



JAVA Y LAS REDES

(1^a Parte de 3)

**Introducción a las redes, a java.io, java.zip, java.net,
java.nio, a RMI y a CORBA**

Miguel Ángel Abián

JAVA Y LAS REDES

(1^a Parte de 3)

Resumen: En este tutorial, dividido en tres partes, se presenta un panorama general de las comunicaciones en red mediante Java y de la E/S de Java. Para ello se explican las redes, los paquetes java.io, java.nio, java.net y java.zip, así como RMI y CORBA, desde un planteamiento sinérgico y pragmático; pero permitiendo la consulta independiente de los apartados dedicados a cada paquete. Se presentan muchos ejemplos del funcionamiento de esos paquetes (un navegador, un servidor HTTP, ejemplos de E/S, de RMI, de CORBA, etc.) y el ejemplo principal de un chat (con java.io/CORBA, con java.io/java.net y, después, con java.nio/java.net).

Abstract: In this tutorial, divided in three parts, a general view of network communications using Java and Java I/O is presented. For this, networks, the packages java.io, java.nio, java.net and java.zip, and also RMI and CORBA, are explained from a pragmatic and synergic point of view; but allowing the independent consultation of the sections dedicated to each package. Many examples of code using these packages are presented (a browser, a HTTP server, I/O examples, RMI examples, CORBA examples, etc), together with the main example of a chat application (using java.io/CORBA, using java.io/java.net and, after, using java.nio/java.net).

Keywords: java.nio, java.io, java.zip, java.rmi, java.net, NIO, protocols, API Socket, layer, TCP/IP, OSI, FTP, SMTP, POP3, HTTP, CGI, Unicode, UTF-8, UTF-16, socket, interoperability, client-server, distributed computing, distributed objects, client-server, [WebMethod], N-tier, RMI, RMI registry, remote invocation, CORBA, POA, servants, CORBA objects, IOR, transient IOR, persistent IOR, web services, server socket, non-blocking sockets, channels, buffers, chat

La imagen de la página anterior corresponde a un goteado de Jackson Pollock y el copyright pertenece a sus herederos o a cualquier institución que los represente. Se reproduce sin ánimo de lucro.

ÍNDICE DE LA PRIMERA PARTE

1. Introducción	Página 5
2. Fundamentos de las comunicaciones en red	Página 15
2.1. Algunas definiciones	Página 15
2.2. Protocolos de comunicaciones	Página 22
2.3. TCP/IP: Un modelo de capas	Página 23
2.4. TCP/IP: Un conjunto de protocolos	Página 33
2.5. El modelo de referencia OSI estaba vestido para el éxito, pero el éxito no le llegó	Página 49
2.6. El principio de “extremo a extremo”	Página 52
2.7. Sockets	Página 55
2.7.1. Introducción. Todo empezó en UNIX	Página 55
2.7.2. Sockets. Tipos de sockets	Página 56
2.7.3. Un ejemplo de sockets en C	Página 67
2.7.4. Ventajas e inconvenientes de los sockets	Página 69
2.8. Algunos servicios de la capa de aplicación en la arquitectura TCP/IP	Página 70
2.8.1. Introducción	Página 70
2.8.2. El servicio de administración de redes	Página 70
2.8.3. El servicio de transferencia de archivos	Página 71
2.8.4. El servicio de correo electrónico	Página 77
2.8.5. La World Wide Web	Página 90
3. El paquete java.io de Java	Página 100
3.1. La clase java.io.File	Página 101
3.2. La jerarquía de clases java.io.InputStream	Página 106
3.3. La jerarquía de clases java.io.OutputStream	Página 117
3.4. Cuando un byte no basta: comunicaciones en un mundo plurilingüe	Página 131
3.5. La clase java.io.InputStreamReader	Página 139
3.6. La clase java.io.OutputStreamWriter	Página 141
3.7. La clase java.io.BufferedReader	Página 143
3.8. La clase java.io.BufferedWriter	Página 146
3.9. La clase java.io.PrintWriter	Página 148
3.10. El patrón decorador y el paquete java.io	Página 150
4. Aplicaciones y sistemas distribuidos	Página 156
4.1. Introducción. De los mainframes a los sistemas distribuidos	Página 156
4.2. Aplicaciones y sistemas distribuidos	Página 175
4.3. Dos ejemplos de arquitecturas distribuidas: CORBA y los servicios web	Página 180

5. RMI: Llamando desde lugares remotos	Página 189
5.1. Fundamentos de la RMI: el modelo de objetos distribuidos de Java	Página 189
5.2. Anatomía de las aplicaciones RMI: adaptadores y esqueletos. La arquitectura RMI. El servicio de registro remoto RMI	Página 194
5.2.1. Anatomía de las aplicaciones RMI: adaptadores y esqueletos	Página 194
5.2.2. La arquitectura RMI	Página 203
5.2.3. El servicio de registro remoto RMI	Página 205
5.3. Recorrido rápido por el paquete java.rmi	Página 209
5.4. Ejemplo completo del desarrollo e implementación de una aplicación con RMI	Página 215
5.5. Distribución de las aplicaciones RMI: carga dinámica de clases con RMI	Página 230
5.6. Ventajas e inconvenientes de la RMI	Página 248
6. CORBA: Llamando desde más lejos aún. CORBA y Java	Página 250
6.1. Introducción. ¿Para qué se usa CORBA?	Página 250
6.2. El modelo de objetos de CORBA	Página 252
6.3. Vocabulario básico de CORBA	Página 256
6.4. Dentro de la arquitectura de CORBA	Página 259
6.5. CORBA y RMI: puntos de intersección, puntos de fuga	Página 282
6.6. El problema del océano Pacífico: náufragos en los mares de CORBA	Página 285
6.7. Java y CORBA: un par de ejemplos	Página 286

JAVA Y LAS REDES

Introducción a las redes, a java.io, java.zip, java.net, java.nio, a RMI y a CORBA

(1^a Parte de 3)

Fecha de creación: **27.07.2004**

**Miguel Ángel Abián
mabian AT aidima DOT es**

Copyright (c) 2004, Miguel Ángel Abián. Este documento puede ser distribuido sólo bajo los términos y condiciones de la licencia de Documentación de javaHispano v1.0 o posterior (la última versión se encuentra en <http://www.javahispano.org/licencias/>).

Many web generations ago, the ARPA knights started a revolution against the telco circuit-switching empire and determined to destroy the death stars and their twisted pairs. I was one of those knights when our leader Vint Cerf, Father of the Internet, crossed over to the telco side of the force. Cerf Vader and legions of imperial stormlawyers are now defending the death stars against the insignificant ispwooks. The previous speaker in this forum series, the Father of the Web, Tim-Berners-Lee-Po --who speaks over 6,000,000,000 dialects-- has been captured by Java the Hutt. You are Luke and Leia. The death stars must be destroyed and I foresee it.

Bob Metcalfe, padre de Ethernet, rememora en clave cómica las batallas contra AT&T

La decisión de hacer de la Web un sistema abierto fue necesaria para que fuese universal. Si hubiésemos patentado la tecnología, probablemente, no hubiera despegado. No puedes proponer que algo sea un espacio universal si, al mismo tiempo, mantienes el control sobre ello.

Tim Berners-Lee

Hay dos grandes productos que vienen de Berkeley: LSD y [Unix] BSD. No creemos que esto sea una coincidencia.

Jeremy S. Anderson

La red no es nada más que una masa inerte de metal, plástico y arena. Somos los únicos seres vivientes en el ciberespacio.

Richard Barbrook

1.- Introducción

El propósito de este tutorial, dividido en tres partes, es desarrollar un panorama general de las comunicaciones en red mediante Java, tanto para JDK 1.2-1.3 como para JDK 1.4-1.5 (el JDK 1.5 se conoce ahora como 5.0), excluyendo los *applets*. Para ello se explicarán con detalle cuatro paquetes de Java, así como las tecnologías RMI y CORBA:

- **java.io**
- **java.zip**
- **java.net**
- **java.nio (incorporado con JDK 1.4 y mantenido en JDK 1.5 o 5.0)**

Debido a como Java trata las comunicaciones en red, bien podría este tutorial titularse “**Entrada y salida en Java**”. Parte de mi motivación para escribirlo reside en que no existe todavía ningún libro que trate de forma sinérgica los cinco paquetes (al menos, yo no he podido encontrarlo; mi búsqueda se ha limitado a libros en castellano, inglés y francés). Los pocos textos que abordan de forma precisa los paquetes `java.nio` o `java.rmi` no suelen presentar con detalle los otros, y viceversa. Aparte, los pocos libros en castellano que proclaman cubrir las novedades de JDK 1.4 (oficialmente, *Java 2 SDK standard version 1.4*) caen de pleno en la categoría de *Realidad, falta de adecuación a...*.

Doy por hecho que el lector conoce la sintaxis de Java, las estructuras básicas de control y los hilos (*threads*). Si no es así, puede recurrir a los tutoriales de javaHispano. Un buen comienzo son los tutoriales *Java básico con ejemplos*, de Abraham Otero, y *Threads*, de Scheme the API man (*sic*). Mi idea es dar unos cuantos ladrillos básicos para que cada uno pueda construir sus propios edificios virtuales, y proporcionar trucos y consejos que no suelen explicarse en los textos ni en las aulas.

En el **apartado 2** se introducen los fundamentos necesarios para entender los conceptos utilizados en redes: protocolos, arquitecturas, *sockets*, puertos, etc. Aunque el lector tenga conocimientos de redes, recomiendo su lectura para que sepa cuál es la terminología que usaré durante el resto del tutorial y su significado. Comprender qué son los datagramas, los protocolos TCP/UDP y la API Socket explica el comportamiento de las clases del paquete `java.net`, imprescindibles para desarrollar con Java aplicaciones en red. El trabajo que se invierte en entender bien el funcionamiento de las redes TCP/IP tiene sus dividendos: uno aprende enseguida a manejar `java.net`.

Este apartado lo he escrito en *pianissimo*, pues tenía que poner, una tras otra, las bases para llegar al concepto de *socket*, que resulta fundamental para las comunicaciones en red. Y no me refiero sólo a las comunicaciones en red con Java: este lenguaje usa el mismo modelo de *sockets* que el sistema operativo UNIX BSD, modelo que ha devenido estándar *de facto*. He omitido intencionadamente cualquier contenido relativo a la historia y desarrollo de Internet, pues resulta muy fácil acceder a esa información mediante la propia red de redes.

La última parte del apartado está dedicada a varios servicios de la capa de aplicación (administración de redes, transferencia de archivos, correo electrónico, *World Wide Web*). En el apartado 8 veremos qué clases tiene Java para trabajar con algunos de estos servicios.

En el **apartado 3** se explica el paquete `java.io`. Casi todas las clases del paquete `java.net` permiten abrir objetos `java.io.InputStream` o `java.io.OutputStream`, que pueden usarse directamente para el intercambio de datos. El trabajo en red con Java resulta asombrosamente sencillo: basta usar la clase apropiada de `java.net`, extraer de ella un flujo de entrada o salida y escribir en el flujo o leer de él. Dentro de apartado se hace especial hincapié en las clases derivadas de `java.io.Reader` y `java.io.Writer`, indispensables para permitir comunicaciones plurales en cuanto al lenguaje, y en el papel que desempeña el patrón decorador como estructura interna de `java.io`. Incluyo también algunos consejos para manejar este paquete (cómo cerrar flujos, cómo tratar posibles excepciones en la E/S, etc.), además de unos cuantos ejemplos de uso para las situaciones más frecuentes. Si el lector tiene

un buen conocimiento de este paquete, puede pasar directamente al siguiente apartado.

Los sistemas distribuidos se tratan en el **apartado 4**. Para comprender por qué son necesarios los modernos sistemas distribuidos, se parte de los sistemas monolíticos (basados en *mainframes* y terminales “tontos”) y se acaba en los ejemplos de CORBA y los servicios *web*. Entre el punto de inicio y el final, se explican las arquitecturas cliente-servidor de dos y tres capas, de N capas y se definen los conceptos en que se basan las aplicaciones distribuidas y los problemas más acuciantes con los que tienen que tratar.

Quizás mis opiniones finales sobre los servicios *web* o sobre el [WebMethod] de C# sean un tanto polémicas. Podría haberlas omitido y así no levantaría suspicacias sobre mis intereses u opiniones; pero **no tengo dios ni patria ni amo en cuanto a tecnologías informáticas**: carezco de cualquier interés por vender o promocionar un producto, o por predisponer al lector a favor de una o en contra de otra (los vendedores y promotores de los servicios *web*, que florecen cual almendros en primavera, no pueden decir lo mismo). Mis opiniones se basan en criterios técnicos, en comparaciones con otras tecnologías y en pruebas. La venta y la propaganda no me interesan; ya hay demasiada gente dedicada a ellas. No negaré que no hay ninguna tecnología perfecta, pero algunas son más imperfectas que otras.

La invocación remota de métodos (RMI: *Remote Method Invocation*) de Java se aborda en el **apartado 5**. El paquete `java.rmi` es demasiado extenso y complejo como para tratarlo de forma completa aquí; pero se exponen las clases e interfaces más usadas para desarrollar aplicaciones distribuidas. Supongo que los creadores de RMI, cuando la acabaron, harían lo mismo que Frank Sinatra tras interpretar una buena canción: aflojarse el nudo de la corbata, sonreír con picardía y encender un cigarrillo (lo último, dicho de paso, no es muy popular en estos tiempos). Es un paquete del cual los ingenieros de Sun pueden, con razón, sentirse orgullosos.

La RMI de Java es una obra maestra de la ingeniería del software; está bien diseñada, bien implementada y bien documentada. Por añadidura, funciona muy eficazmente y sin la complejidad de arquitecturas distribuidas como CORBA. Si se explica antes del paquete `java.net` es porque se basa en la serialización de objetos, explicada en el apartado anterior.

El **apartado 6** está dedicado a CORBA, una arquitectura para la construcción de sistemas distribuidos que ha influido mucho en todas las tecnologías distribuidas actuales. En este apartado se explica por qué usar CORBA, cuál es su estructura, cómo se integra con Java. Además, se presentan dos ejemplos del uso de CORBA con Java (el segundo corresponde a una aplicación de *chat*).

Sun ha basado en parte su J2EE en CORBA y se ha preocupado, junto con IBM, en conseguir la mayor integración posible entre los productos Java y la tecnología CORBA.

En el **apartado 7** se estudia el paquete `java.zip`; si bien éste no se utiliza en las comunicaciones en red tanto como se debería, resulta muy útil cuando se desea ahorrar tiempo de transmisión. Gracias a él, se puede usar GZIP o ZIP para comprimir la información que se envía a la red, ya sea mediante flujos de E/S o mediante el envío de objetos serializados. Cuanto mayor es la redundancia de los datos, mayor es la reducción del tiempo de transmisión y del tráfico en la red.

En el apartado 8 se explica el paquete `java.net`; todas las clases de este último paquete (`ServerSocket`, `Socket`, `DatagramSocket`, `URL`, etc.) se ilustran con muchos ejemplos (un primitivo navegador, un servidor HTTP, etc.). En el apartado 9 se presenta el ejemplo de una aplicación cliente-servidor de tipo *chat*.

En el apartado 10 se explican las novedades de `java.nio`, incluido por vez primera en el JDK 1.4, y se aborda su utilización para las comunicaciones en red. Conceptos como selectores, canales, buffers (instancias de las clases `ByteBuffer`, `IntBuffer`, `DoubleBuffer`, etc.) y sockets sin bloqueo se ven con detenimiento en dicho apartado. Cualquier programador que use Java para programar aplicaciones en red debería tener muy en cuenta el paquete `java.nio`, pues proporciona E/S sin bloqueo, característica que permite programar aplicaciones sumamente escalables usando *sockets*. Gracias a ella, se evitan los problemas derivados de usar múltiples hilos que aparecen cuando la E/S sí es bloqueante: sobrecarga del sistema, violaciones de la seguridad de los hilos, bloqueos, etc. En el apartado 11 se presenta el ejemplo de una aplicación cliente-servidor de tipo *chat* escrita con `java.nio`, y se compara con la versión del apartado 9.

Todo el código de los ejemplos puede conseguirse enviándome un correo (mabian AT aidima DOT es) con el asunto “Java y las redes”.

Desde luego, este tutorial no trata de unir dos puntos (las redes y Java) mediante un arabesco o un ocho; pero tampoco intente trazar una línea absolutamente recta, pues hay ideas y conceptos que no caen en el segmento recto entre dichos puntos, si bien son interesantes para entender por qué Java es como es y para desarrollar aplicaciones útiles y eficaces.

Incluir un apartado dedicado a `java.io` no es tarea ociosa. Si bien Java incluye el paquete `java.net` para trabajar en red, las entradas y salidas que corresponden a las conexiones mediante *sockets*, a la clase `URI` y a las clases `HTTP` acaban usando clases del paquete `java.io` (o de `java.nio`, si así lo desea el programador). Como el modelo de E/S utilizado por Java no presta atención al lugar de donde proceden los datos, resulta conveniente conocer bien unas cuantas clases del paquete `java.io`. Aparte, la mayor parte de los libros dedican poco espacio a las codificaciones de caracteres y a la forma en que Java trata los caracteres **Unicode**, características muy importantes a la hora de construir **aplicaciones en red para usuarios que usen distintos idiomas**. En algunos casos, las simplificaciones de los textos llevan a engaño. Así, en muchos sitios de la documentación oficial de Java –y de Windows– se afirma que Unicode es un sistema de codificación de dieciséis bits, lo cual es **manifestamente falso** (tal y como se verá en el apartado 3).

Por supuesto, dar una explicación completa y exhaustiva del paquete `java.nio` aquí queda fuera de lugar; pero se han tratado los aspectos más ligados con las comunicaciones en red. Pese a que no parece que haya despertado mucho interés en los programadores, es un paquete con muchísimas mejoras importantes con respecto a `java.io`, y que acerca a Java al sueño de ser un lenguaje tan válido como C/C++ para realizar computaciones complejas y programar controladores de dispositivos. Si nos restringimos al ámbito de las comunicaciones en red, su uso aporta sencillez al código y mejora espectacularmente el rendimiento (mejoras del 50%-80% son frecuentes).

Si bien cada apartado se ha escrito pensando en que refuerce a los otros, se pueden consultar independientemente, de modo que el lector puede espigar a su gusto. El lector que sólo quiera saber qué ofrece un paquete puede pasar directamente al apartado correspondiente. No obstante, para extraer el máximo provecho del tutorial recomiendo su lectura ordenada.

En varios apartados he incluido recuadros que incluyen avisos, advertencias curiosidades. Mi idea es advertir al lector de detalles que pueden devenir importantes o que son fuente común de errores o confusiones.

El dilema de escoger entre teoría y práctica se ha solventado aquí mezclando ambas a partes casi iguales, gramo más, gramo menos. No digo que sea la mejor manera posible, pero si es la menos mala que conozco. Se puede afirmar que hay que conocer a la perfección la teoría de redes para entender lo que hace Java (o C o C++), pero discrepo de esa opinión (si pensara así, no escribiría obras de divulgación). Aunque algunos piensen que un conocimiento a medias es peligroso, creo que esta postura, aparte de elitista y académica, anda errada: es preferible sentar unas bases, aun incompletas, siempre que no se desvirtúen los conceptos e ideas ni se mellen innecesariamente las aristas y bordes de la materia, a remitir de entrada a libros *definitivos*, que a menudo no pueden entenderse o valorarse hasta que uno ya tiene experiencia en la materia tratada. Situaciones como la de emplear libros formalistas y axiomáticos para enseñar termodinámica a alumnos que tienen el primer contacto con la materia se me antojan aberrantes: ninguna disciplina nace formada y axiomatizada (excepto las que ya nacen muertas). Olvidar para qué se desarrollan los conocimientos científicos y técnicos (dicho de otro modo: desdeñar qué problemas quieren resolver) es una de las tragedias de la Universidad española. No crean que dramatizo, que las tragedias son muchas: no se necesita la calculadora para evaluar los reconocimientos internacionales a la ciencia desarrollada en España. El único premio Nobel científico generado –a medias, pero ésa es otra (larga) historia– por el sistema universitario español data de 1906.

La primera versión de este tutorial, que termine en febrero de 2004, ocupaba más de cuatrocientas páginas. Sin embargo, no me satisfacía en absoluto: había prestado demasiada atención a las jerarquías de clases de los paquetes arriba mencionados, y el tutorial casi se había convertido en una recopilación exhaustiva de clases y métodos. Sin rodeos: me había perdido en la inmensidad de las entrañas de ese sofisticado lenguaje conocido como Java.

Afortunadamente, el camino admitía retorno: decidí reescribir el tutorial desde cero, considerando estas ideas clave:

- ▶ Describir sólo las clases y métodos imprescindibles para la mayoría de las comunicaciones en red.
- ▶ Apoyarme, en la medida de lo posible, en ejemplos.

Hay tres motivos para obrar así: a) la documentación de todas las clases de Java está disponible para cualquier desarrollador (aunque por doquier hay libros que parecen pensar que no es así); b) muchos programadores sólo usan unas pocas clases de los paquetes anteriores –casi siempre las mismas–; y c) los ejemplos, pese a su incompletitud, son la mejor manera de fijar ideas y de comprender para qué puede servir cada clase.

Por su naturaleza, este trabajo peca de omisiones, pues es imposible abarcar en un tutorial todo lo que se conoce sobre redes. Hay una omisión que lamento mucho: la de la historia de Internet. No me refiero a la historia oficial, expurgada, beatificada y santificada, sino a la verdadera historia de la red de redes, una historia que comienza con un puñado de *hackers*, de bolcheviques de salón, de hijos de Marx y de la Coca Cola, de izquierdista de derechas, muchos de ellos californianos, y que acaba en el NASDAQ, en la especulación y en situaciones tan absurdas e ilógicas como ver cotizar las acciones de Amazon a seiscientos dólares, o las de Terra a más de cien euros.

Tal como cuenta Bruce Sterling en *The Hacker Crackdown* (la traducción es mía):

[...] Las raíces genuinas del moderno movimiento subterráneo de los *hackers* pueden rastrearse más precisamente hasta un movimiento *hippie* anarquista ahora bastante olvidado que se conoció como los *Yippies*. Los *Yippies*, quienes tomaron su nombre del bastante ficticio Youth International Party [Partido Internacional de los Jóvenes], llevaron a cabo una ruidosa y vitalista política de subversión surrealista y exagerada maldad política. Sus premisas básicas eran la promiscuidad sexual flagrante, el consumo abundante y público de drogas, el derrocamiento político de cualquier personaje poderoso de más de 30 años de edad y un final inmediato para la guerra de Vietnam, empleando cualquier medio que fuera necesario, incluida la levitación psíquica del Pentágono. Los dos *yippies* más notorios fueron Abbie Hoffman y Jerry Rubin. [...]

[...]

Se dice que Abbie Hoffman ha provocado que el Federal Bureau of Investigation [FBI] acumulara la más voluminosa ficha jamás abierta a un sólo ciudadano estadounidense [...]. Fue un publicista de talento, que reconocía los medios electrónicos como campo de juego y como arma a la vez. Disfrutó de la manipulación activa de las cadenas de TV y de otros medios hambrientos de imágenes, con extrañas mentiras, rumores extraordinarios, disfraces y suplantaciones y otras siniestras distorsiones, siempre con la garantía absoluta de crear problemas y dificultades a la policía, a los candidatos presidenciales y a los jueces federales [...].

En cuanto a relevancia política, los *yippies* fueron un completo cero a la izquierda; sabían gritar sus consignas cuando había una cámara de televisión cerca y tenían el aspecto que la policía presuponía en cualquier sospechoso, pero ahí acababa todo. En cuanto a conciencia política, eran un completo fraude. Su activismo político estuvo más cercano a un espectáculo bufonesco que a otra cosa. Quizás ellos creyeran que estaban haciendo la sacrosanta Revolución, pero fueron poca cosa más que unos oportunistas que sabían gritar las consignas necesarias para salir en los medios de comunicación y para asustar a la temerosa clase media norteamericana (como Marilyn Manson, pero sin tanto maquillaje y sin prejuicios hacia los fumadores): "Mata a tus padres" (veinte años después, Hoffman lo convirtió en "Ama a tus padres"), "No te fíes de nadie de más de treinta años, seguro que se ha vendido al sistema". Entonces, ¿cómo se va uno a fiar de alguien de menos de treinta años, si sabe que acabará "vendiéndose"? Incuestionablemente, una persona como Rosa Banks –la humilde costurera de Alabama que se negó a ceder su asiento a un hombre blanco, a lo cual estaba obligada por ley– representaba un peligro un millón de veces mayor para unos Estados Unidos blancos y anglosajones que personas como Hoffman y Rubin. Sin embargo, éstos influyeron de manera reconocible en muchas personas relacionadas con las redes que acabarían siendo parte de lo que conocemos hoy como Internet.

Cuando se escriba la historia completa de Internet, alguien deberá explicar cómo un producto derivado de la guerra fría, al igual que la carrera espacial, se nutrió del trabajo de tanta gente que se definía como pacifista, libertaria (en el sentido

estadounidense del término, no en el europeo) o que, cuando menos, mantenía un gran recelo hacia el control gubernamental (pese a que era el gobierno de los Estados Unidos el que subvencionaba, directa o indirectamente, sus trabajos e investigaciones). Muchos investigadores y *hackers* que se identificaban con el *Captain America* de *Easy Reader*, con Timothy Leary o con Abbie Hoffmann participaron en un proyecto financiado al principio por el Pentágono. ¿Cuánto influyó en ello que la Internet financiada por el Pentágono fuera pública de verdad, esto es, de acceso libre y gratuito?

Ese alguien también deberá explicar por qué se puso toda la tecnología e infraestructura desarrollada con dinero público (del Pentágono y, luego, de la *National Science Foundation*) en manos de unas pocas empresas privadas. Desde luego, en Europa se ve raro (y seguramente es ilegal) que una tecnología se pague con fondos públicos y que luego se regale a unas cuantas grandes empresas. Sería como obligar al sector público a que gastase sumas multimillonarias en investigación y desarrollo para que los frutos se los quedaran unos pocos, en lugar de la sociedad. Dicho sin tapujos tecnológicos o económicos: no se puede pretender que unos den de pastar a la vaca para que otros se beban la leche de sus ubres; el dinero público no puede ser la versión económica del **sastre de Campillo**, que cosía de balde y ponía el hilo. Por otro lado, Europa consideraría que una práctica así es competencia desleal: se perjudica tanto a las empresas a las que no se cede esa tecnología e infraestructura como a las empresas de otros países donde no hay dinero público para esos fines.

Afortunadamente, en el desarrollo de Internet han existido y existen muchas personas más preocupadas en divulgar sus conocimientos que en el lucro. Por ejemplo, la *World Wide Web* existe tal como la conocemos gracias al espíritu desinteresada de su inventor, el físico británico Berners-Lee. En una entrevista publicada el 30 de junio de 2004 en el diario *Expansión*, explica por qué entregó gratis su invento al mundo: “Si hubiese pedido dinero, hoy no hubiera existido la red mundial WWW, sino pequeñas redes aisladas en Internet”. Sus palabras me recuerdan a las que dijo Marie Curie cuando discutía la idea de conseguir la patente sobre el radio: “Es imposible e iría contra el espíritu científico [...]. Los investigadores siempre deben publicar sus resultados por completo. Si nuestro descubrimiento tiene aplicación comercial, es algo de lo que no debemos sacar provecho. Si el radio va a usarse en el tratamiento de ciertas enfermedades, me parece inadmisible beneficiarnos de ello”. Puede que Marie Curie fuera una ingenua y que Berners-Lee también lo sea; pero ojalá el mundo estuviera más lleno de ingenuos.

Una historia completa de Internet también deberá contar que algunas compañías creadas al calor de la red no se preocuparon por sus empleados y accionistas –lo que hubiera hecho que el dinero público invertido en Internet repercutiera en la sociedad que, a fin de cuentas, lo había generado–, sino que se dedicaron a llenarse los bolsillos con el dinero de los incautos que invertían en ellas (en la bolsa suele decirse que “un ignorante y su dinero no permanecen mucho tiempo juntos”). ¿Cómo consiguieron el dinero ajeno? Fue sencillo; gritaban con fuerza “nuevos modelos de negocio”, “las técnicas tradicionales de valoración son inútiles con nuestras empresas”, “el potencial de Internet es ilimitado”, “no pierda esta oportunidad” y paparruchas por el estilo. Los folletos de las OPV insistían en que invertir en las *puntocom* era peligroso: no pagaban dividendos, no tenían beneficios ni se preveía que los tuvieran en mucho, mucho tiempo, pero..., ya sabe lo que se dice de los ignorantes, ¿verdad?

Quizá se pregunte usted qué relación puede haber entre la especulación bursátil y Java o las redes. La hay, desde luego: Sun alcanzó cotizaciones jamás imaginadas gracias a la publicidad intensiva que hizo de Java. Muchas empresas pequeñas y medianas consiguieron inversores anunciando que habían trasladado todas sus aplicaciones a Java o que ya sólo iban a trabajar con Java. Algunas, faltas de

experiencia o incapaces de ver que Java no podía satisfacer sus objetivos, protagonizaron algunos de los más vergonzosos y sonrojantes retornos a lenguajes o modelos de negocio tradicionales que aún se recuerdan. Corel es mi ejemplo favorito, aunque poca gente lo recuerda: la intención de escribir todos los productos de la empresa en Java tuvo que abandonarse tras un contundente fracaso técnico y económico. Por si fuera poco, Corel no aprendió la lección: decidió sacar todas sus aplicaciones para el SO Linux y sacar su propia versión del SO; finalmente, ha tenido que abandonar todos sus proyectos concernientes a Linux, tal y como hizo con los de Java. Con respecto a las redes, el hundimiento de las *puntocom* ha ocasionado recortes en los presupuestos empresariales para el desarrollo de nuevas tecnologías y cierta reticencia en los inversores a la hora de financiar novedades. Además, algunas redes de fibra óptica han cambiado de manos tras la debacle. Y las nuevas manos están demasiado ocupadas intentando paliar las pérdidas que dejaron las viejas como para pensar en ampliar o mejorar las infraestructuras de telecomunicaciones. No se puede negar que el agujero dejado por la economía virtual ha tenido consecuencias relevantes en la economía real. Y las redes, no lo olvidemos, forman parte de esta última.

Durante los años noventa se pensaba que las empresas *puntocom* no tenían límites, que el cielo era la última frontera. Pero se olvidó que, por mucho que crezca un árbol, las ramas nunca alcanzan el cielo. Ahora todos sabemos en qué acabó todo ello. De una forma hipócrita, casi cínica, hasta el *Wall Street Journal* tuvo que publicar las verdades del barquero cuando acabó la fiesta y hubo que recoger los platos rotos:

Mito número 1: Las empresas de tecnología pueden generar beneficios impresionantes en ingresos, ventas y producción durante los próximos años.

Mito número 2: Las empresas tecnológicas no están sujetas a las fuerzas económicas ordinarias, como una compañía lenta o un aumento de los tipos de interés.

Mito número 3: Los monopolios crean increíbles oportunidades.

Mito número 4: El crecimiento exponencial de Internet acaba de comenzar, y si cambia será para acelerar.

Mito número 5: Las perspectivas futuras son más importantes que los ingresos inmediatos.

Mito número 6: Esta vez, las cosas serán diferentes...

Desde luego, envidio a quien escriba la historia completa de Internet. Dispondrá de un argumento que parece sacado de una comedia griega: hay pasión, rebeldía, engaño, juventud, cólera, avaricia, celos, envidias... Y al final, como siempre, los dioses ridiculizarán a los avariciosos antes de destruirlos.

Si no me engaño, Menéndez y Pelayo escribió que el autor que comienza un libro es discípulo del que lo termina. Aun no siendo este texto un libro, me reconozco en su frase. Al escribirlo, he visto como muchas relaciones entre conceptos se iban explicitando. No dudo que las relaciones estuvieran ahí desde el principio; pero ¿para qué nos sirve aquello de lo que no somos conscientes? Al intentar pensar la red del tutorial, he visto como unos hilos reforzaban a otros, manifestando vínculos que me habían pasado inadvertidos hasta el momento.

Si bien el hipertexto es una estupenda herramienta, no puede hacer por nosotros el trabajo de enlazar lógicamente datos y de buscar semejanzas o analogías. Abundancia de datos no equivale a información ni a conocimiento, por más inmediatez

con que se nos ofrezcan los datos: una cantidad excesiva de datos sueltos, dispersos, faltos de trabazón, no es información, del mismo modo que un montón de tableros no constituye un mueble. Acumular datos sin ton ni son no es **ciencia ni tecnología**, sino algo muy distinto: **filatelia**. Creo que la escritura lineal sigue siendo importante para dar sentido de conjunto a unos datos o hechos, para dotarlos de coherencia y argumento. En una palabra, para transformarlos en verdadera información, en auténtico conocimiento. No niego que esta opinión viene motivada, probablemente, porque nací antes de que existiera el hipertexto e Internet. Si el lector quiere interpretar mis palabras como el lamento quejoso y moribundo de un ser anacrónico-analógico, acataré gustoso su veredicto: desde niño he sentido simpatía por el pájaro dodo.

La abundancia de datos inconexos no constituye el único obstáculo para la búsqueda de información en Internet. Muchos servicios que ofrecen información se comportan como algunos programas de televisión: prometen muchas, muchísimas cosas en los anuncios previos a la emisión o en las pausas para la publicidad (novedades informativas, entrevistas, actuaciones, regalos, cuerpos ligeros de ropa, etc.), siempre enseguida, siempre ahora mismo, de modo que intentan mantener la atención del espectador –que quizás piensa, desesperado y predispuesto al engaño, que la programación va a mejorar por fin–, pero al final no cumplen sus promesas o no lo hacen como imaginaba el espectador.

Así las cosas, este trabajo intenta evitar al lector el esfuerzo de extraer un vasito de información a partir de la cascada de datos sobre redes y Java que ofrecen Internet y la bibliografía técnica. El resultado final, con sus aciertos y sus fallos, está ante sus ojos. Si no le parece interesante, lo siento. Lo haré mejor la próxima vez. Palabra.

En febrero de 2003 se publicó en javaHispano un tutorial mío titulado *Cómo hacer un chat en Java (JDK 1.2-1.3 y JDK 1.4-1.5): fundamentos, desarrollo e implementación*. Aunque en él se tratan parcialmente las comunicaciones en red y el paquete **java.net** y se explica cómo programar un *chat*, hay demasiadas diferencias entre ambos como para que este tutorial se pueda considerar una nueva versión o una ampliación de aquél:

- El propósito de este tutorial no es solamente hacer un *chat*: es aprender a usar Java para las comunicaciones en red. Con todo, no he renunciado al ejemplo de un *chat*, pues es difícil encontrar problemas donde se manejen a un tiempo tantos conceptos de comunicaciones en red.
- El apartado común de fundamentos de las comunicaciones en red se ha ampliado. Ahora ocupa más espacio que el tutorial *Cómo hacer un chat en Java*. Aparte del interés que la materia que muestra tiene por sí misma, resulta absurdo intentar programar para redes sin saber qué hay bajo ellas.
- Se han incluidos apartados inexistentes en aquél. Así, los dedicados a **java.io**, **java.nio**, **java.zip**, a los sistemas distribuidos, a RMI y a CORBA.
- Se trata a fondo la internacionalización con **java.io**, imprescindible para escribir aplicaciones multilingües.
- Se ha añadido mucho código (ejemplos de E/S, un primitivo navegador, un sencillo servidor HTTP, etc), no relacionado con lo que se precisa para programar un *chat*.
- Se ha mejorado mucho el código dedicado al *chat* (sincronización de métodos, control de los tiempos de conexión sin actividad, eliminación de hilos muertos, etc.), para que el lector entienda las complejidades que se presentan al escribir aplicaciones para redes.
- Se ha incluido el código completo de un *chat* en **java.nio**, el cual apenas estaba esbozado en el tutorial del chat. También se incluye el código de un *chat* basado en CORBA.

Por lo dicho, considero que este tutorial es independiente del otro. *Java y las redes* resulta más idóneo para quienes deseen una introducción al abultado hexagrama **java.io/java.nio/java.net/java.zip/RMI/CORBA**. *Cómo hacer un chat en Java* se orienta más hacia quienes sólo deseen echar un vistazo a **java.net** para ver si les interesa o no. Mi idea es mantener ambos en javaHispano.

2. Fundamentos de las comunicaciones en red

2.1. Algunas definiciones

Para entender cómo se producen las comunicaciones en Internet o en cualquier red TCP/IP, conviene definir de entrada ciertos conceptos básicos (casi todos se detallarán con más exactitud conforme avance el tutorial):

- **Un proceso** es un programa en ejecución; cada proceso es independiente de los otros y tiene reservado un cierto espacio de direcciones de memoria. En general, los procesos son controlados por el sistema operativo. Un proceso consiste en a) un espacio de memoria virtual para el código ejecutable y los datos; b) unos recursos del sistema operativo; c) un estado del procesador (especificado por los valores que toman las direcciones de memoria física, los registros, etc.); y d) una especificación de seguridad, en la que se almacenan los permisos del proceso y de su propietario. En las redes, los sockets son un mecanismo para permitir la comunicación entre procesos.
- **Un máquina o equipo** es un hardware capaz de ejecutar procesos. A veces, también se usa *sistema* o *dispositivo* con ese sentido.
- **Una red** es un conjunto de máquinas o equipos (no necesariamente ordenadores: pueden ser teléfonos móviles, agendas electrónicas, electrodomésticos, etc.) que comparten un mismo medio físico de comunicación (cable coaxial, fibra óptica, ondas de radio, láser, etc.) y un mismo conjunto de reglas para comunicarse entre ellos. Si los dispositivos están todos en una zona geográfica limitada y no muy extensa –por ejemplo, un edificio o un campus–, se habla de **redes de área local** o LANs (*Local Area Networks*); en caso contrario, se habla de **redes de área extensa** o WANs (*Wide Area Networks*). Por ejemplo, los ordenadores de las sucursales de un banco forman parte de una WAN.
- **Un anfitrión (*host*)** es una máquina (habitualmente, un ordenador) conectada a una red. Dicho de otro modo: es un nodo de la red. Este término también suele usarse para referirse a máquinas que ofrecen servicios a otros computadores (FTP, Telnet, etc.), caso en que equivale a **servidor u ordenador central**. En los textos anglosajones suele llamarse *host address* a la dirección de Internet o dirección IP de un nodo.
- **Un protocolo** es un conjunto de reglas y procedimientos necesario para que dos máquinas intercambien información. Existen muchos conjuntos de protocolos para las comunicaciones en red entre ordenadores: IBM desarrolló SNA (*Systems Network Architecture*) y APPN (*Advanced Peer-To-Peer-Networking*); DEC desarrolló DNA (*Digital Network Architecture*); Apple creó Appletalk; Novell usa su SPX/IPX (*Sequenced Packet Exchange/Internet Packet Exchange*); Xerox usa XNS (*Xerox Network Services*), etc. Sin embargo, el conjunto más popular de protocolos de red es **TCP/IP**, que se estudiará en los siguientes subapartados.

