORBA se tienen que definir con el **lenguaje de definición de interfaces IDL** (*Interface Definition Language*) que, como su propio nombre indica, se limita a definir la interfaz del objeto y no su implementación. La **definición del objeto** incluye la definición de sus atributos, sus métodos (denominados operaciones) y las excepciones que eleva. La **implementación del objeto** se tiene que realizar con algún lenguaje de programación que tenga enlaces con IDL (en la actualidad existen enlaces con lenguajes como C, C++, Java, Smalltalk, Ada, etc.).

IDL proporciona herencia múltiple de interfaces, de manera que las interfaces derivadas heredan las operaciones y los tipos definidos en las interfaces base. Todas las interfaces derivan de una interfaz raíz, denominada `Object`, la cual proporciona servicios que son comunes a todos los objetos CORBA, como duplicación y liberación de referencias a objetos, etc.

La compilación de una definición de objetos en IDL genera, entre otras cosas, un sustituto (*stub*) y un esqueleto (*skeleton*), para el cliente y servidor, respectivamente. El sustituto crea una petición de servicio al ORB a instancias del cliente y representa al objeto servidor. El esqueleto, por su parte, entrega las peticiones a la implementación del objeto CORBA.

Es posible utilizar para la definición de los objetos una serie de tipos basicos y construidos que no tienen la categoría de objetos y que son similares a otros tipos encontrados en la mayoría de los lenguajes de programación: *char*, *boolean*, *array*, *struct*, etc.

4.1.4 El repositorio de interfaces

Las interfaces de objetos se almacenan en un **repositorio de interfaces** (IR, *Interface Repository*), que proporciona un almacenamiento persistente de las declaraciones de interfaces realizadas en IDL. Los servicios proporcionados por un IR permiten la navegación por la jerarquía de herencia de un objeto y proporcionan la descripción de todas las operaciones soportadas por un objeto.

La función principal del IR es proporcionar la información de tipos necesaria para realizar peticiones utilizando la **Interfaz de Invocación Dinámica**, aunque puede tener otros propósitos, como servir de almacenamiento de componentes reutilizables para los desarrolladores de aplicaciones.

4.1.5 La Interfaz de Invocación Dinámica

La compilación de las declaraciones IDL en algún lenguaje de programación permite a los clientes invocar operaciones en objetos conocidos, pero algunas aplicaciones necesitan poder realizar llamadas a objetos sin tener conocimiento de sus interfaces en tiempo de compilación. En esencia, la **Interfaz de Invocación Dinámica** (DII, *Dynamic Invocation Interface*) es un *stub* genérico de clientes capaz de enviar cualquier petición a cualquier objeto, interpretando en tiempo de ejecución los parámetros de la petición y los identificadores de la operación.

4.1.6 Adaptadores de objetos

CORBA permite que la implementación de los objetos sea todo lo variada que se quiera. En unos casos, varias interfaces IDL pueden estar implementadas por un solo programa, y, en otros casos, una interfaz IDL puede estar implementada por una serie de programas, uno para cada operación.

Un **Adaptador de Objetos** (OA, *Object Adapter*) proporciona los medios por los que varios tipos de implementaciones de objetos utilizan los servicios del ORB, como por ejemplo:

- generación de referencias de objetos.
- invocación de métodos de objetos.
- seguridad.
- activación y desactivación de objetos e implementaciones.

Dependiendo del ORB subyacente, un OA puede elegir entre proporcionar estos servicios delegando en el ORB o realizando el trabajo él mismo. En cualquier caso, las implementaciones de los objetos no van a estar al tanto de la diferencia, ya que únicamente usan la interfaz proporcionada por el OA.

4.1.7 Referencias

Para que un cliente pueda realizar una petición a un objeto servidor, deberá utilizar una **referencia** al objeto. Una referencia siempre refiere el mismo objeto para la que fue creada, durante tanto tiempo como exista el objeto. Las referencias son tanto inmutables como opacas, de manera que un cliente no puede “entrar” en una referencia y modificarla. Sólo el ORB sabe que es lo que hay “dentro” de la referencia.

Cuando se pasan objetos como parámetros en invocaciones a métodos, lo que realmente se pasan son referencias a dichos objetos. El paso de objetos como parámetro es, por tanto, por referencia.