- **Un protocolo fiable** es, desde el punto de vista de las comunicaciones, un protocolo que incluye detección de errores y, a menudo, corrección de errores.
- **Una tarjeta de red o una tarjeta adaptadora de red** es el hardware que se conecta físicamente con el cableado de la red. La tarjeta se gestiona con el software del controlador de la tarjeta. Las tarjetas de red se encargan de la transferencia física de la información entre las máquinas de la red.
- **Una dirección física o de hardware** es un código que identifica únicamente un nodo o anfitrión de una red. En las redes que siguen los estándares IEEE 802 (como Ethernet), cada tarjeta de red tiene su propio identificador único, de 48 bits de longitud: la dirección MAC (*Media Access Control*), grabada en un chip dentro de ella. Por ejemplo, el número hexadecimal 00:A0:C9:12:C3:25 corresponde a una dirección MAC válida, asociada a una tarjeta de red fabricada por Intel Corporation. La dirección MAC de un ordenador es la dirección MAC de su tarjeta de red. Otras redes usan como direcciones físicas direcciones DLC (*Data Link Control*).
- **Ethernet** es la tecnología dominante para redes de ordenadores del tipo LAN. Viene definida por el protocolo IEEE 802.3 y especifica exactamente el cableado y las señales eléctricas que debe usar la red. Toda máquina en una red Ethernet tiene asignado un número único de 48 bits, conocido como dirección Ethernet o dirección MAC. La asignación de direcciones Ethernet se hace de modo que no existan dos máquinas con la misma dirección MAC. Actualmente, las redes Ethernet utilizan cable coaxial delgado (10Base-2), cable coaxial grueso (10Base-5), cable de par trenzado (10Base-T) y fibra óptica (10Base-F).
- **Internet** es la mayor red pública TCP/IP que existe. Está formada por la interconexión de un gran número de redes de ordenadores de diferentes tipos, capaces de interoperar gracias al uso común de la familia de protocolos TCP/IP. En Internet se emplean distintos lenguajes de programación, sistemas operativos, redes, hardware, conectores, etc.; sin embargo, TCP/IP hace que todas estas diferencias no importen.
- **Una intranet** es una red privada, perteneciente a una organización, que utiliza los protocolos TCP/IP. Las intranets son parte de Internet, pero se administran independientemente y suele tener fronteras configurables en cuanto a seguridad y acceso. Habitualmente, se componen de varias redes LAN enlazadas entre sí. Técnicamente, una Intranet viene a ser una versión en miniatura de Internet. Cualquier aplicación o servicio disponible en Internet está disponible en las intranets.
- **Una extranet** es la unión de dos o más *intranets*. Las extranets están usándose para permitir el comercio electrónico entre empresas y para integrar partes de sus sistemas informáticos. Por ejemplo, la aplicación que controla el sistema de producción de un fabricante puede integrarse mediante una extranet con la aplicación que gestiona los pedidos de un distribuidor.

- **Un servicio (de una red)** es una función que presenta a sus usuarios. Internet, por ejemplo, ofrece servicios de archivos, de correo electrónico, de grupos de noticias, de foros, etc. Todo servicio ofrece un conjunto de operaciones que el usuario puede aprovechar. Así, cualquier servicio de archivos ofrece leer, escribir, borrar, intercambiar y enviar archivos.
- **Los URL (*Uniform Resource Locator: localizador universal de recursos*)** identifican cualquier servicio o recurso de una red. Un URL identifica el ordenador de la red que proporciona el servicio o recurso e identifica qué servicio o recurso solicita el usuario.
- **Una capa o nivel** es una abstracción que agrupa distintos problemas de comunicaciones relacionados entre sí.
- **Los paquetes (de datos)** son unidades de datos en cualquier capa de un conjunto de protocolos (la estratificación en capas de los protocolos se estudiará más adelante). Todo paquete consta de dos partes: una **cabecera** (donde se incluye la información de la máquina que lo originó y de la máquina a la que va dirigido) y un **área de datos** (donde va la información).
- **Los datagramas** son paquetes de datos de la capa IP o de red (se verá en el apartado 2.3). A veces se usa *datagrama* como sinónimo de *paquete*.
- **Una trama (frame)** es un paquete preparado para su transmisión por un medio físico. El proceso de preparación (**entramado o framing**) suele consistir en añadir delimitadores para indicar el principio y fin del paquete, así como campos de control. En una red Ethernet, verbigracia, durante el proceso de entramado se convierten las direcciones IP en direcciones físicas o de Ethernet, asociadas al hardware.
- **Una trama física (physical frame) o trama de red física** es un paquete tal y como es transmitido en un medio físico (fibra óptica, ondas de radio, microondas, cable coaxial, cable de par trenzado, etc.). Puede, por tanto, ser una secuencia de señales eléctricas, ópticas o electromagnéticas. Decir que una trama, un datagrama o un paquete circula por una red equivale a decir que una trama física circula por ella, y viceversa. Por ello, muchas veces se usa *trama*, *datagrama* o *paquete* en lugar de *trama física*. En este tutorial usaré *paquete* en el sentido general descrito en la página anterior, reservaré *datagrama* para los paquetes de la capa de red, y escribiré *trama física* cuando quiera subrayar la conversión de bits en señales físicas. He aquí varios ejemplos de mi uso de estos términos: *La capa de red genera datagramas; El paquete recorre todas las capas; El módem convierte las tramas en tramas físicas.*
- **Los encaminadores o encaminadores IP (routers o IP routers)** son dispositivos que encaminan los paquetes de datos entre subredes, intentando encontrar el mejor camino posible para cada paquete. Físicamente, pueden ser dispositivos electrónicos que conectan entre sí subredes heterogéneas (basadas en fibra óptica, Ethernet, etc.) o anfitriones en los que se ha instalado software de encaminamiento. Suele reservarse el término **pasarela (gateway)** para los dispositivos

que mueven datos entre diferentes conjuntos de protocolos (por ejemplo, entre TCP/IP y el SNA de IBM), y **encaminador (router)** para los que mueven datos entre subredes que operan con los mismos protocolos.

- **Un cortafuegos (firewall)** es un medio de regulación del acceso a una red, ya sea mediante hardware o software. Su principal misión es limitar el acceso a intranets o extranets (aunque se usan cada vez más para proteger anfitriones aislados) filtrando los paquetes que entran y salen, de forma que sólo se permita el flujo de paquetes autorizados. Un cortafuegos puede, por ejemplo, impedir el paso de paquetes procedentes de ciertas máquinas o el envío de paquetes a otras.
- **Los flujos o corrientes (streams)** son tuberías o canales de comunicaciones: tienen dos extremos entre los cuales fluyen los datos de manera continua. Obedecen literalmente la ley de Bennet ("Si un cable tiene un extremo, probablemente tiene otro").
- **Una plataforma** es una combinación específica de hardware y de sistema operativo. Por ejemplo, un PC con una arquitectura Pentium y con el sistema operativo Red Hat pertenece a la plataforma Intel/Linux.
- **Mainframe** equivale a macroordenador, gran ordenador, ordenador central o servidor corporativo. Viene a ser un término industrial para los ordenadores grandes. Las máquinas con la arquitectura 390 de IBM son un ejemplo de *mainframe*. La palabra viene de los armazones metálicos donde se guardan estos ordenadores. Hoy día, los *mainframes* no se caracterizan por ser voluminosos, sino por su eficiencia a la hora de acceder a sistemas de almacenamiento (discos) y de transferir los datos del disco a la máquina.
- **Un demonio (daemon o demon)** es un programa o proceso al que no se llama explícitamente, pero que permanece en estado latente hasta que se cumplen ciertas condiciones. El término viene de UNIX. Los demonios se usan mucho en procesos relacionados con redes, porque permanecen a la espera de peticiones de los clientes sin impedir la ejecución de otros programas. En inglés, *daemon* es la grafía antigua para *demon* (demonio, o espíritu bueno o malo). En los textos de telecomunicaciones se usa a veces **dragón (dragon)** como sinónimo de demonio.
- **Un RFC (Request For Comments: petición de comentarios)** es un documento relacionado con algún estándar de Internet o de los sistemas relacionados con ella. Protocolos como HTTP, SMTP, POP3 o FTP se han definido mediante unos RFC. Uno de los RFC más famosos, pese a su difícil implementación, es el RFC 1149 (<http://www.faqs.org/rfcs/rfc1149.html>), cuyo nombre es *Standard for the transmission of IP datagrams on avian carriers* (Estándar para la transmisión de datagramas IP en aves mensajeras), que define el protocolo CPIP. Actualmente ya se dispone de una implementación del CPIP.

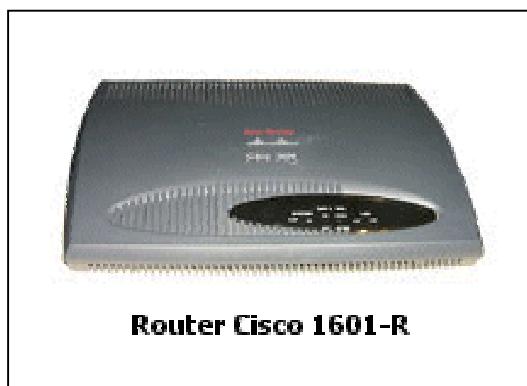


Figura 1. Un router muy popular. Extraído de la publicidad de Cisco Systems, Inc.

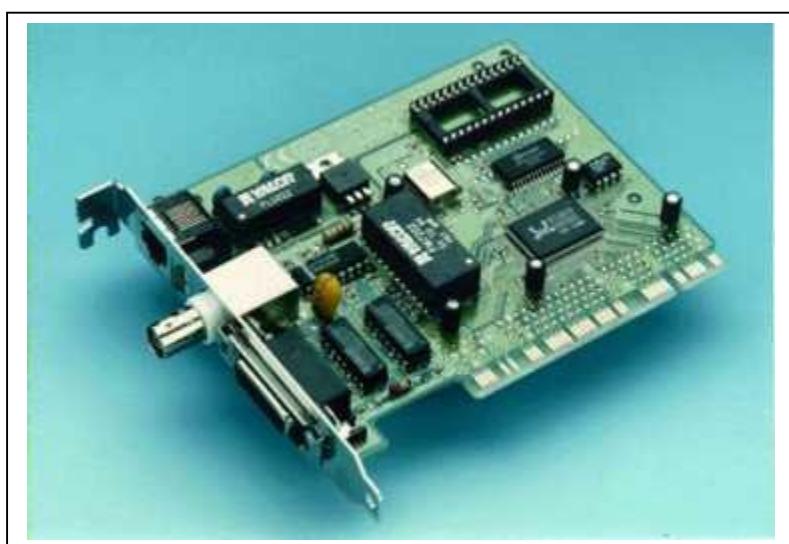


Figura 2. Una tarjeta de red

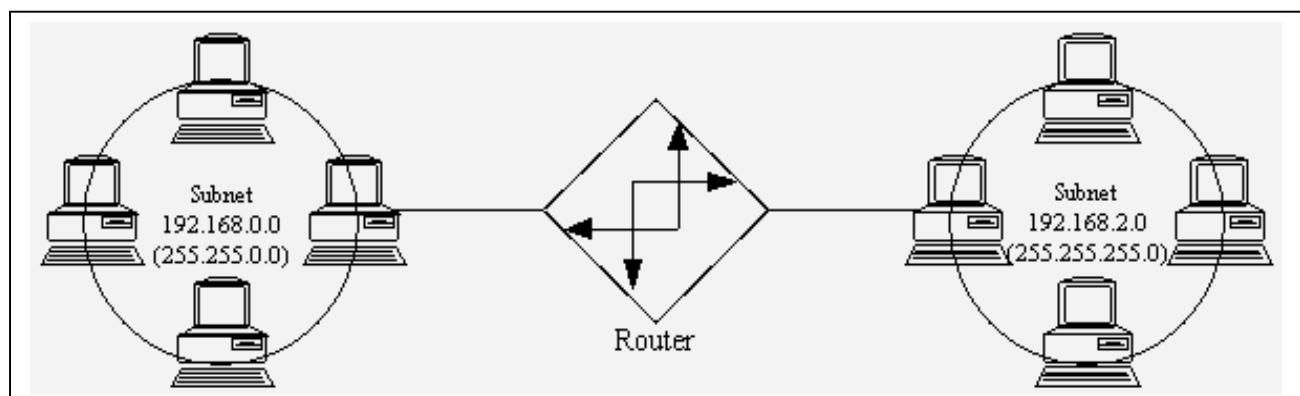


Figura 3. Una extranet muy sencilla. Dibujo escaneado de *Daryl's TCP/IP Primer*

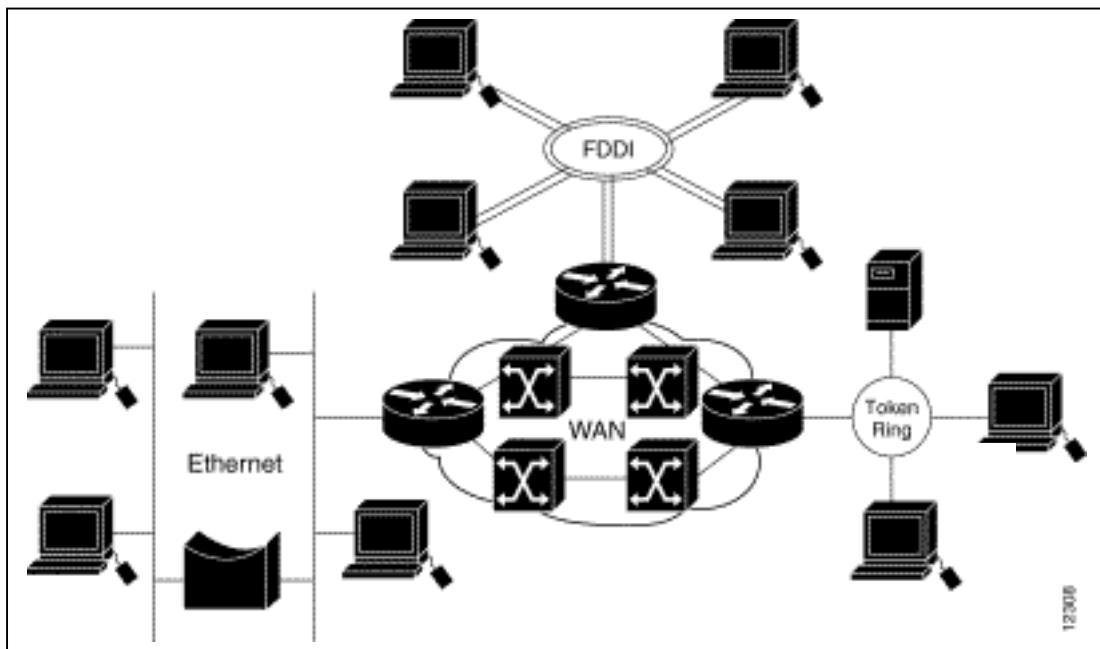


Figura 4. Una extranet más compleja que la de la figura 3

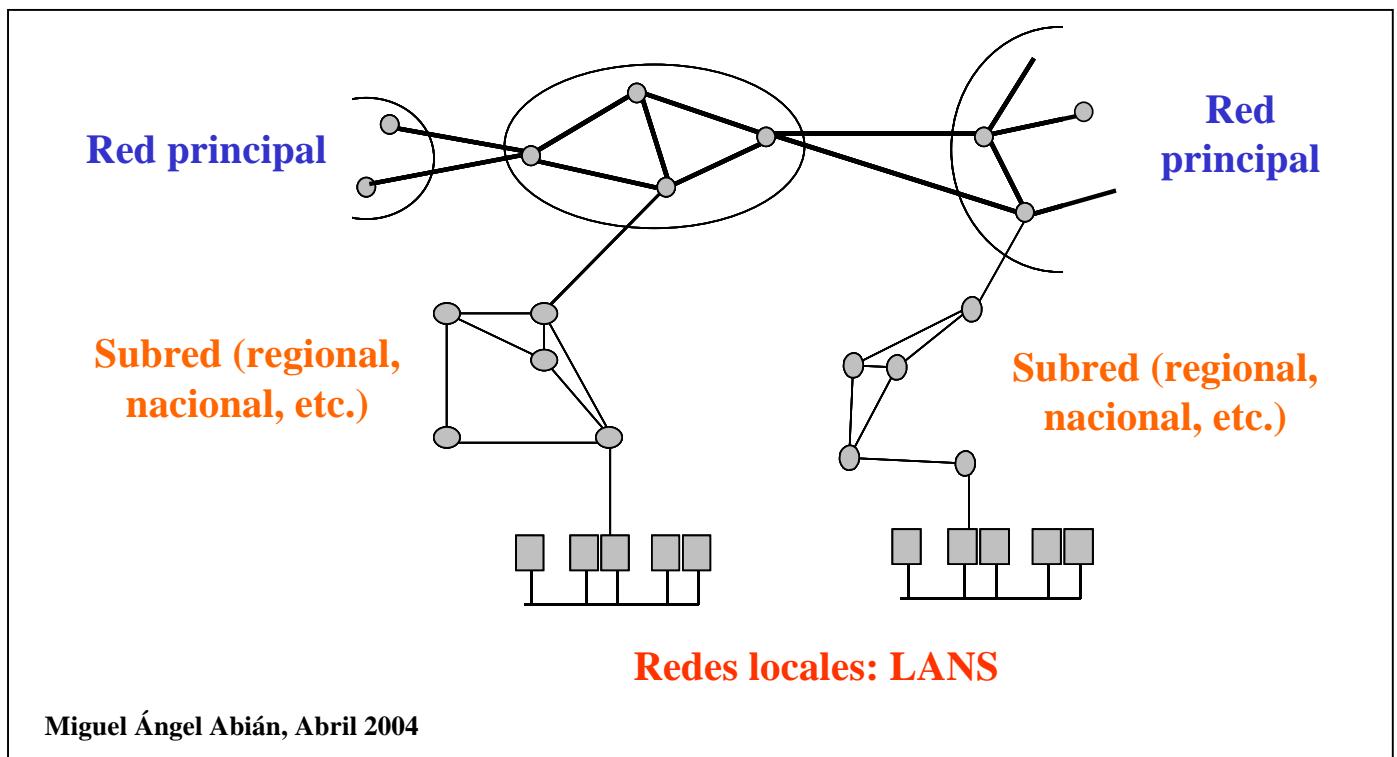


Figura 5. Esquema muy simplificado de la estructura de Internet

Una trama Ethernet



- **Preámbulo:** 8 bytes, secuencia de ceros y unos usada para la sincronización.
- **Dirección de destino:** 6 bytes, dirección física del nodo de destino (dirección MAC).
- **Dirección de origen:** 6 bytes, dirección del nodo de origen.
- **Tipo:** 2 bytes, especifica el protocolo de la capa superior usado.
- **Datos:** entre 46 y 1500 bits, información de las capas superiores.
- **CRC:** 4 bytes, *Cyclic Redundancy Check*, es una secuencia de comprobación de la trama.

Miguel Ángel Abián, Abril 2004

Figura 6. Esquema de una trama Ethernet

2.2. Protocolos de comunicaciones

En [An Internet Encyclopedia](#) se define protocolo como “una descripción formal de los formatos de mensaje y reglas que dos o más máquinas deben seguir para intercambiar esos mensajes”. Los protocolos de comunicaciones establecen la manera como se realizan las comunicaciones entre ordenadores. La meta última del uso de protocolos es conseguir que ordenadores con distintos sistemas operativos y arquitecturas de hardware puedan comunicarse si siguen al pie de la letra los protocolos; dicho de otro modo, se persigue la independencia del proveedor. El espíritu que guía el diseño de protocolos para redes se resume en la frase “Sea conservador en lo que haga, sea liberal en lo que acepte de otros”.

Los protocolos fundamentales de Internet son el TCP (*Transmission Control Protocol*) y el IP (*Internet Protocol*). Para evitar confusiones, conviene aclarar que las siglas TCP/IP se usan para designar conceptos íntimamente relacionados, pero no idénticos:

- 1) Por un lado, **un modelo de capas**.
- 2) Por otro lado, **un conjunto o familia de protocolos** (TCP, IP, UDP, Telnet, FTP, etc.) con un comportamiento común, no la mera suma de los protocolos TCP e IP. Algunos de estos protocolos se verán más adelante.

Un conjunto de capas y protocolos forma una **arquitectura de comunicaciones**.

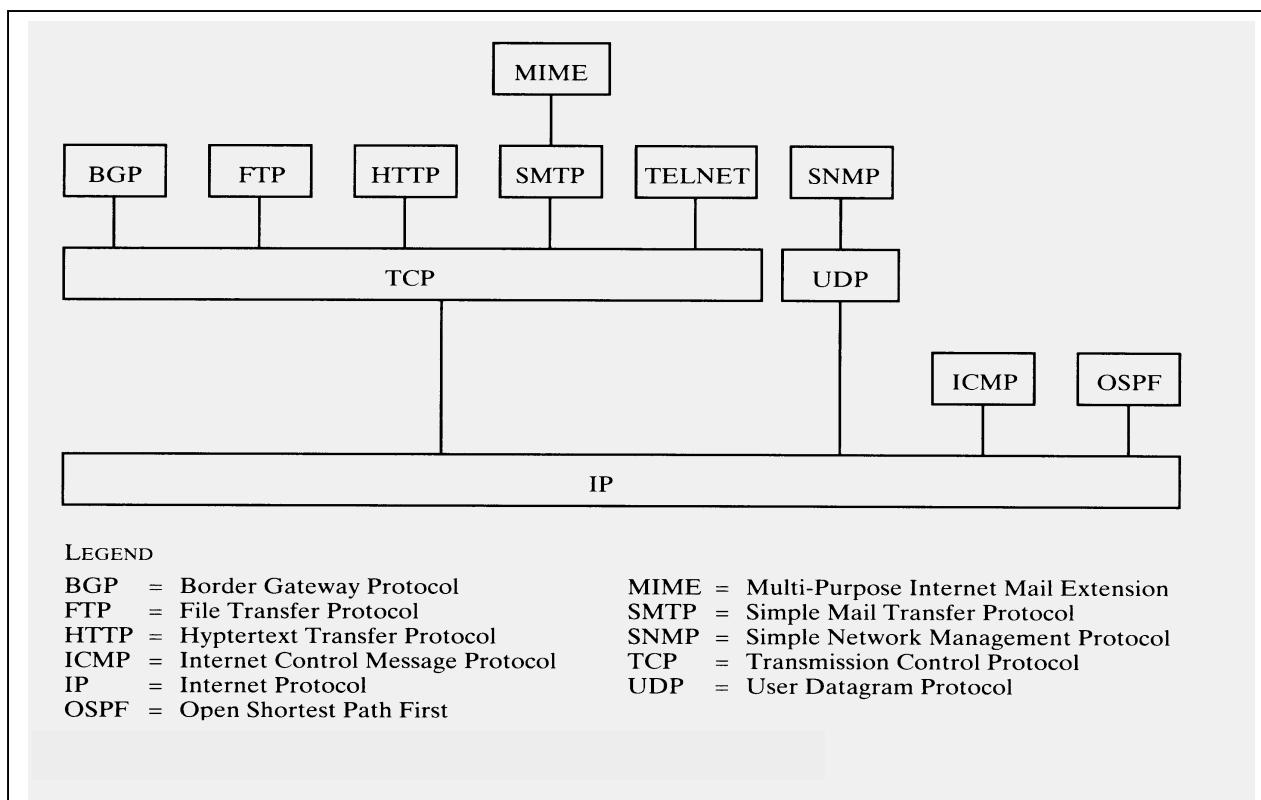


Figura 7. Algunos protocolos de la familia TCP/IP. Dibujo escaneado de *Data and Computer Communications*

2.3. TCP/IP: Un modelo de capas

En general, entender cualquier proceso de comunicación es tarea compleja. Dentro de procesos como enviar correo electrónico, consultar páginas web o, simplemente, llamar por teléfono, se ocultan muchos subprocessos que se entrelazan de forma no trivial.

Como es tendencia inherente al género humano el reducir la complejidad de los problemas para intentar resolverlos, resulta natural que las comunicaciones también se aborden desde una aproximación simplificadora. La manera más sencilla de simplificar cualquier proceso consiste en dividirlo en pequeños problemas de más sencilla manipulación: fragmentar facilita la comprensión. Así, un proceso complejo de comunicaciones se fragmenta en subproblemas de menor complejidad. Las **capas** (o **niveles**) agrupan subproblemas relacionados entre sí.

Las capas son objetos conceptuales que modelan situaciones. Para cada capa puede definirse una forma de funcionamiento (o varias) que resuelve uno o más subproblemas asociados a la capa: estas formas de funcionamiento se denominan **protocolos**. Con otras palabras: los protocolos en un modelo de capas son **posibles modos de funcionamiento de una capa**, que la definen funcionalmente. Un protocolo puede convertirse en un estándar, ya sea *de jure* o *de facto*; en ese caso, cualquier fabricante de productos de hardware o de software puede diseñar y construir productos que implementen ese estándar. Así, el protocolo de área local Ethernet (correspondiente a la norma IEEE 802.3) es implementado por muchos fabricantes que comercializan productos Ethernet compatibles entre sí, pero distintos en cuanto a circuitería.

Cada capa lleva a cabo un conjunto de funciones bien definidas y sólo puede comunicarse con otras mediante una **interfaz**, especificada por algún protocolo. El término **capa** resulta muy apropiado, pues las comunicaciones se realizan mediante el paso de los datos de una capa a la que está inmediatamente bajo ella, y así sucesivamente. Entre dos máquinas, las comunicaciones desde la capa N de una de ellas hasta la capa N correspondiente a la otra se comportan según los protocolos definidos para esa capa concreta.

A una capa le pueden corresponder varios protocolos; pero un protocolo no puede corresponder a varias capas (se violaría la definición de capa). Las capas se construyen de modo que la capa N de la máquina receptora recibe exactamente el mismo objeto enviado por la correspondiente capa N de la máquina emisora.

Capas de un servicio de correos

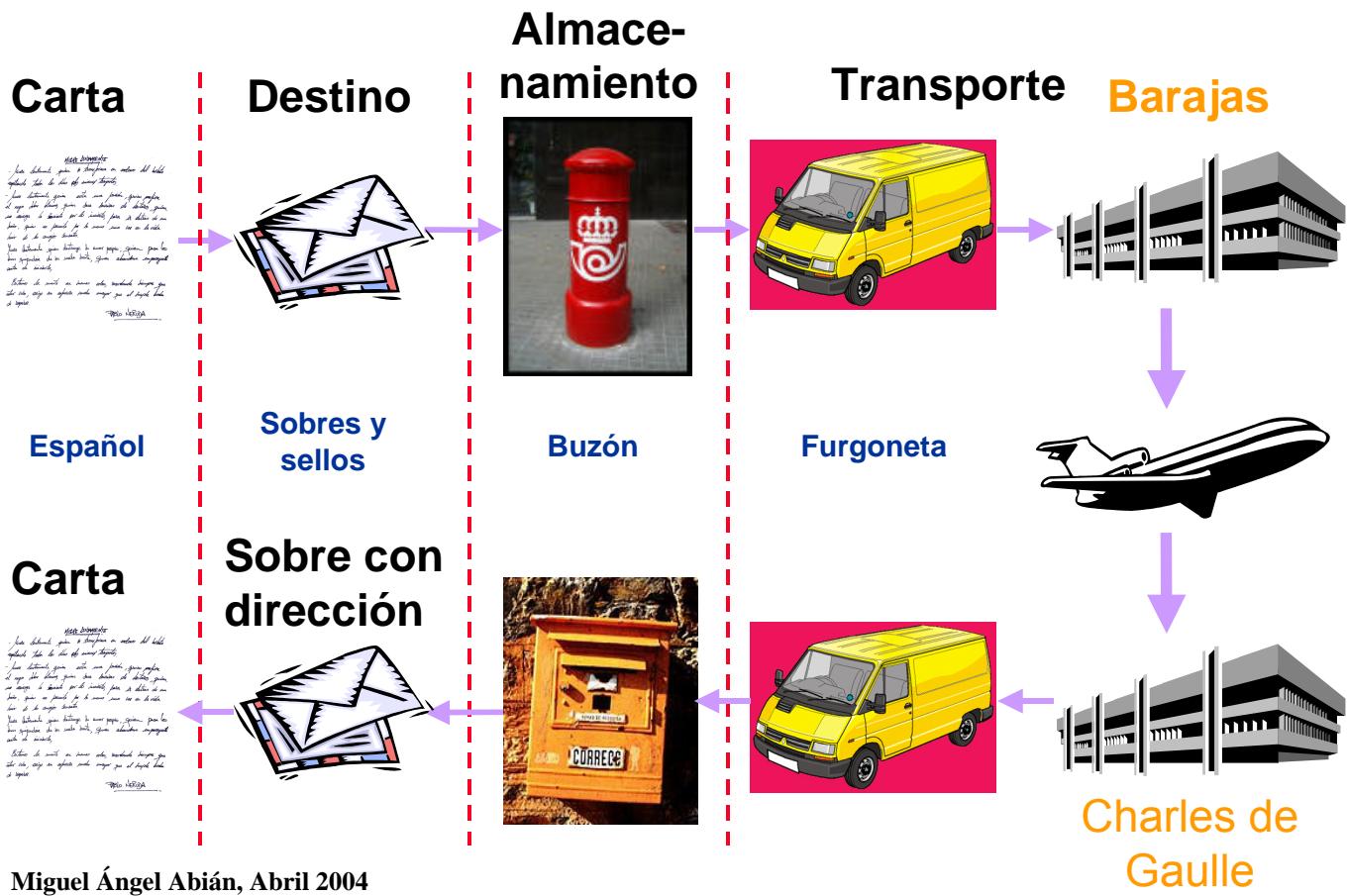


Figura 8. Un modelo de capas para un sistema de comunicaciones que todos hemos utilizado

El sencillo modelo en capas de la figura 8 encapsula en capas los diversos subproblemas que se plantean para que una carta llegue a su destino y muestra algunas posibles soluciones. He aquí algunos de los problemas que se resuelven en las capas:

- ¿En qué lenguaje se debe escribir la carta para que la persona que la reciba la entienda?
- ¿Qué formato debe usarse para que el sistema de correos sepa cuál es el destino de la carta?
- ¿Cómo se localiza el lugar geográfico donde se encuentra la persona a quien se envía la carta?
- ¿Cómo se transporta la carta (es decir, la información) de un sitio a otro?
- ¿Qué rutas son las más adecuadas para llevar lo antes posible cada carta a su destino?

Por sencillas que nos parezcan las soluciones –tengamos en cuenta que cualquiera de nosotros sabe desde niño cómo funciona un servicio de correos–, las preguntas que hemos hecho se vuelven a plantear para el transporte de información entre redes (el cual no solemos conocer desde niños).

Las soluciones descritas, que aparecen en azul, son protocolos. Dependiendo de las circunstancias, se usará un protocolo u otro. Así, si una carta tiene su origen y destino en la misma ciudad, lo normal será llevarla en motocicleta (al menos, eso me ha asegurado el funcionario de Correos a quien he preguntado). Usar furgonetas o motocicletas equivale a usar protocolos distintos para un mismo subproblema.

La estratificación en capas de los protocolos es consecuencia lógica de la estratificación del problema en capas: todos los protocolos que resuelven subproblemas de una misma capa se agrupan dentro de esa capa.

El desarrollo en capas de los protocolos de comunicaciones proporciona ventajas substanciales. Por un lado, las capas permiten modelar y simplificar el estudio y desarrollo de los protocolos, lo que facilita desarrollarlos. Por otro lado, la división de los protocolos en capas fomenta la interoperabilidad.

Las dos mejores definiciones de **interoperabilidad** que conozco proceden del proyecto IDEAS ("la capacidad de un sistema o un producto para trabajar con otros sistemas o productos sin especial esfuerzo de parte del cliente") y de la norma ISO 16100 ("la capacidad para compartir e intercambiar información usando una sintaxis y una semántica comunes para conseguir una relación funcional específica mediante el uso de una interfaz común").

La estratificación en capas de los protocolos permite que un sistema que use los protocolos de una capa N pueda comunicarse con otros sistemas que también usen protocolos de esa capa, sin que importen los detalles exactos de las capas N-1, N-2, etc., ni las diferencias entre ambos sistemas (arquitecturas, software, hardware, periféricos, etc.). En consecuencia, dado un protocolo de la capa N, se pueden cambiar las características internas de los protocolos de las capas inferiores, o cambiarlos por otros, sin que se necesite modificar los de la capa N. Esta propiedad se encuadra a la perfección dentro de las definiciones anteriores de interoperabilidad.

Nota: Por precisión, conviene señalar que la interoperabilidad derivada del uso de capas es una interoperabilidad **débil**. Nadie nos asegura que los datos intercambiados sean debidamente procesados o comprendidos. Si se usa XML, se puede asegurar que existe interoperabilidad en cuanto a los datos, pero no en cuanto a la semántica: el receptor y el emisor pueden entender las etiquetas XML de formas muy distintas. Para una interoperabilidad completa se necesita usar ontologías. Para tener un primer contacto con las ontologías, recomiendo consultar esta dirección:
http://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html . Disfruten del Côtes d'Or.

El lector que conozca la terminología de la orientación a objetos reconocerá que las capas vienen a ser como clases bien encapsuladas, con interfaces públicas e implementaciones internas privadas. Si se cambia la implementación interna de una clase y se respeta la interfaz, no se necesita modificar el resto de las clases.

La relación funcional entre capas se representa en la figura 9.

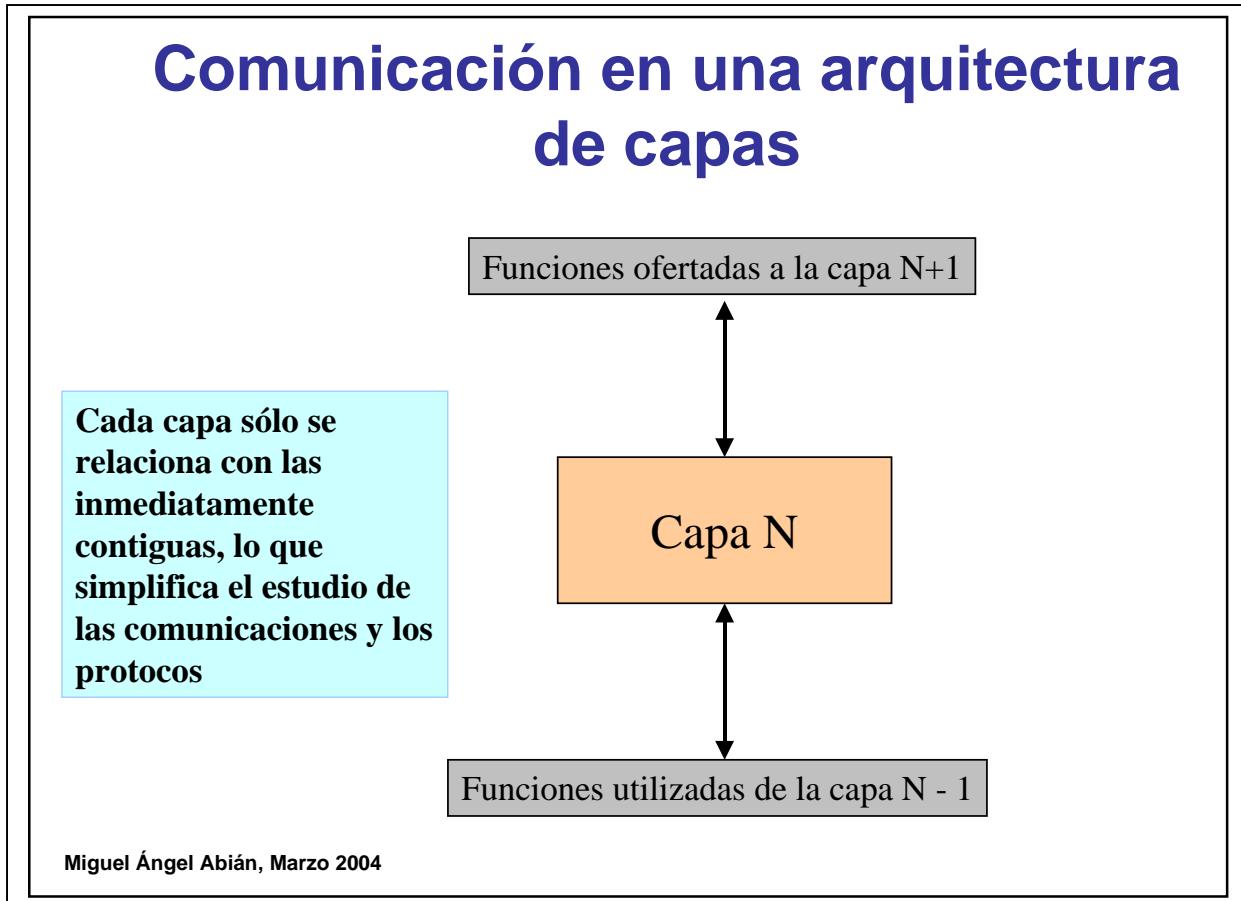


Figura 9. Comunicación entre capas

La estructura en capas resulta muy habitual en telecomunicaciones. Consideremos, por caso, una conversación telefónica acerca de anzuelos entre un informático alemán y un pescador australiano, ambos hablantes de esperanto (cada uno desconoce la lengua vernácula del otro). Aunque las redes telefónicas a las que están conectados son completamente distintas, permiten que cada uno oiga al otro: **son capaces de interoperar**. Por otra parte, cada hablante puede escuchar –no sólo oír– las palabras que dice el otro porque usan un lenguaje común (esperanto). Por último, si la conversación tiene sentido para ambos (en otras palabras, si existe comunicación efectiva) es porque versa sobre asuntos o intereses que ambos comparten (la pesca, en este caso). Estamos, pues, ante una arquitectura de dos capas –red telefónica y lengua–, en la que existen tres protocolos –el asociado a la transmisión física de la voz, el esperanto y la pesca–; como hoy día la palabra *arquitectura* se usa con la ligereza del helio, espero que nadie se ofenda por la imprecisión del término.

En este ejemplo, cada capa es lógicamente independiente de las otras (que se pueda establecer comunicación telefónica no implica que los interlocutores hablen el mismo idioma o que tengan intereses comunes); pero todas son necesarias para la comunicación. La capa física corresponde a la red telefónica, pues ésta permite la comunicación física (la transmisión de voz). Los dos usuarios no necesitan saber, al igual que ocurre en Internet, nada de los detalles de la capa física; basta con que sepan cómo llamar por teléfono y conozcan los prefijos internacionales.

A este sencillo modelo de comunicaciones se le pueden añadir más capas. Para incorporar una nueva, podemos imaginar que ambos hablantes no hablan esperanto y que cada uno se comunica mediante un intérprete que sabe esperanto y el idioma del hablante con que está. Entonces, se habría añadido una nueva capa (intérprete) sin modificar el número de protocolos. La escena descrita resulta enrevesada –no lo negaré–; pero situaciones mucho más extrañas se han visto en la apasionante historia de las redes de comunicaciones.

Otro ejemplo de arquitectura de capas en procesos de telecomunicaciones nos lo proporciona la propia historia: una situación habitual en el París ocupado por los alemanes en la II Guerra Mundial era la del espía soviético que transmitía a Moscú un mensaje cifrado con un libro de claves (libro de una vez), mediante radiotransmisiones de onda larga en código Morse. Se solía usar la banda de treinta y nueve metros para recibir, y la de cuarenta y nueve para emitir. En Moscú se descifraba el mensaje con el libro de claves complementario del anterior y se dirigía a quien correspondiera. Una vez leído, se enviaba un mensaje de respuesta al emisor, también cifrado con un libro de claves. En la siguiente figura se muestra una versión simplificada de la arquitectura de comunicaciones resultante (no considero ningún problema derivado del uso de distintas lenguas).

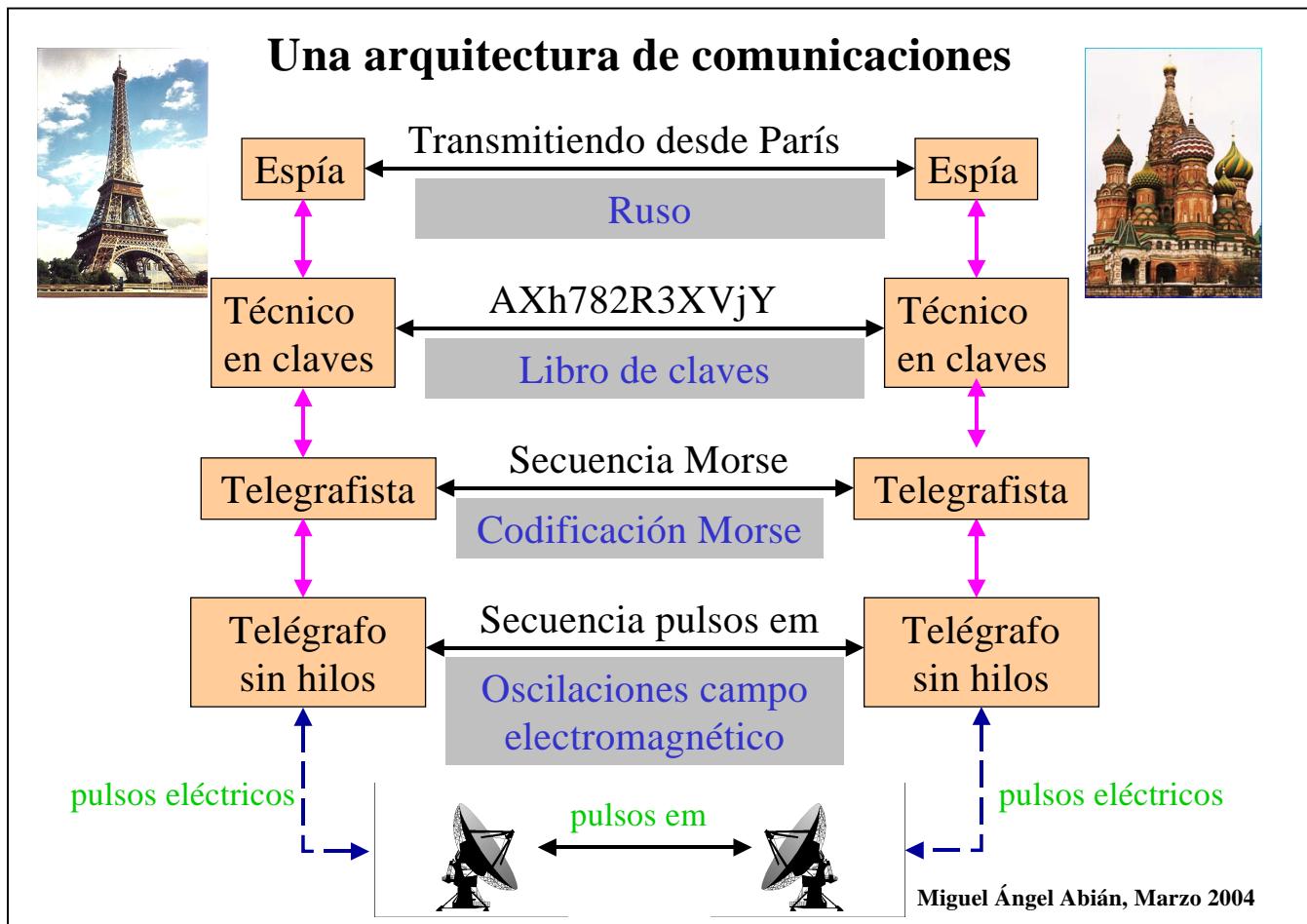


Figura 10. Transmitiendo desde la Francia ocupada

El proceso mostrado en la figura 10 es simple: el espía parisense redactaba su informe; el técnico en claves lo cifraba en una secuencia alfanumérica; y el telegrafista convertía el mensaje cifrado en rayas y puntos Morse y lo enviaba a Moscú mediante telegrafía sin hilos. Si la capa Heaviside tenía a bien no convertir el mensaje en ruido estático o en un manojo de interferencias incordiantes, un proceso similar pero inverso se llevaba a cabo en Moscú: un telegrafista recibía el código Morse y lo transcribía a un documento; un técnico en claves descifraba el documento, y un agente del espionaje soviético lo interpretaba. Para no faltar a la verdad, he de decir que los papeles de espía, técnico en claves y telegrafista solían recaer en una misma persona; aquí considero que había una persona distinta para desempeñar cada papel.

Al igual que el ejemplo anterior, éste nos manifiesta ya la encapsulación de las capas: el espía no tenía por qué saber nada de cifrado ni de código Morse; el especialista en claves no necesitaba conocer nada sobre técnicas de espionaje o telegrafía sin hilos; y el telegrafista no tenía por qué saber nada de claves ni de espionaje. Por otro lado, el espía en Moscú tampoco necesitaba saber nada acerca de cómo había llegado el mensaje a sus manos: sólo necesitaba evaluar su veracidad y formular una respuesta. A todos los efectos, todo sucedía como si se comunicaran directamente el espía parisense y el ruso. En todo modelo de capas, aunque la comunicación real se realiza **verticalmente**, todo sucede como si se realizara **horizontalmente**.

Curiosidad: El método de transmisión de información cifrada descrito arriba no es tan arcaico como podría uno pensar. También fue usado por muchos espías de la extinta RDA en el Berlín de los años sesenta y setenta, así como por agentes occidentales infiltrados en países tras el Telón de Acero.

En España hay constancia de que existieron transmisiones de ese tipo en los años sesenta y setenta (se supone que procedían de espías soviéticos, pero nunca hubo comunicados oficiales). Hace siete años, pude ver en el INTA (Instituto Nacional de Técnica Aeroespacial) los coches con antenas que se usaban en Madrid para localizar los focos de las transmisiones.

La **arquitectura TCP/IP** (en el sentido de conjunto de capas y protocolos) consta de cuatro capas:

- **Capa de enlace.** Suele incluir la tarjeta de red del ordenador, el controlador de la tarjeta para el sistema operativo utilizado y el medio de transmisión utilizado (cable, láser, fibra óptica, radio, etc.). Esta capa describe las características físicas del medio de transmisión, gestiona las conexiones físicas, proporciona la transmisión física de los datos (cadenas de bits) a través del medio usado y se encarga de manejar todos los detalles del hardware que interacciona con el medio de transporte de los datos. Asimismo, se encarga de especificar cómo han de transportarse los paquetes de datos por el medio de transmisión. Dicha especificación incluye obligatoriamente el **entramado (framing)**: la manera como se especifica el comienzo y el fin de los paquetes. Por ejemplo, una de las funciones de esta capa consiste en transformar una secuencia de bits en una señal eléctrica, en un modo de una fibra óptica, en ondas electromagnéticas, etc. Esta capa admite muchos protocolos alternativos (ATM, Ethernet, Token ring, etc.).
- **Capa de red (también conocida como capa IP).** Permite que se transmitan datos entre máquinas de una red basada en TCP/IP, independientemente de la naturaleza de las redes por las cuales pase la información. Los datos se encaminan hacia su destino por medio de esta capa. IP es su protocolo más conocido.
- **Capa de transporte.** Proporciona servicios de entrega de los datos desde el origen al destino. Los dos protocolos dentro de TCP/IP que suelen implementar esta capa son TCP (*Transmission Control Protocol*: protocolo de control de la transmisión) y UDP (*User Datagram Protocol*: protocolo de datagramas del usuario).
- **Capa de aplicación.** Proporciona toda la lógica correspondiente a las aplicaciones para los usuarios. Los protocolos que implementan esta capa se encargan de definir las condiciones exactas para solicitar y recibir servicios, y de proporcionar interfaces para los usuarios. Los protocolos más conocidos de esta capa son HTTP (*HyperText Transfer Protocol*: protocolo de transferencia de hipertexto), Telnet, FTP (*File Transfer Protocol*: protocolo de transferencia de archivos), SMTP (*Simple Mail Transfer Protocol*: protocolo de transferencia de correo electrónico).

Mail Transfer Protocol: protocolo de transferencia de correo simple), Kerberos, DNS (*Domain Name System*: sistema de nombres de dominio) e ICMP (*Internet Control Message Protocol*: protocolo de mensajes de control de Internet).

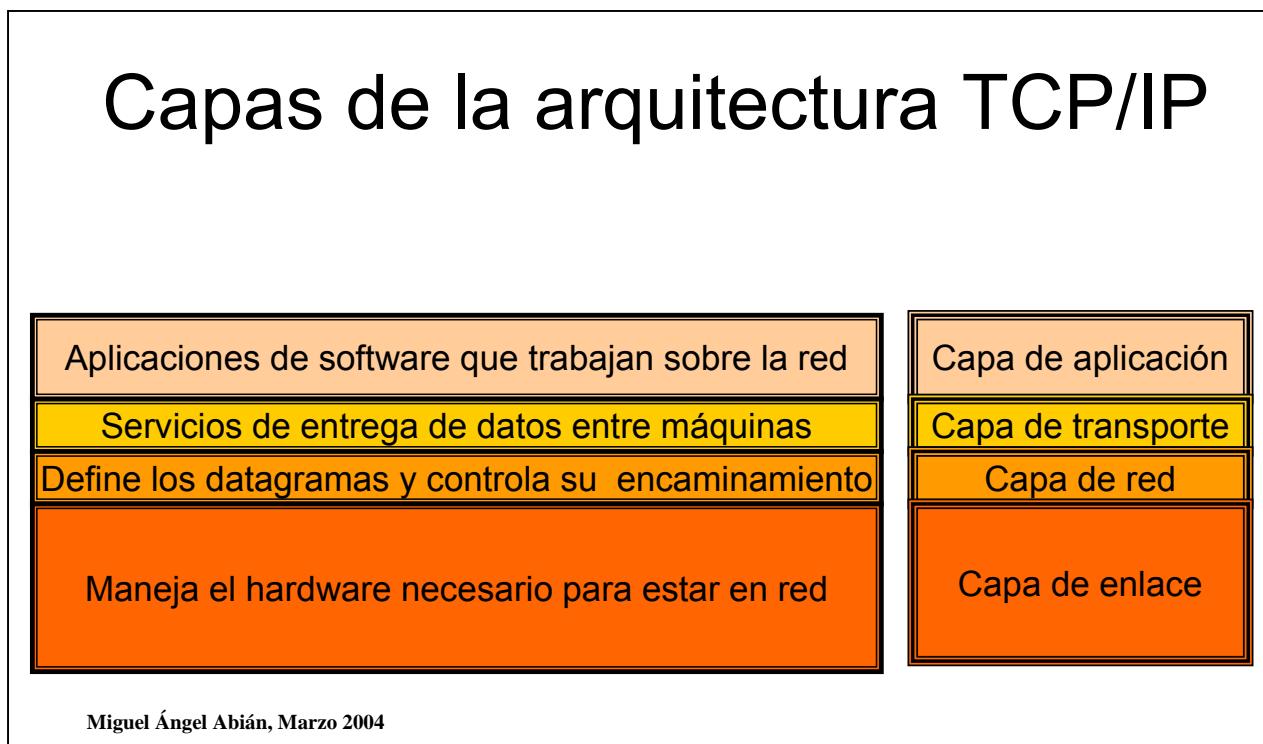


Figura 11. Las capas de la arquitectura TCP/IP

Algunos autores sostienen que la capa de enlace debería manejar sólo los detalles del hardware que interacciona con el medio físico de transporte de los datos: niveles de voltaje, sincronización de cambios de voltaje, distancia entre cables, etc.; no los detalles del medio físico. Si se acepta esta idea, debe incluirse una nueva capa en el modelo: la **capa física**, encargada de transmitir los datos en un medio físico. Así, un módem o una tarjeta de red realizarían funciones en la capa de enlace y en la física, pues transforman secuencias de bits en señales normalizadas que pueden transmitirse mediante un medio físico, y gestionan la activación y desactivación de la conexión física.

Si se incorpora la capa física al modelo TCP/IP de cuatro capas, el reparto de funciones entre la capa física y la de enlace queda así:

- Capa física.** Se encarga de convertir las secuencias de bits en señales físicas y de transmitirlas al medio físico; para ello, define la representación de los bits en estados físicos (medio de transmisión, asignación de voltaje a ceros y unos, definición del tiempo asignado a cada bit, definición de establecimiento de conexión y de su fin, etc.).
- Capa de enlace.** Define las tramas (tamaño, secuencia de bits para el comienzo y el fin). En la máquina emisora, la capa de enlace descompone los datos procedentes de capas más altas en secuencias de tramas y envía las tramas a la capa física en forma de secuencias de bits. En la máquina receptora, recibe secuencia de bits de la capa física,

las traduce en tramas –comprobando también si son tramas válidas– y envía tramas de correcta recepción a la máquina emisora.

TCP/IP funciona sobre el concepto de **modelo cliente-servidor**. Suele usarse el término **servidor** para significar una aplicación (o proceso) que se ejecuta en una máquina en red (o en varias) y realiza dos funciones: a) acepta peticiones de procesos que se ejecutan en otras máquinas en red; y b) contesta a dichas peticiones proporcionándoles un servicio. Los procesos que realizan las peticiones se llaman **clientes**. Por regla general, los clientes son activos (hacen solicitudes) y los servidores pasivos (esperan solicitudes). Su tiempo de vida no coincide: los clientes dejan de ejecutarse cuando se paran las aplicaciones de las cuales forman parte; los servidores se ejecutan de forma continua. Una **aplicación cliente-servidor** admite la división en dos o más procesos donde cada uno es cliente o servidor. Esta separación es lógica, no física: una aplicación cliente-servidor puede ejecutarse en una máquina (con sendos procesos para el cliente y el servidor) o en anfitriones separados por miles de kilómetros. En una aplicación en red, resulta habitual que a la separación lógica le corresponda una física.

La distinción entre clientes y servidores no siempre resulta tan clara como la presento. Un servidor puede actuar como servidor para un proceso y como cliente para otro. Así, por ejemplo, un servidor que necesite consultar a otro para ofrecer su servicio a los clientes será cliente para el segundo. Para este tutorial, usaré cliente y servidor para designar **los papeles mutuos desempeñados en una sola comunicación**.

Las palabras *cliente* y *servidor* también suelen usarse para referirse a los anfitriones o nodos en los que se ejecutan procesos de uno u otro tipo.

Una clasificación muy extendida de los modelos cliente-servidor es ésta:

- **Modelos cliente-servidor de presentación descentralizada.** La presentación (gestión de la interacción con el usuario, gráficos, etc.) corre a cuenta del cliente. El servidor gestiona la lógica de la aplicación (o reglas de negocio) y los datos. En un sistema bancario, por ejemplo, una regla de negocio puede ser “No se permite sacar dinero de cuentas con saldos negativos”.
- **Modelos cliente-servidor de lógica distribuida.** El cliente se encarga de la presentación y de parte de la lógica de la aplicación; el servidor se encarga del resto de la lógica de la aplicación y de todos los datos. Por ejemplo, en un sistema bancario de lógica distribuida, los clientes pueden encargarse de avisar de la introducción de valores inválidos: códigos de cuenta incompletos, códigos fiscales erróneos, etc.
- **Modelos cliente-servidor de lógico descentralizada.** En el cliente reside la presentación y toda la lógica de la aplicación. El servidor sólo se encarga de los datos; viene a ser un almacén de datos.

La *World Wide Web*, que usa el protocolo HTTP, obedece fielmente al modelo cliente-servidor. Los clientes son las aplicaciones que permiten consultar páginas *web* (navegadores); y los servidores, aquellas que suministran (“sirven”) páginas *web*. Cuando un usuario introduce una dirección *web* en el navegador (un URL), éste solicita, mediante el protocolo HTTP, la página *web* al servidor *web* que se ejecuta en el ordenador servidor donde reside la página. El servidor *web* envía la página por Internet al cliente (navegador) que se la solicitó, y este último la muestra al usuario.

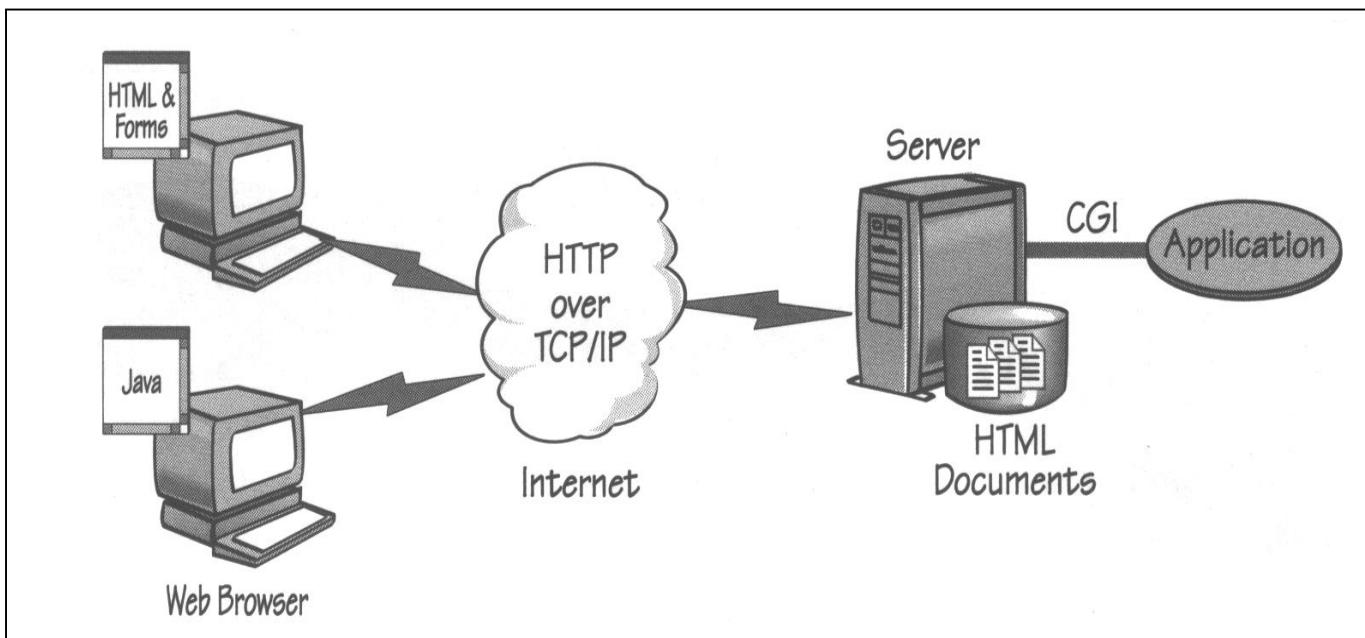


Figura 12a. Ejemplo de aplicación cliente-servidor: la Web

EL MODELO CLIENTE-SERVIDOR EN LA WEB

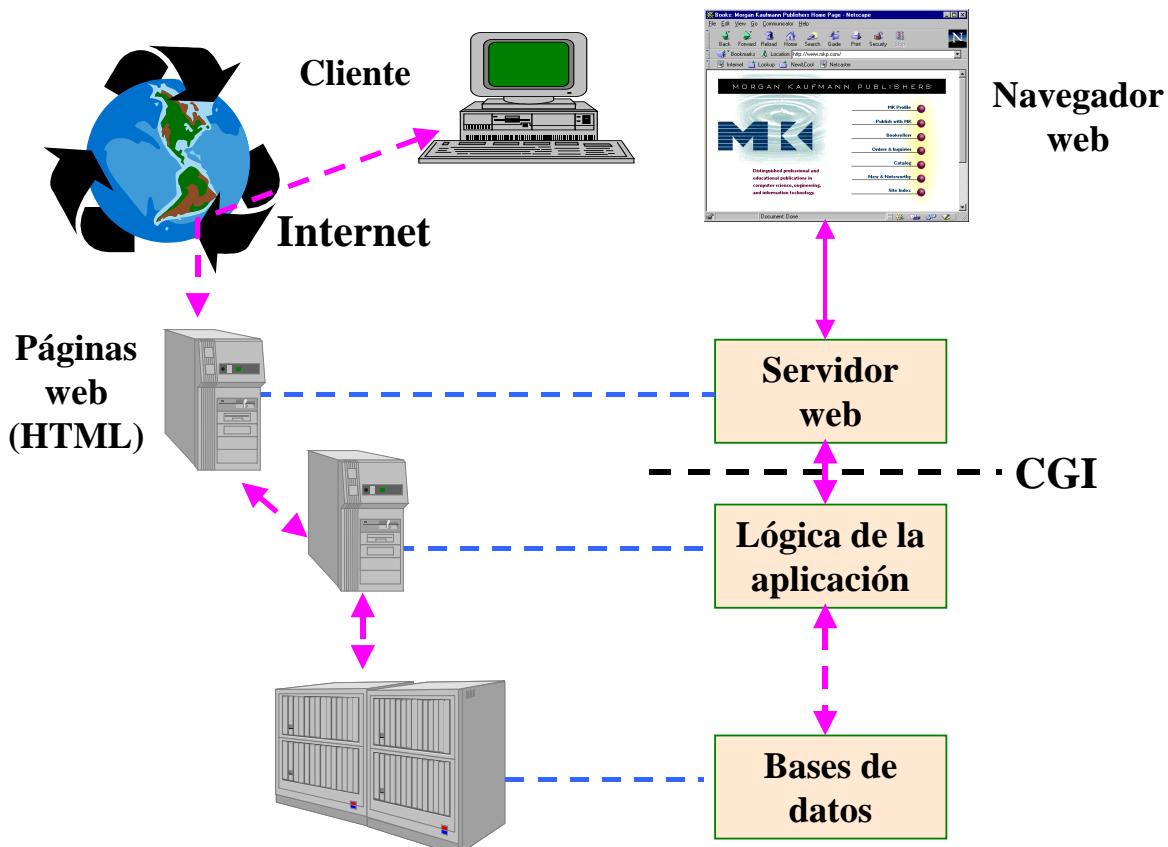


Figura 12b. Otra vista de la figura 12a

2.4. TCP/IP: Un conjunto de protocolos

Tal como se dijo al principio del subapartado 2.2, las siglas TCP/IP se usan también para designar un **conjunto o familia de protocolos** (IP, TCP, UDP, Telnet, FTP, Telnet, etc.), correspondientes a las capas vistas en el subapartado 2.3.

Ya se adelantó en 2.3 que los protocolos son posibles modos de funcionamiento de una capa. En la capa de transporte, por ejemplo, existen dos protocolos (UDP y TCP) que definen dos formas alternativas de funcionamiento (con conexión o sin ella). Donde más protocolos alternativos existen es en la capa de aplicación; dependiendo del tipo de servicio que precise el usuario, se usará uno u otro. La navegación por Internet se hace posible gracias al protocolo HTTP, que no forma parte del paquete estándar de protocolos TCP/IP. Protocolos más veteranos, como Telnet o FTP, sí se incluyen siempre en este conjunto de protocolos.

La unión de las capas vistas en 2.3 con los protocolos que se verán en este apartado forma una completa arquitectura de comunicaciones, mostrada en la figura 13.

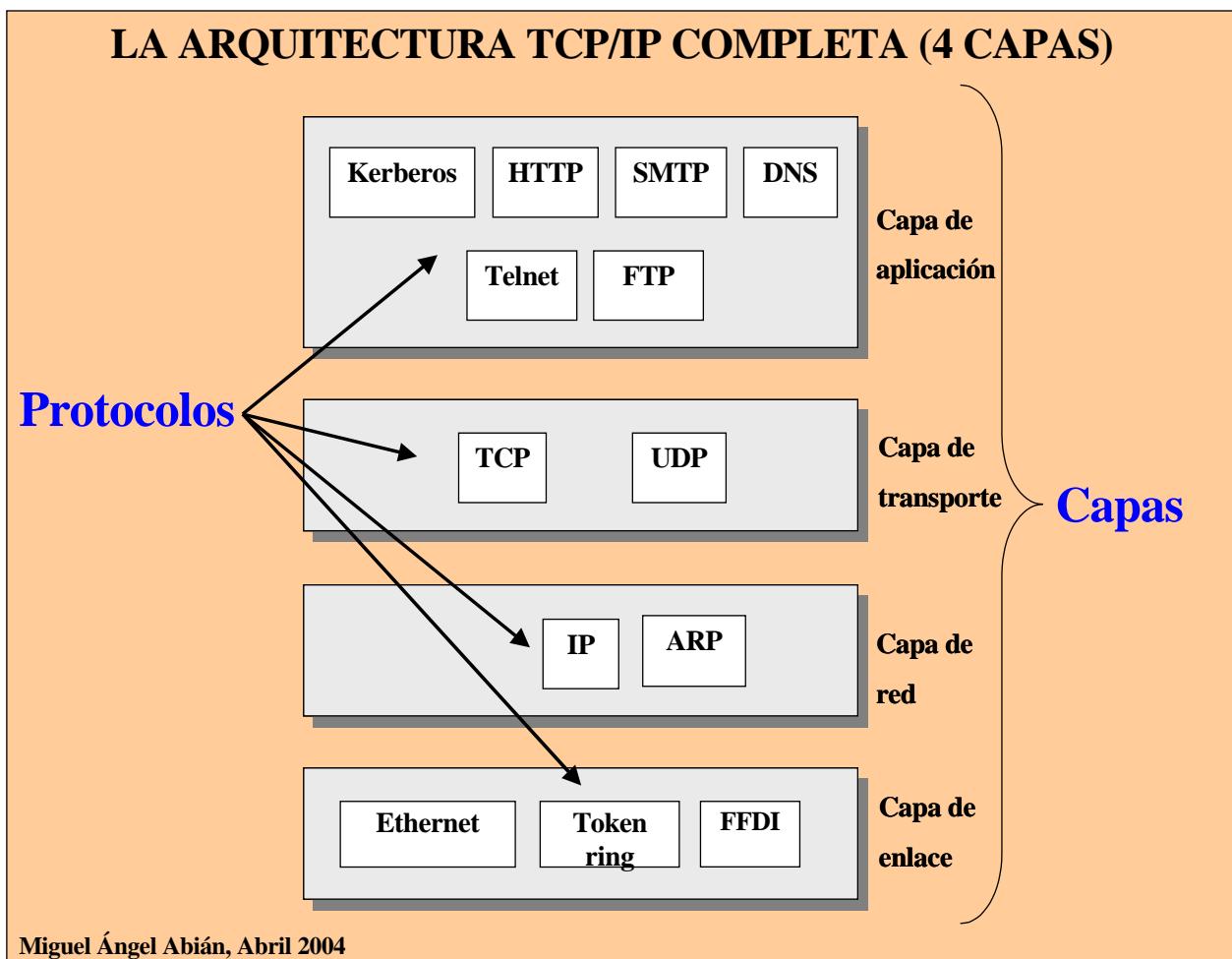
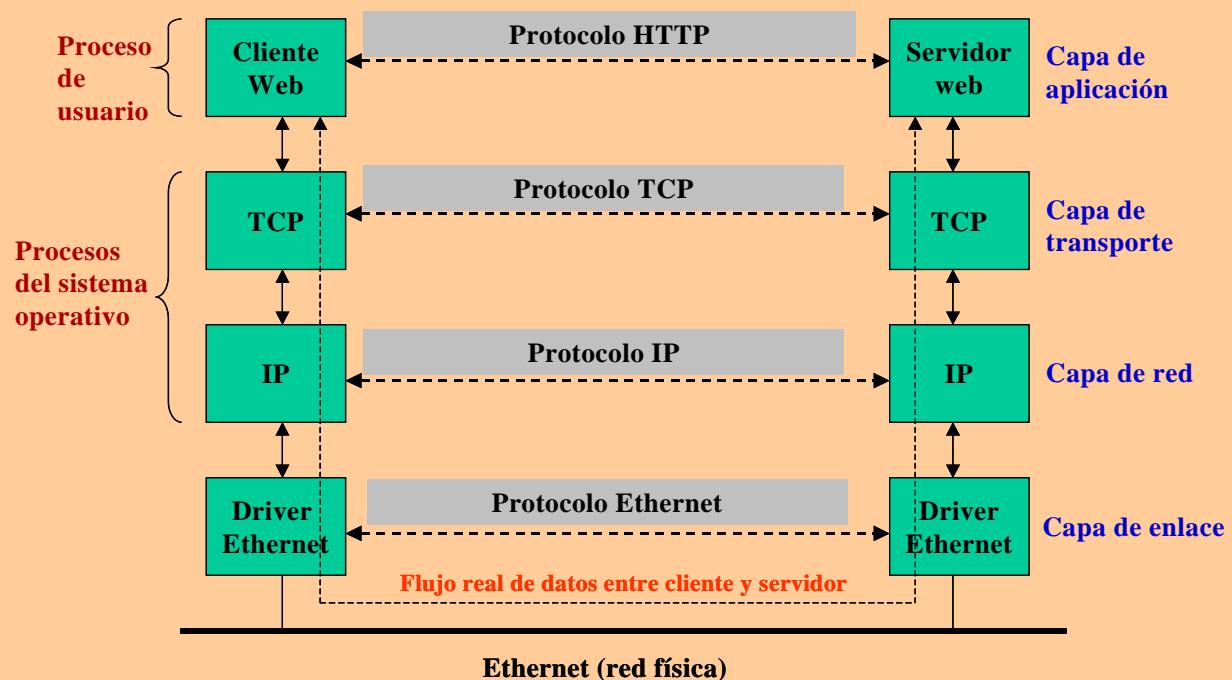


Figura 13. La arquitectura TCP/IP: un conjunto de capas y protocolos

En la figura 14 se muestra cómo encaja el protocolo de aplicaciones HTTP en la arquitectura TCP/IP (supongo que la red física corresponde al tipo Ethernet):

Acceso a una página web desde la perspectiva de la arquitectura TCP/IP



Miguel Ángel Abián, Marzo 2004

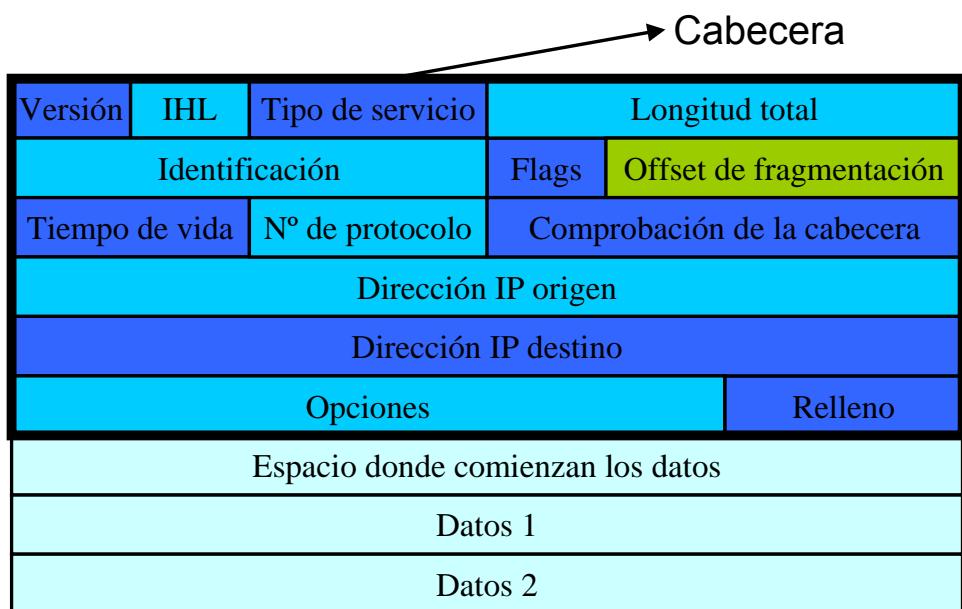
Figura 14. El ejemplo de la figura 12 desde un punto de vista arquitectónico.

El **protocolo IP** es el protocolo de red utilizado para enviar datos entre los ordenadores conectados a Internet o que forman parte de intranets o extranets. Es la piedra angular sobre la cual se sostiene el caótico edificio que es –aparentemente– Internet; pues los paquetes de datos contienen paquetes IP o datagramas, es decir, paquetes con la forma definida por este protocolo. Simplificando un poco, este protocolo realiza tres funciones básicas:

- Define el **datagrama**, el paquete atómico de información en la capa de red de cualquier arquitectura TCP/IP (Internet es la red TCP/IP más grande que existe).
- Define el esquema de direcciones (IP) de las redes TCP/IP.
- Se encarga de trasladar los datos entre la capa de enlace y la de transporte.
- Controla el **proceso de fragmentación y ensamblado de los datagramas**. Ninguna red puede transmitir tramas de tamaño ilimitado: todas tienen una unidad de transferencia máxima (*Maximum transfer unit* o MTU) que define el tamaño máximo de las tramas que pueden ser enviadas a través de una red física dada. Si un datagrama no cabe

dentro de una trama física, este protocolo especifica cómo ha de dividirse y cómo los fragmentos han de ensamblarse en el destino.

Esquema de un datagrama



Miguel Ángel Abián, Marzo 2004

Figura 15. Esquema de un datagrama, el átomo lógico de Internet

Los datagramas (a veces se llaman datagramas IP) son paquetes de datos, definidos por el protocolo IP. Todo datagrama cuenta con un origen y un destino; pero no existe conexión: no hay dos extremos en la comunicación. Asimismo, es independiente de los enviados antes o después de él. En ocasiones, un datagrama debe dividirse en otros de menor tamaño; esta situación resulta corriente cuando se intentan pasar datagramas entre subredes físicas muy distintas. Básicamente, todo datagrama consta de dos partes: datos y cabecera (donde figura la fuente, el destino y el tipo de datos que contiene).

IP es un protocolo **sin conexión** y **sin fiabilidad**. Sin conexión, porque no incorpora ningún mecanismo para intercambiar información de control antes de enviar datos a una máquina receptora ni mantiene información de estado entre datagramas sucesivos (cada uno se maneja independientemente de los otros, y pueden alcanzar su destino en un orden distinto al de salida). Sin fiabilidad, porque no existe garantía de que un datagrama llegue correctamente a su destino: no se encarga de reenviar los paquetes perdidos o defectuosos (suele llamarse *chernobylgram* –*Chernobyl datagram*– a aquel paquete tan defectuoso que causa la “fundición” de la máquina receptora” –es decir, que la deja fuera de funcionamiento–; los creadores de protocolos

sólo se inspiran en las realidades más dramáticas para dar nombres a sus conceptos: nadie usa *lovegram*, *peacegram* o términos similares).

Bajo el párrafo anterior subyace una importante pregunta: ¿cómo se pueden construir comunicaciones fiables sobre un protocolo base (IP) carente de toda fiabilidad? La respuesta se verá cuando se explique el protocolo TCP.

Cada anfitrión en una red TCP/IP (o, simplemente, IP) se identifica mediante, **al menos**, una dirección IP (existen protocolos que transforman las direcciones físicas en direcciones IP, y viceversa). Estas direcciones son lógicas, no físicas: son independientes del hardware y de la naturaleza física de las redes.

Cada dirección IP tiene dos partes; una identifica a una red, y la otra a un anfitrión o nodo dentro de la red. Actualmente, la versión en vigor de este protocolo es la IPv4, que utiliza 4 bytes (32 bits) para cada dirección IP. En formato decimal, una dirección IP consiste en cuatro números enteros, menores de 256, separados por puntos (por ejemplo, 147.156.203.5).

Que una máquina tenga más de una dirección IP es, en ocasiones, imprescindible. Una máquina que actúe como encaminador, por ejemplo, interconecta redes distintas. Como una dirección IP implica una red, resulta imprescindible que el encaminador tenga varias direcciones IP, cada una correspondiente a una conexión de red.

Mediante el **DNS** (*Domain Name Service*: servicio de nombres de dominio), los usuarios pueden referirse a los anfitriones con nombres (por ejemplo, www.aidima.es) en lugar de con ristas de números. DNS es un servicio de nombres distribuido que permite a los usuarios de Internet referirse a los anfitriones con nombres sencillos de recordar en lugar de con direcciones IP. La representación de direcciones IP mediante nombres de dominio resulta muy conveniente para los usuarios; pero no es empleada por el protocolo IP, que sólo maneja direcciones IP. Así pues, en toda petición hecha con un nombre de dominio debe convertirse el nombre de dominio en una dirección IP usando el protocolo DNS.

Aunque muchos autores usan *datagrama* para cualquier paquete de datos que viaje por una red, no me parece que sea una elección acertada: creo que ocasiona problemas conceptuales y que resulta beneficioso precisar un poco más y manejar con cautela los términos. Ninguna red transmite directamente datagramas: en realidad, por el medio de transmisión se transmiten **tramas físicas** (*physical frames*) o **tramas físicas de red**; cuando un datagrama viaja por una red, lo hace encapsulado dentro de la trama física usada por la red que lo alberga. Una trama física es la representación física (mediante señales eléctricas, pulsos de láser, ondas electromagnéticas, etc.) de una secuencia de bits bien delimitada; la secuencia de bits constituye la **trama**, a secas. Una trama es un paquete que ha sido codificado para ser enviado por un medio físico. Los protocolos de la capa de enlace definen la forma de las tramas y se encargan de la correcta representación de la información digital en las tramas físicas correspondientes al medio físico subyacente. En el caso de un módem, cuando recibe una trama física (analógica), la convierte en una trama (digital).

Una trama física lleva dentro de ella un datagrama encapsulado, pero no es un mero datagrama. Por decirlo de una manera simplificada, un datagrama siempre es un viajero lógico en una trama física. Si la información que contiene un datagrama tiene que pasar de una red a otra mediante un encaminador (*router*), se extrae el datagrama de la trama física que lo alberga y se pone en una nueva trama física de la otra red (que puede ser completamente distinta a la primera). Como un datagrama es un concepto lógico, no físico, puede viajar, con distintas tramas físicas, entre redes

Ethernet, Token ring, ATM, X2.5, etc. El vehículo cambia, es de quitaipón, pero el mensajero siempre es el mismo. Vagones de metro pintarrajeados de *graffitis* que cualquiera rasparía con saña si apareciesen en su casa, motocicletas, autobuses que circulan por vías de peaje, bicicletas, motocicletas, aviones militares, etc.: todos estos medios de transporte, convenientemente traducidos al mundo digital, serán usados – tarde o temprano – por casi todos los mensajeros.

Si al lector le agrada la metáfora antropomorfa del mensajero, puede considerar que todo datagrama es un mensajero encargado de viajar a lejanos lugares para entregar un mensaje (los datos) que siempre lleva consigo. La capa de enlace da visado al pasajero y lo embute, amable pero firmemente, en un medio de transporte. El mensajero, por supuesto, siempre tiene que tener a mano el visado, pues se lo exigen cada vez que cruza una frontera (encaminador). Si el visado se deteriora (por ejemplo, si se borra el lugar de nacimiento o el de destino), el mensajero queda en una posición comprometida, pues tendrá problemas para salir de la zona donde se encuentra. Asimismo, si pierde o deteriora el mensaje que transporta, su viaje pasa a carecer de sentido.

Para diferenciar entre datos correspondientes a distintas capas de TCP/IP, recomiendo usar esta terminología para las unidades de datos:

- **Mensajes**, para la capa de aplicación.
- **Segmentos** (TCP o UDP), para la capa de transporte.
- **Datagramas**, para la capa de red o IP.
- **Tramas**, para la capa de enlace.
- **Tramas físicas**, para la capa física.

Más adelante, usaré todos estos términos para mostrar el proceso completo de transmisión de datos con el protocolo UDP.

Para ilustrar cómo se transmite físicamente la información entre redes TCP/IP, consideraré un caso muy sencillo: la solicitud de una página web. No olvidemos que Internet es **una red compuesta por subredes sumamente heterogéneas, interconectadas unas con otras mediante encaminadores (routers)**. Cuando el usuario hace su petición mediante un navegador, su petición se va encapsulando (el proceso se muestra en la figura 16; se añade una cabecera por cada capa) hasta que acaba en una trama (un datagrama encapsulado, listo para ser transmitido por un medio físico), la cual se envía a la subred local transformada en una señal física (una onda eléctrica portadora, un modo no evanescente de una fibra óptica, una onda electromagnética, etc.), llamada trama física. La subred local dirige la trama física hacia Internet mediante su encaminador.