4.1.8 Paso de parámetros por valor

En la última revisión importante de CORBA (revisión 2.3, de Junio de 1999) se introdujo la propuesta de **objetos-por-valor** (*objects-by-value*), que extienden el modelo de objetos de CORBA tradicional para permitir el paso de objetos por valor dentro de los parámetros de los métodos. Para conseguirlo se introduce un nuevo tipo IDL denominado el tipo **valor** (*value*).

Cuando un ORB encuentra un objeto de tipo valor dentro de un parámetro, automáticamente pasa una copia del estado del objeto al receptor. En contraste, el ORB pasará por referencia cualquier objeto declarado vía una interfaz IDL.

Un tipo valor se puede entender a medio camino entre una interfaz IDL y una estructura (del estilo de las del lenguaje C). La sintaxis del valor permite especificar algunos detalles de implementación (por ejemplo, su estado), que no forman parte de una interfaz IDL. Además, se pueden especificar métodos locales para operar con dicho estado. A diferencia de las interfaces, los métodos de los tipos valor no pueden ser invocados remotamente.

4.1.9 Servicios de objetos

Los **CORBAservices** son un conjunto de servicios que aumentan y complementan los servicios proporcionados por el ORB. Todos ellos son accesibles a través de un conjunto de interfaces especificadas en IDL. Se relacionan, a continuación, los servicios incluidos en el estándar:

- *Life Cycle Service*: define las operaciones para crear, copiar, mover y eliminar objetos.
- *Persistence Service*: proporciona una interfaz para almacenar objetos de manera persistente en una amplia variedad de servidores de almacenamiento: bases de datos orientadas a objetos (ODBMSs), bases de datos relacionales (RDBMSs) y ficheros tradicionales.
- *Naming Service*: permite a los objetos localizar otros objetos, incluso utilizando servicios de directorio como X.500, NIS+, NDS, LDAP, etc.
- *Event Service*: permite a los objetos registrar dinámicamente su interés por eventos específicos. Este servicio define el objeto denominado *event channel*, que recolecta y distribuye eventos entre los objetos, que no necesitarán conocerse entre sí.
- *Concurrency Control Service*: proporciona un gestor de cerraduras (*lock*, en inglés).
- *Transaction Service*: proporciona una coordinación de dos fases (*two-phase commit*) utilizando transacciones planas o anidadas.
- *Relationship Service*: proporciona una forma de crear asociaciones dinámicas entre objetos que no se conocen entre sí.
- *Externalization Service*: proporciona una manera estándar de introducir/enviar al exterior datos de un objeto utilizando un mecanismo de tipo flujo de bytes (*stream*).
- *Query Service*: proporciona operaciones de consulta sobre objetos. Es un superconjunto de SQL.

- *Licensing Service*: proporciona operaciones para medir la utilización de objetos para asegurar una compensación justa por su uso. Permite controlar el coste por sesión, por nodo, por creación de instancia, etc.

- *Properties Service*: proporciona operaciones que permiten asociar valores con nombre (propiedades) a cualquier objeto.

- *Time Service*: proporciona interfaces para sincronizar relojes en un entorno de objetos distribuidos. También proporciona operaciones para definir y gestionar eventos dependientes del tiempo.

- *Security Service*: proporciona seguridad en el entorno de objetos distribuidos, soportando autenticación, listas de control de acceso, confidencialidad, etc.

- *Trader Service*: proporciona "páginas amarillas" para los objetos. Permite a los objetos publicitar sus servicios y pujar por la realización de trabajos.

- *Collection Service*: proporciona interfaces para crear y manipular las collecciones más comunes.

4.2 Construcción de aplicaciones con CORBA y JAVA

A continuación se describe el proceso de construcción de una aplicación consistente en la interacción entre un cliente y un servidor.

1. Definir la interfaz remota. Se define, en primer lugar, la interfaz del objeto remoto en IDL. Dicha interfaz permitirá generar, de manera automática, el código fuente del *stub* y el *skeleton* (ver más arriba) así como todo el código necesario para comunicarse con el ORB. Evidentemente, en el caso de que sólo se fuera a implementar el cliente porque el servidor ya existiese, se tendría que proporcionar el fichero IDL correspondiente a la interfaz que expone el servidor.
2. Compilar la interfaz remota. El compilador *idlj* genera todo el código fuente mencionado en el paso anterior.
3. Implementar el servidor. A partir de los esqueletos que genera el compilador *idl* es sencillo implementar el servidor. Además de los métodos que implementan la interfaz remota, el código del servidor implementa un mecanismo para arrancar el ORB y esperar por una invocación de un cliente.
4. Implementar el cliente. De una manera similar al servidor, el cliente hace uso de los stubs generados en el paso 2. El cliente se basa en el stub para arrancar su ORB, encontrar el servidor utilizando el servicio de nombrado, obtener una referencia al objeto remoto e invocar sus métodos.
5. Arrancar los programas. Una vez está todo implementado, se arranca el servicio de nombrado, el servidor y, finalmente, el cliente