Una vez cruzada el encaminador de la subred emisora, la trama física va de encaminador en encaminador a través de Internet hasta que pueda entregarse de forma directa. Cada vez que la trama llega a un encaminador, suceden dos hechos claramente diferenciados: a) el hardware convierte la trama física en una trama; b) luego, el software IP extrae el datagrama encapsulado dentro de la trama, selecciona el siguiente encaminador en algún camino hacia la máquina de destino, y lo envía a través de alguna subred a la que tenga acceso directo, transformándolo antes en una trama física adaptada a la nueva subred subyacente.

Cuando se da el caso de que un encaminador tiene la máquina de destino en alguna de sus subredes, éste envía la trama física a la máquina correspondiente. En

esta última se llevan a cabo dos procesos: a) la capa de enlace (hardware y controladores) transforma la trama física en una trama; b) a continuación, la trama pasa por las tres capas superiores; c) finalmente, la capa de aplicación se encarga de pasar el mensaje contenido en el datagrama encapsulado a la aplicación adecuada (servidor web), que enviará el archivo HTML solicitado hacia la máquina que hizo la petición. La situación se representa en la figura 17.

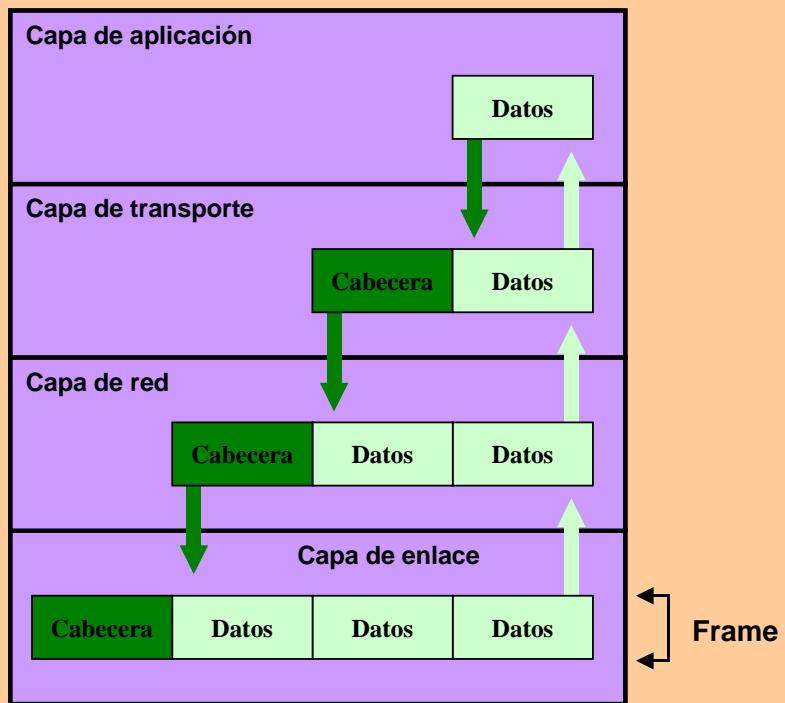
En todo lo dicho en el párrafo anterior se sobreentiende que la máquina a la que se solicita la página no forma parte de la subred a la que pertenece la máquina cliente, pues si así fuera no se necesitarían encaminadores: la entrega sería directa. De hecho, ni siquiera sería necesario usar el protocolo IP, pues el direccionamiento físico de los números de las tarjetas de red bastaría para entregar correctamente la petición.

¿Cómo sabe un encaminador si la trama va destinada a sus subredes? El datagrama dentro de la trama tiene una cabecera en la que consta cuál es su destino, que es examinada por cada encaminador. Si la máquina de destino que figura en la cabecera no pertenece a ninguna máquina de las subredes correspondientes al encaminador, éste envía la trama de vuelta a Internet, hacia otros encaminadores. La comprobación de la cabecera no se hace en la capa de aplicación –sería como pagar una carta contra reembolso sin comprobar que el mensaje es para uno–, sino en la capa de enlace.

Hilando fino, uno podría preguntarse cómo sabe un encaminador si la máquina de destino pertenece a sus subredes. La respuesta reside en que todas las direcciones IP de las máquinas dentro de una misma red incluyen un prefijo común (una secuencia de dígitos), tal como ya se dijo antes. En consecuencia, el encaminador sólo tiene que mirar la dirección IP del datagrama encapsulado y comparar el prefijo de ésta con los de sus subredes.

Nota: Cuando en una comunicación han intervenido encaminadores, la capa de red de la máquina receptora **no** recibe exactamente el mismo objeto enviado por la correspondiente capa de red de la máquina emisora. Lo usual es que se modifique algún campo de la cabecera. Para la capa de transporte y la de aplicación se cumple a rajatabla la identidad entre lo emitido y lo transmitido.

Encapsulación progresiva de los datos transmitidos



Miguel Ángel Abián, Marzo 2004

Figura 16. Encapsulación de datos en una sesión HTTP

Los encaminadores no se limitan a enviar paquetes de un lado para otro; también se encargan de escoger las mejores rutas o caminos para ellos (entiéndase por "mejor camino" el más rápido). Para ello, cada encaminador mantiene actualizada una tabla de encaminamiento (*routing table*) que registra las mejores rutas para ciertos destinos de la red. Para saber que una ruta es mejor que otras se usan algoritmos que involucran factores como el tráfico de la red, el ancho de banda, la fiabilidad, etc. Como puede suponerse, las tablas de encaminamiento se van actualizando regularmente (los parámetros anteriores varían con el tiempo); de esto se encargan los protocolos de enrutamiento: OSPF (*Open Shortest Path First*), BGP (*Border Gateway Protocol*), etc.

Cuando el encaminador (o parte de él) es un ordenador, no se necesita obligatoriamente que sea un equipo con grandes prestaciones (memoria, microprocesador, etc.). Un encaminador no guarda información sobre las máquinas dentro de las redes a las que se haya conectado, sólo sobre las subredes dentro de estas redes. En consecuencia, la información que necesita guardar depende del número de subredes dentro de cada red a la que está conectado, no del número de ordenadores de las redes. Un encaminador no utiliza directamente el nodo de destino cuando encamina una trama, sólo la red de destino.

En media, cada vez que se envía un correo electrónico o se solicita una página web, los datagramas asociados atraviesan entre diez y veinte redes. Un datagrama suele tardar unos seiscientos milisegundos en cruzar veinte redes (es decir, veinte

encaminadores). En todas las comunicaciones se pueden perder tramas (y, por tanto, datagramas) o pueden quedarse huérfanas: las redes físicas pueden fallar, las máquinas de destino (o de origen) pueden desconectarse de Internet, de la intranet o la extranet, etc. Cuando uno aprende de qué manera se encaminan los datagramas por Internet, suele preguntarse si de verdad este caótico juego de ping-pong da lugar a comunicaciones fiables. Pues las da: muy mal deben de estar las redes intermedias para que se registre más de un cinco por ciento de paquetes perdidos. Sea cual fuere el estado de las redes que actúen como infraestructuras de telecomunicación (siempre que exista alguna, por supuesto), existen maneras de garantizar la entrega y recepción de paquetes, tal como veremos más adelante.

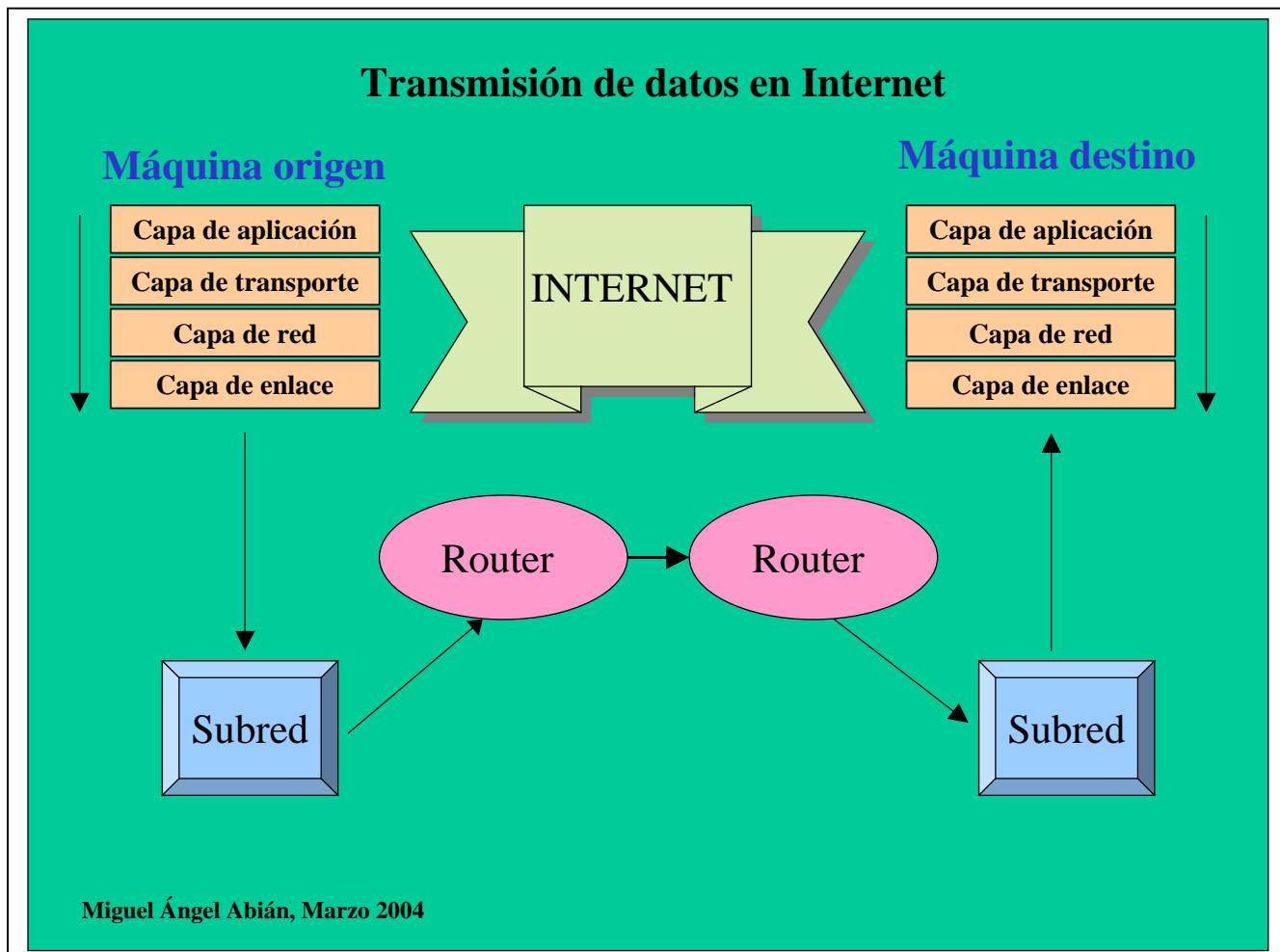


Figura 17. Esquema de la transmisión física de datos en Internet

Cualquier usuario de un ordenador sabe que puede ejecutar varios procesos al mismo tiempo. Por ejemplo, no es raro que se consulten a la vez distintas páginas web y se use a la vez alguna aplicación cliente de correo. Se puede, en consecuencia, intercambiar información con más de un anfitrión a la vez. La ejecución simultánea de distintos procesos en una misma máquina resulta posible gracias a los puertos, que permiten distinguir entre distintos procesos que se ejecutan en la misma máquina. Un **puerto** es una abstracción lógica, pues no corresponde a ningún objeto del mundo real (no tiene ninguna relación con un puerto paralelo para impresoras, que sí es algo tangible). Igual que un anfitrión se identifica de forma única por su dirección IP, un

proceso se identifica de forma única por la dirección IP de la máquina donde se ejecuta y por el número de puerto que tiene asociado. Mediante el uso de puertos, los protocolos TCP Y UDP usan **una capa extra de abstracción sobre la capa de red**.

Al igual que el protocolo IP permite la comunicación entre ordenadores de Internet (o de una extranet o intranet), los puertos permiten la comunicación entre programas o aplicaciones específicas –genéricamente, entre procesos– que se ejecutan en aquellos ordenadores. Cuando un datagrama atraviesa Internet, lleva en su interior la dirección IP del sistema que lo generó y el puerto del sistema local al que está ligado. Los encaminadores que mueven datagramas entre redes no leen ni manipulan la información relacionada con el puerto; la máquina de destino es la única que emplea esta información.

Cuando se considera una arquitectura cliente-servidor (por ejemplo, el ejemplo anterior del servidor web y el navegador cliente), el programa servidor permanece a la escucha de peticiones por parte de los clientes a través de los puertos que tiene activos. Las informaciones solicitadas se envían a los clientes mediante los puertos que éstos han utilizado para realizar las peticiones. En general, los servicios más conocidos de Internet tienen asignados puertos por defecto. Por ejemplo, el protocolo HTTP utiliza el puerto estándar 80, y el protocolo FTP el puerto estándar 21. Por ello no es preciso escribir en un navegador <http://www.aidima.es:21>, basta con escribir <ftp://www.aidima.es>.

Es posible utilizar puertos distintos de los estándar para los servicios de Internet, si bien no es habitual. Se podría perfectamente, por ejemplo, asignar el puerto 2378 al protocolo HTTP y el puerto 3216 al protocolo FTP. Los clientes necesitarán, para intercambiar datos con el servidor, conocer los nuevos puertos asociados a cada servicio.

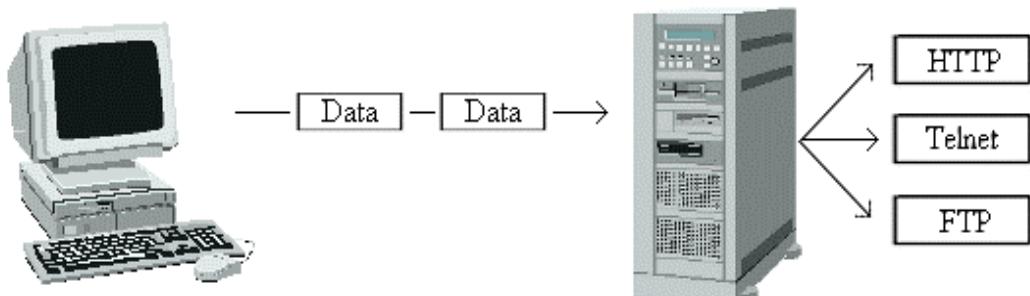
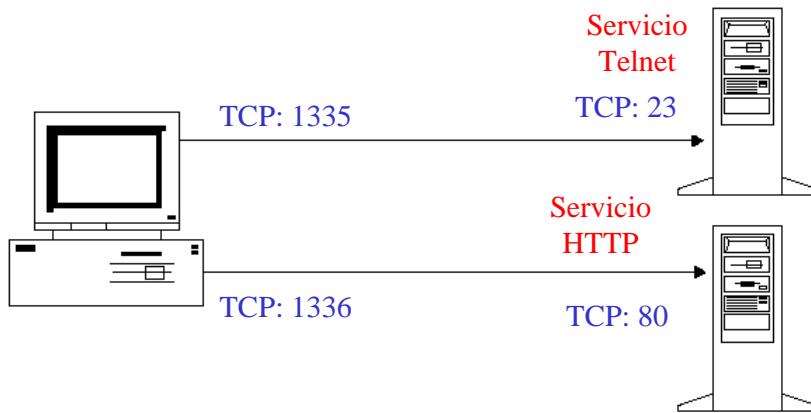


Figura 18. Un servidor proporciona distintos servicios mediante distintos puertos

Ejemplo de peticiones simultáneas por distintos puertos



Mediante la identificación mediante IP y números de puerto, el servidor sabe qué aplicación de un cliente solicita un determinado servicio.

Figura 19. Ejemplo de peticiones simultáneas por distintos puertos

En la siguiente tabla aparecen los puertos asociados a los servicios más habituales de la capa de aplicación en una arquitectura TCP/IP:

Servicio	Número Puerto
Canal de datos FTP	20
Canal de control FTP	21
Telnet	23
Simple Mail Transfer	25
Gopher	70
Finger	79
Hipertexto (HTTP)	80
USENET	119

Figura 20. Ejemplo de algunos puertos y servicios populares

Existen dos protocolos de transporte (TCP y UDP) en la arquitectura TCP/IP que se encargan de enviar datos de un puerto a otro para hacer posible la comunicación entre aplicaciones o programas.

El protocolo TCP (*Transmission Control Protocol*: protocolo de control de la transmisión) ofrece tres características fundamentales:

- **Tolerancia a fallos (fiabilidad).** Las transmisiones de datos mediante TCP se dividen en segmentos; si se pierden o resultan dañados, el protocolo lo detecta y se encarga de volver a transmitir los segmentos que faltan o que llegaron incompletos.
- **Manejo de corrientes continuas de datos.** Una vez establecida una conexión, el protocolo permite que los datos se transmitan de forma continua.
- **Orientación a la conexión.** Siempre se transmite información de control antes de comenzar la comunicación; lo mismo ocurre antes de una desconexión. Suele usarse la expresión *circuito virtual* para referirse a las comunicaciones entre las dos máquinas finales.

TCP funciona así: cuando la capa de aplicación envía un flujo (*stream*) de bytes que corresponden a una petición, el flujo se convierte en una sucesión de segmentos, cada uno con un tamaño máximo de 64 K (muchas veces se usan 1.500 bytes). A cada paquete se le añade una cabecera (que incluye un *checksum*, además de una secuencia numérica si la petición consta de más de un paquete). Luego, cada uno se envía a la capa de red, junto con la dirección IP de destino y el número de puerto, donde se transforma en un datagrama.

Nota: Un *checksum* es un número que se calcula usando un algoritmo sobre los datos que envía una máquina. Cuando son recibidos por la máquina destino, se vuelve a aplicar otra vez el algoritmo. Si el número obtenido a partir de los datos recibidos coincide con el inicial, se considera que los datos se han recibido correctamente.

Cuando los datagramas lleguen a su destino, transmitidos dentro de tramas físicas, serán entregados a un proceso TCP, que reconstruirá el flujo original de bytes enviados por la máquina origen o fuente.

La transmisión de datos mediante este protocolo es asíncrona: existe un retraso voluntario en las transmisiones. Cada vez que se envía un paquete, se tarda un tiempo en enviar el siguiente. Este tiempo basta para que el paquete de datos llegue a la aplicación TCP en el destino, para que ésta envíe un mensaje de aceptación y para que este mensaje llegue a la máquina origen. En el mensaje de aceptación se especifica cuál es el próximo paquete esperado por la aplicación TCP del destino. Si el tiempo se acaba sin que se haya recibido ningún mensaje de aceptación, la máquina emisora reenvía el paquete. Gracias a este mecanismo de espera, **pueden conseguirse comunicaciones fiables a partir de un protocolo base intrínsecamente falto de fiabilidad (IP)**.

Vemos, pues, que la vida del mensajero de la página 37 es limitada: dispone de un tiempo límite para alcanzar su destino y entregar su mensaje; es como si viajara con un cronómetro que ha comenzado su cuenta atrás. Si no puede entregarlo en dicho tiempo, es sustituido por otro mensajero idéntico a él, y así sucesivamente. La travesía del primer viajero no tiene por qué coincidir con la del segundo o con la de cualquier otro que lo sustituya. En el tiempo transcurrido entre el comienzo del viaje de uno y el

del otro, la ruta de viaje puede variar por muchísimos motivos: un puente puede haberse hundido, una zona puede haber quedado inundada, una autopista puede tener menos tráfico que antes, etc.

Por completitud, a continuación detallo la secuencia exacta de pasos que realiza cualquier proceso que utilice TCP (se omite cualquier detalle relativo a los pasos intermedios de encaminamiento de los paquetes, ya vistos en la página 37):

- 1) La capa de aplicación de la máquina emisora transforma la petición del proceso de usuario en un flujo de bytes (mensaje) que transfiere a la capa de transporte.
- 2) La capa de transporte trocea el flujo en segmentos TCP y añade una cabecera a cada segmento TCP, la cual contiene el *checksum* del segmento y un número de secuencia. Acto seguido, los segmentos TCP se pasan a la capa de red.
- 3) La capa de red crea un datagrama a partir de cada segmento que recibe de la capa de transporte. El área de datos del datagrama contiene los datos del segmento; la cabecera lleva las direcciones IP de la máquina de origen y de la máquina de destino, además de un nuevo *checksum*. Esta capa también determina la dirección física (dirección MAC o Ethernet en una red Ethernet) de la máquina de destino o del encaminador más próximo. Luego pasa el datagrama a la capa de enlace.
- 4) La capa de enlace transforma el datagrama en una trama, colocando los datos del datagrama en el área de datos de la trama y añadiendo una nueva cabecera. En este paso, se reemplazan las direcciones IP por direcciones físicas de red. En las redes Ethernet y Token ring se usa para ello el protocolo ARP (*Address Resolution Protocol*).
- 5) La capa física transforma la trama en una trama física y la envía a la red.
- 6) Cuando la trama física llega a la máquina de destino, la capa de enlace extrae de ella la trama que contiene. A continuación, descarta la cabecera de la trama y transfiere los datos, en forma de datagrama, a la capa de red.
- 7) La capa de red calcula el *checksum* del datagrama. Si no coincide con el valor que figura en la cabecera, se descarta el datagrama (los datos se han dañado en el trayecto: el mensaje del recadero se ha estropeado); en este caso, la máquina origen volverá a enviarlo al poco tiempo.
- 8) Si el *checksum* coincide con el valor en la cabecera, la capa de red elimina la cabecera IP y envía el segmento TCP resultante a la capa de transporte.
- 9) La capa de transporte calcula el *checksum* del segmento. Si no coincide con el valor que figura en la cabecera, se descarta el segmento. Si coincide, se comprueba que el número de secuencia es correcto.
- 10) Si el número de secuencia y el *checksum* del segmento TCP son correctos, la capa de transporte envía un mensaje de recepción correcta

a la máquina de destino y pasa los datos del segmento a la capa de aplicación.

- 11) La capa de aplicación ensambla en el orden correcto los datos que le vienen de la capa de transporte, simulando un flujo continuo de bytes.

Los pasos anteriores se reflejan, simplificados, en la figura siguiente. Se considera una aplicación cliente-servidor de tipo web que funciona en una red Ethernet, y se emplea una representación en cinco capas de la arquitectura TCP/IP.

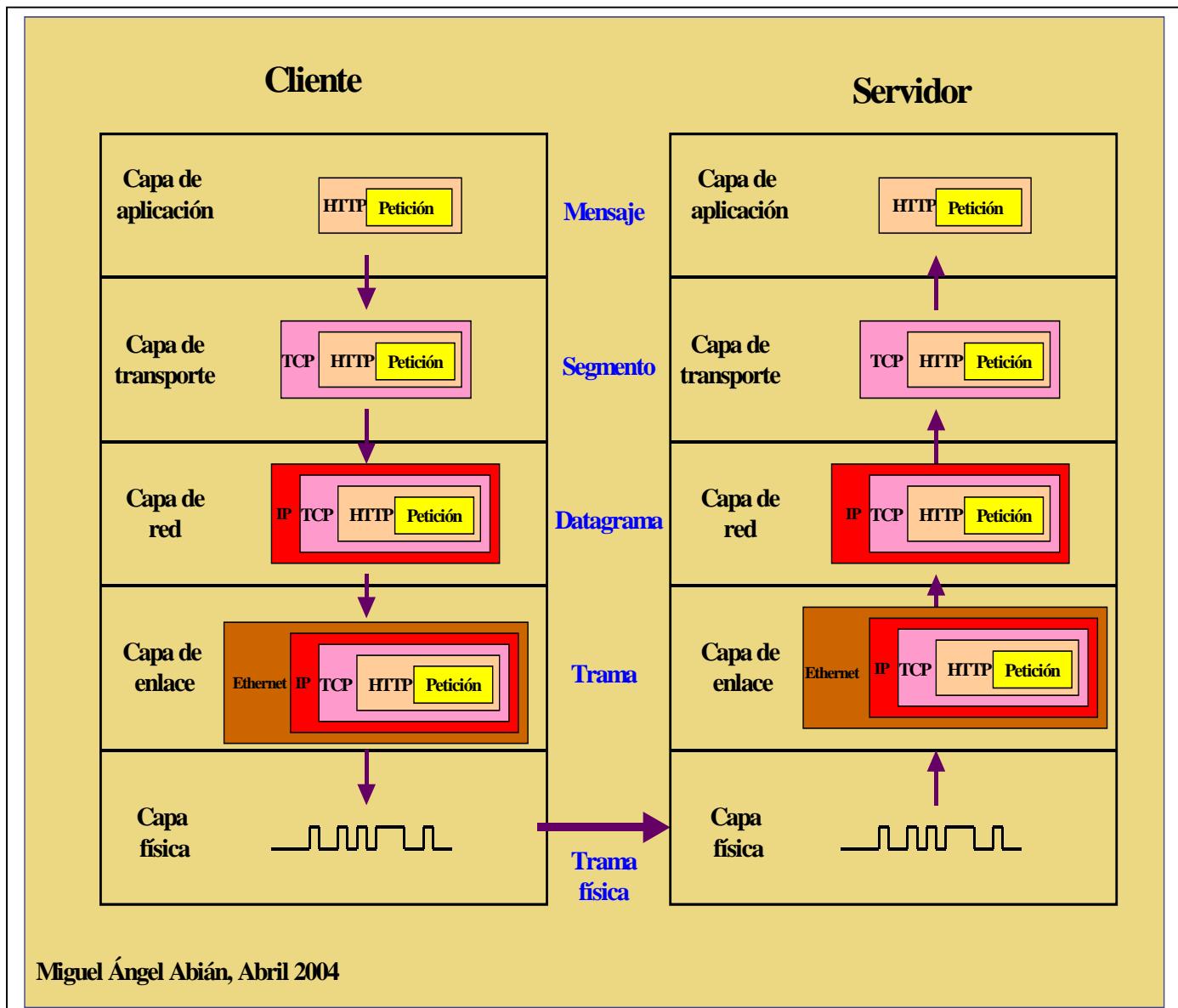


Figura 21. Representación de la solicitud de una página web a un servidor

TCP es el complemento perfecto de IP: éste envía los paquetes desde el origen hacia el destino con la máxima velocidad que puede, pero no garantiza su correcta entrega. TCP realiza las comprobaciones oportunas; en el caso de que algún paquete se haya perdido o haya llegado en mal estado, avisa al origen para que vuelva a enviarlo. Cuando hay problemas en la transmisión o recepción de un paquete, IP lo descarta y envía un mensaje genérico de error a la máquina origen. TCP interpreta el error y se encarga de que la máquina que envió el paquete con problemas lo reenvíe.

Cuando se dice que se ha instalado TCP (o TCP/IP) en un ordenador, no se quiere decir que se ha instalado sólo el protocolo TCP (o TCP e IP). En realidad, se busca significar que se ha instalado software que realiza todas las funciones del conjunto de protocolos TCP/IP (IP, en la capa de red; TCP/UDP, en la capa de transporte; HTTP, FTP, Telnet, etc., en la capa de aplicación). Conviene no olvidar que TCP/IP es un conjunto de protocolos, esto es, un conjunto de reglas, no un software específico. Muchos fabricantes implementan estos protocolos de formas muy diversas.

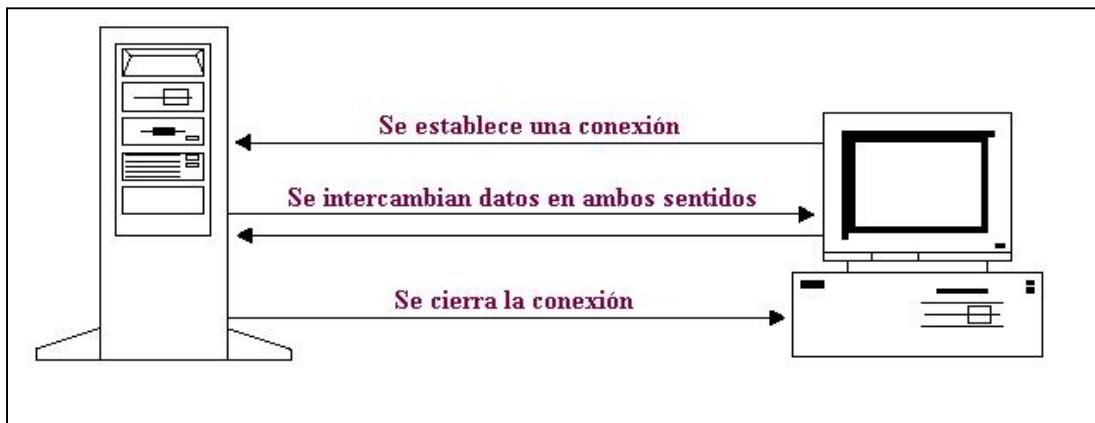


Figura 22. Ejemplo típico de una sesión TCP

El protocolo UDP (*User Datagram Protocol*), en cambio, ofrece estas otras tres características principales:

- **Carencia de tolerancia a fallos.** En este protocolo no hay mecanismos que adviertan de los errores.
- **Orientación a mensajes.** Las aplicaciones que lo usan envían mensajes unitarios (datagramas) de forma discontinua.
- **Carencia de conexión.** Un paquete UDP se puede enviar en cualquier momento a cualquier destino. Antes de transmitir datos, las aplicaciones basadas en UDP no comprueban si la aplicación destino se halla preparada para recibir los datagramas.

Advertencia: Muchas veces se usan términos como *conexiones UDP* o *sesiones UDP*. Como no creo que su uso sea correcto (salvo que se quiera expresar que las aplicaciones finales simulan conexiones o sesiones), evitaré su uso.

Desde luego, la comparación entre TCP y UDP no favorece mucho al último. La única ventaja de UDP sobre TCP es la velocidad: la ausencia de comprobaciones hace que sea mucho más rápido. ¿Velocidad o seguridad? La respuesta al dilema depende de las necesidades de cada aplicación. UDP suele usarse para transmisiones de voz y de vídeo.

UDP funciona así: cuando la capa de aplicación envía un flujo (*stream*) de bytes que corresponden a una petición, el flujo se convierte en una sucesión de paquetes o segmentos que no sobrepasan 64 K de tamaño. A cada paquete se le añade una cabecera, en la cual es optativo el *checksum*. Luego, cada uno se envía a la capa de red, junto con la dirección IP de destino y el número de puerto, donde se transforma en un datagrama.

Tanto el protocolo TCP como el UDP utilizan puertos, aunque son distintos para cada uno; es decir, una aplicación puede usar simultáneamente el puerto número 1025 con TCP y otro puerto número 1025 con UDP. Una máquina tiene 65.536 puertos UDP y 65.536 puertos TCP. Los puertos con números inferiores a 1024 suelen reservarse para procesos del sistema, y no es conveniente usarlos.

En la capa de aplicación pueden usarse muchos protocolos distintos. Algunos forman parte de TCP/IP, pues se han incluido en él desde el comienzo (por ejemplo, Telnet o FTP). Otros son más recientes, y no se incluyen todavía en TCP/IP (así, HTTP o HTTPS). El protocolo HTTP (*HyperText Transfer Protocol*: protocolo de transferencia de hipertexto) proporciona un excelente ejemplo de protocolo popular que no forma parte de TCP/IP. Para poder usarlo no basta con tener un ordenador donde se haya configurado y parametrizado TCP/IP: también se necesita un navegador (la aplicación cliente). Lo cual implica instalar, como mínimo, los dos componentes que éste lleva:

- El que implementa el protocolo HTTP.
- El que se encarga de la presentación de las páginas web y de atender los eventos que genera el usuario (pulsaciones de ratón, pulsación de teclas, etc.).

Para cerrar este subapartado, se va a detallar con un ejemplo lo que ocurre, **ciñéndose sólo a los protocolos**, cuando se accede a una página web (el protocolo HTTP se verá en 2.8.5):

- 1) En un navegador se escribe esta URL (y se pulsa la tecla correspondiente para enviar la petición):
<http://www.aidima.es/aidima/index.htm>
- 2) El protocolo DNS convierte www.aidima.es en la dirección IP 192.168.1.20.
- 3) El protocolo HTTP (recordemos el `http` inicial), construye un mensaje GET /aidima/index.htm, que se envía al anfitrión con IP 192.168.1.20.
- 4) El protocolo TCP (el que usa HTTP) establece una conexión con el anfitrión con IP 192.168.1.20, por el puerto 80 (el estándar de HTTP), y envía el mensaje GET /aidima/index.htm.
- 5) El protocolo IP envía los paquetes TCP (en forma de datagramas) al anfitrión con IP 192.168.1.20, el cual ha enviado antes a la máquina del cliente un mensaje de que está disponible para recibir peticiones.

- 6) Un protocolo dependiente del medio físico usado en la red (fibra óptica, cable coaxial, etc.) codifica los paquetes IP/TCP/HTTP y los envía a la red en forma de tramas físicas. Para ello, hay un proceso de substitución de la dirección IP 192.168.1.20 por la correspondiente dirección física de red.

Los retrasos en atender las peticiones de páginas web pueden deberse a muchas causas: saturación de las redes, excesivo número de peticiones al servidor, ancho de banda insuficiente, etc. Ahora bien, si uno quiere creer en explicaciones más esotéricas –y, para qué negarlo, más divertidas– le recomiendo la siguiente (extraída de *Dave Barry In Cyberspace*), fácilmente extrapolable a muchos países: "Una página web puede tardar un poco en aparecer en su pantalla. La razón para el retraso es que, cuando escribe una dirección Web, su computadora la pasa a otra computadora, que a su vez la pasa a otra computadora, y así sucesivamente a través de cinco computadores, hasta que finalmente alcanza la estación de trabajo de un descontento empleado del Servicio Postal de los Estados Unidos, que la arroja a la basura".

2.5. El modelo de referencia OSI estaba vestido para el éxito, pero el éxito no le llegó

El modelo de capas TCP/IP no es el único existente. La ISO (*International Standard Organization*: organización internacional de normas), encargada de la elaboración de normas internacionales, desarrolló un modelo para las conexiones entre sistemas que estén abiertos a comunicaciones con otros sistemas, conocido como **modelo de referencia OSI** (*Open System Interconnection*: interconexión de sistemas abiertas).

Este modelo tiene siete capas: capa física, capa de enlace de datos, capa de red, capa de transporte, capa de sesión, capa de presentación y capa de aplicación. Las tres últimas vendrían a equivaler a la capa de aplicación del modelo TCP/IP. Parece que el número de capas no obedece a ninguna lógica conceptual: sorprende que la especificación del modelo OSI dedique tanto espacio a la capa física y a la de enlace de datos, y tan poco a las de presentación y aplicación. Es más: en casi todas las implementaciones del modelo y de los protocolos asociados, la capa de presentación no existe y la presencia de la de aplicación es testimonial. La única explicación que he encontrado a este hecho se encuentra en *Computer Networks 3rd Edition* [Andrew S. Tanenbaum, 1998] y la traduzco a continuación:

A pesar de que casi nadie lo admite públicamente, el verdadero motivo por el que el modelo OSI tiene siete capas es que en el momento en que se diseñó, IBM tenía un protocolo patentado de siete capas llamado SNA (*Systems Network Architecture*: arquitectura de redes de sistemas). En esa época, IBM dominaba la industria de la computación hasta tal punto que todo el mundo, incluidas las empresas telefónicas, las de ordenadores de la competencia e incluso los principales gobiernos tenían miedo de que IBM usase su fuerza en el mercado para obligar a todos, prácticamente, a emplear SNA, que podría ser modificado en el momento que se quisiese. Con OSI se pretendía crear un modelo de referencia y una pila de protocolos semejante al de IBM que pudieran convertirse en el estándar mundial y que estuvieran controlados no por una empresa, sino por una organización neutral: la ISO.

Como ya se dijo, TCP/IP se usa tanto para designar a un modelo de capas como a una familia de protocolos. En consecuencia, es legítimo y exacto hablar de la arquitectura TCP/IP, pues una arquitectura de comunicaciones es un modelo de capas y un conjunto de protocolos, cada uno asociado a una capa. El modelo OSI, en cambio, es sólo un modelo, no una arquitectura de comunicaciones, puesto que no define los protocolos que puede usar cada capa. Existen normas ISO que sí especifican los protocolos que cada capa debe tener, pero no forman parte del modelo OSI. Por lo que he podido ver en el Perinorm (un programa de búsqueda de normas de todos los países del mundo), hace mucho tiempo que no se han revisado.

Si bien he creído interesante mencionar el modelo OSI, no voy a extenderme mucho en sus detalles, pues hace ya tiempo que las redes eligieron caballo ganador: TCP/IP. El modelo OSI sólo se usa ya en los libros de texto (especialmente en los europeos).

Las principales diferencias internas entre ambos modelos son éstas:

- Dentro del modelo OSI, se trabaja con conexiones; por tanto, cada aplicación que desea comunicarse con otra debe establecer primero un camino hasta el extremo de destino. Como hemos visto, TCP/IP trabaja con paquetes.

- El modelo OSI coloca ciertas funciones en la red, de manera que las aplicaciones no tienen que gestionar ciertos aspectos. En TCP/IP, se sigue el principio de “extremo a extremo”: las redes deben encargarse del menor número posible de funciones; deben limitarse a mover paquetes de un punto a otro, sin analizar su contenido ni diferenciar entre unos paquetes y otros. En este último modelo, son las aplicaciones y los ordenadores de los extremos los que deben asumir la mayor parte de las funciones exigibles (reordenación de paquetes, confirmación de llegada correcta de los paquetes, reenvío de los paquetes defectuosos, etc.)
- El modelo OSI se basa en un concepto un tanto idealista de las redes, que no corresponde a las redes reales. TCP/IP es totalmente empírico: surgió de la experiencia sobre redes que ya existían.

El principio de “extremo a extremo”, que se verá con más detalle en el siguiente subapartado, fue una guía para los diseñadores de las redes que acabaron dando lugar a la Internet que hoy conocemos. El acierto de esta decisión de diseño queda fuera de cualquier duda: la suma de miles de redes individuales, absolutamente heterogéneas y dispares, jamás hubiera sido posible si las redes hubiesen tenido que encargarse de casi todas las funciones necesarias para conseguir comunicaciones fiables. Si Internet es tan flexible, se debe a este principio de diseño de las redes: los paquetes de datos pueden enviarse para un conjunto de aplicaciones muy variadas (navegación por páginas web, transmisión de audio, de vídeo, realidad virtual, etc.).

Varias han sido la causa del fracaso del modelo OSI frente al TCP/IP. Incluyo aquí algunas de ellas para que el lector perciba las dificultades inherentes a cualquier normalización:

Complejidad. El modelo OSI es demasiado general. Su especificación es larga y densa (para mejorar la legibilidad de la documentación, la parte que describe la capa de enlace de datos y la de red tuvo que recurrir a dividirlas en muchas subcapas). Cuando se añaden al modelo las normas ISO que especifican los protocolos de cada capa, las cosas empeoran y la documentación se convierte en algo que nadie querría cargar sobre su espalda. Cuando leí las especificaciones del modelo, hace seis años, me llamó la atención la existencia de subcapas “de pega”, esto es, subcapas que han sido incluidas en el modelo para salvar las diferencias entre las características de cualquier tipo de red que pueda existir y las que el modelo puede describir. Me recuerda a las teorías de supercuerdas, cuyo lema parece ser éste: si una teoría física no explica todos los fenómenos existentes y concebibles, aumente el número de dimensiones (diez, luego trece) hasta que sea tan general que cubra cualquier fenómeno que pudiera existir.

Mal momento de aparición. El modelo OSI se creó cuando los protocolos de red aún se estaban desarrollando y perfeccionando. Como no se orientó hacia ninguna familia de protocolos, se quedó en tierra de nadie. Por otro lado, cuando existieron productos comerciales basados en el modelo OSI, tuvieron que competir con familias de protocolos mucho más específicas y que ya llevaban un tiempo en el mercado.

Falta de adaptación a la tecnología de redes. El modelo OSI se centra en las comunicaciones. Su especificación dedica un espacio marginal a los problemas inherentes al software de redes o al software en general. Algunas especificaciones son perfectas para un mundo de cables, teléfonos y transistores, pero resultan sumamente difíciles de implementar en software. Frases como “Una entidad quiere responder a un suceso” o “Ha llegado la respuesta a una petición preliminar” son demasiado abstractas como para admitir una rápida implementación en software.

Abundancia de fallos iniciales. Las primeras implementaciones del modelo OSI y de los protocolos relacionados tuvieron muchos problemas: se colocaron protocolos en capas donde no correspondían, se usaron sólo protocolos con conexión (lo cual contrastaba con los protocolos que usaban casi todas las LAN), la capa de enlace sólo era aplicable exactamente a redes de tipo punto a punto, etc. Todos estos problemas no alentaron el espíritu inversor de los clientes en una época en que el mundo de las telecomunicaciones andaba muy revuelto (ya saben, “se tarda una vida en conseguir un cliente, pero sólo un minuto en perderlo”).

Razones políticas y psicológicas. La tradicional desconfianza estadounidense hacia las organizaciones extranjeras hizo que el modelo OSI fuera visto como una imposición de la Comunidad Económica Europea y, más tarde, de la administración estadounidense. Si bien es cierto que las compañías telefónicas de varios países europeos influyeron en la elaboración del modelo, también lo hicieron compañías estadounidenses. En Estados Unidos, a finales de los ochenta, el gobierno quiso imponer de forma obligatoria lo que se conoció como GOSIP (*Goverment Open Systems Interconnect Profile*: perfil de interconexiones de sistemas abiertos del gobierno), que dejó de ser obligado en 1995. Como era de esperar, la imposición por parte del gobierno estadounidense de una cierta normalización se consideró como un recorte de las libertades individuales.

Retrocedamos un poco en el tiempo y situémonos mentalmente en 1985, en la Universidad de Berkeley (California). Consideremos un investigador que trabaje con UNIX y con los protocolos TCP/IP. Resulta probable que manifieste un cierto desprecio hacia la Autoridad. Después de todo, en los años sesenta y setenta muchos alumnos y profesores tomaban drogas psicodélicas; predicaban apasionadamente la revolución – sexual y política–, antes de convertirse en ejecutivos de ventas, empresarios o profesores; y desconfiaban de un gobierno cuyo presidente tenía serios problemas con el alcohol, con los barbitúricos y con la verdad (sólo lo último acabó forzándole a dimitir).

¿Cómo se sentiría nuestro ficticio investigador ante la mera suposición de que una potencia extranjera o el Gobierno de los Estados Unidos está tratando de forzar a los investigadores a que adopten un modelo foráneo? ¿No pensaría que se trata de una injerencia en el poder personal e intransferible de decisión de cada individuo? ¿No estaría dispuesto a sabotear cualquier (supuesto) intento de coartar su derecho a elegir? Como puede suponerse, el modelo OSI fue un estrepitoso fracaso en las universidades norteamericanas, que marcaban la pauta en cuanto a investigación en redes, y sólo ha tenido un tibio éxito en algunas compañías telefónicas europeas y japonesas.

Si el lector consulta algún libro publicado entre 1987 y 1991, puede ser que encuentre que los autores hablan maravillas del OSI y que lo presentan como una apuesta segura. Este optimismo se reveló infundado: casi nadie lo usa fuera de las aulas. El modelo OSI presenta ciertas ventajas pedagógicas sobre el modelo TCP/IP de cuatro capas; pero las ventajas se desvanecen cuando se usa el modelo TCP/IP de cinco capas (detallado al final de 2.3).

La arquitectura TCP/IP no es la Santa Perfección Canónica; pero tiene una ventaja sobre el modelo OSI y sus protocolos asociados que se reveló crucial para que las redes TCP/IP dominen el mercado sin competidores de peso: se desarrolló para que en ella cupieran los protocolos TCP/IP. Por consiguiente, la arquitectura es como un traje a medida para estos protocolos. Debido a su carácter específico, no resulta útil para describir familias de protocolos arbitrarios; pero eso no es un problema en un mundo donde casi todas las redes usan TCP/IP o son compatibles con él.

2.6. El principio de “extremo a extremo”

El principio de “extremo a extremo” o de conectividad de extremo a extremo, mencionado en el subapartado anterior, merece una cierta atención, pues cualquier aplicación basada en la arquitectura TCP/IP lo tiene en cuenta, ya sea de forma directa o indirecta.

Este principio fue enunciado como tal en el artículo *End-to-end arguments in system design* (<http://web.mit.edu/Saltzer/www/publications/endtoend/endtoend.pdf>), obra de Jerome H. Saltzer, David P. Reed y David D. Clark y publicado en noviembre de 1984. Este principio establece que

La función en cuestión [de una aplicación en red] puede implementarse completa y correctamente solamente con el conocimiento y ayuda de las aplicaciones que permanecen en los extremos del sistema de comunicación. Por lo tanto, proporcionar la función considerada como una característica del propio sistema de comunicación no es posible.

Y también que

[...] las funciones colocadas en los niveles bajos del sistema deben ser redundantes o de poco valor cuando se comparan con el coste de proporcionarlas en ese bajo nivel. Los ejemplos comentados en este artículo incluyen la supresión de los mensajes duplicados y la confirmación de la entrega.

Este principio lleva a modelos de redes “tontas” y terminales “listos”, modelos contrarios a los que predominaban antes de Internet. Seguirlo lleva aparejadas muchas ventajas:

- Los fallos en las redes intermedias no pueden destruir de forma irrevocable las comunicaciones de extremo a extremo (sólo los fallos en los extremos pueden). En un modelo en que la red se encargue de muchas funciones (como el reenvío de los paquetes defectuosos, por ejemplo), un fallo en ella acabará con la comunicación, pues se perderán datos y no habrá manera de recuperarlos.
- Las infraestructuras de las redes son simples: deben ocuparse sólo de entregar los paquetes de la manera más eficaz posible.
- Los paquetes se transportan sin modificación desde el origen hasta el destino.
- No hay terminales privilegiados: todos son nodos de una red. Tampoco hay paquetes con privilegios; todos se tratan de la misma forma. El principio de “extremo a extremo” implica un principio de “no discriminación” para los nodos y paquetes.

- La escritura de aplicaciones se torna homogénea: el programador no cambia su código cada vez que se cambia de red.
- Permite definir arquitecturas independientes de las redes usadas y que admiten con facilidad nuevos servicios y protocolos.

El principio de “extremo a extremo” está siendo puesto a prueba durante los últimos años. En la Internet actual hay subredes que lo vulneran. Un ejemplo interesante de incumplimiento del principio nos lo da NAT.

En el RFC 3022 se describe la técnica NAT (*Network Address Translation*: traducción de direcciones de red). NAT se propuso para paliar el progresivo agotamiento de las direcciones IP; es un mecanismo que permite tener en las subredes direcciones IP repetidas, esto es, direcciones que ya corresponden (o pueden corresponder) a algún anfitrión de Internet. Mediante esta técnica se pueden reescribir las direcciones IP de los paquetes que pasan a través de un encaminador o un cortafuegos.

Una red con NAT se conecta mediante uno o más “traductores NAT de direcciones”; en ella, las direcciones IP de los nodos de la red son privadas y no resultan accesibles directamente desde fuera de la red, por lo que pueden coincidir con las de otros anfitriones en otras redes de Internet. Cuando un anfitrión de la red que usa NAT desea acceder a un anfitrión de alguna otra red de Internet, el traductor NAT manipula los paquetes salientes y convierte la dirección IP del anfitrión emisor (privada y quizás duplicada) en una dirección IP válida (única en Internet). Cuando el anfitrión de destino conteste la petición, los paquetes llevarán como dirección de destino la dirección IP válida asignada por el traductor NAT. Éste se encargará de convertir esa dirección a la dirección IP privada del anfitrión que hizo la petición.

Como vemos, NAT modifica el contenido de los paquetes entre el nodo emisor y el destinatario. En consecuencia, cualquier protocolo TCP/IP que confíe en el principio de “extremo a extremo” no funcionará correctamente o hará que NAT sea inservible. Por ejemplo, el protocolo IPSEC (*IP Security*: seguridad IP) se basa en cifrar los bits de los paquetes. Debido al cifrado, un traductor NAT no puede modificar correctamente los bits donde se almacena la dirección IP, con lo cual se vuelve inútil.

Muchos cortafuegos también chocan de frente con el principio de “extremo a extremo”. El cifrado de los paquetes basándose en este principio entra en conflicto con la necesidad que tienen muchos cortafuegos de inspeccionar las direcciones IP de los paquetes entrantes, con el fin de impedir el paso de paquetes procedentes de direcciones “peligrosas” o “sospechosas” (en realidad, son las personas las peligrosas o las sospechosas, no las direcciones).

Las amenazas para el principio no acaban ahí. En Internet, casi todas las empresas propietarias de redes tienen acuerdos para permitir un cierto tráfico de paquetes “forasteros”, esto es, paquetes que no han sido generados en la red que atraviesan ni tienen como destino ningún nodo de esa red. Póngase en el lugar de una empresa que tenga redes propias y que ofrezca servicios de Internet mediante suscripción. ¿Cómo podrá atraer clientes? Pues ofreciendo servicios y contenidos de calidad (grandes anchos de banda, buenas películas, servicios meteorológicos actualizados, precios baratos, etc.). Ahora bien, ¿le interesaría a esa empresa dar a los paquetes “forasteros” el mismo ancho de banda o la misma calidad de servicio que proporciona a sus suscriptores? No (salvo que cobre por ello). Es pura lógica comercial: ¿en qué se diferenciaría esta empresa de sus competidores si diera a los clientes de éstos la misma calidad que a los suyos? Un ancho de banda grande es una ventaja comercial sólo cuando pocas empresas no lo ofrecen. Sucede como con el

dinero: ¿qué privilegios o ventajas daría el dinero si todos lo tuvieran en abundancia? Como es obvio, una empresa como la descrita carecerá de interés en preservar el principio de “no discriminación”, pues tiene un acicate económico para alterar el funcionamiento transparente e igualitario de Internet.

Otra amenaza para el sufrido principio de “extremo a extremo” viene del ofrecimiento de garantías en la calidad de los servicios de Internet. Garantizar un cierto ancho de banda en Internet, por ejemplo, no resulta fácil, pues los caminos que siguen los paquetes varían continuamente. En las aplicaciones críticas (bancarias, médicas, etc.) suele ser imprescindible un nivel garantizado de servicio y eficacia. Una manera de conseguirlo consiste en crear subredes dentro de Internet donde pueda controlarse y medirse la calidad del servicio, sujetas a la supervisión de terceros. Esta solución, pese a ser imprescindible a veces, es contraria al principio de “extremo a extremo”, pues implica un “trato preferente” a ciertos paquetes y un control de los paquetes por parte de las redes.

Las situaciones expuestas causan que, dentro de Internet, haya “islas” con problemas para interoperar con el resto de la red de redes, que sigue el principio de “extremo a extremo”. Dos son las principales soluciones para conciliar el devaluado principio y las necesidades actuales de Internet:

- Adoptar la versión número seis del protocolo IP (IPv6), la cual permitirá usar un número de direcciones IP (2^{128}) mucho mayor que el actual (2^{32}).
- Establecer protocolos que permitan asignar distintos niveles de prioridad a distintos flujos de datos. El mayor obstáculo a esta solución reside en los proveedores de servicios de Internet, poco interesados en asignar de forma prioritaria recursos a flujos de datos que no proceden de sus suscriptores.

2.7. Sockets

2.7.1. Introducción. Todo empezó en UNIX

Las abstracciones vistas en los subapartados anteriores (incluidos los puertos) no bastan para permitir las comunicaciones en red. Mientras se ejecutan, las aplicaciones o procesos de usuario permanecen dentro del espacio de memoria reservado para el usuario. El software que implementa a TCP/IP, en cambio, forma parte del sistema operativo. La situación se torna más complicada cuando consideramos aplicaciones que se ejecutan en distintas plataformas, pues deben comunicarse procesos que se ejecutan sobre arquitecturas de hardware y sistemas operativos diferentes.

Para que haya comunicación se necesita una API común (*Application Programming Interface*: interfaz de programación de aplicaciones) entre las aplicaciones de usuario y los protocolos de transporte. Una API define cómo los programadores deben usar una cierta característica o función de un sistema, sea de software o hardware. La API más utilizada para la comunicación en redes es la **API Socket**, que fue diseñada inicialmente para la versión 4.1 de UNIX BSD (*Berkeley Software Distribution*), desarrollado en la Universidad de California (Berkeley). La versión 4.2 de UNIX BSD (1983) fue la primera en introducir la familia de protocolos TCP/IP dentro del sistema operativo; TCP/IP había sido introducido en UNIX BSD ya en 1981, pero no como parte del SO.

Las malas lenguas dicen que no es casualidad que el LSD y el UNIX BSD surgieran en la Universidad de Berkeley. Tras indagar en las reacciones que provocó en la comunidad UNIX la aparición de la API Socket, puedo asegurar que la reacción que tuvieron muchos programadores se ve magníficamente representada por el rostro alucinado del soldado de la película *Apocalypse Now* que, al poco de tomar ácido, contemplaba extasiado y ensimismado las explosiones aéreas y el fuego de artillería, absorto en la catarata de sonidos e imágenes en que se transmutaba, por obra y gracia de la farmacología moderna, la desoladora realidad bélica. Quizás nadie en *Apocalypse Now* supiera quién era el oficial al mando, ni siquiera Francis Ford Coppola; pero los usuarios de UNIX reconocieron en la API Socket al oficial al mando.

Como Java usa el mismo modelo de **sockets** que UNIX BSD (hoy día, la API Socket es un estándar de facto), vale la pena recordar su origen en UNIX. En este sistema operativo, antes de que un proceso de usuario realice cualquier operación de E/S, se llama a un método *open()* para especificar qué archivo o dispositivo (en UNIX, todo es un archivo) va a usarse y obtener el permiso necesario. La llamada al método devuelve un entero (el **descriptor de archivo**); cuando un proceso ejecuta operaciones de E/S –como *read()* o *write()*–, el descriptor del archivo se usa como uno de los argumentos de la operación. La secuencia de pasos que se realiza en UNIX para trabajar con la E/S se conoce como Abrir-Leer-Escribir-Cerrar (*Open-Read-Write-Close*).

Los *sockets* (“enchufes”) no son más que una generalización del mecanismo que usa UNIX para manipular archivos, adecuada para manejar comunicaciones de red: si para acceder a un archivo o a un dispositivo se necesita un descriptor de archivo, para permitir comunicaciones entre procesos (estén o no en máquinas distintas) se necesitan descriptores de *sockets*. Ahora bien, un descriptor de archivo nace vinculado a un archivo o dispositivo; mientras que un descriptor *socket* puede corresponder a *sockets* no enlazados a direcciones específicas de destino (dicho de otro modo, no vinculados a pares dirección IP-número de puerto). Al igual que sucede con los archivos y dispositivos de UNIX, con los *sockets* pueden usarse métodos como *read()*, *write()* y *close()*. No hay, en definitiva, grandes diferencias entre las operaciones de transferencias de datos para *sockets* y para archivos. En un archivo, por ejemplo, *write()* transfiere datos de una aplicación o proceso de usuario al archivo: en un socket,

`write()` transfiere datos de un proceso de usuario (emisor) a otro proceso de usuario (receptor).

La API Socket, como toda API, es un conjunto de declaraciones de funciones o métodos, que admite múltiples implementaciones. En todas las distribuciones actuales de UNIX que la usan se implementa como parte del núcleo del sistema operativo. Casi todos los sistemas operativos actuales permiten usar sockets. Windows, por ejemplo, usa la API WinSock (implementada como una biblioteca), mientras que UNIX System V usa la API TLI (*Transport Layer Interface*: interfaz de la capa de transporte). Curiosamente, este último (que en sus primeras implementaciones no admitía sockets) ganó la batalla comercial contra UNIX BSD, pero tuvo que adoptar el modelo de sockets de UNIX BSD y, por ende, su forma de trabajar en red.

2.7.2. Sockets. Tipos de sockets

Los sockets definidos por la API Socket son abstracciones del sistema operativo que existen mientras se ejecutan aplicaciones de red; pueden usarse para acceder a distintos protocolos de transporte, no sólo a TCP/IP. Por medio de los sockets, los procesos envían o reciben mensajes. Un socket viene a ser una especie de pasadizo, residente en el sistema que lo crea, que permite que los procesos de la capa de aplicación puedan enviar y recibir mensajes de otros procesos de aplicación, se ejecuten o no en la misma máquina. En la figura 23 se muestra dónde se ubicarían conceptualmente los sockets.

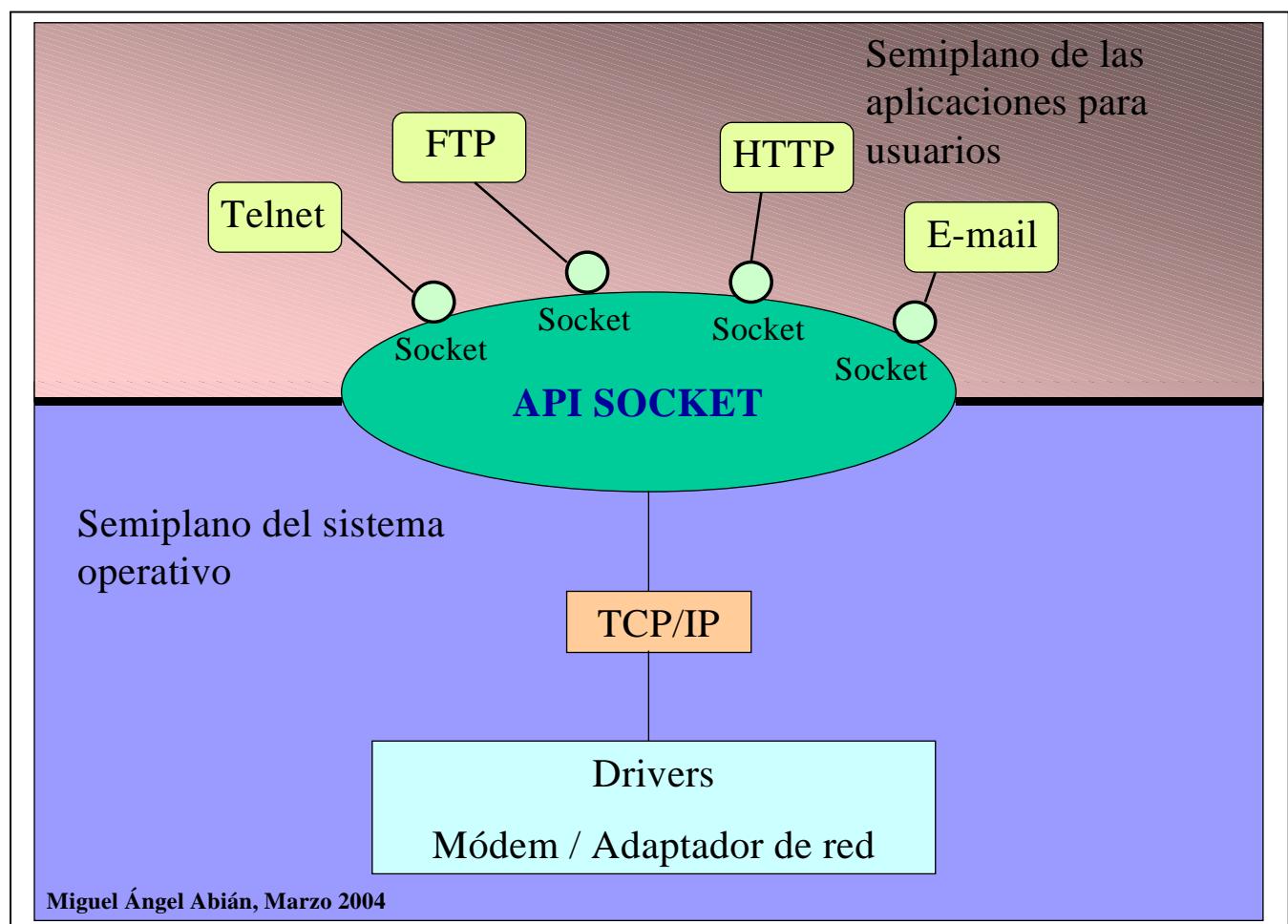


Figura 23. Una abstracción sobre abstracciones

En *Thinking in Java 3rd Edition*, Bruce Eckel describe así los *sockets*:

El *socket* es la abstracción de software usada para representar los *terminales* de una conexión entre dos máquinas. Para una conexión dada, hay un *socket* en cada máquina, y puedes imaginar un *cable* hipotético corriendo entre las dos máquinas con cada extremo del *cable* enchufado a un *socket*. Desde luego, el hardware físico y el cableado entre máquinas es completamente desconocido. El punto fundamental de la abstracción es que no necesitamos conocer más de lo necesario.

Un *socket* es al sistema de comunicaciones entre procesos lo que el buzón de la figura 8 al sistema de comunicación por correo: un punto de comunicación entre dos procesos que permiten intercambiar información (el envío y la recogida de las cartas, en el caso del correo; el envío y la recepción de datos binarios, en el caso de los *sockets*).

Desde un punto de vista pragmático, los *sockets* son herramientas que permiten comunicar procesos a través de Internet, de extranets, de intranets e incluso en ordenadores aislados.

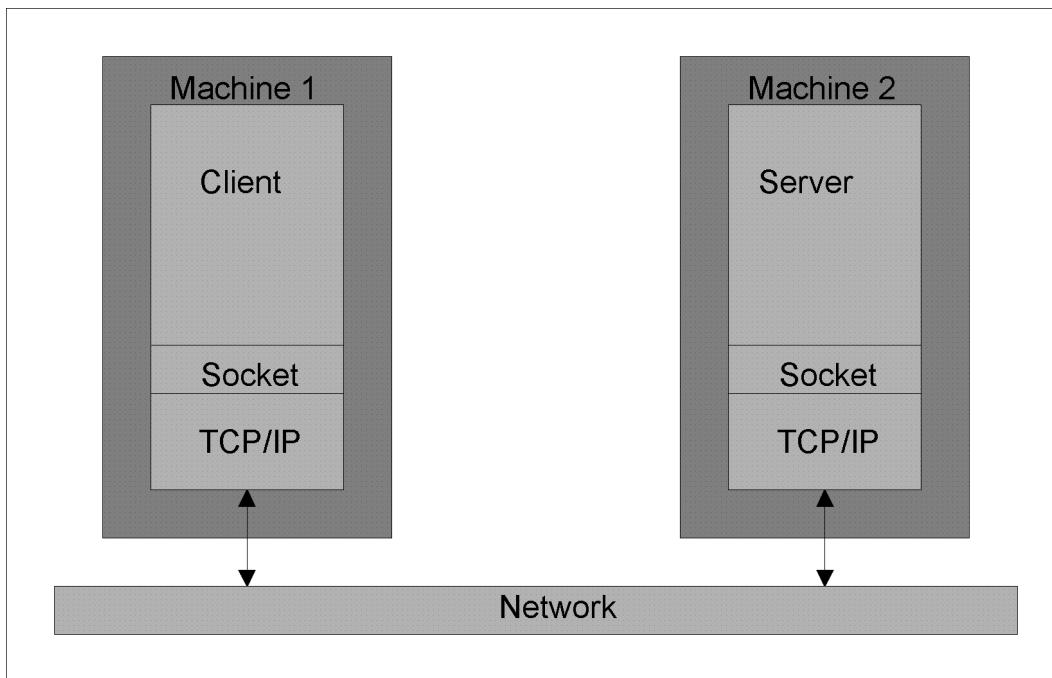


Figura 24. Ejemplo de conexión cliente-servidor

Hay dos tipos de *sockets*:

- **Activos**, los cuales están conectados mediante una conexión abierta, que puede permitir la transmisión de datos.
- **Pasivos**, los cuales no están conectados. Por tanto, no pueden usarse para transmitir datos. Se usan para permanecer a la espera de peticiones de conexión; cuando reciben una, generan *sockets* activos.

Los protocolos TCP y UDP utilizan *sockets* para comunicar programas entre sí en una arquitectura cliente-servidor. Todo *socket* tiene asociado la dirección IP del anfitrión donde se ejecuta el programa servidor o cliente y la dirección del puerto utilizado por el programa cliente o servidor.

UDP y TCP usan *sockets*, pero los *sockets* UDP y TCP son muy distintos. Un *socket* TCP está ligado a una sola máquina y permite la transmisión bidireccional de datos mediante flujos de datos (después, la capa de red los convierte en datagramas). En cambio, un *socket* UDP puede estar ligado a muchas máquinas y permite el envío o recepción de datos –siempre unidireccional– en forma de paquetes discretos que a veces también se llaman como los paquetes definidos por la capa de red (datagramas).

Los **sockets TCP** (también conocidos como *sockets* de flujo) ofrecen, entre otras, las funciones que aparecen en la siguiente tabla:

Acción	Nombre en la API Socket	Descripción
SOCKET	socket()	Creación de un socket
BIND	bind()	Asignación de una dirección IP a un socket
CLOSE	close()	Cierre de la conexión
LISTEN	listen()	Declaración de aceptación de conexiones entrantes
ACCEPT	accept()	Espera hasta que llegue alguna conexión
CONNECT	connect()	Intento de establecer conexión
SEND	send()	Envío de datos por medio de la conexión
RECEIVE	recv()	Recepción de datos por medio de la conexión

Figura 25. Algunas funciones de los sockets TCP

En el modelo cliente-servidor, al cliente le corresponden las funciones *socket()*, *connect()*, *send()/recv()* y *close()*; al servidor, *socket()*, *bind()*, *listen()*, *accept()* y *close()*. La función *accept()* permite usar las funciones *recv()/send()* y *close()* sobre los *sockets* que crea. Aunque no incluye las funciones *write()* y *read()*, también se pueden usar; pero ofrecen menos control sobre la transmisión de datos que *send()* y *recv()*, más especializadas en las comunicaciones entre procesos.

En una comunicación TCP, los clientes crean *sockets* activos (por brevedad, omito TCP) y los conectan a *sockets* activos del servidor generados a partir de *sockets* pasivos del servidor. A su vez, cada *socket* está asociado a una dupla con dos componentes: dirección IP y número de puerto. Como hay conexión, no se precisa

enviar la información (IP de la máquina y puerto al que está ligado) contenida en el *socket* del cliente ni en el *socket* del servidor cada vez que se intercambian flujos de datos. Dicho de otro modo: como un *socket* activo en el servidor está conectado a un *socket* activo del cliente mientras dura la comunicación, el primero siempre sabe adónde enviar sus respuestas. Cuando se habla de conexiones TCP que atraviesan Internet, un *socket* es globalmente único; pues viene caracterizado por cinco datos: el protocolo usado (FTP, HTTP, etc.), dos direcciones IP (la de la máquina local y la de la máquina remota) y dos puertos (uno local y otro remoto). Si la comunicación TCP se realiza dentro de una red local, el *socket* es único en esa red.

Los pasos que se siguen para establecer, mantener y cerrar una conexión TCP se muestran aquí:

- Se crean los *sockets* en el cliente y el servidor.
- El servidor establece el puerto por el que proporcionará el servicio.
- El servidor permanece a la escucha de las peticiones de los clientes.
- Un cliente conecta con el servidor.
- El servidor acepta la conexión.
- Se realiza el intercambio de datos.
- El cliente o el servidor, o ambos, cierran la conexión.

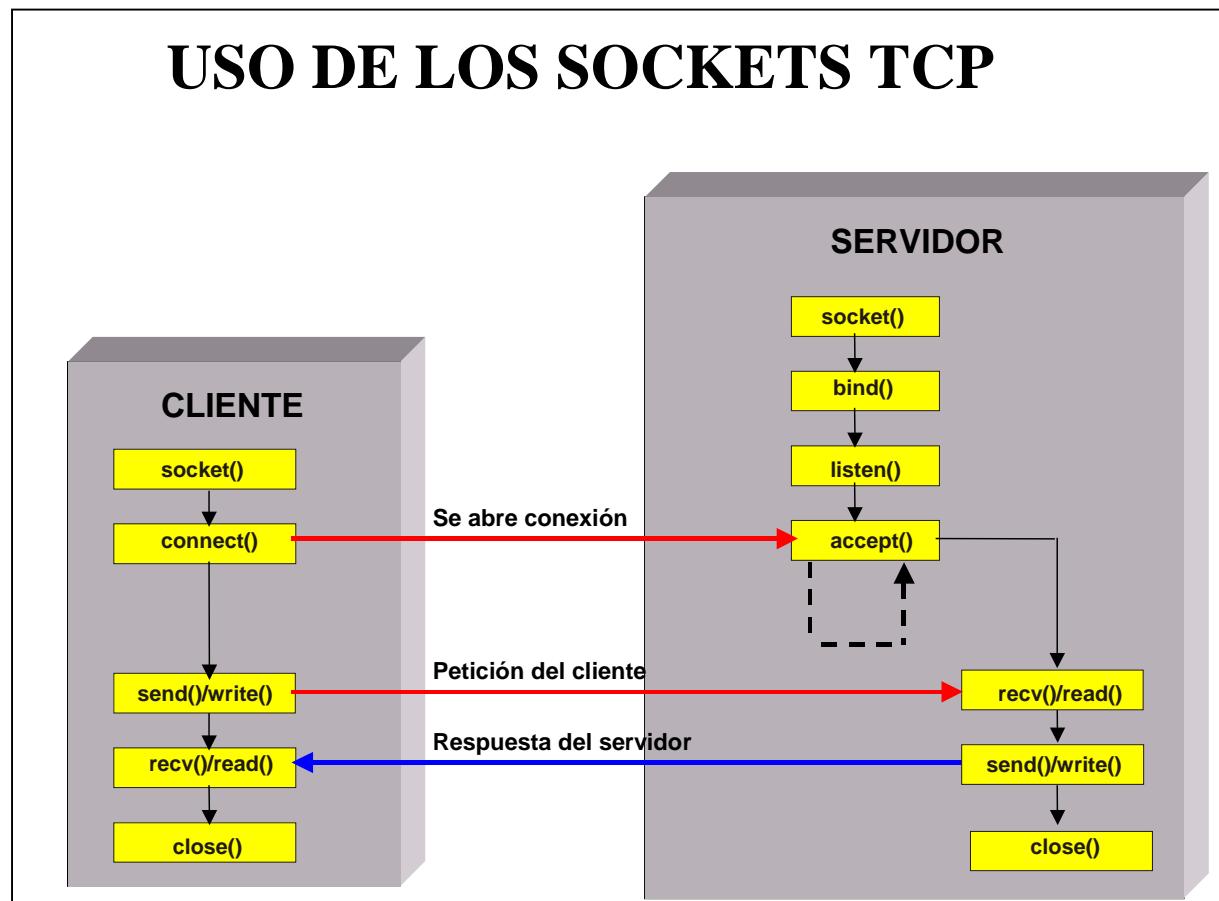


Figura 26. Esquema del uso de los sockets TCP

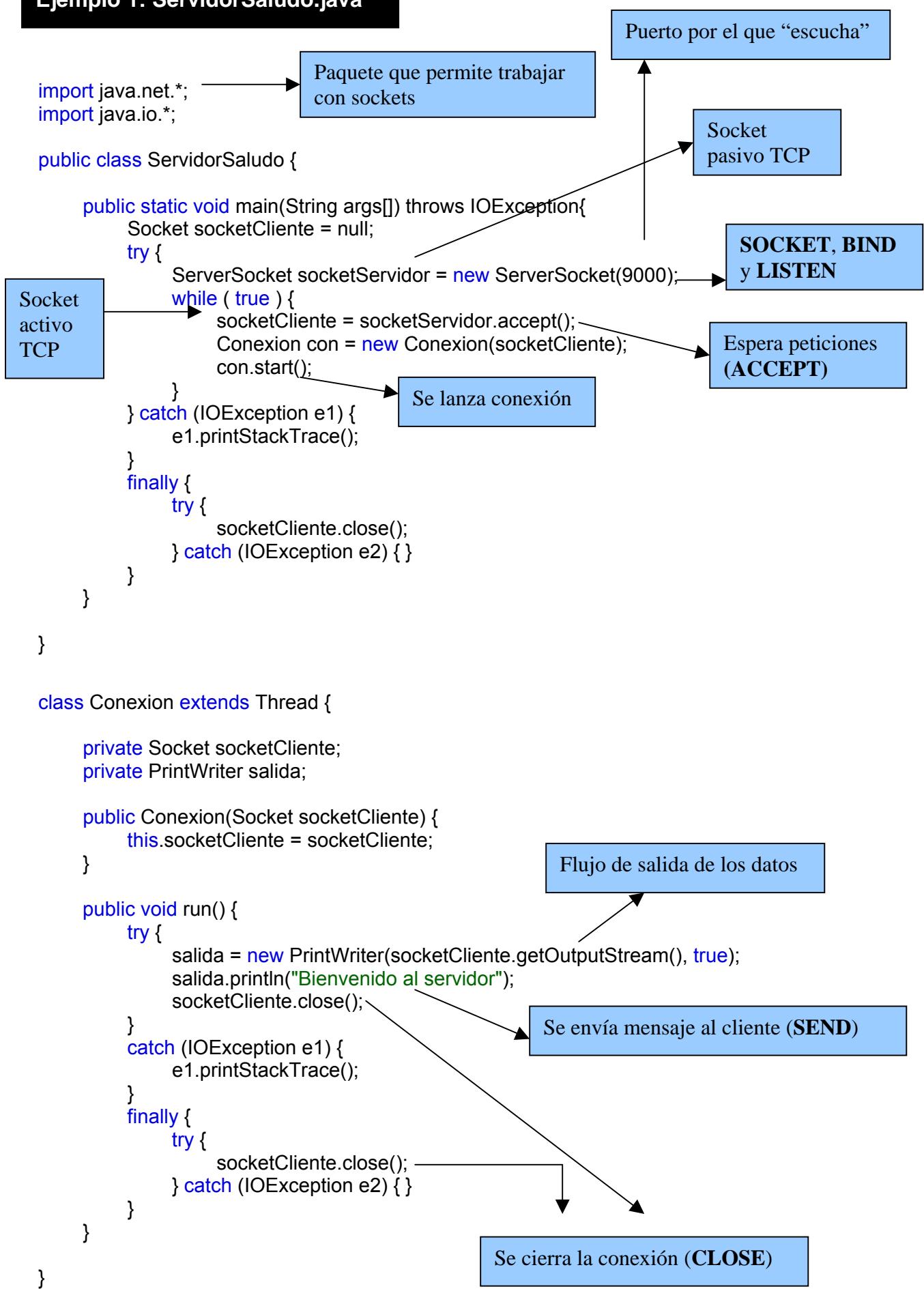
Java proporciona clases para la implementación de *sockets*, ya sean TCP o UDP. La interfaz que ofrece Java para la programación con *sockets* se basa completamente en la API Socket de UNIX BSD; pero la ha simplificado ostensiblemente, amén de darle una buena capa de orientación a objetos. Por ejemplo, todas las funciones de la figura 25 se obtienen con las clases `Socket` y `ServerSocket` de `java.net`, paquete que se verá en el apartado 8. La explicación previa de los protocolos UDP y TCP hará que este paquete sea muy sencillo de explicar.

Programar *sockets* con Java es muy sencillo, mucho más que usar C (véase el siguiente subapartado). Internamente, Java usa la JNI (*Java Native Interface*: interfaz nativa de Java) para llamar a las bibliotecas en C que implementan las funciones de los *sockets*, pero el programador no debe preocuparse de ello.

Para exemplificar el uso de *sockets* en Java, en la página siguiente se muestra el código correspondiente a un programa servidor multihilo que envía mediante TCP un saludo a cada cliente que se le conecta. Si bien los métodos de `Socket` y `ServerSocket` se explicarán detenidamente en el apartado 8, prefiero que el lector cuyas retinas estén deslumbradas o chamuscadas por la ausencia de código Java en las primeras cincuenta y nueva hojas pueda ver ya la estrecha relación entre la API Socket y algunas clases de `java.net`, así como la materialización en este lenguaje de los pasos arriba expuestos. Una vez leído el apartado 8, el lector puede volver hacia atrás y repasar este código. Como puede suponerse, la clase `ServerSocket` proporciona todo lo necesario para escribir programas servidores en Java: incorpora métodos que esperan conexiones por un determinado puerto y métodos que devuelven un objeto `Socket` cuando se recibe una conexión, así como métodos para recibir y enviar datos.

Si el programa se ejecuta en la máquina local, para verlo funcionar hay que ejecutarlo y acceder después a la dirección <http://localhost:9000> desde un navegador. Si, en cambio, se ejecuta en una máquina remota (www.uv.es, por ejemplo), se debe llamar mediante la dirección correspondiente (<http://www.uv.es:9000>, siguiendo con el ejemplo).

Ejemplo 1: ServidorSaludo.java



En este sencillo ejemplo vemos que las clases `ServerSocket` y `Socket` ofrecen al programador un `InputStream` o un `OutputStream`, mediante métodos como `getOutputStream()`. Las clases `InputStream` y `OutputStream`, así como muchas de sus subclases, se explicarán en el apartado 3.