4.2.1 Ejemplo

Se describe, a continuación, el proceso de construcción del ejemplo típico "Hola mundo". El programa "Hola mundo" tendrá una sola operación, que consiste en devolver la cadena de caracteres a mostrar.

4.2.1.1 Escritura de la interfaz

La interfaz IDL juega el papel de contrato entre el cliente y el servidor, especificando qué operaciones y qué atributos están disponibles.

Para el ejemplo, crear un fichero denominado *Hola.idl* y escribir en él el siguiente texto:

```
module AplicacionHola {
              interface Hola
              {
                  string diHola();  // Única operación
              };
};
```

Un módulo (*module*) CORBA es algo así como un paquete (*package*) de Java. Actúa como un contenedor para interfaces y declaraciones. Una vez compilado, se generará un paquete Java para cada módulo CORBA.

La interfaz (*interface*) especifica el contrato entre cliente y servidor. Cuando se compile el fichero, se generará una interfaz Java a partir de la correspondiente CORBA.

Por su parte, las operaciones CORBA se convertirán en métodos Java.

Compilación de la interfaz

Desde el intérprete de comandos ejecutamos el compilador con la siguiente sintaxis:

```
> idlj -fall Hola.idl
```

Si se echa un vistazo al contenido del directorio en el que reside *Hola.idl* se puede comprobar que se ha creado un subdirectorio denominado *AplicacionHola* que contiene varios ficheros. El fichero *Hola.java* tiene la forma siguiente:

```
package AplicacionHola;
/**
* AplicacionHola/Hola.java
* Generated by the IDL-to-Java compiler (portable), version "3.0"
* from Hola.idl
* miércoles 23 de mayo de 2001 12H46' CEST
*/


public interface Hola extends HolaOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity
{
} // interface Hola
```

Con una interfaz tan sencilla es fácil deducir las correspondencias que existen entre sentencias IDL y sentencias Java:

Sentencia IDL	Sentencia Java
`module AplicacionHola`	`package AplicacionHola;`
`interface Hola`	`public interface Hola`
`string diHola();`	`String diHola();`

Interpretación de los ficheros resultado de la compilación de la interfaz

Los 5 ficheros generados por *idltojava* son:

- *_HolaImplBase.java*: esta clase abstracta es el esqueleto del servidor, que proporciona funcionalidad básica CORBA al mismo. Implementa la interfaz *Hola.java*.

- *_HolaStub.java*: esta clase es el stub del cliente y proporciona funcionalidad CORBA para él. Implementa la interfaz *Hello.java*.
- *Hola.java*: esta interfaz contiene la versión Java de la interfaz IDL. Contiene el método *diHola*. Además, deriva de org.omg.CORBA.Object, proporcionando también funcionalidad CORBA.
- *HolaHelper.java*: esta clase final proporciona funcionalidad auxiliar, en particular el método *narrow* necesario para convertir referencias a objetos CORBA en sus tipos apropiados.
- *HolaHolder.java*: esta clase final mantiene una instancia pública de tipo *Hola*.

4.2.1.2 Desarrollo del programa cliente

Un cliente CORBA en Java es como cualquier otro programa Java. Es necesario importar los paquetes de librería (*packages*) necesarios, declarar la clase aplicación, definir un método inicial (*main*) y escribir código para gestionar las excepciones.

Importación de paquetes

En un fichero denominado *ClienteHola.java*, escribir el código siguiente:

```
import AplicacionHola.*;    // Paquete que contiene los stubs
import org.omg.CosNaming.*; // ClienteHola utilizara el
                            // servicio de nombrado
import org.omg.CORBA.*;     // Necesario en todas las
                            // aplicaciones CORBA
```

Declaración de la clase cliente

En el mismo fichero, declarar la clase cliente:

```
public class ClienteHola {
      // Añadir aquí el método main en el paso siguiente
}
```

Definir el método inicial

Toda aplicación Java necesita el método *main*. Se declarará en el ámbito de la clase *ClienteHola*:

```
public static void main(String args[]) {
      // Añadir aquí el bloque try-catch en el paso siguiente
}
```

Gestionar las excepciones CORBA

Dado que todos los programas CORBA pueden generar excepciones CORBA en tiempo de ejecución, es habitual introducir todo el código del método *main* en un bloque *try-catch*.

```
try {
     // Añadir el resto del código de ClienteHola aquí
} catch(Exception e) {
     System.out.println("ERROR : " + e);
     e.printStackTrace(System.out);
  }
```

Creación de un objeto ORB

Un cliente CORBA necesita un objeto ORB para realizar su trabajo. Todo cliente instancia un objeto *org.omg.CORBA.ORB* y lo inicializa pasándole cierta información acerca de sí mismo.

Dentro del bloque *try-catch* anterior, declarar e inicializar la variable ORB:

```
ORB orb = ORB.init(args, null);
```

Localización del servidor Hola

Una vez que el cliente tiene un ORB, puede pedirle que localice el servicio que necesita: en este caso, el servidor *Hola*. Hay varias formas en las que un cliente CORBA puede obtener una referencia inicial a un objeto; en el ejemplo se utilizará el **COS** (CORBA Object Services) **Naming Service** especificado por OMG y proporcionado con Java IDL.

PASO 1: El primer paso en la utilización del servicio de nombrado consiste en **obtener el contexto de nombres inicial**. En el bloque *try-catch,* después de la inicialización del ORB, se tiene que llamar a *orb.resolve_initial_references(“NameService”)* para obtener una referencia a objeto del servidor de nombres:

```
org.omg.CORBA.Object objRef =
                orb.resolve_initial_references(“NameService”);
```

La cadena de caracteres “NameService” está definida para todos los ORBs CORBA. Cuando se le pasa dicha cadena, el ORB devuelve el contexto de nombres inicial, una referencia a objeto del servidor de nombres.

PASO 2: convertir (*narrow*) **la referencia a objeto**. Como ocurre con todas las referencias a objetos CORBA, *objRef* es un objeto CORBA genérico. Para utilizarlo como un objeto *NamingContext* hay que convertirlo a dicho tipo:

```
NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

El objeto *ncRef* es ahora un *org.omg.CosNaming.NamingContext* y se puede utilizar ya para acceder al servicio de nombrado y encontrar otros servicios.

PASO 3: encontrar el servicio. Los nombres pueden tener diferentes estructuras, en función de la implementación del servicio de nombrado. Es por eso que los servidores de nombres de CORBA gestionan nombres complejos utilizando objetos *NameComponent* (componente de un nombre). Cada *NameComponent* contiene una sola parte o elemento del nombre completo. Un array de objetos *NameComponent* puede contener un camino completo a un objeto en un sistema de ficheros de un ordenador.

Para encontrar al servidor *Hola* es necesario un *NameComponent* que contenga la cadena de caracteres que lo identifica.

```
NameComponent nc = new NameComponent(“Hola”,””);
```

La sentencia anterior asigna al campo *id* de *nc* el valor “Hola” y al campo *kind*, una cadena vacía.

Dado que el camino al objeto *Hola* tiene un solo elemento, se crea a continuación un array de un solo elemento. El método *NamingContext.resolve* utilizará dicho array:

```
NameComponent path[] = {nc};
```

Finalmente, se pasa *path* al método *resolve* del servicio de nombrado para obtener una referencia a objeto del servidor Hola y se convierte (*narrowing*) en un objeto *Hola*:

```
Hola holaRef = HolaHelper.narrow(ncRef.resolve(path));
```

Aquí se puede ver la utilidad de la clase *HolaHelper*. El método *resolve* devueve un objeto CORBA genérico que debe ser, por tanto, convertido (*narrow*) en un objeto *Hola*, referencia necesaria para completar el trabajo.

Invocación del método diHola

Las invocaciones CORBA tienen idéntico aspecto a las invocaciones a objetos locales. Toda la complejidad del empaquetado de los datos, su envío, desempaquetado en el servidor e invocación del método del objeto servidor se realizan de manera transparente para el programador del objeto cliente.

Dentro del bloque *try-catch* de *ClienteHola.java*, introducir el siguiente código:

```
String hola = holaRef.diHola();
System.out.println(hola);
```

4.2.1.3 Desarrollo del programa servidor

La estructura de un programa servidor es la misma que la de cualquier aplicación Java. Se importan los paquetes necesarios, se declara la clase servidora, se define el método *main* y se escribe código para gestionar las excepciones.

Importación de paquetes

En un fichero denominado *ServidorHola.java*, escribir el código siguiente:

```
import AplicacionHola.*;    // Paquete que contiene los stubs
import org.omg.CosNaming.*; // ServidorHola utilizara el
                            // servicio de nombrado
import org.omg.CosNaming.NamingContextPackage;
                            // Paquete que contiene excepciones
                            // especiales lanzadas por el
                            // servicio de nombrado
import org.omg.CORBA.*;     // Necesario en todas las
                            // aplicaciones CORBA
```

Declaración de la clase servidora

En el mismo fichero, declarar la clase servidora:

```
public class ServidorHola {
      // Añadir aquí el método main en el paso siguiente
}
```

Definir el método inicial

Toda aplicación Java necesita el método *main*:

```
public static void main(String args[]) {
      // Añadir aquí el bloque try-catch en el paso siguiente
}
```

Gestionar las excepciones CORBA

Dado que todos los programas CORBA pueden generar excepciones CORBA en tiempo de ejecución, es habitual introducir todo el código del método *main* en un bloque *try-catch*.

```
try {
     // Añadir el resto del código de ServidorHola aquí
```

```
} catch(Exception e) {
     System.out.println("ERROR : " + e);
     e.printStackTrace(System.out);
  }
```

Creación de un objeto ORB

Como el cliente, un servidor CORBA necesita un objeto ORB local. Todo cliente instancia un objeto *org.omg.CORBA.ORB* y registra sus objetos sirvientes de manera que el ORB puede encontrar el servidor cuando recibe una invocación para él.

Dentro del bloque *try-catch* anterior, declarar e inicializar la variable ORB:

```
ORB orb = ORB.init(args, null);
```

Gestión del objeto sirviente

Un servidor es un proceso que instancia uno o más objetos sirvientes. El sirviente implementa la interfaz generada por *idlj* y realiza realmente el trabajo especificado por las operaciones de dicha interfaz. *ServidorHola* necesitará un *SirvienteHola*.

PASO 1: **instanciación del objeto sirviente**. Dentro del bloque *try-catch*, inmediatamente después de la llamada a *init*, instanciar el objeto sirviente:

```
SirvienteHola  holaRef = new SirvienteHola();
```

Esta clase sirviente no está definida aún; se hace en el paso siguiente. A continuación se conecta el sirviente con el ORB, de manera que el ORB podrá reconocer invocaciones al mismo y pasárselas al sirviente correcto:

```
orb.connect(holaRef);
```

PASO 2: **definición de la clase sirviente**. Al final del fichero *ServidorHola.java*, fuera de la clase *ServidorHola*, definir la clase para el objeto sirviente:

```
class SirvienteHola extends _HolaImplBase {
    public String diHola() {
        return “\nHola, mundo!!\n”;
    }
}
```

Como se puede comprobar, el sirviente es una subclase de *_HelloImplBase*, de manera que hereda toda la funcionalidad general CORBA generada por el compilador

Utilización del servicio de nombrado (COS Naming Service)

El objeto *ServidorHola* trabaja con el servicio de nombrado para hacer disponibles a los clientes las operaciones del objeto sirviente. El servidor necesita una referencia a objeto al servicio de nombrado, de manera que pueda registrarse a sí mismo y asegurar que las invocaciones a la interfaz Hola sean enrutadas a su objeto sirviente.

PASO 1: obtener el contexto de nombres inicial. En el bloque *try-catch*, después de la inicialización del ORB, se tiene que llamar a *orb.resolve_initial_references* para obtener una referencia a objeto del servidor de nombres:

```
org.omg.CORBA.Object objRef =
                orb.resolve_initial_references(“NameService”);
```

La cadena de caracteres “NameService” está definida para todos los ORBs CORBA. Cuando se le pasa dicha cadena, el ORB devuelve el contexto de nombres inicial, una referencia a objeto del servidor de nombres.

PASO 2: convertir (*narrow*) **la referencia a objeto**. Como ocurre con todas las referencias a objetos CORBA, *objRef* es un objeto CORBA genérico. Para utilizarlo como un objeto *NamingContext* hay que convertirlo a dicho tipo:

```
NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

El objeto *ncRef* es ahora un *org.omg.CosNaming.NamingContext* y se puede utilizar ya para acceder al servicio de nombrado y encontrar otros servicios.