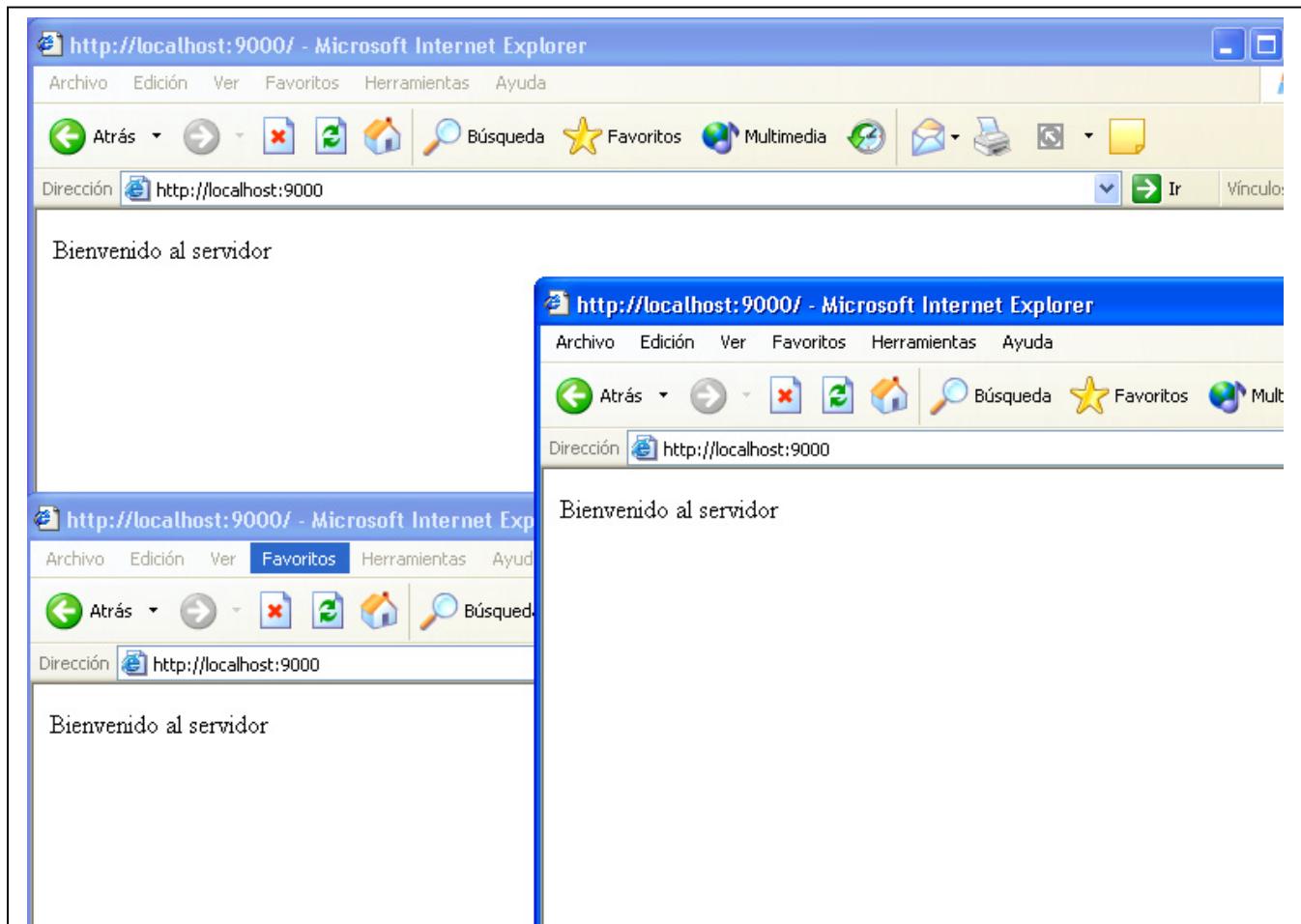


Figura 27. El ejemplo 1 en funcionamiento

Para el programador de Java, un *socket* TCP es la representación de una conexión para la transmisión de información entre dos ordenadores distintos o entre un ordenador y él mismo. Esta abstracción de alto nivel permite despreocuparse de los detalles que yacen bajo ella (correspondientes a protocolos subyacentes). En resumen, un *socket* TCP permite conectarse a un equipo a través de un puerto, enviar o recibir datos y cerrar la conexión establecida.

Los **sockets UDP** (también conocidos como *sockets de datagramas*) ofrecen, entre otras, las funciones que aparecen en la siguiente tabla:

Acción	Nombre en la API Socket	Descripción
SOCKET	socket()	Creación de un socket
BIND	bind()	Asignación de una dirección IP a un socket
CLOSE	close()	Cierre del socket
SEND TO	Sendto()	Envío de datos
RECEIVE FROM	recvfrom()	Recepción de datos

Figura 28. Algunas funciones de los sockets UDP

En el modelo cliente-servidor, al cliente le corresponden las funciones *socket()*, *sendto()*/*recvfrom()* y *close()*; al servidor, *socket()*, *bind()*, *sendto()*/*recvfrom()* y *close()*. Aunque no incluyo las funciones *write()* y *read()*, también se pueden usar; pero ofrecen menos control sobre la transmisión de datos que *send()* y *recv()*, más especializadas en las comunicaciones entre procesos.

En una comunicación UDP, los clientes y los servidores se comunican enviando paquetes de datos entre los *sockets* de cada uno, sin que exista conexión. Así pues, un *socket* del servidor necesita enviar con cada paquete la IP de la máquina cliente y el número de puerto asociado al *socket* del cliente; lo mismo sucede, *mutatis mutandis*, con los *sockets* del cliente. En caso contrario, los paquetes acabarían en algún Triángulo de las Bermudas de Internet.

Al no existir conexión, los *sockets UDP* no son globalmente únicos, a diferencia de lo que sucede con los de tipo TCP, ya que no existe conexión: siempre se refieren a la máquina local. Lógicamente, los *sockets TCP* ocupan más memoria RAM que los UDP, pues necesitan mantener más información. Además, los primeros necesitan reservar memoria para dos *buffers* (uno para recibir y otro para transmitir; un *buffer* es una memoria de almacenamiento temporal), mientras que los segundos sólo necesitan un *buffer* para recibir o transmitir.

USO DE LOS SOCKETS UDP

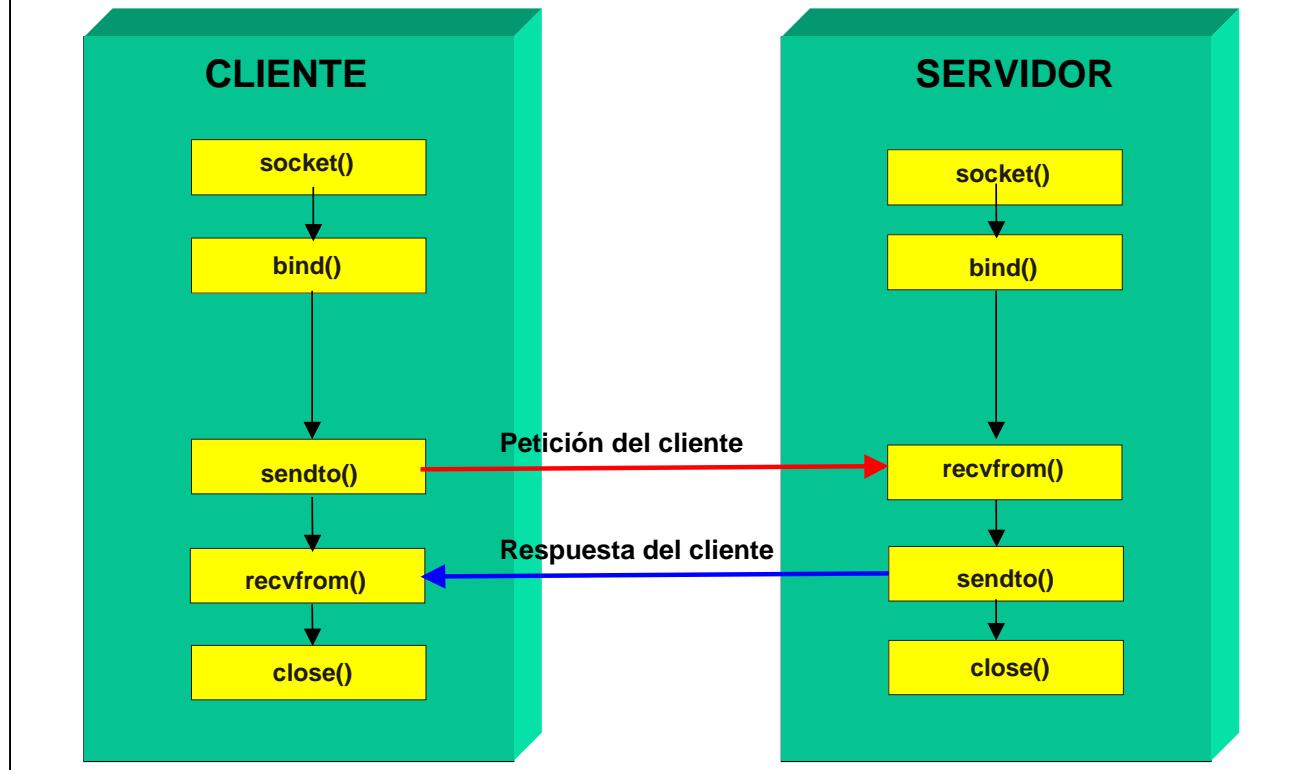


Figura 29. Esquema del uso de los sockets TCP

En la página siguiente aparece el código para una aplicación cliente-servidor que devuelve un saludo a cada cliente mediante el protocolo UDP, indicando el puerto por donde recibe la petición y la fecha de ésta. El comportamiento exacto de las clases `DatagramSocket` y `DatagramPacket` se explicará en el apartado 8, pero el lector puede usar lo aprendido sobre sockets y el esquema de la figura 29 para intuir cómo funcionan. El código del cliente se incluye porque un navegador web no valdría como cliente UDP, pues el protocolo HTTP usa TCP, no UDP. Por no alargar en exceso el código, prescindo de usar hilos. Tal y como está escrito, se supone que el servidor y el cliente se ejecutan en la máquina local. Si la aplicación de servidor residiera en una máquina remota, habría que modificar la línea

```
InetAddress destino = InetAddress.getByName( "localhost" );
```

del ejemplo 2b indicando el nombre de la máquina donde se ejecuta el servidor.

Ejemplo 2a: ServidorSaludoUDP.java

```
import java.net.*;
import java.io.*;

public class ServidorSaludoUDP {

    private static byte[] buffer;
    private static byte[] datos;

    public static void main(String args[]) {
        try {
            DatagramSocket socket = new DatagramSocket(9000);
            buffer = new byte[1024];
            while( true ) {
                DatagramPacket datorama = new DatagramPacket(buffer, buffer.length);
                socket.receive(datorama);
                InetAddress hostDestino = datorama.getAddress();
                int puertoDestino = datorama.getPort();
                datos = datorama.getData();
                String cadena = new String(datos, 0, datos.length);
                System.out.println("Bienvenido al servidor. Envío por el puerto " +
                    puertoDestino + " el mensaje: " + cadena);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

RECEIVE FROM

Puerto por el que “escucha”

Socket activoUDP

SOCKET y BIND

Ejemplo 2b: ClienteUDP.java

```
import java.net.*;
import java.io.*;
import java.util.*;

public class ClienteUDP {

    private byte[] buffer;
    private static String cadena = "Enviando datagrama en la fecha: " + new Date();

    public static void main(String args[]) {

        try {
            byte[] buffer = cadena.getBytes();
            InetAddress destino = InetAddress.getByName("localhost");
            DatagramPacket datorama = new DatagramPacket(buffer, buffer.length,
                destino, 9000);
            Socket activo
            UDP
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Socket activo UDP

DatagramSocket socket = new DatagramSocket();

```

CLOSE }          socket.send(datagrama); → SEND TO
        }          socket.close();
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}

```

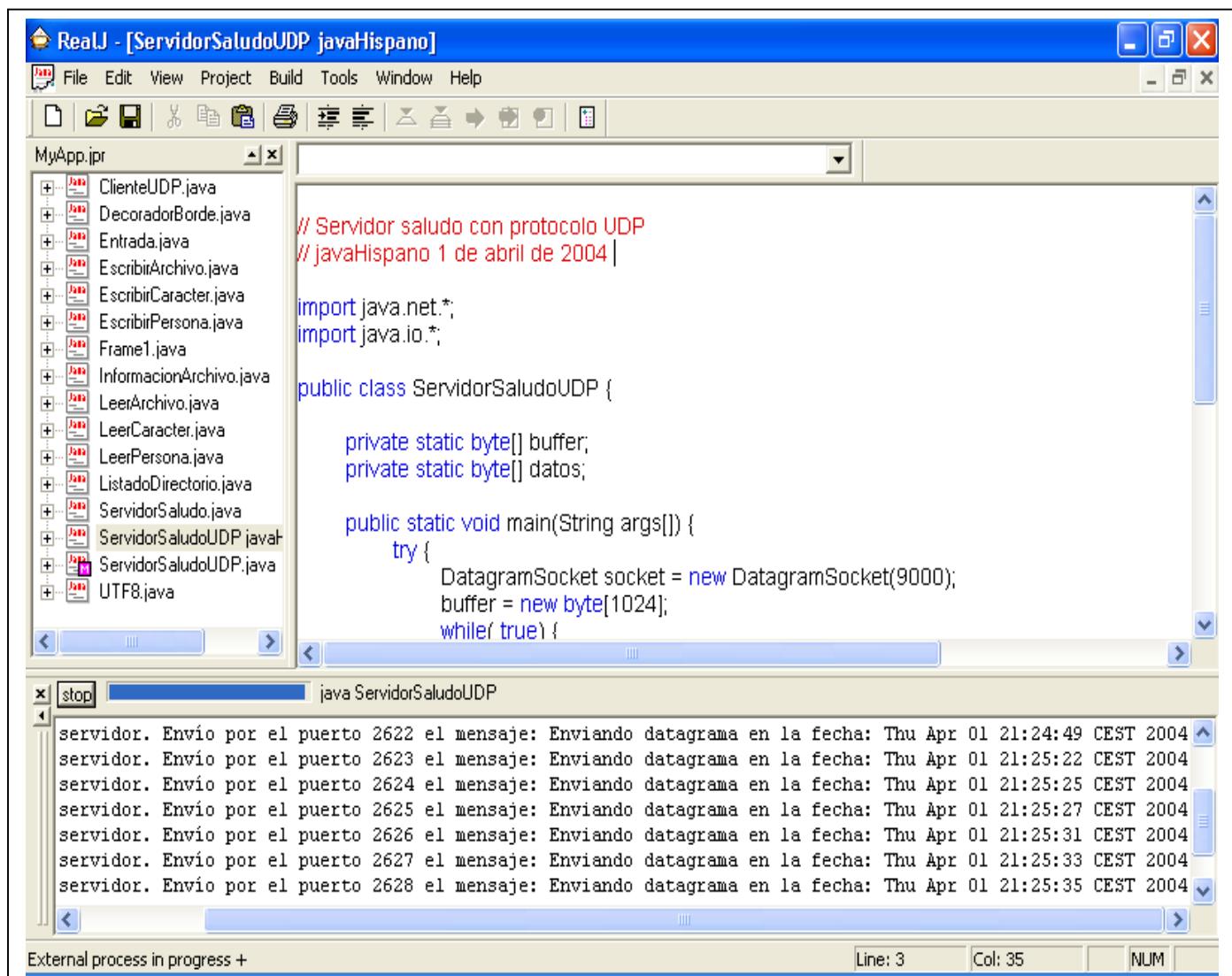


Figura 30. El código del ejemplo 2 en funcionamiento

2.7.3. Un ejemplo de sockets en C

Como curiosidad, incluyo aquí el código C correspondiente a un programa servidor que atiende a un cliente (sólo a uno) y le envía un mensaje de bienvenida y otro de despedida. Es la versión en C del ejemplo 1 (`ServidorSaludo.java`), pero sin hilos; el cliente puede ser un simple navegador. Con él, sólo pretendo que el lector intuya la complejidad de la programación de sockets en C y que pueda comparar más adelante con la sencillez y elegancia de Java para las comunicaciones en red.

Desde luego, esta sencillez y elegancia no es gratuita (lo único que da gratis el universo es hidrógeno, y algo de helio): Java sólo puede trabajar con TCP/IP, mientras que la API Socket de UNIX BSD y la API TLI pueden funcionar con casi cualquier protocolo de red. En principio, ello no constituye un gran obstáculo: los sockets de Java funcionan en Internet, en extranets e intranets.

Como mis conocimientos de las bibliotecas en C para redes están muy apolillados, no puedo garantizar que este código sea todo lo eficaz que podría ser o que aproveche las características de las bibliotecas más recientes de C. Agradeceré mucho cualquier sugerencia de los lectores al respecto.

```
// Programa servidor TCP escrito en C
// Manda un saludo a cada cliente que se le conecta

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define PUERTO 9000 // Puerto por el que se escuchan las peticiones

int main(void) {

    // dasocket y nuevodasocket: descriptores de archivo de sockets
    // longcliente: almacena el tamaño de la dirección del cliente
    int dasocket, nuevodasocket, longcliente;

    // sockeaddr_in se define en <netinet/in.h>, se usa para representar dir de Internet
    // dir_servidor y dir_cliente: direcciones de Internet del servidor y el cliente
    struct sockaddr_in dir_servidor, dir_cliente;

    // Buffer de 1 K para almacenar caracteres
    char buffermensaje[1024];

    // Almacena el estado de la conexión.
    int estado;

    // Se inicializa la estructura dir_servidor, llenando sus campos de ceros
    bzero((char *) &dir_servidor, sizeof(dir_servidor));

    // Se inicializa el buffer, llenando sus campos de ceros
    bzero(buffer, 1024);

    // Se crea un socket TCP
    dasocket = socket(AF_INET, SOCK_STREAM, 0);
    // Se comprueba que se pudo crear
    if (dasocket < 0) {
```

```
        perror("No se pudo abrir el socket.");
        exit(1);
    }
/* Se define la dirección del servidor mediante las líneas siguientes */
// Se establece el tipo de servidor
dir_servidor.sin_family = AF_INET;
// Se establece el orden de los bytes usado en la red
dir_servidor.sin_addr.s_addr = INADDR_ANY;
// Se establece la dirección IP de la máquina
dir_servidor.sin_port = htons(PUERTO);

/* Se asigna el estado del socket */
// bind liga un socket a una dirección. En este caso, la dirección del servidor.
// bind toma tres argumentos: el descriptor de archivo del socket, la dirección a la cual
// está ligada y su tamaño.
estado = bind(dasocket, (struct dirsocket *) &dir_servidor, sizeof(dir_servidor))

/* Se comprueba el estado. Si no se ha podido ligar el socket al puerto, bind retorna -1 */
if (estado<0) {
    perror("Error al intentar enlazar el socket al servidor.");
    exit(1);
}

/* El servidor queda a la espera de sockets entrantes */
// listen toma dos argumentos: el descriptor de archivo del socket y el número máximo de
// conexiones que pueden estar en espera mientras el servidor maneja una conexión
// (suele usarse 5).
listen(sockfd,5);
longcliente = sizeof(dir_cliente);
/* El servidor acepta conexiones de los clientes */
// accept paraliza el servidor hasta que llegan conexiones de los clientes. Devuelve un
// nuevo descriptor de archivo, que se usara para intercambiar información con el cliente.
nuevodashcket = accept(dasocket, (struct dirsocket *) &dir_cliente, &longcliente);

/* Se comprueba que la conexión es correcta */
if (nuevodashcket < 0) {
    perror("No pudo aceptarse la conexión del cliente.");
    exit(1);
}

/*Se envía un mensaje de bienvenida al cliente usando write */
if ( write(nuevodashcket, "Bienvenido al servidor", 22) <0 ) {
    perror("Problemas de comunicación con el cliente.");
    exit(1);
}

/* Se envía un mensaje de despedida al cliente usando send */
buffermensaje = "El servidor dice adiós\n";
send(nuevodashcket, buffermensaje, strlen(buffermensaje), 0);

/* Se cierra el socket nuevo */
close(nuevodashcket);

return 0;
}
```

2.7.4. Ventajas e inconvenientes de los sockets

Las principales ventajas que ofrecen los *sockets* se detallan aquí:

- Las aplicaciones que los usan son muy rápidas y eficaces. Como los *sockets* son entidades de bajo nivel, se comunican rápida y eficazmente con el sistema operativo e introducen poca sobrecarga en las aplicaciones.
- Son el mecanismo más difundido para las comunicaciones entre procesos. Todos los sistemas operativos y lenguajes de programación actuales les dan cabida. Escasos son los protocolos de transporte que no usan *sockets*.

La otra cara de la moneda nos la dan estos inconvenientes de los *sockets*:

- Son abstracciones de bajo nivel. Por ello, la programación intensiva con *sockets* no resulta muy cómoda para el programador medio. Cuando uno se ha acostumbrado a mensajes de error del estilo de “Conversión errónea de tipos”, “El tipo del argumento no corresponde a la definición del método” o “Número erróneo de argumentos”, no entusiasma la idea de encontrarse con errores como “Conexión rechazada”, “No se encontró el anfitrión”, “Puerto ocupado por otro proceso”, etc.
- El código escrito con *sockets* depende de la plataforma. Debido al bajo nivel de los *sockets*, están muy ligados a la plataforma donde se ejecutan.
- No permiten el envío directo de argumentos. Es el programador quien debe encargarse de abrir y cerrar los flujos de E/S y de introducir en ellos los argumentos que se quieran pasar, así como de extraer los resultados. Por ejemplo, si se necesita que el cliente envíe un argumento `int` y que el servidor devuelva un `double`, el programador debe encargarse de escribir en el cliente el código que introducirá el `int` en un flujo de salida, el código en el servidor que leerá el flujo de entrada y extraerá un `int` y el código en el cliente que leerá un `double` del flujo de entrada enviado por el servidor.
- El código con *sockets* es difícil de reutilizar. Aparte de los problemas derivados por el bajo nivel de éstos, los clientes necesitan conocer la dirección de la máquina donde se ejecuta el servidor y el número de puerto para acceder a éste. Si el servidor cambia de ubicación, hay que recomilar los clientes o, al menos, reconfigurarlos.
- No ofrecen metainformación sobre los servicios de los servidores y sus localizaciones. No hay “mapas” que digan “El servidor A está en B y ofrece estos servicios”.

2.8. Algunos servicios de la capa de aplicación en la arquitectura TCP/IP

2.8.1. Introducción

Los protocolos de la capa de aplicación especifican

- 1) los tipos de mensaje intercambiados (peticiones y respuestas, por ejemplo);
- 2) la sintaxis de los posibles mensajes: los campos de éstos y sus formatos;
- 3) el significado de los campos;
- 4) las reglas para responder a las peticiones o para enviarlas.

Ninguno de estos protocolos especifica o detalla ninguna implementación, que queda libre para los programadores.

En los siguientes subapartados veremos algunos servicios de la capa de aplicación, basados en protocolos que nos resultarán útiles cuando estudiemos el paquete `java.net`.

2.8.2. El servicio de administración de redes

En cualquier red, por sencilla que sea, el proceso de administración resulta imprescindible para controlar los recursos y compartirlos adecuadamente. El **protocolo SNMP** (*Simple Network Management Protocol*: protocolo de administración de redes simples) se basa en UDP. Mediante SNMP, las aplicaciones pueden recopilar información sobre el funcionamiento de los nodos de una red TCP/IP (eficacia, existencia de problemas, desconexiones, etc.). SNMP simplifica la administración de redes mediante el envío de órdenes a través de las redes, en lugar de mediante cambios en las configuraciones físicas de las máquinas.

Este protocolo se estructura en dos partes: el administrador SNMP y los agentes SNMP. El administrador SNMP es una aplicación que se ejecuta en la máquina encargada de administrar la red y que se comunica con los agentes mediante la red. Los agentes SNMP se ejecutan en los nodos de la red (máquinas, dispositivos periféricos: impresoras, etc.), en pasarelas, en encaminadores, etc., y mantienen un registro de su funcionamiento y de su estado actual. Al conjunto de todos los registros se le llama MIB (*Management Information Base*: base de información de la administración). La MIB se divide en grupos que contienen información acerca de distintos aspectos de la red: nombre del dispositivo, nombre de la red, interfaz de la red, hardware y software de cada nodo, velocidad de trasmisión de los paquetes, estadísticas de los paquetes enviados y no recibidos, de los datagramas UDP, de las conexiones TCP en marcha, etc.

La comunicación siempre se realiza entre el administrador y un agente, pues los agentes no pueden intercambiar información entre ellos. Éstos emiten respuestas a las órdenes enviadas por el administrador (lo normal es una respuesta de un agente por cada orden enviada por el administrador). El protocolo SNMP define el formato de las órdenes o mandatos, así como el de las respuestas de los agentes.

El uso de UDP es muy conveniente, pues las comunicaciones entre el administrador y los agentes se basan en peticiones de datos y en respuestas con los datos solicitados. La falta de fiabilidad de UDP no constituye problema alguno para ese tipo de comunicaciones. Frente al TCP, UDP no exige el mantenimiento de una

conexión entre el administrador y cada agente, lo cual resulta muy razonable: las comunicaciones del tipo consulta-respuesta suelen ser esporádicas y tienen pocos datos para intercambiar. Ahora bien, la falta de fiabilidad obliga a que el administrador SNMP compruebe cada cierto tiempo todos los nodos bajo su control, para detectar sucesos inesperados (conexiones y desconexiones de nodos, pongamos por caso). Todos los sucesos relevantes de los agentes son transmitidos al administrador, el cual se encarga de averiguar los detalles exactos mediante consultas a los agentes.

Obvio aquí cualquier consideración de seguridad, pues es una materia bastante compleja. Con todo, el lector puede imaginarse la importancia que tiene la seguridad en la administración de redes. Si los agentes no dispusieran de un modo de asegurarse de que los mensajes vienen de la máquina donde se ejecuta el administrador SNMP, podrían dar información de su estado y de sus características a cualquier máquina que lo solicitara. Consecuentemente, una máquina ajena a la red podría controlar todos los dispositivos, suplantando a la verdadera máquina administradora.

2.8.3. El servicio de transferencia de archivos

El servicio de transferencia de archivos, basado en el protocolo de aplicación FTP, fue uno de los servicios más populares de Internet. Por regla general, FTP se usa para transferir archivos entre anfitriones de una red TCP/IP. No obstante, su función no termina ahí: permite navegar en los sistemas de archivos de las máquinas local y remota, así como crear archivos y ficheros en ambas máquinas (si uno cuenta con los permisos necesarios). En la especificación del protocolo se destaca que está diseñado principalmente para usarlo dentro de programas; si bien los usuarios pueden utilizarlo directamente, sin recurrir a programas que ejerzan de interfaz para el protocolo.

FTP define un conjunto de órdenes que se envían como texto US-ASCII. En el subapartado 3.4 se comentará el estándar de codificación US-ASCII, así como otros muchos más modernos y flexibles; asimismo, se comentarán varias cuestiones concernientes a la internacionalización de las aplicaciones. Por ahora, podemos aceptar que una codificación da una tabla de equivalencia entre a) números, letras y símbolos y b) sus representaciones en bits. Por ejemplo, en US-ASCII, la letra "A" se codifica como 01000001.

Cada orden FTP tiene hasta cuatro caracteres seguidos por cero o más argumentos. Una respuesta del cliente a una orden tiene un número de tres dígitos seguido de una explicación opcional en texto US-ASCII:

Número de 3 dígitos	Respuesta de texto
---------------------	--------------------

Advertencia: Para que se pueda interpretar que una orden está completa (sea de este protocolo o de otro), debe ir seguida de uno o varios caracteres que indiquen un salto de línea. Casi todos los sistemas usan uno o dos caracteres de control (no mostrables por pantalla) para ese propósito: <LF> o <CR>, o <CR> seguido de <LF>. <LF> denota avance de línea (*Line Feed*); <CR>, retorno de carro (*Carriage Return*). Windows, por ejemplo, usa <CR><LF> para indicar una nueva línea; UNIX emplea <LF>.

En la codificación US-ASCII, <CR> tiene el código 13 en decimal y el 0D en hexadecimal. Asimismo, <LF> tiene el código 10 en decimal y el 0A en hexadecimal.

En lenguajes como C, C++ y Java, “\r” denota un retorno de carro (<CR>) y “\t” un avance de línea.

FTP se basa en el modelo cliente-servidor (descrito en el subapartado 2.3): el cliente es cualquier proceso que inicia la transferencia de un archivo, ya sea hacia un anfitrión remoto o desde él; el servidor es el proceso del anfitrión remoto que transfiere el archivo. Casi todos los clientes FTP actuales permiten, además de transferir archivos, recorrer los sistemas de archivos del anfitrión remoto y del local, así como crear y borrar archivos y directorios en dichas máquinas, siempre y cuando uno cuente con los permisos necesarios.

La especificación del FTP incluye la posibilidad de transmitir muchos tipos de archivos; pero las implementaciones actuales del protocolo suelen admitir sólo archivos de textos y archivos binarios.

Los ficheros de texto que se transmiten son convertidos a formato US-ASCII (“texto plano”). Antes de ser enviada, a cada línea se le añade el código US-ASCII de retorno de carro. En el cliente FTP se realiza la conversión de los archivos de texto al formato específico de la plataforma donde se ejecuta el cliente.

Los archivos binarios se transfieren sin modificaciones. Se envían como flujos continuos de bytes, mediante el protocolo TCP. Por la fiabilidad de éste, FTP no tiene que encargarse de controlar los datos perdidos o descartados ni de volver a enviarlos.

Cualquier aplicación que use el protocolo TCP mantiene dos conexiones TCP durante la descarga de cualquier archivo. Una se usa para la transferencia de información de control (emplea el puerto 21 por defecto), y la otra para la transferencia de datos (usa por defecto el puerto 20). Con la primera se intercambian órdenes FTP y respuestas del cliente; con la segunda, los datos de los archivos. Los recursos gastados en mantener dos conexiones durante la transferencia de archivos se compensan con las siguientes ventajas:

- Se puede detener la transmisión de datos en cualquier momento, mediante la orden ABOR (de *abort*, abortar). Si sólo se usara una conexión, no se podría cancelar la transferencia de datos hasta que todos ellos se hubieran enviados. Para evitar el inconveniente anterior, se podría implementar un sistema de control que comprobara cada cierto tiempo que el cliente no hubiera emitido ninguna orden de finalización; pero sería muy ineficaz para transferencias de archivos voluminosos.

- El servidor FTP mantiene en todo instante una suerte de "estado", esto es, la información sobre el directorio actual y sobre la autenticación hecha por el cliente al comienzo.

ESQUEMA DE UNA CONEXIÓN FTP

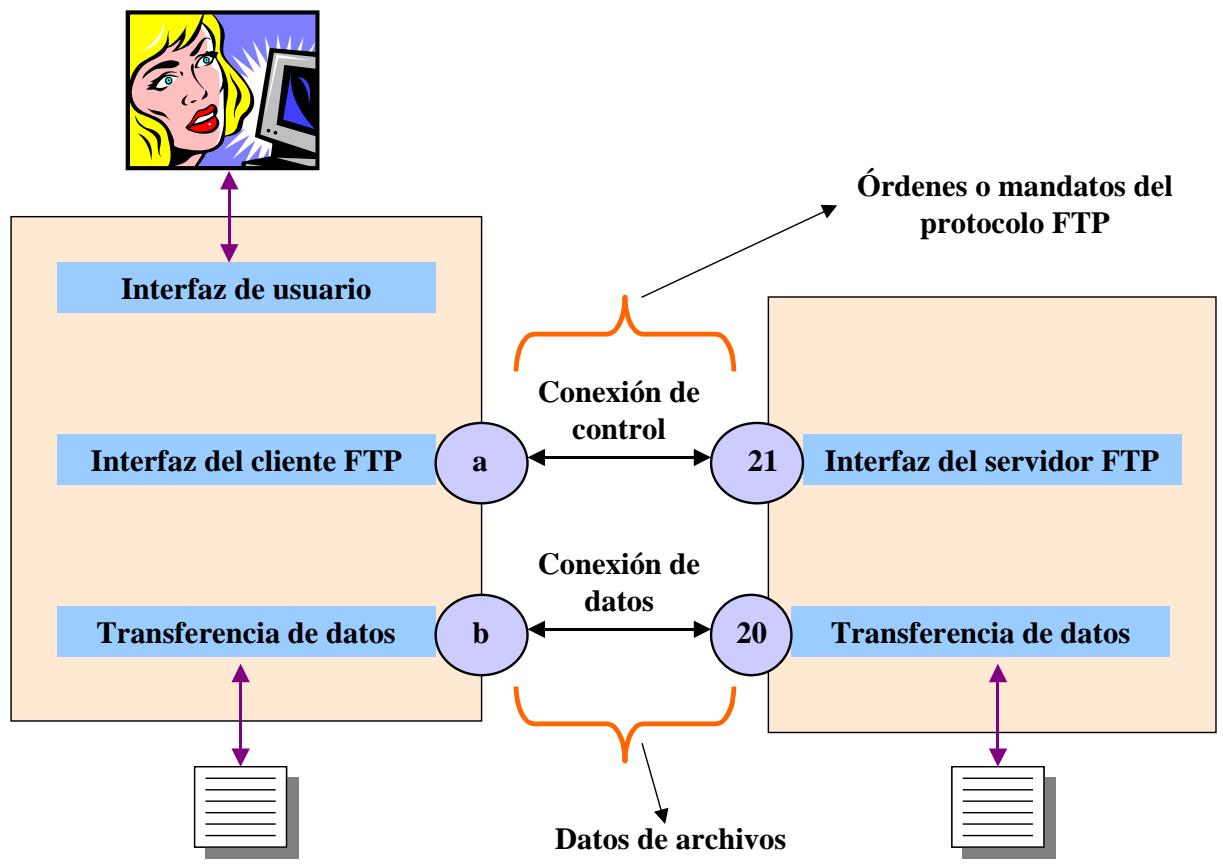


Figura 31. Esquema simplificado de una conexión FTP

Cuando se comienza una sesión FTP (el puerto por defecto es el 21), el servidor FTP envía educadamente un mensaje de bienvenida (código 220). A continuación, el cliente debe enviar un nombre de usuario (orden `USER nombreusuario`). Si el servidor contesta con el código 331 (correspondiente a *Need password for username*: se necesita contraseña para el nombre de usuario), el cliente solicita al usuario una contraseña y la remite al servidor mediante la orden `PASS contrasenya`. Si la contraseña es válida, el servidor envía al cliente la respuesta 230 (acceso autorizado). Para ver los archivos y subdirectorios del directorio actual en el servidor, se usa la orden `DIR`. Cuando el servidor recibe dicha orden, ejecuta otras dos. La primera, del estilo `PORT a, b, c, d, e1, e2`, establece la dirección IP del cliente (`a, b, c, d`) y un número de puerto del cliente (`e2 + 256 × e1`). La segundo (`LIST`) hace que el servidor abra una conexión TCP caracterizada por los argumentos de la primera orden, que envíe la lista de subdirectorios y archivos y que cierre la conexión.

Si se desea descargar un fichero desde un ordenador remoto, se usa primero una orden `PORT` para declarar el puerto que se va a emplear para la conexión; luego, se envía una orden `RETR nombrefichero` que especifique el nombre del archivo que se quiere descargar. Cuando el fichero ha sido transferido correctamente, el servidor FTP cierra la conexión.

FUNCIONAMIENTO DE UNA CONEXIÓN FTP (1)

ftp> sagan.txt

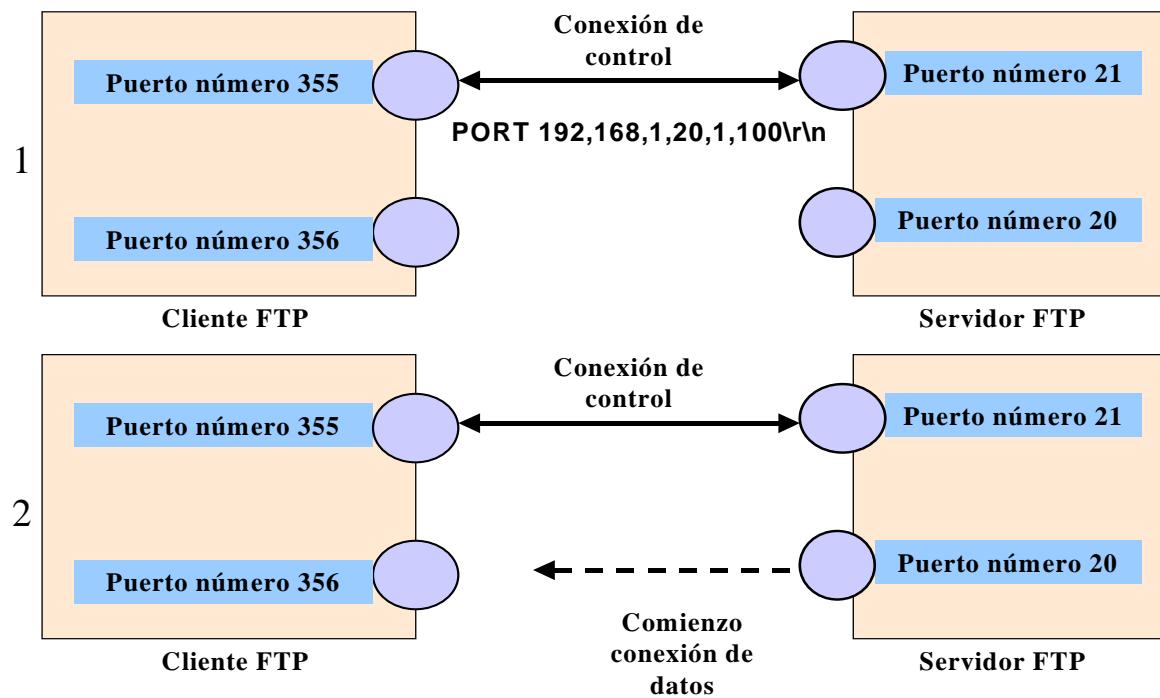


Figura 32. Los dos primeros pasos de una conexión FTP

FUNCIONAMIENTO DE UNA CONEXIÓN FTP (2)

ftp> sagan.txt

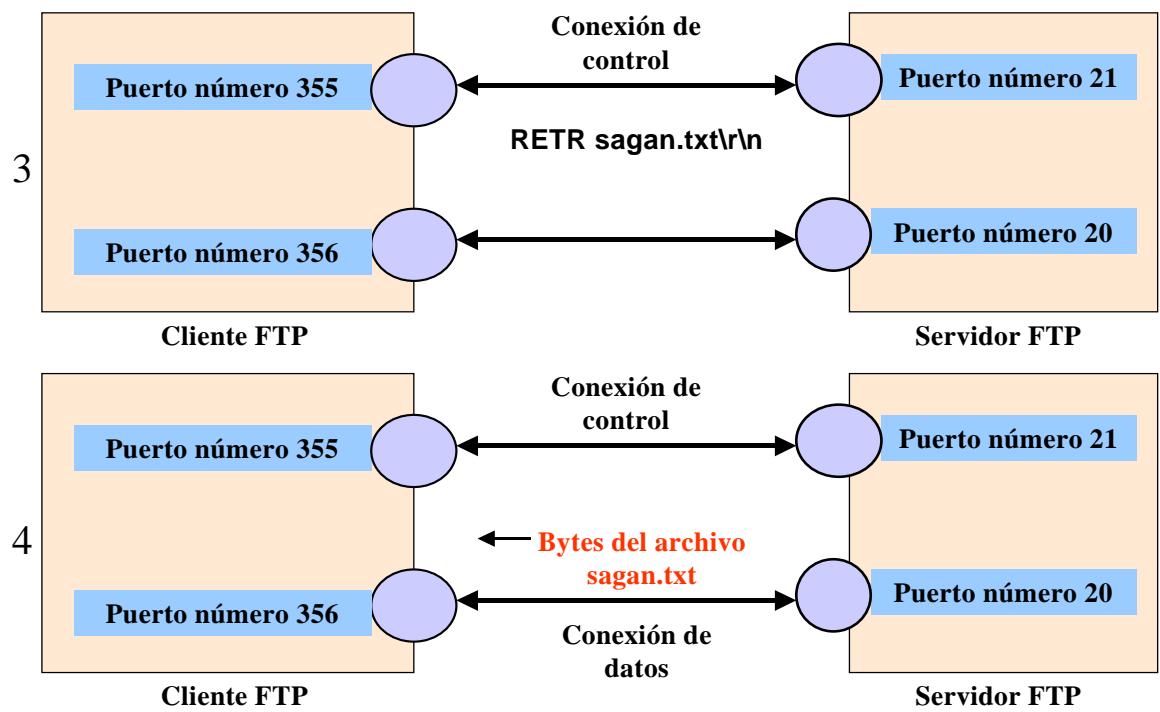


Figura 33. Los dos siguientes pasos de una conexión FTP

FUNCIONAMIENTO DE UNA CONEXIÓN FTP (3)

ftp> sagan.txt

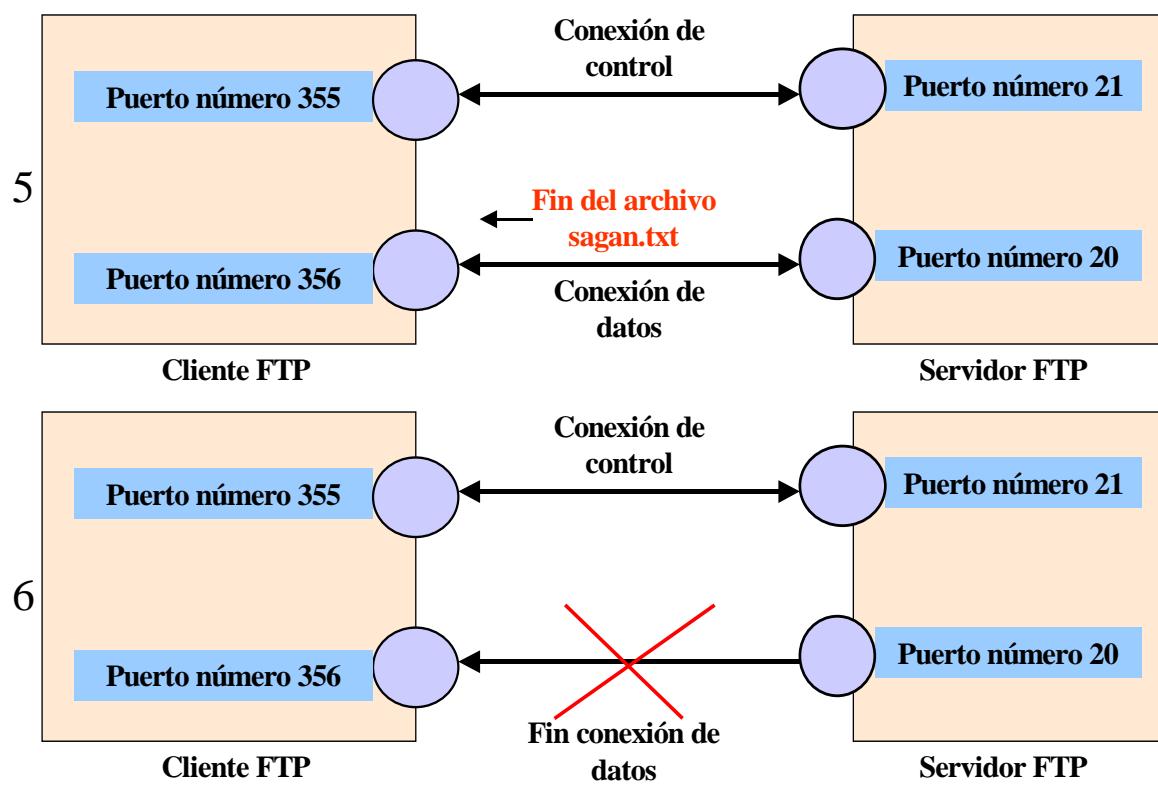


Figura 34. Los dos siguientes pasos de una conexión FTP

FUNCIONAMIENTO DE UNA CONEXIÓN FTP (4)

ftp> sagan.txt

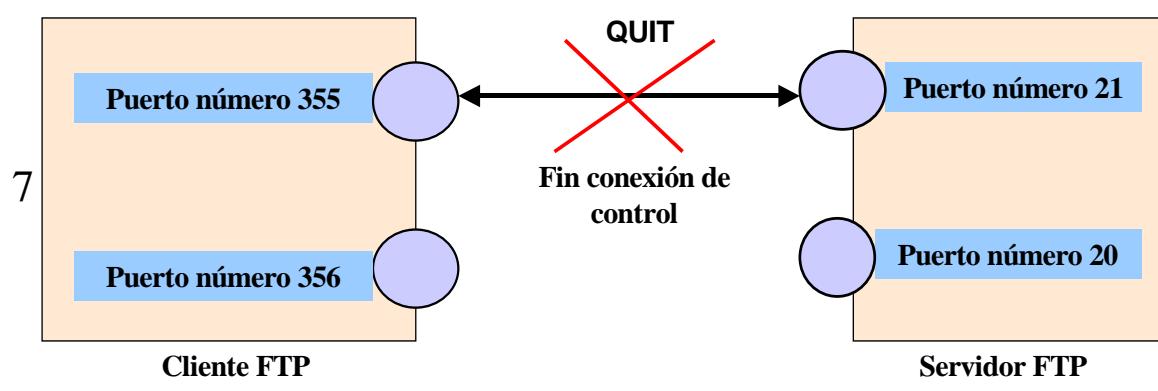


Figura 35. Último paso de una conexión FTP

Para que el lector aprecie las posibilidades del FTP, incluyo aquí unas cuantas órdenes de este protocolo. Lo normal hoy día es usar aplicaciones FTP que proporcionen una interfaz amigable al usuario; pero siempre se pueden usar directamente las órdenes.