PASO 3: registrar el sirviente en el servidor de nombres. En primer lugar, crear un *NameComponent* que contenga la cadena de caracteres que lo identifica.

```
NameComponent nc = new NameComponent("Hola","");
```

La sentencia anterior asigna al campo *id* de *nc* el valor "Hola" y al campo *kind*, una cadena vacía. Dado que el camino al objeto *Hola* tiene un solo elemento, se crea a continuación un array de un solo elemento. El método *NamingContext.resolve* utilizará dicho array:

```
NameComponent path[] = {nc};
```

Finalmente, se pasa *path* y el objeto sirviente al servicio de nombrado, de manera que queden asociados objeto sirviente y el identificador "Hola":

```
ncRef.rebind(path, holaRef));
```

Ahora, cuando el cliente invoque *resolve("Hola")* sobre el contexto de nombres inicial, el servicio de nombrado devolverá una referencia al sirviente *Hola*.

Espera por invocaciones

El servidor ya está listo. Sólo necesita esperar a que el cliente solicite sus servicios. Para conseguirlo, introducir el código siguiente al final (y dentro de) el *bloque try-catch*:

```
java.lang.Object sync = new java.lang.Object();
synchronized(sync) {
    sync.wait();
}
```

Esta forma de *Object.wait* requiere que *ServidorHola* esté vivo hasta que llegue una invocación del ORB. Dada su ubicación en *main*, después de que una invocación haya finalizado y *diHola* retornado, el servidor esperará de nuevo.

4.2.1.4 Compilación y ejecución de la aplicación

Compilación del programa cliente

```
javac ClienteHola.java AplicacionHola\*.java
```

Compilación del programa servidor

```
javac ServidorHola.java AplicacionHola\*.java
```

Ejecución de la aplicación

Desde un intérprete de comandos, lanzar el servidor de nombres:

```
tnameserv -ORBInitialPort 2222
```

Desde un segundo intérprete de comandos, lanzar el servidor Hola:

```
java ServidorHola -ORBInitialHost nameserverhost
                  -ORBInitialPort 2222
```

Desde un tercer intérprete de comandos, lanzar el cliente:

```
java ClienteHola -ORBInitialHost nameserverhost
```

```
                                -ORBInitialPort 2222
```

El cliente deberá mostrar el mensaje siguiente:

```
Hola, mundo!!
```

CAPÍTULO 5 RMI

5.1 Introducción

RMI (*Remote Method Invocation*) [Sun98] fue diseñado para permitir la invocación de métodos remotos de objetos entre distintas máquinas Java (ubicadas en el mismo o en distintos computadores) de manera transparente. Integra directamente un modelo de objetos distribuidos en el lenguaje Java a través de un conjunto de clases e interfaces.

5.2 Objetos RMI

Un **objeto RMI** es un objeto cuyos métodos pueden ser invocados desde otra máquina Java, incluso a través de una red. Cada objeto remoto implementa una o más **interfaces remotas** que especifican qué operaciones pueden ser invocadas por los clientes. Cualquier otra operación pública que tenga el objeto pero que no aparezca en la interfaz remota no podrá ser utilizada por los clientes remotos. Los clientes invocan dichos métodos exactamente igual que si fueran métodos locales, quedando ocultos los detalles de la comunicación. Se utilizan los denominados **sustitutos** (*stub*) y **esqueletos** (*skeleton*), que actúan de intermediarios entre los objetos local y remoto. Los sustitutos y los esqueletos son generados de manera automática por el compilador *rmic*.

5.3 Modelo de programación

Desde el punto de vista del programador, los objetos remotos se manipulan de la misma manera que los locales, a través de referencias. Realmente, una **referencia** a un objeto remoto apunta a una referencia a un sustituto local que lo representa, y que es el encargado de transmitir los parámetros de la invocación al computador remoto y recibir de él el valor de retorno. La manipulación de objetos remotos tiene que tener en cuenta:

- Cómo obtener una referencia a un objeto remoto.
- Gestión de excepciones específicas de la invocación de métodos remotos.

Por su parte, un esqueleto es el encargado de recoger los parámetros que recibe del sustituto a través de la red, invocar de manera local al objeto remoto y devolver el resultado de nuevo a través de la red al sustituto del objeto que realizó la invocación.

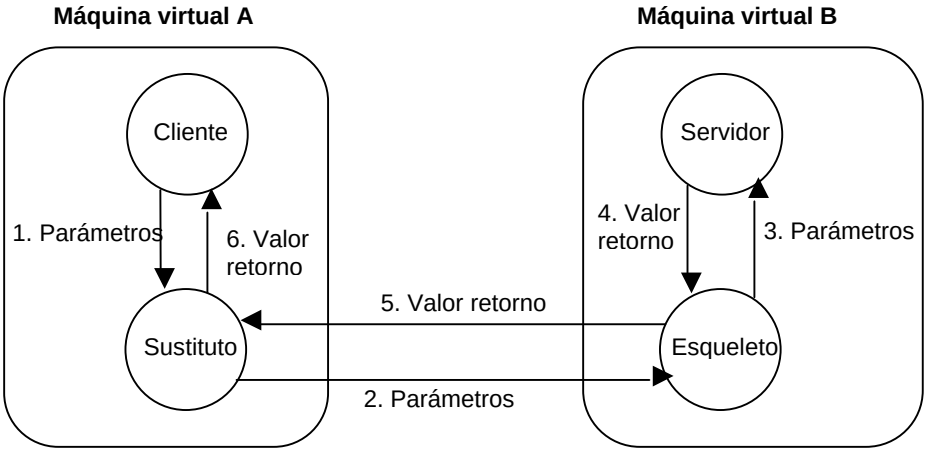


Figura 4.9. Uso de sustitutos y esqueletos en la invocación remota.

A diferencia de una invocación local, una invocación RMI pasa los objetos locales que forman parte de la lista de parámetros por copia (por valor), dado que una referencia a un objeto local sólo sería útil en una máquina virtual única. RMI utiliza el **servicio de serialización de objetos** para aplanar el estado de un objeto local y colocarlo en el mensaje a enviar a la máquina virtual remota. Si el objeto es no-serializable, no se puede usar como parámetro. También son pasados por valor los parámetros de tipos primitivos. Por otro lado, RMI pasa los objetos remotos (objetos que implementan una interfaz remota) por referencia.

5.4 Servicios

RMI proporciona interfaces y clases para buscar objetos remotos, cargarlos y ejecutarlos de manera segura. Adicionalmente, proporciona un **servicio de nombrado** un tanto primitivo (servicio de nombrado no persistente y espacio de nombres plano) que permite localizar objetos remotos y obtener referencias a ellos, de manera que se puedan invocar sus métodos.

La utilización de objetos remotos lleva consigo la aparición de nuevas situaciones de error, de tal forma que el programador deberá manejar el conjunto de nuevas excepciones que pueden ocurrir durante una invocación remota.

RMI incluye una característica de recolección de basura distribuida, que recolecta aquellos objetos servidores que no son referenciados por ningún cliente de la red.

5.5 Construcción de aplicaciones con RMI

A continuación se describe el proceso de construcción de una aplicación consistente en la interacción entre un cliente y un servidor.

6. Definir la interfaz remota. La interfaz remota especificará los métodos accesibles de manera remota por los clientes.
7. Implementación de la interfaz remota (o lo que es lo mismo, implementar el servidor). Un objeto RMI remoto debe implementar, al menos, una interfaz remota. También puede implementar otras interfaces (locales y/o remotas) así como métodos públicos (que sólo estarán disponibles de manera local).

8. Implementar el cliente. El cliente sólo podrá ser implementado una vez que las interfaces remotas han sido definidas, dado que utilizarán los métodos disponibles a través de ellas.
9. Generar los sustitutos (*stubs*) y esqueletos (*skeletons*) para clientes y servidores, respectivamente.
10. Arrancar los programas. Se arrancarán, en este orden, el servicio de nombrado, el servidor y el cliente.

5.5.1 Ejemplo

Se describe, a continuación, el proceso de construcción del ejemplo típico “Hola mundo”. El programa “Hola mundo” tendrá una sola operación, que consiste en devolver la cadena de caracteres a mostrar.

5.5.1.1 Escritura de la interfaz remota

La interfaz remota especifica el conjunto de métodos que pueden ser invocados remotamente por los clientes:

- Tiene que ser declarada pública.
- Tiene que extender la interfaz *java.rmi.Remote*.
- Todos sus métodos deben lanzar la excepción *java.rmi.RemoteException*.
- Si los métodos remotos utilizan como parámetros o resultados objetos remotos, deberán ser interfaces y no clases.