- **USER** (*User name*). Los clientes que se conectan a un servidor FTP envían primero este mandato una vez verificada la conexión. Con **USER**, el usuario se identifica ante el servidor y se comprueba el nivel de acceso que tiene a los archivos y directorios de anfitrión donde se ejecuta el servidor. El nombre de usuario se pasa como argumento; por ejemplo: **USER jmlopez**.
- **PASS** (*Password*). Se envía después de la orden **USER**. Como la contraseña se pasa como argumento, el cliente FTP debe encargarse de codificarla si no desea dejar la seguridad en manos del azar.
- **CWD** (*Change Working Directory*). Permite que el usuario cambie a un directorio distinto del actual, ya sea para almacenar archivos o para recuperarlos (asumiendo que cuente con los permisos necesarios). El directorio de trabajo donde se desea trabajar se pasa como argumento.
- **REIN** (*Reinitialise*). Se encarga de terminar la conexión del usuario cuando acaba la transferencia en curso, suponiendo que exista alguna.
- **QUIT**. Cierra la conexión del usuario. Si hay alguna transferencia en marcha, el servidor la cancela.
- **ABOR** (*Abort*). Obliga a que el servidor cancele el último mandato FTP y a que anule cualquier transferencia de datos que pudiera estar en marcha.
- **PORT**. Permite especificar el puerto para las transferencias de datos. Como argumento toma una dirección IP y un número de puerto.
- **TYPE**. El argumento de esta orden especifica la representación de los datos que se transferirán. Dos argumentos frecuentes son **A** (de ASCII) e **I** (de *Image*).
- **RETR** (*Retrieve*). Hace que el servidor transfiera al cliente una copia del archivo cuyo nombre figura como argumento (siempre que el usuario disponga de los permisos apropiados).
- **STOR** (*Store*). Hace que el servidor acepte los datos transferidos por el cliente y que los almacene en un directorio del anfitrión servidor (siempre que se tengan los permisos necesarios).
- **APPE** (*Append*). Hace que el servidor acepte datos provenientes del cliente y que los almacene en el fichero especificado como argumento, añadiéndolos al final del archivo si ya existe.
- **RNFR** (*Rename From*). Especifica el nuevo nombre para el fichero que va a ser renombrado.
- **DELE** (*Delete*). Hace que el fichero especificado en el argumento sea borrado en el servidor, siempre que el usuario disponga de los permisos apropiados.

- **MKD** (*Make Directory*). Hace que el directorio especificado como argumento se borre, si el usuario dispone de los permisos de rigor.
- **PWD** (*Print Working Directory*). Hace que se devuelva el nombre del directorio actual.
- **HELP**. Hace que el servidor envíe información sobre su implementación, configuración, etc.
- **LIST**. Hace que el servidor envíe al cliente una lista de todos los archivos y subdirectorios dentro del directorio actual de trabajo.
- **NOOP** (*No Operation*). Obliga al servidor a enviar una respuesta que confirme que la conexión sigue activa.

2.8.4. El servicio de correo electrónico

El correo electrónico es uno de los servicios más populares de Internet. A un sistema de correo electrónico se le deben exigir estas funciones:

- **Composición:** correspondiente al proceso de creación de los mensajes (rellenado de los campos de un mensaje, existencia de una lista de direcciones).
- **Transferencia:** correspondiente al proceso de mover un mensaje del emisor al destinatario. Es usual conseguir la transferencia mediante la entrega del mensaje a alguna máquina intermedia, la cual se encarga de reenviarlo hasta su destino final.
- **Generación de informes:** correspondiente a la generación de información que indique al emisor lo que ha ocurrido con su mensaje (si ha sido recibido, si se perdió, si fue rechazado, etc.)
- **Presentación de la información:** correspondiente a la presentación de los mensajes al usuario. Si los mensajes sólo tienen texto, implementar esta función resulta sencillo; pero la situación se complica cuando contienen imágenes, vídeos, archivos de audio, etc. El sistema de correo electrónico debe saber qué hacer con cada tipo de archivo dentro del mensaje.
- **Disposición:** se refiere a la gestión de los mensajes (borrado, reenvío, listas de correo, almacenamiento en buzones, etc.)

Los sistemas actuales de correo electrónico suelen constar de dos partes bien diferenciadas, llamadas **agente de usuario** y **agente de transferencia de mensajes**. La primera permite que se escriban y se lean mensajes; por ello, dispone de una serie de órdenes para componer los mensajes, para recibirlos, mostrarlos y contestarlos. La segunda se encarga de mover los mensajes hasta su destino final. Cualquier aplicación que permita a los usuarios leer correos o enviarlos constituye, total o parcialmente, un agente de usuario.

El servicio de correo electrónico se basa en el protocolo de aplicación SMTP (*Simple Mail Transfer Protocol*: protocolo de transferencia de correo sencillo), definido en el RFC 821. El adjetivo “sencillo” procede de su limitación a mensajes sencillos de correo (sin archivos adjuntos ni caracteres no US-ASCII). Algunos textos traducen equivocadamente SMTP como “protocolo simple de transferencia de correo”, cuando lo simple es el correo, no el protocolo.

Este protocolo se diseñó a principios de los años 80. Como ya existían sistemas de correo electrónico anteriores a él, se construyó para que pudiera implementarse sobre cualquier sistema de comunicaciones capaz de manejar líneas de hasta 1.000 caracteres US-ASCII. Así pues, con este protocolo se pueden enviar mensajes a través de redes que no usen TCP/IP; en las redes TCP/IP, TCP es el mecanismo de transporte de los mensajes.

SMTP sigue el modelo cliente-servidor: los procesos que transmiten mensajes operan como clientes; aquellos que los reciben, como servidores. Durante la vida de una conexión SMTP, el cliente y el servidor establecen una conversación: el cliente envía peticiones al servidor, que las atiende.

El formato de los mensajes SMTP se define en el RFC 822:

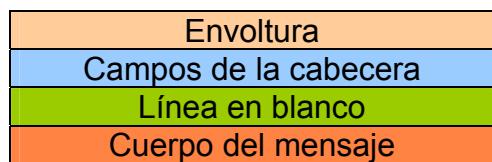


Figura 36. Formato de un mensaje según el RFC 822

Cada campo de la cabecera tiene una línea de texto US-ASCII con el nombre del campo, dos puntos y un valor (algunos campos admiten varios valores). Los campos de la cabecera se muestran en la figura 37. Cuando un usuario usa para escribir correo un agente de usuario, éste crea un mensaje con el formato anterior

Cabecera	Significado
To:	Dirección o direcciones de correo electrónico de los destinatarios primarios.
Cc:	Dirección o direcciones de correo electrónico de los destinatarios secundarios.
Bcc:	Dirección o direcciones con copia oculta
From:	Persona que creó el correo
Sender:	Dirección de correo del remitente
Received:	Línea añadida por cada agente de transferencia a lo largo de la ruta de destino
Return-Path:	Especifica una ruta de retorno al remitente

Figura 37a. Campos obligatorios de la cabecera de un mensaje SMTP

Además de los campos anteriores, una cabecera puede contener otros campos optativos, detallados en la figura 37b.

Cabecera:	Significado:
Date:	Fecha y hora de envío del mensaje.
Reply-To:	Dirección de correo a la que dirigir la contestación
Message-Id:	Número único que sirve para referirse al mensaje
In-Reply-To:	Identificador del mensaje al cual responde este mensaje
References:	Otros identificadores del mensaje
Keywords:	Palabras clave elegidas por el usuario
Subject:	Descripción breve del mensaje

Figura 37b. Campos optativos de la cabecera de un mensaje SMTP

El RFC 822 permite que los usuarios y los programas de correo electrónico inventen nuevas cabeceras. Para crearlas basta con empezarlas con X- (por ejemplo, X-Prioridad-Personal: 7).

SMTP estaba pensado originalmente para mensajes cuyo cuerpo estuviera escrito usando la codificación US-ASCII (volveremos a ella en el subapartado 3.4), la cual resulta inapropiada para muchas lenguas, algunas alfábéticas (español, francés, alemán, ruso, griego, etc.) y otras no alfábéticas (japonés, chino, etc.). Si se intenta usar dicha codificación para una lengua latina como la nuestra, los resultados suelen ser de este tipo:

Desde que a principios de año cambié de programa de correo electrónico, algunos reciben caracteres extraños en mis mensajes =BFQué estás sucede?

Un mensaje escrito en chino:



puede acabar convertido en

TyōRGi hA īš7ōö ËÜMR ötOöÛ ×ðGüfuÃ«#4ª kLcw·âv ,L#@

En cuanto el correo electrónico se fue popularizando en los países no angloparlantes, se hizo evidente que el protocolo SMTP debía dar cabida a muchísimas lenguas que usaban caracteres no US-ASCII. Como la necesidad de usar US-ASCII aparece en el RFC 821, hubo que definir unos RFC que permitieran usar en el cuerpo caracteres no US-ASCII: el RFC 1521 y el 1522, en los cuales se define la especificación MIME (*Multipurpose Internet Mail Extensions*: extensiones multipropósito de correo de Internet). MIME permite la inclusión de caracteres no ingleses y de información binaria en el cuerpo del mensaje. Cualquier texto no US-ASCII o cualquier archivo (de vídeo, de audio, etc.) se codifica como US-ASCII, para así poder enviarlo mediante el protocolo SMTP.

MIME define cinco nuevas cabeceras de mensaje (se describen en el RFC 2045):

Cabecera MIME	Significado
MIME-Version:	Identifica la versión MIME
Content-Type:	Tipo del mensaje
Content-Description:	Cadena legible que describe el contenido del mensaje.
Content-Id:	Identificador único
Content-Transfer-Encoding:	Especifica cómo se codifica el cuerpo del mensaje

Figura 38. Cabeceras de un mensaje MIME

He aquí un ejemplo de mensaje MIME:

```
MIME-Version: 1.0
Content-Type: text/plain; charset=us-ascii
Content-Transfer-Encoding: 7bit
Content-Description: mensaje muy simple MIME
Content-ID: <part00909@jlopez.emple>
```

Este es el cuerpo del mensaje.

La cabecera `MIME-Version` indica la versión de la especificación MIME seguida por el mensaje.

La cabecera `Content-Type` declara el formato original del cuerpo del mensaje. Dentro del primer valor (`text/plain`), `text` corresponde al tipo del cuerpo; `plain`, al subtipo del cuerpo. El tipo es una clasificación genérica del contenido (por ejemplo, `text`); y el subtipo es una clasificación más concreta (por ejemplo, `plain`, `html` o `xml`). El tipo del cuerpo admite estos valores: `text`, `image`, `audio`, `video`, `application` y `message`.

El campo `Content-Transfer-Encoding` indica cómo se codifican los caracteres en bits. Se puede usar la codificación US-ASCII (`us-ascii`), ISO Latin 1 (`iso-8859-1`), etc. Un mismo mensaje puede enviarse con distintas codificaciones.

Cuando se envían mensajes binarios con el cuerpo (un documento PDF o un ejecutable, por ejemplo), suelen codificarse mediante la codificación de base 64. Ésta divide cada grupo de veinticuatro bits en grupos de seis bits, y codifica cada grupo como un carácter ASCII mostrable (no de control). El sistema de codificación es “A” para el cero, “B” para el uno, y así sucesivamente. Cuando se acaban las mayúsculas (“Z” corresponde al veinticinco), se continúa con las minúsculas, luego con los caracteres del “0” al “9” (el “0” corresponde al cincuenta y dos; el “9” al sesenta y uno) y, luego, con “+” (sesenta y dos) y “-“ (sesenta y tres). Los caracteres “=” y “==” se usan para indicar que el último grupo del archivo contiene sólo ocho o dieciséis bits, respectivamente.

0 A	17 R	34 i	51 z
1 B	18 S	35 j	52 0
2 C	19 T	36 k	53 1
3 D	20 U	37 l	54 2
4 E	21 V	38 m	55 3
5 F	22 W	39 n	56 4
6 G	23 X	40 o	57 5
7 H	24 Y	41 p	58 6
8 I	25 Z	42 q	59 7
9 J	26 a	43 r	60 8
10 K	27 b	44 s	61 9
11 L	28 c	45 t	62 +
12 M	29 d	46 u	63 /
13 N	30 e	47 v	
14 O	31 f	48 w	
15 P	32 g	49 x	
16 Q	33 h	50 y	

Figura 39. Tabla de la codificación de base 64: valor y código

Un ejemplo valdrá más que mil palabras: imaginemos que tenemos un archivo binario cuyos primeros veinticuatro bits son:

000100010000010001000011

El segmento se dividirá en grupos de seis bits:

000100 010000 010001 000011

que equivalen en decimal a 4, 16, 17 y 3.

Cada grupo se codificará como una letra, de acuerdo con el esquema expuesto:

EQRD

Por ejemplo, en la codificación de base 64, el siguiente galimatías:

SVNBKjAwKiAgICAgICAgICAgMDAqICAgICAgICAgICowMSo5ODc2NTQzMjEgICAg
ICAgMTIqODAwNTU1MTIzNCAGICAgKjkxMDYwNyowMTEExK1UqMDAYMDAqMTEwMDAw
Nzc3KjAqVCo+CkdTK1BPKjk4NzY1NDMyMSo4MDA1NTUxMjM0KjkyMDUwMSoyMDMy
Kjc3MjEqWCowMDIwMDMKU1QqODUwKjAwMDAwMDAwMQpCRUcqMDAqTkuqTVMxMTEy
Kio5MjAlMDEqKkNPTlRSQUNUIwpSRUYqSVQqODEyODgyNzc2MwpOMSpTVCpNQVZF
Uk1DSyBTWVNURU1TCK4zKjMzMTIgTkVXIEhBTVBTSelSRSBTVFJFRVQKTjQqU0FO
IEpPU0UqQ0EqOTQ4MTEKUE8xKjEqMjUqRUEqKipWQypUUDhNTSpDQipUQVBFOE1N
C1BPMSoYKjMwKkVBKioqVkMqVFAXLzQqO1qVEFQRTEvNE1OQ0gKUE8xKjMqMTI1
KkVBKioqVkMqRFNLMzEvMipDQipESVNLmzUKQ1RUKjMKU0UqMTEqMDAwMDAwMDAx
CkdFKjEqNzcyMQpJRUEqMSoxMTAwMDA3NzCK

equivale a

```

ISA*00*          *00*          *01*987654321          *12*8005551234      *910=
607*0111*U*00200*110000777*0*T*>
GS*PO*987654321*8005551234*920501*2032*7721*X*002003
ST*850*000000001
BEG*00*NE*MS1112**920501**CONTRACT#
REF*IT*8128827763
N1*ST*MAVERICK SYSTEMS
N3*3312 NEW HAMPSHIRE STREET
N4*SAN JOSE*CA*94811
PO1*1*25*EA***VC*TP8MM*CB*TAPE8MM
PO1*2*30*EA***VC*TP1/4*CB*TAPE1/4INCH
PO1*3*125*EA***VC*DSK31/2*CB*DISK35
CTT*3
SE*11*000000001
GE*1      *7721
IEA*1*110000777

```

Este ejemplo corresponde a una petición de comercio electrónico con una empresa de Estados Unidos.

En la siguiente tabla se exponen algunos valores que puede tomar la cabecera Content-Type, definidos en el RFC 1521:

Tipo	Subtipo	Descripción
text	plain	Texto sin formato
	richtext	Texto que incluye órdenes simples de formato
image	gif	Imagen en formato GIF
	jpeg	Imagen en formato JPEG
audio	basic	Archivo de audio
video	mpeg	Archivo en MPEG
application	octet-stream	Secuencia de bytes sin interpretación
	postscript	Documento imprimible en Postscript
message	rfc822	Mensaje MIME RFC 822
	partial	Mensaje dividido para la transmisión
	external-body	El mensaje mismo debe obtenerse de la red

Figura 40. Valores posibles de la cabecera Content-Type

Por ejemplo, cuando en un mensaje aparece el campo Content-Type: image/jpeg, el cliente ya sabe que debe tratarlo como una imagen.

Nota: Si ha trabajado con sistemas UNIX y peina alguna que otra cana, seguramente le sonarán los programas `Uuencode` y `Uudecode`. Su comportamiento es similar al de MIME. Con el primero, un archivo binario se convierte en un archivo de texto US-ASCII mediante una codificación no mucho más compleja que la descrita hace tres páginas. `Uudecode` descodifica el archivo de texto US-ASCII y lo devuelve a su formato original (binario). Con el uso generalizado de MIME, dichos programas apenas se utilizan.

En principio, una implementación del SMTP proporciona todo lo necesario para enviar y recibir correos electrónicos. Una comunicación SMTP comienza cuando la máquina de origen abre una conexión TCP con la máquina de destino mediante el puerto número 25 (asociado por defecto al SMTP). El servidor SMTP envía un mensaje de reconocimiento al cliente, del tipo 220 Server Ready. El formato de las respuestas SMTP es muy similar al que vimos para las respuestas FTP: primero, un código de tres dígitos; a continuación, un mensaje de texto. Un servidor SMTP puede rechazar conexiones mediante la respuesta 421 Service not available. Si la conexión se acepta, el cliente envía la orden HELO nombreanfitrion (por ejemplo, HELO jmgarcia.uv.es). Una vez que el cliente recibe una respuesta 250 OK, puede empezar el envío del mensaje.

El envío del mensaje comienza cuando el cliente indica de quién es el mensaje, a quién se envía, y luego despacha el contenido del mensaje. De quién procede el mensaje se especifica con la orden MAIL FROM <dirección>, que avisa al destinatario de que va a recibir un nuevo mensaje. La dirección entre “<” y “>” es el camino de retorno para el mensaje, esto es, la dirección a la cual se enviará cualquier mensaje de error. Si se usa MAIL FROM: <>, no se enviarán mensajes de error. A quién se envía el mensaje se especifica con la orden RCPT TO: <dirección> (si hay varios destinatarios, se precisa un RCPT para cada uno).

Cada destinatario que reciba el mensaje contestará con una respuesta 250 OK; si el servidor rechaza el mensaje, se enviará una respuesta 550 (siempre que el servidor no reconozca el mandato del cliente, responderá con un código 500 ó 502). Una vez enviada la información concerniente al remitente y a los destinatarios, el contenido del mensaje se envía mediante la orden DATA. En primer lugar se envía una orden DATA y se espera a recibir una respuesta del tipo 354 Start mail input; end with <CRLF> (el texto puede variar de unos servidores a otros; por ejemplo, en otros servidores la respuesta será 354 Enter mail, end with “.” on a line by itself). Cuando le llega, el cliente envía el mensaje como una sucesión de líneas de texto US-ASCII (sólo se permiten caracteres mostrables). Finalmente, se indica con <CRLF> que el mensaje ha terminado (en el otro ejemplo, se indicaría con “.”). Si la transferencia ha sido correcta, el servidor responde con un mensaje 250 OK. Como no se envía para cada línea ningún mensaje de recepción correcta, cualquier error obliga a reenviar todos los datos del mensaje.

Las direcciones que se indican en la cabecera del correo no se emplean en la entrega del mensaje, sólo se usan los argumentos de los mandatos RCPT TO:.

El conjunto de órdenes que puede usar un cliente SMTP se detalla en esta tabla:

Orden	Significado
HELO nombre_anfitrion	Identifica el origen de la conexión
MAIL FROM: <direccion_anfitrion>	Identifica al remitente
RCPT TO: <direccion_destino>	Identifica al destinatario
DATA	Señala el comienzo de la introducción de datos
RSET	Aborta la conexión con el servidor SMTP
QUIT	Cierra la conexión con el servidor SMTP
HELP	Muestra ayuda sobre las órdenes aceptadas
EXPN <direccion_correo>	Expande una lista de correo
VRFY <direccion_correo>	Comprueba la existencia de una dirección de correo.

Figura 41. Órdenes a disposición de un cliente SMTP

Advertencia: El RFC 821 especifica que las órdenes se interpretan sin tener en cuenta si se escriben con mayúsculas o minúsculas (o con una combinación de ambas). Así, daría igual escribir DATA que DaTa o data. Sin embargo, algunos servidores SMTP se han implementado de manera que sólo entienden mandatos escritos en mayúsculas. En este tutorial sigo el criterio de escribirlos siempre con mayúsculas. Cualquier argumento de una orden debe ser de tipo US-ASCII. Así, no están permitidos argumentos como núñez@mail.es, kurtnaßbenger@mail.dt o sánchez@mail.es.

A continuación expongo un ejemplo de transferencia SMTP (disculpe que me use como ejemplo por partida doble, pero es difícil inventar esta clase de ejemplos):

```

220 aidima.es Sendmail SMI-8.6/SVR4 ready at Fri, 9 Jul 2004
16:34:25 GMT
HELO mabian.aidima.es
250 aidima.es OK mabian.aidima.es [192.168.1.20], pleased to
meet you
MAIL FROM: <mabian@aidima.es>
250 <mabian@aidima.es>... Sender ok
RCPT TO <mabian2@aidima.es>
250 <mabian2@aidima.es>... Recipient ok
DATA
354 Enter mail, end with "." On a line by itself
Hola, lectores y lectoras de javaHispano:
Este es un ejemplo del protocolo SMTP.
Saludos.
.
250 TAA11108 Message accepted for delivery
QUIT
221 mabian@aidima.es closing connection

```

El carácter de fin
de mensaje es
indicado por el
servidor

Cuando comenzó a usarse SMTP, era común abrir sesiones Telnet en una máquina remota donde se ejecutaba un servidor SMTP y escribir el correo electrónico tal y como se detalla en el ejemplo. Enseguida fueron apareciendo programas cliente (agentes de usuario) que liberaban al usuario de la tarea de conocer y escribir las órdenes, aunque la interfaz gráfica de los primeros clientes era tan amigable como suave es el puercoespín. Hoy día, los clientes de correo ofrecen interfaces gráficas excelentes, que ponen el correo electrónico a disposición de quien desee usarlo. Además, ofrecen utilidades impensables para aquellos clientes que se ejecutaban en pantallas monocromas y sin ratón. A saber: filtros, reglas de selección y almacenamiento, mensajes predefinidos (“Estoy de vacaciones. Volveré el 28 de agosto.”), etc.

Siento cierta nostalgia por los programas UNIX de correo con que trabajé en mi tesis. Funcionaban exclusivamente por teclado y su interfaz gráfica parecía sacada de un terminal “tonto” que hubiera escapado del desguace. Con todo, aún los empleo de vez en cuando, por razones que no vienen al caso. Viéndolos en retrospectiva, debo reconocer que permiten hacer el 90 por ciento de las cosas que hacen los modernos clientes de correo, llenos de colorines y gráficos y empeñados en consumir unos recursos desproporcionados si los comparamos con aquéllos.

El protocolo SMTP se encarga de establecer una conexión TCP entre la máquina de destino (cliente) y la receptora (servidor) y de enviar directamente el mensaje a través de ella. En consecuencia, basta este protocolo para entregar mensajes directamente a los usuarios. Ahora bien, pocas veces interesa la entrega directa de los mensajes a los destinatarios. Veamos algunos problemas que plantea el uso directo de SMTP:

- a) Si SMTP no consigue enviar un mensaje a su destino, lo reintenta a intervalos cada vez más largos, durante varios días, antes de enviar un mensaje de error a la dirección de camino de retorno. Como las máquinas receptoras actúan como servidores, si no están encendidas y conectadas a Internet las veinticuatro horas del día puede suceder que el servidor esté apagado, desconectado, o ambas cosas, cada vez que el cliente intenta enviarle el mensaje.
- b) Muchas veces, el uso de conexiones TCP está restringido por motivos de seguridad. A los cortafuegos les desagradan las conexiones a puertos aleatorios.
- c) Cuando el destinatario usa un protocolo de correo y el remitente otro, la comunicación directa se torna imposible. Las incompatibilidades entre dos protocolos de correo electrónico pueden ser legión: uno puede admitir campos de cabecera que el otro no tiene, pueden haberse construidos sobre familias de protocolos incompatibles, pueden existir diferencias semánticas entre los campos de la cabecera (aun cuando tengan el mismo nombre), los formatos de los argumentos de los mandatos pueden ser distintos o tener diferentes representaciones binarias... Aunque los problemas expuestos no son triviales, palidecen ante los derivados de la falta de compatibilidad entre los campos del cuerpo. Imagine que envía un mensaje en cuyo

cuerpo hay una referencia a un archivo disponible mediante FTP (por ejemplo, <ftp://aidima.es/publico/furnitureexplorer.html>). ¿Cómo podrá interpretarlo un cliente de correo que trabaje con un protocolo del modelo OSI, donde no existe FTP? No podrá hacerlo directamente. Es más: posiblemente tampoco podrá hacerlo indirectamente (mediante “traductores” o pasarelas, que funcionan bien para mensajes de texto US-ASCII), pues las diferencias de implementación entre los protocolos TCP/IP y los OSI son casi insalvables.

Estos problemas suelen evitarse **servidores de correo electrónico**, que se encargan de almacenar, recibir y transmitir correos electrónicos. El propósito básico de un servidor de correo electrónico es actuar como un almacén de correos al cual pueden acceder los destinatarios. En un servidor de correo electrónico, cada usuario autorizado dispone de uno o varios buzones, donde se van almacenando los mensajes que recibe, que luego pueden descargarse en una máquina local para su posterior lectura.

Un protocolo de aplicación sencillo y eficaz para recuperar mensajes de un buzón de un servidor de correo es el **POP3** (*Post Office Protocol Version 3*: versión 3 del protocolo de oficina de correos), definido en el RFC 1225. Este protocolo tiene órdenes para que un usuario establezca una sesión, obtenga sus mensajes, los borre y cierre la sesión. Con ellas, el usuario puede obtener los mensajes de sus buzones remotos (ubicados en servidores de correo) y almacenarlos en su máquina local para leerlos cuando desee.

POP3 usa el protocolo TCP y, por defecto, el puerto TCP estándar 110. También se basa en el modelo cliente-servidor: los clientes reclaman sus correos, y el servidor se los envía. Cuando un cliente se conecta a un servidor POP3 para leer su correo, obtiene la respuesta **+OK** POP3 server ready. POP3 emplea **+OK** y **-ERR** para comunicar al cliente su aceptación o rechazo de la última orden recibida. Basta con que el cliente lea el primer carácter de una respuesta (“-” o “+”) para que sepa si se ha producido un error o no. El siguiente paso por parte del cliente consiste en enviar al servidor el mandato **USER nombreusuario**. Si el servidor devuelve una respuesta **+OK**, el cliente debe enviar un mandato **PASS contrasenya**. Si la contraseña es válida, el cliente recibirá un mensaje del estilo **+OK nombreusuario has x message(s) (y octets)**, donde **x** es el número de mensajes sin leer e **y** los bytes que ocupan. Durante el proceso de identificación, si el cliente introduce una contraseña incorrecta o irreconocible por el servidor recibirá un mensaje del estilo **-ERR**.

Advertencia: En este tutorial uso los términos *byte* y *octeto* de forma intercambiable. Si se consultan otros textos, conviene tener presente que no siempre un byte equivale a 8 bits (octeto): hay sistemas operativos antiguos, pero que aún se usan, que trabajan con bytes de 7 bits. Raro sería que el lector se encontrara con alguno, pero más vale estar prevenido.

El mandato **STAT** devuelve el número de mensajes que no han sido leídos (**num_mensajes**) y su tamaño total en bytes (**num_bytes**), con una respuesta del tipo **+OK num_mensajes num_bytes**. El mandato **LIST** sirve para averiguar el tamaño de cada mensaje. **RETR** se usa para retirar mensajes del buzón: el servidor envía una respuesta **+OK** y envía el mensaje completo al ordenador local del usuario. La orden

TOP num_mensaje num_lineas lista la cabecera del mensaje con número num_mensaje, más num_lineas del cuerpo del mensaje.

En la tabla siguiente se muestran las órdenes más habituales de los clientes POP3:

Orden	Significado
USER usuario	Identifica al usuario
PASS contrasenya	Identifica la contraseña del usuario
STAT	Devuelve el número de mensajes y sus tamaños en bytes
LIST	Lista los mensajes en el buzón y sus tamaños
RETR num_mensaje	Lee un mensaje
DELE num_mensaje	Marca un mensaje para borrarlo al cerrar la sesión
QUIT	Acaba la sesión. Cuando se ejecuta, se borran todos los mensajes marcados.
TOP num_mens num_lineas	Lista la cabecera del mensaje más las líneas del cuerpo que se marcan

Figura 42. Órdenes a disposición de un cliente POP3

He aquí un ejemplo de una sesión POP3:

```
+OK POP3 server ready <aidima.es>
USER mabian
+OK Password required for mabian.
PASS 34XsdSak9512
+OK mabian has 7 messages.
STAT
+OK 7 1345
LIST
+OK 7 messages (1345 octets)
RETR 1
+OK 546 octets
DELETE 1
+OK message 1 deleted
LIST
+OK 1 messages (799 octets)
TOP 1 20
-ERR message 1 has been deleted
TOP 2 4
+OK 799 octets
Received: from aidima.es by aidima.es id LAA01134; Fri, 9 Jul
2004 18:12:01 GMT
Message ID: <003f01c412a$13a2b510$900000@aidima.es>
From: Jose Luis Gracia jlgracia@aidima.es
To: <mabian@aidima.es>
Subject: Convención sobre interoperabilidad en Riga
Date: Fri, 9 Jul 2004 20:11:37 +0100
```

```
MIME-version: 1.0
Content-Type: text/plain; Charset="iso-8859-1"
Content-Transfer-Encoding: 7 bit
Hola, Miguel Ángel:
¿Tienes ya algo preparado para la convención en Riga? Aún falta,
pero más vale tenerlo en cuenta.
Dime algo.
QUIT
```

Como puede suponerse, no es usual emplear directamente las órdenes, puesto que los agentes de usuario proporcionan interfaces gráficas mucho más cómodas para tratar con los servidores POP3 y descargar mensajes. Existen, sin embargo, algunas ocasiones en las que puede venir bien establecer una sesión de Telnet en un servidor POP3 e introducir a pelo las órdenes:

- Cuando se quiere eliminar un mensaje defectuoso que bloquea al cliente de correo cada vez que lo intenta descargar a la máquina local.
- Cuando se quieren eliminar mensajes sin haberlos descargado (por su tamaño o su peligrosidad, por ejemplo).
- Cuando se quiere comprobar el contenido de los mensajes antes de bajarlos, viendo los campos de la cabecera o las primeras líneas del cuerpo, pongamos por caso.
- Cuando se quieren leer los correos desde cualquier parte del mundo sin necesidad de navegadores (hay servicios que permiten conectarse al buzón de correo de uno mediante un navegador web). ¿Por qué complicarse tanto la vida con POP3 en lugar de usar un navegador y uno de esos servicios? Existen motivos de seguridad para ello, pero aquí no me voy a extender en ellos: no es propósito del tutorial desvelar las vulnerabilidades de ningún sistema.

En la figura 43 se muestra un sencillo esquema del proceso de envío y entrega de correos electrónicos.

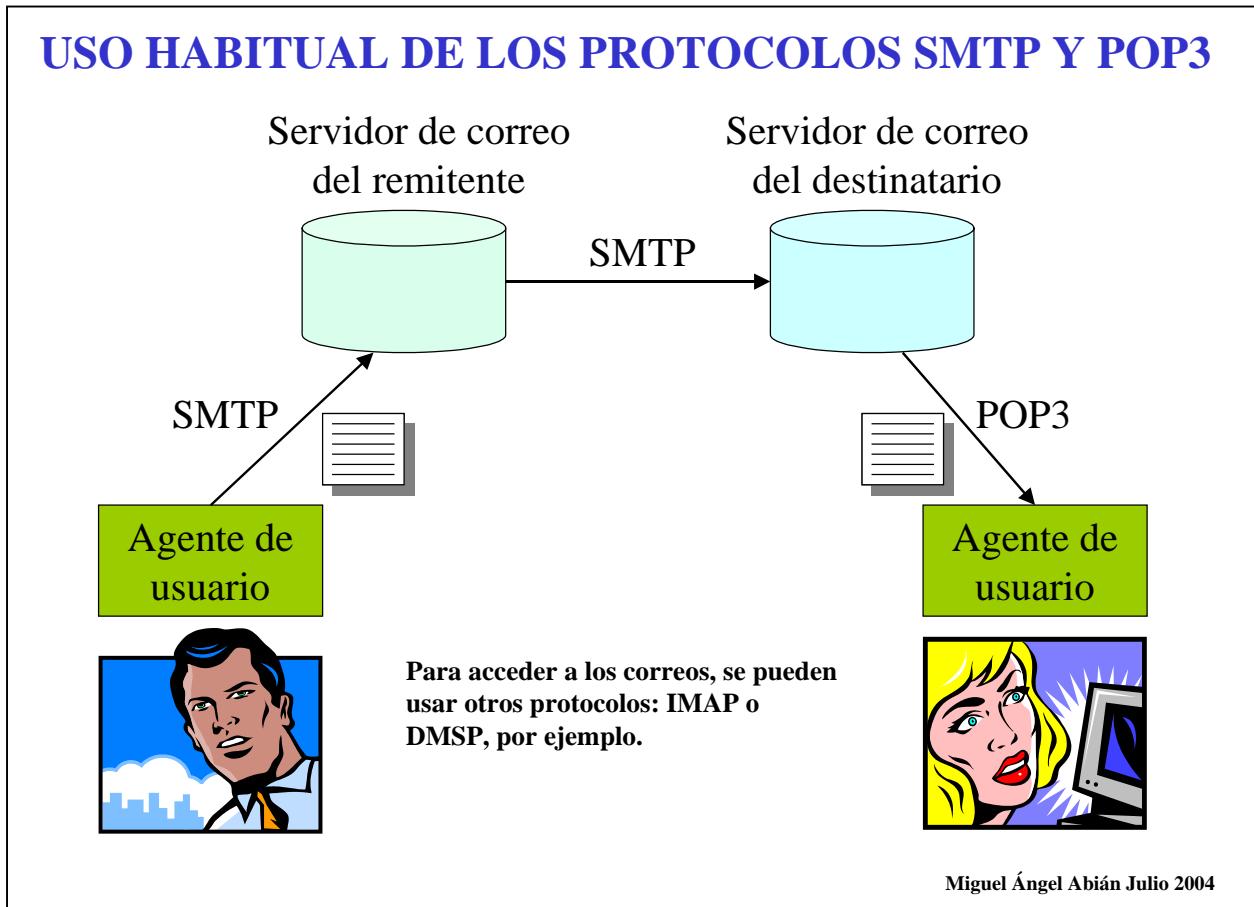


Figura 43. Esquema del uso combinado de los protocolos SMTP y POP3

Además de POP3, existen otros protocolos más modernos y perfeccionados para el envío de correo electrónico desde el servidor a la máquina local empleada por el usuario. Dos de ellos son el IMAP (*Interactive Mail Access Protocol*: protocolo interactivo de acceso al correo) y el DMSP (*Distributed Mail System Protocol*: protocolo de sistema de correo distribuido).