El ejemplo definirá una interfaz remota como la siguiente:

```
public interface InterfazHola extends java.rmi.Remote {
    // Este método será invocado por clientes remotos
    public String diHola() throws java.rmi.RemoteException;
}
```

5.5.1.2 Implementación de la interfaz remota

Se trata, de alguna manera, de implementar la clase que proporciona la funcionalidad del servidor. Dicha clase (que implementa la interfaz remota) debería, como mínimo:

- Declarar las interfaces remotas a implementar.
- Definir un constructor para el objeto remoto.
- Proporcionar una implementación para cada método remoto de las interfaces remotas

Para que el objeto remoto del servidor sea accesible para los clientes, es necesario que:

- Se creen una o más instancias del objeto remoto.
- Al menos una de dichas instancias se registre en el servicio de nombrado de RMI (RMI *registry*).

Adicionalmente, para que un objeto pueda presentar comportamiento remoto debe extender la clase *java.rmi.RemoteServer* o una de sus subclases (por ejemplo,

java.rmi.server.UnicastRemoteObject, que define un esquema de objetos servidores no replicados que son válidos únicamente si el proceso servidor está vivo).

El código del servidor quedaría de la forma siguiente:

```
import java.rmi.*
import java.rmi.server.*;
public class ServidorHola extends UnicastRemoteObject implements
InterfazHola {

    public ServidorHola() throws RemoteException {
        super();  // Llamada al constructor de su superclase, que
                  // pone en conocimiento del motor de RMI en
                  // tiempo de ejecución de que el objeto remoto
                  // existe y puede aceptar invocaciones
    }

    public String diHola() throws RemoteException {
        return "\nHola, mundo!!\n";
    }

    public static void main(String[] args) {
        try {
           // Instanciar el objeto
           ServidorHola obj = new ServidorHola();
           // Registrar el objeto en el servicio de nombrado
           Naming.rebind("/ServidorHola", obj);
           System.out.println("Objeto ServidorHola creado\n");
        } catch (Exception e) { System.err.println(e); }
    }
}
```

5.5.1.3 Implementación del cliente

El cliente que hace uso del objeto remoto es muy sencillo: en primer lugar, tiene que obtener una referencia al objeto remoto consultando el servicio de nombrado. Una vez dispone de la referencia, sólo tiene que invocar el método deseado (en este caso, sólo hay definido un método).

```
import java.rmi.*;

public class ClienteHola {

    public static void main(String[] args) {
        if (System.getSecurityManager() == null)
           System.setSecurityManager(new RMISecurityManager());
        try {
           InterfazHola obj =
               (InterfazHola) Naming.lookup("/ServidorHola");
           String mensaje = obj.diHola();
           System.out.println(mensaje);
        } catch (Exception e) {
```

```
            System.out.println("Excepcion en el cliente " + e);
          }
      }
}
```

En el código del cliente todo es bastante normal (hasta esperable) excepto, quizá, la utilización de un gestor de seguridad (*security manager*). El gestor de seguridad RMI proporciona un contexto de seguridad bajo el cual se ejecutan las aplicaciones RMI. Si no se crea uno, los sustitutos (*stubs*) y las clases únicamente pueden ser cargadas desde el *classpath* local. Con el fin de no complicarse la vida, será necesario utilizar un fichero de políticas de seguridad con un contenido similar a este:

```
grant {
  // Permitirlo todo
  permission java.security.AllPermission;
};
```

5.5.1.4 Compilación del cliente y el servidor

La compilación de ambos se realiza de la manera siguiente:

```
javac ClienteHola.java
javac ServidorHola.java
```

5.5.1.5 Generación de sustitutos y esqueletos

La generación de los sustitutos (stubs) y esqueletos (skeletons) es inmediata. Se realiza utilizando el compilador rmic aplicado sobre el fichero que describe el servidor:

```
rmic ServidorHola
```

Obviamente, este paso se debe dar una vez haya tenido éxito la compilación del servidor.

5.5.1.6 Ejecución de la aplicación

Ejecución de la aplicación

Desde un intérprete de comandos, lanzar el servidor de nombres:

```
rmiregistry
```

Desde un segundo intérprete de comandos, lanzar el servidor Hola:

```
java ServidorHola
```

Desde un tercer intérprete de comandos, lanzar el cliente:

```
java ClienteHola –Djava.security.policy=hola.policy
```

El cliente deberá mostrar el mensaje siguiente:

```
Hola, mundo!!
```