El IMAP se define en el RFC 1064. A diferencia del POP3, no copia el correo en la máquina local del usuario, pues se diseñó para que el usuario pueda acceder a su correo desde cualquier máquina (un PC, una agenda electrónica, un ordenador portátil, etc.). Los servidores de correo que usan IMAP siempre mantienen un depósito de mensajes al cual los usuarios autorizados tienen acceso desde cualquier máquina. Además, IMAP permite guardar los mensajes mediante atributos (fecha de envío, nombre del remitente, palabras clave, etc.). Así, el correo puede consultarse con órdenes que equivalen a “Muéstreme todos los mensajes recibidos el 23/07/04”, “Muéstreme todos los mensajes de Fulanito”, etc.

El DMSP se describe en el RFC 1056. A diferencia del IMAP, permite descargar correo del servidor a una máquina local. Sin embargo, no acaba ahí su trabajo: una vez descargado el correo y cerrada la conexión, los usuarios pueden leer su correo y contestarlo; cuando reabren su conexión, todo el nuevo correo se transferirá al servidor y se sincronizarán los mensajes en el servidor y en la máquina local. Por ejemplo, los correos borrados mientras el usuario estaba desconectado desaparecerán del servidor cuando se establezca una reconexión.

2.8.5. La World Wide Web

La *World Wide Web* (también conocida como WWW o Web) es, para muchos usuarios, Internet. Desde luego, es con diferencia el servicio más popular de la capa de aplicación. Dentro de la arquitectura TCP/IP, la *Web* introduce protocolos de aplicación y servicios nuevos. Los navegadores *web*, los servidores *web* y el protocolo HTTP se ubican en la cima de la familia de protocolos TCP/IP. La *Web* no es un sistema cerrado ni estático; según se van necesitando, se le añaden nuevos protocolos y aplicaciones.

Dependiendo de quien use el término *Web*, suele emplearse una definición u otra. A continuación doy seis que son bastante comunes (no será preciso que le diga cuáles son oficiales, enseguida lo notará):

- El universo de la información accesible de la red, la encarnación del conocimiento humano.
- Un sistema distribuido de hipertexto, de tipo cliente-servidor y que funciona en Internet, que proporciona una única interfaz de usuario para acceder a un gran conjunto de información almacenada como informes, notas, bases de datos, documentación de ordenadores.
- Un servicio de información multimedia, distribuido, basado en hipertexto y que funciona en Internet: es interactivo, dinámico, distribuido y multiplataforma.
- Un conjunto de servicios (correo electrónico, foros, transferencia de archivos, HTTP, etc.) a los cuales se puede acceder mediante un navegador web.
- Un conjunto de archivos de hipertexto, de imágenes, vídeos y sonidos ubicados en los servidores web.
- Un conjunto de protocolos que permiten la consulta y la transmisión de páginas *web* en Internet, en las intranets y en las extranets.

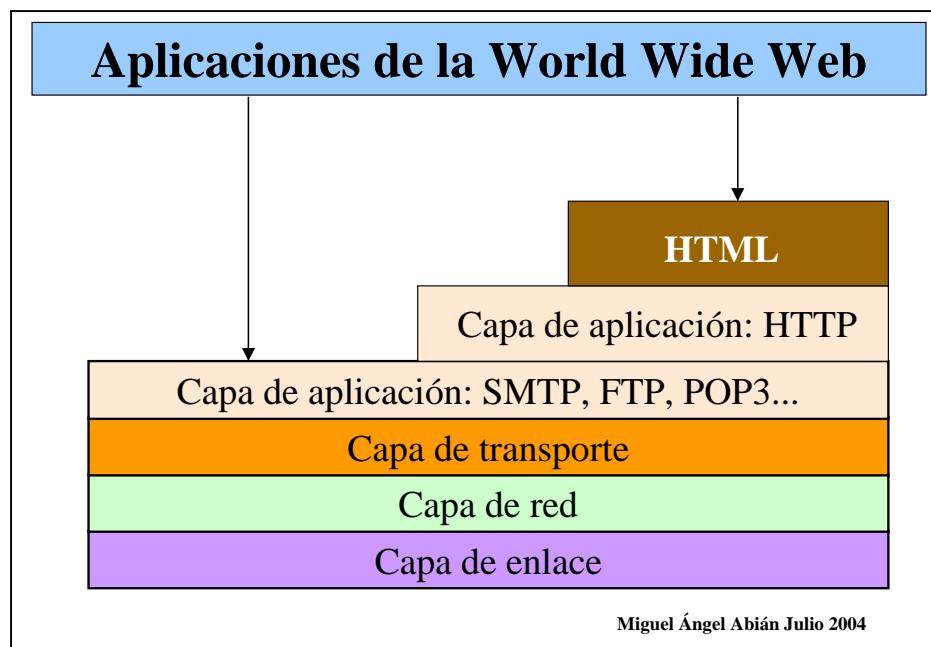


Figura 44. La WWW dentro de la arquitectura TCP/IP

La característica más relevante de la *Web* es su estructura hipertextual: la WWW contiene un gran conjunto de documentos de hipertexto (llamados páginas *web*). Cualquier documento de hipertexto contiene enlaces que conectan con otros documentos; un enlace puede ser una palabra, una frase o un gráfico. A través de los enlaces, se puede acceder a otros documentos, imágenes, vídeos, sonidos, etc. En general, se dice que la red permite acceder a **recursos**. Un recurso es una página *web* o algún tipo de contenido susceptible de ser almacenado en un archivo, binario o no, y presentado al usuario de una forma inteligible. Un recurso puede ser una imagen, un directorio, un vídeo, un documento Word, PDF, PostScript, etc. También puede ser una referencia a una consulta a una base de datos o a un motor de búsqueda (como Google).

Como los enlaces de una página a otros recursos pueden no seguir caminos lógicos o directos, la *Web* es un entramado de documentos, vídeos, imágenes, animaciones y sonidos. De ahí lo apropiado del nombre (*web* significa telaraña o malla). Los documentos de la *Web* no tienen porque ser necesariamente estáticos: pueden generarse de forma dinámica, como respuesta a las acciones o peticiones de los usuarios. Por ejemplo, cuando se hace una compra electrónica, la página *web* que nos muestra los detalles de la transacción no existía antes de la compra.

Como supongo que si está leyendo esto sabe ya muy bien qué aspecto presenta la *Web* y cómo se navega por ella, obviaré explicarlo y me dedicaré a explorar sus entrañas.

El servicio de la *World Wide Web* se basa, como tantos otros, en el modelo cliente-servidor: el cliente solicita acceder a páginas *web*, y el servidor se las suministra. Tres son los componentes fundamentales de este servicio:

- Una arquitectura cliente-servidor que usa el protocolo de aplicación HTTP (*HyperText Transfer Protocol*: protocolo de transferencia de hipertexto) y se basa en TCP/IP (tanto en la acepción de familia de protocolos como en la de modelo de capas). Por debajo del HTTP quedan los protocolos de las capas de red, de transporte y de enlace. A diferencia de FTP o Telnet, HTTP no se considera como parte estándar de la familia de protocolos TCP/IP, aunque es casi imposible encontrar algún producto comercial que no lo implemente. HTTP viene a ser el protocolo vernáculo de la *Web*.
- Los URL (*Uniform Resource Locator*: localizador uniforme de recursos).
- Un lenguaje de etiquetado de hipertexto (HTML o *HyperText Markup Language*: lenguaje de etiquetado de hipertexto). Este lenguaje, basado en etiquetas, es el que permite escribir páginas *web* y establecer enlaces entre ellas.

En este apartado consideraré sólo los dos primeros componentes de la *Web*. En ella se pueden encontrar muchísimos manuales de HTML, gratuitos y de buena calidad. Nótese la naturaleza metalingüística de este hecho: el HTML se usa para describirse a sí mismo.

Para localizar un recurso en la *Web*, se usan los **URL** (*Uniform Resource Locator*: localizador uniforme de recursos), que vienen a ser punteros a los recursos de la *Web*.

Los URL ya han sido mencionados antes e incluso se adelantó una definición provisional en el subapartado 2.1. En un URL hay la siguiente información:

- El protocolo que debe usarse para acceder al recurso.
- El nombre de la máquina donde está el recurso.
- El número de puerto por el cual la máquina permite solicitar el recurso.
- La ubicación del recurso dentro de la máquina.

Un URL tiene la forma

esquema : localización-según-el-esquema

La primera parte (esquema) especifica el protocolo de aplicación que se usará para acceder al recurso. Un URL como <ftp://ftp.upv.es/comun/temario.doc> identifica un archivo que puede conseguirse mediante el protocolo FTP (visto en 2.8.3). Los URL de HTTP son los más comunes para localizar recursos a los que se puede acceder mediante el protocolo HTTP. El lector puede encontrar más información de los tipos de URL y de sus formatos en

<http://archive.ncsa.uiuc.edu/SDG/Software/Mosaic/Demo/url-primer.html>

En este mismo ejemplo, `http` nos indica el protocolo necesario para acceder al recurso. La cadena `archive.ncsa.uiuc.edu` nos indica que el recurso está ubicado en un anfitrión cuyo nombre DNS es `archive.ncsa.uiuc.edu`. La cadena `/SDG/Software/Mosaic/Demo/` nos indica el camino dentro del anfitrión `archive.ncsa.uiuc.edu` donde se encuentra el recurso. Por último, `url-primer.html` nos indica el nombre del recurso.

En un URL se puede especificar un número de puerto. Tal como ya se comentó, si éste no se especifica, se sobreentiende que se usa el puerto por defecto correspondiente al protocolo usado. En el caso de HTTP, el puerto por defecto es el 80. Podríamos, por tanto, haber escrito

<http://archive.ncsa.uiuc.edu:80/SDG/Software/Mosaic/Demo/url-primer.html>

Los URL siempre han tenido un defecto inherente a su formato: todo URL hace referencia a un anfitrión concreto; no permite apuntar a un recurso sin especificar dónde se encuentra. Para los recursos muy solicitados, sería interesante disponer de copias del recurso, ubicadas en distintos anfitriones. Así se podrían distribuir las peticiones entre varios anfitriones distantes y se repartiría el tráfico de datos entre redes alejadas geográficamente. Sin embargo, los URL no permiten solicitar recursos independientemente de los anfitriones donde se encuentran o se generan estos recursos.

Una solución a este problema la dan los **URN** (*Uniform Resource Name*: nombre uniforme de recursos). Vienen a ser unos punteros independientes de la localización. Gracias a ellos, un recurso puede ser copiado en muchos lugares distintos. Si una copia no se encuentra disponible en un sitio dado, el recurso puede ser encontrado en

otro sitio. Los URN permiten referirse a los *recursos mediante nombres*, sin decir dónde están aquéllos. He aquí un ejemplo de un URN:

[urn:isbn:0232343789](#)

Los **URI** (*Uniform Resource Identifier*: identificador uniforme de recursos) son otra vuelta de tuerca a concepto de URL: constituyen una abstracción que incluye a los URL y URN. Un URI asigna un nombre a un recurso, lo describe y permite encontrarlo.

En la arquitectura cliente-servidor de la *Web*, los clientes suelen usar **navegadores web** (*web browsers*). Un navegador es una aplicación que se encarga de obtener los recursos *web*, de interpretar las etiquetas HTML y de mostrarlas por pantalla.

Los navegadores se comunican con los servidores *web* mediante el protocolo HTTP, que veremos más adelante, y usan los URL para localizar los recursos. Casi todos permiten el uso de otros protocolos (FTP, HTTPS, etc.) además del HTTP. En 2004, los navegadores más populares son Internet Explorer, Opera, Safari y los basados en Mozilla.

La guerra de navegadores entre Microsoft y Netscape (comprada por AOL a finales de 1998) provocó la aparición de extensiones del HTML no del todo compatibles, lo cual ha retrasado mucho la estandarización del lenguaje de marcado. Un perfecto ejemplo de la falta de interoperabilidad nos lo dan los letreros del tipo “Optimizado para Internet Explorer 5.0 y para una resolución de 800x600”.

En la parte del servidor de la *Web* encontramos **servidores web**. Un servidor *web* es una aplicación o proceso que implementa al protocolo HTTP para recuperar recursos que vienen identificados por sus URL. En consecuencia, servidor *web* es sinónimo de servidor HTTP. Estos servidores se implementan mediante demonios; por ejemplo, en UNIX los servidores *web* se llaman `Httpd` (la letra “d” es de “demonio”). También se usa “servidor *web*” para referirse a los anfitriones encargados de “servir” recursos (casi siempre páginas *web*) mediante el protocolo HTTP.

El servidor *web* más popular es Apache, seguido por el Internet Information Server, de Microsoft. En la actualidad, a cualquier servidor HTTP “serio” se le debe pedir que sea capaz de realizar estas tareas:

- Llevar el registro de las actividades (conexiones, desconexiones, fallos, intentos de acceso no autorizado, etc.)
- Permitir la identificación de los usuarios.
- Permitir enviar datos a subrutinas o procedimientos que los procesen.
- Encargarse de la creación y gestión de directorios virtuales.
- Permitir trabajar con otros protocolos además del HTTP: HTTPS, SSL, FT, Gopher, etc.

Un **sitio web** (*website*) es una colección de páginas *web*, que tiene un común la raíz de sus URL. No necesariamente tienen que estar localizadas físicamente en un mismo anfitrión, aun cuando esto es lo usual.

El protocolo HTTP, que usa la codificación US-ASCII, es el protocolo *de facto* para la transferencia de archivos web y el acceso a los recursos de la Web,. La versión 1.1 de este protocolo se documenta en el RFC 2616 (la versión 1.0, ahora obsoleta, se documenta en el RFC 1945).

El RFC 2068 afirma que el protocolo HTTP funciona por lo general sobre una conexión TCP, pero que no depende de una capa de transporte específica. Tal y como se define, HTTP es un protocolo abierto, extensible, sin conexión y sin estado. Es abierto y extensible porque, cuando transmite información a un cliente, incluye una cabecera MIME que informa a éste de la clase de datos que siguen a la cabecera. Con ese conocimiento, los clientes saben qué aplicaciones necesitan para interpretar el recurso (reproductores de vídeo, de audio, etc.). Si se crean recursos nuevos, HTTP puede incorporarlos sin grandes problemas. Su carencia de conexión y estado se debe a que, una vez contestada la petición de un cliente y pasado un tiempo, la conexión entre servidor y cliente se desecha y no se registra. Para el servidor, cada petición es "nueva", pues no la asocia a peticiones anteriores o a clientes concretos. Si fuera humano, HTTP sería amnésico.

Toda comunicación HTTP consiste en tres partes:

- Una línea inicial, que puede corresponder a una respuesta o a una petición.
- Una cabecera (optativo).
- Un cuerpo (optativo).

En el **lado del cliente**, una conexión HTTP sigue estos pasos:

- 1) El cliente (un navegador) abre un *socket* TCP y se conecta al servidor HTTP por medio de un puerto (por defecto, el 80). Mediante el *socket* envía su petición. La línea inicial de una petición consiste en el método de petición (en este protocolo, las órdenes se llaman métodos), la dirección del recurso solicitado y la versión del protocolo HTTP con que se trabaja. Por ejemplo: GET /aidima/index.html HTTP/1.1. Esta línea indica una petición con el método GET del recurso index.html (un archivo HTML), así como la versión del HTTP usado: la 1.1.
- 2) De manera optativa, el cliente puede enviar una cabecera con información acerca de su configuración y de sus preferencias a la hora de mostrar documentos. He aquí un ejemplo:

```
User-Agent: Mozilla/4.76 (WinNT; I)
Accept: image/gif, image/x-xbitmap, image/jpeg,
image/pjpeg, /*
Accept-Encoding: gzip
Accept-Charset: iso-8859-1, *, *utf-8
[línea en blanco para indicar fin de la cabecera]
```

- 3) Tras el envío de la petición y la cabecera, el cliente puede enviar un cuerpo, que contendrá datos enviados por el cliente mediante el método POST. Lo normal es que esos datos se usan en programas de tipo CGI (véase el final de este subapartado).

El **servidor HTTP** contesta a la petición de la siguiente forma:

- 1) Envía al cliente una línea con tres campos: la versión del HTTP usada por el servidor, el código de control y la descripción de este código. Por ejemplo, HTTP/1.1 200 OK indica que el servidor usa la versión 1.1 del protocolo y que devuelve el código 200, cuya descripción es OK. Dicho código significa que la petición del cliente se juzga correcta y que se enviará la información solicitada tras la cabecera.
- 2) Tras la línea anterior, el servidor envía al cliente una cabecera con información sobre sí mismo y sobre el documento solicitado. Veamos un ejemplo:

```
Date: Sat, 10 Jul 2004 09:34:12 GMT
Server: Apache/1.3.12 (Unix) mod_perl/1.21
MIME Version 1.0
Last-modified: Sat, 10 Jul 2004 08:57:56 GMT
Content-Type: text/plain; charset=ISO-8859-1
Content-length: 1239
[línea en blanco para indicar fin de la cabecera]
```

- 3) A continuación se envían los datos solicitados. En nuestro caso, corresponden a un documento HTML (index.html) con un tamaño de 1239 bytes. Pueden tener una forma de este estilo:

```
<HEAD><TITLE>Bienvenido a AIDIMA</HEAD></TITLE>
<BODY>
<H1> Instituto tecnológico del mueble y afines </H1>
...

```

Como vemos en los pasos 2 y 3, el documento que solicitamos está encapsulado mediante MIME (véase 2.8.4). La línea Version 1.0 indica al cliente el cuerpo que vendrá luego está codificado con MIME. La línea Content-type: text/html es la manera con que MIME especifica el tipo de documento o recurso solicitado. Si el cliente no es capaz de trabajar con el tipo de documentos especificado en la cabecera Content-type, no será capaz de mostrarlos. Como ya vimos, los archivos binarios y el texto no US-ASCII se pueden codificar en mensajes MIME, de modo que puedan ser enviados por servidores que sólo admitan US-ASCII. Por ejemplo, si el cliente recibe del servidor una cabecera

```
Content-Type: text/plain; charset=ISO-8859-1
Content-transfer-encoding: base64
```

sabrá que va a recibir, en el cuerpo del mensaje del servidor, un mensaje codificado a partir de la codificación de base 64 y que el mensaje original estaba codificado con ISO-8859-1 (ISO Latin 1).

Veamos otro ejemplo, correspondiente a un archivo PDF:

```
Content-type:application/octet-stream; name=Borrador.pdf
Content-transfer-encoding: base64
CCEhjpVg5EggkkwRSgXWIYQu0AOiRxI0BnfSQDAXR382/MQP/PP8OWXh+AZF
Vb5wo6UYM2kaPr/XqE4Cz38DUEsBAjILFAAAAAGAr0wxI0BnfSQDwgAAAAIEA
AoAAAAAAAAAAUeSDBBQAAAIAK6MMMSNAZ30kW9kMy0xLnBwdOx9
```

De todos modos, TCP permite el envío de flujos de bytes; por tanto, una imagen o un archivo binario se pueden enviar como una secuencia de ceros y unos.

Una de las grandes mejoras de la versión 1.1 del protocolo HTTP frente a la versión 1.0 consiste en que las conexiones TCP permanecen abiertas hasta que el cliente o el servidor las cierran (lo cual suele ocurrir cuando pasa un cierto tiempo sin que se envíen mensajes).

En la versión 1.0, cada petición HTTP implicaba una nueva conexión. Por ejemplo, si una página web tenía muchas imágenes, se necesitaba una conexión TCP para cada una (más la asociada a la página en sí), lo cual provocaba una sobrecarga por la apertura y cierre de conexiones TCP. Como una página puede tener imágenes estáticas, animadas, marcos, *applets*, etc, la versión 1.1 ahorra tener que abrir y cerrar muchas conexiones para mostrar una sola página.

Método	Significado
GET	Permite acceder a recursos del servidor.
HEAD	El formato es similar al de GET pero se solicita al servidor que conteste sólo con cabeceras, no con cuerpos. Suele ser útil para conocer las características de un recurso sin tener que transferirlo al cliente.
POST	Se usa para enviar datos al servidor. Esos datos pueden proceder de un formulario HTML o pueden estar destinados a algún programa en el servidor (CGI)

Figura 45. Métodos más frecuentes del protocolo HTTP

CÓDIGOS DE CONTROL DE HTTP

- | | |
|-------------------------|-----------------------------|
| ★ 200 OK | ★ 403 Forbidden |
| ★ 201 Created | ★ 404 Not Found |
| ★ 202 Accepted | ★ 500 Internal Server Error |
| ★ 204 No Content | ★ 501 Not Implemented |
| ★ 301 Moved Permanently | ★ 502 Bad Gateway |
| ★ 302 Moved Temporarily | ★ 503 Service Unavailable |
| ★ 304 Not Modified | |
| ★ 400 Bad Request | |
| ★ 401 Unauthorized | |

Figura 46. Algunos códigos de control del protocolo HTTP

Las siglas CGI (*Common Gateway Interface*: interfaz común de pasarelas) tienen un doble sentido. Por un lado, es un estándar para ejecutar programas desde servidores web. CGI especifica cómo pasar a un programa en ejecución argumentos como parte de una petición HTTP, y define una serie de variables que dependen del software y hardware empleados. Por otro lado, CGI designa a un programa que se ejecuta en un servidor web y que puede aceptar argumentos de los clientes.

Generalmente, un programa CGI devuelve un documento HTML (una confirmación de la compra de un producto, por ejemplo) que se genera en función de los argumentos pasados por el cliente. Luego, el servidor envía el documento al cliente. En los sistemas UNIX, los servidores de HTTP (`Httpd`) suelen requerir que los programas CGI residan en un directorio `/cgi-bin`.

Antes de los programas CGI, los servidores HTTP eran estáticos: se limitaban a entregar a los clientes las páginas web solicitadas mediante URL.

Los programas CGI obtienen los datos que necesitan mediante los métodos POST y GET del protocolo HTTP. Cuando un usuario rellena el formulario de una página web y aprieta el botón “Enviar”, el navegador envía estos datos al CGI. Si se usa POST, los datos se entregan al CGI por la entrada estándar; si se usa GET, el CGI recibe los datos del formulario mediante la variable de entorno `QUERY_STRING`.

En una llamada a un CGI, ocho son los pasos intermedios:

1. El usuario aprieta el botón “Enviar” de un formulario web, mostrado por un navegador web.
2. El navegador web recoge los datos introducidos en el formulario y los ensambla en una cadena de texto. A continuación, crea una petición HTTP con los datos y la envía al URL que figura en la etiqueta `ACTION` del formulario.
3. El servidor web recibe la petición y usa los datos que hay en ella para configurar las variables de entorno.
4. El servidor web arranca el programa CGI especificado en la petición HTTP del cliente.
5. El CGI recibe el cuerpo de la petición HTTP mediante la entrada estándar o mediante la variable de entorno `QUERY_STRING`, y extrae de él los datos del formulario.
6. El CGI realiza los cálculos o comprobaciones pertinentes y devuelve al servidor, por la salida estándar, una página HTML o un archivo codificado con MIME.
7. El servidor web acepta el archivo enviado por el CGI, añade las cabeceras correspondientes y envía el resultado al navegador del usuario.
8. El navegador muestra al usuario el archivo.

Con la aparición de tecnologías como ASP, JSP o los servlets, la popularidad de los CGI ha ido disminuyendo.

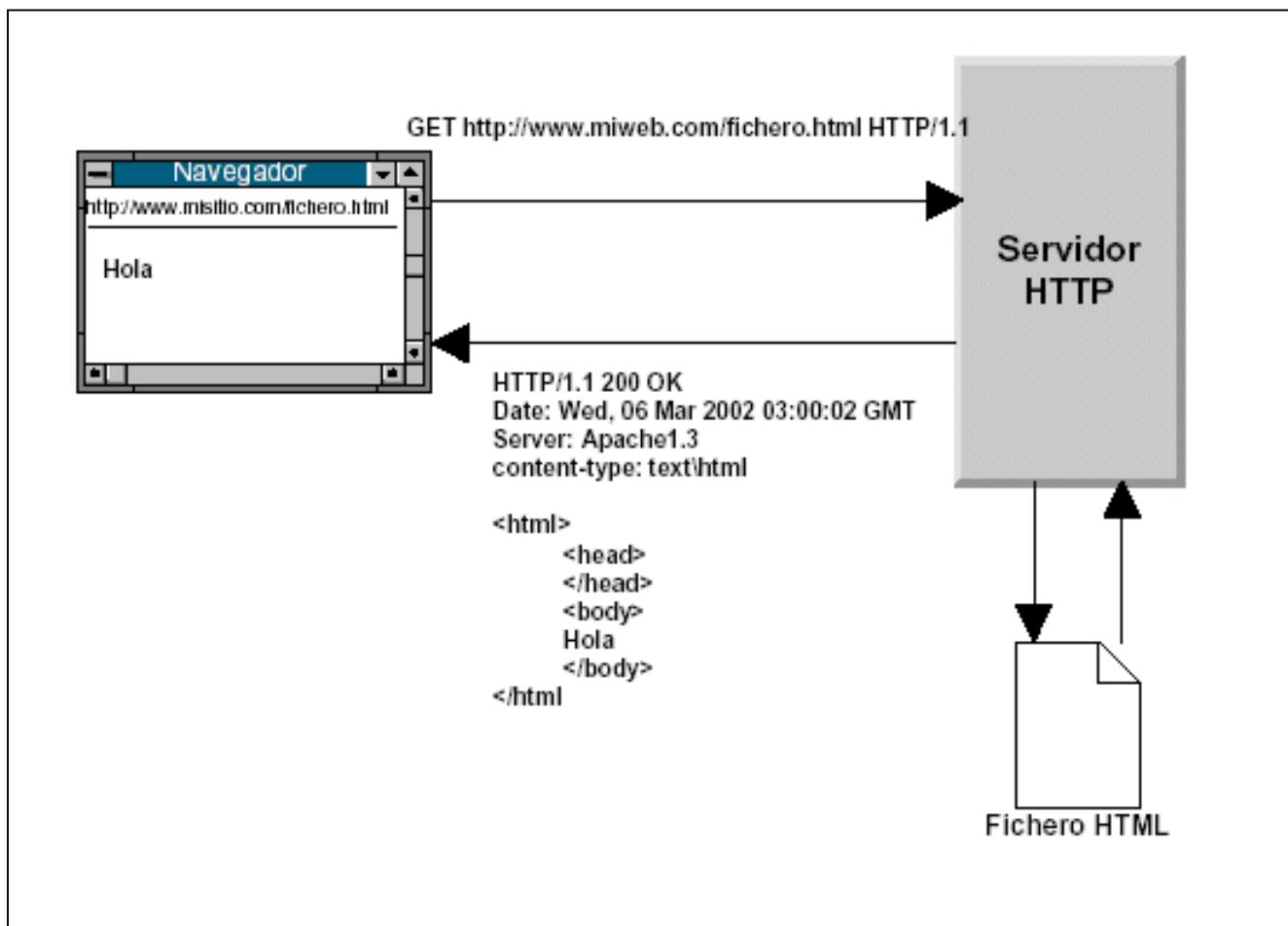


Figura 47a. Esquema de una comunicación HTTP sin CGI

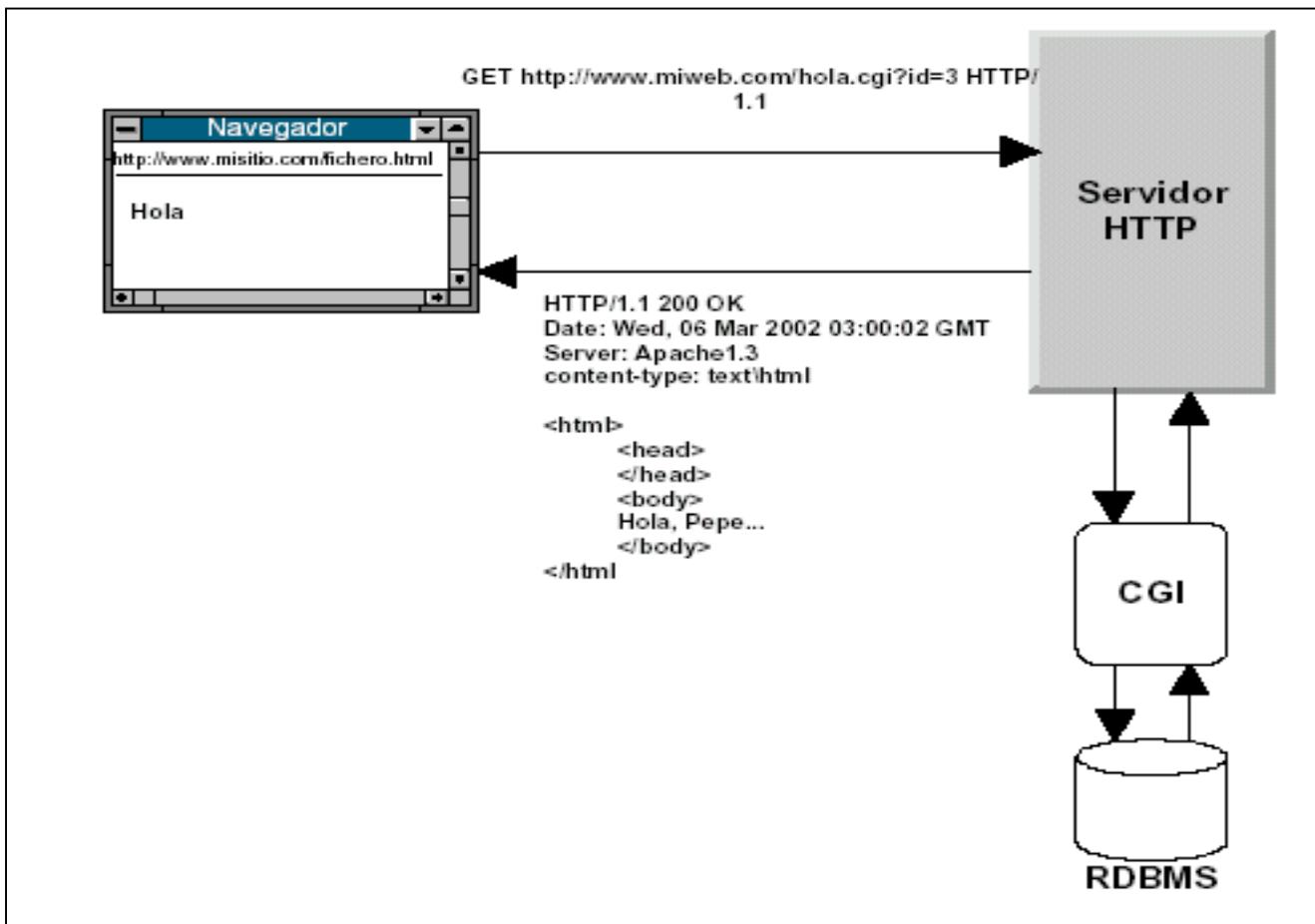


Figura 47b. Esquema de una comunicación HTTP con CGI. En el apartado 4.1 veremos que corresponde a una arquitectura cliente-servidor de cuatro capas

Al lector interesado en aprender más sobre redes y protocolos le recomiendo estos textos: *Computer Networking: A Top-Down Approach Featuring the Internet (2nd Edition)* [J. F. Kurose y K. W. Ross, 2002], *Designing TCP/IP Internetworks* [Geoff Bennett, 1995], *Data and Computer Communications (4th Edition)* [William Stallings, 2000], *Internetworking with TCP/IP Vol. 1: Principles, Protocols, and Architecture (4th Edition)* [Douglas E. Comer, 2000], *Internetworking with TCP/IP Vol. II: ANSI C Version: Design, Implementation, and Internals (3rd Edition)* [Douglas E. Comer y David L. Stevens, 2000] e *Internetworking with TCP/IP, Vol. III: Client-Server Programming and Applications--BSD Socket Version (2nd Edition)* [Douglas E. Comer y David L. Stevens, 1996].

En el libro *Dark Fiber: Tracking Critical Internet Culture (Electronic Culture: History, Theory, and Practice)* [Geert Lovink, 2002] se puede encontrar una interesante visión de la cultura de Internet.

3. El paquete `java.io` de Java

No es propósito de este apartado proporcionar un estudio minucioso o exhaustivo sobre el paquete `java.io` (no se tratan, por ejemplo, los *pipes* ni los ficheros de acceso aleatorio), sino dar una visión general de las clases de Java necesarias para manejar comunicaciones en red. Al lector interesado en un estudio completo de este paquete le remito a la documentación oficial de Sun.

Hasta la aparición del paquete `java.nio` en JDK 1.4, la entrada y salida en java se realizaba exclusivamente mediante `java.io`. Su jerarquía de clases se detalla en el siguiente esquema.

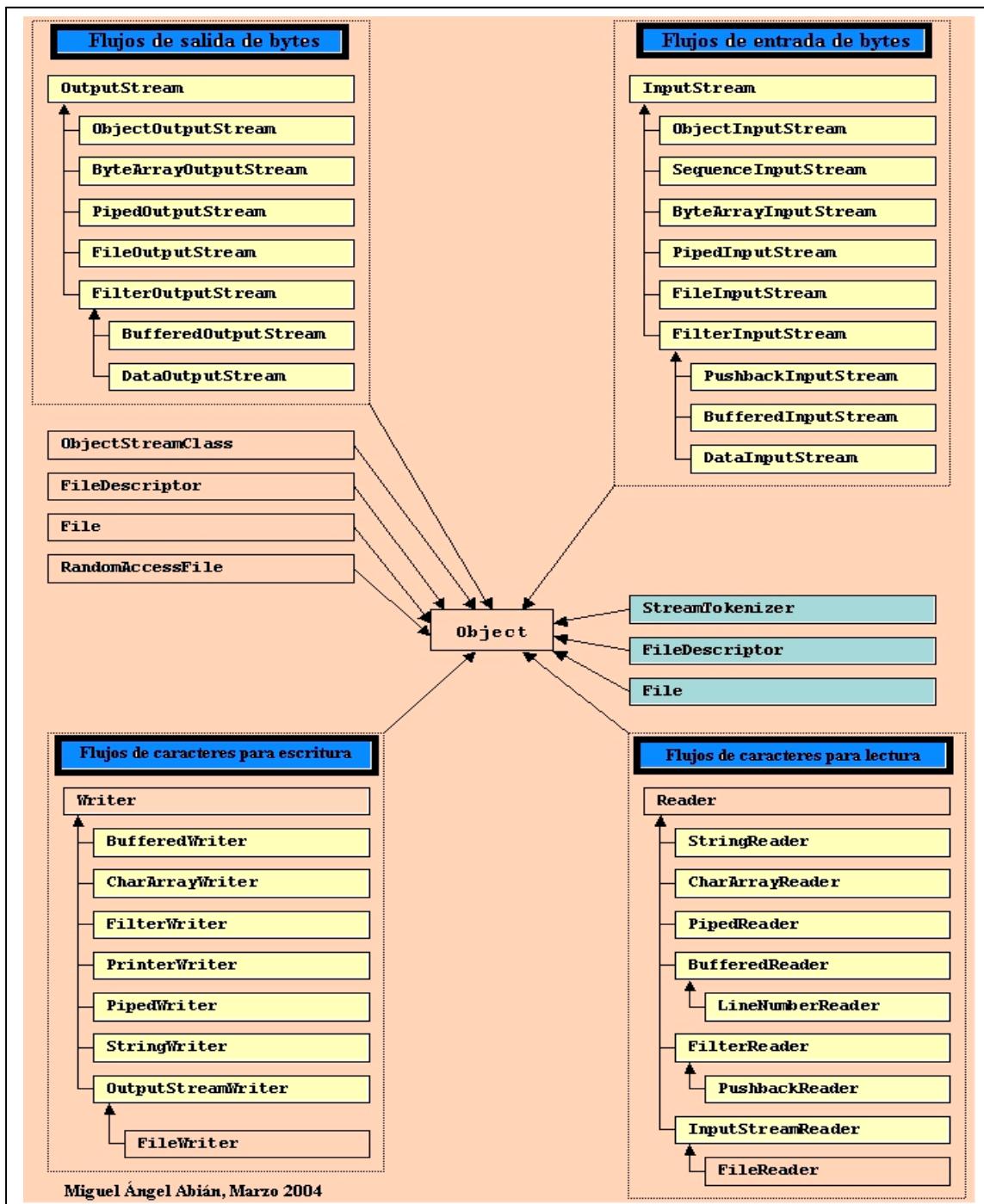


Figura 48. El paquete `java.io`

Antes de abordar ninguna clase, conviene entender cómo procesa Java la entrada y la salida: trabaja con **flujos** o **corrientes** (*streams*), que son secuencias ordenadas de bytes de longitud indeterminada. Los bytes pueden representar caracteres, cadenas o cualquier otro tipo de datos, ya sea primitivo o definido por el usuario.

En Java, todo objeto del que podamos leer una secuencia de bytes es un flujo de entrada (*input stream*); asimismo, cualquier objeto en que podamos escribir una secuencia de bytes es un flujo de salida (*output stream*). Los flujos de entrada mueven bytes desde una fuente externa de datos a un programa Java; los de salida, de un programa Java a algún receptor externo. Existen flujos que mueven bytes entre partes de distintos programas (*pipes*), pero no se van a tratar aquí.

Los flujos de entrada y salida basados en bytes se modelan en Java mediante las clases `java.io.InputStream` y `java.io.OutputStream` (ambas son abstractas), que se explicarán en este apartado. RMI y CORBA, tecnologías que se verán en los próximos apartados, trabajan con flujos de entrada y salida mediante clases que heredan de las dos anteriores.

3.1. La clase `java.io.File`

Una instancia de la clase `File`, pese a su nombre, es una representación abstracta de una ruta de acceso para un archivo o un directorio. Que exista un objeto de esta clase no implica que exista el archivo (o directorio) correspondiente en el sistema de archivos.

Sus constructores son

```
File (String camino)
File (String padre, String hijo)
File (File padre, String hijo)
File (URI uri)
```

Ninguno de los cuatro constructores lanza excepciones: el objeto se crea, existe o no en el sistema de archivos el archivo o el directorio. Veamos algunos ejemplos de uso:

```
// Ejemplo del primer constructor
// La ruta de acceso es relativa
File archivo = new File("tmp.txt");

// Ejemplo del primer constructor
// La ruta de acceso también es relativa
File archivo = new File("/temporal/tmp.txt");

// Ejemplo del segundo constructor
// La ruta de acceso también es relativa
File archivo = new File("/temporal", "tmp.txt");

// Ejemplo del tercer constructor
// La ruta de acceso también es relativa
File archivo1 = new File ("/temporal");
File archivo2 = new File (archivo1, "tmp.txt");
```

```
// Ejemplo del primer constructor en Windows  
// La ruta de acceso es absoluta  
File archivo = new File("C:/temporal/temp.txt");
```

Los cuatro primeros ejemplos usan **rutas de acceso relativas**. Son relativas porque no bastan por sí solas para determinar dónde se encuentran los archivos a los cuales se refieren. Si en una aplicación aparece una ruta de acceso relativa, la MVJ usará el directorio actual del usuario (es decir, el directorio desde donde ejecuta dicha aplicación) para formar una ruta de acceso absoluta.

Por ejemplo, si el archivo que contiene el segundo ejemplo se encuentra –uso un sistema Windows– en C:/java/jdk15, el constructor se referirá a un archivo tmp.txt en C:/java/jdk15/temporal. En un sistema UNIX, si el archivo se encuentra en /usr/local/bin/java/jdk15, el constructor se referirá a un archivo tmp.txt en /usr/local/bin/java/jdk15/temporal.

Para conocer el directorio actual se puede usar este código:

```
System.getProperty("user.dir"));
```

Si se desea conocer la ruta absoluta de un objeto `File` `archivo`, puede usarse éste:

```
System.out.println(archivo.getAbsolutePath());
```

En Windows, un resultado típico de la línea anterior sería así:

```
C:\Documents      and      Settings\miguel      angel\Mis  
documentos\JavaHispano\ persona2.txt.
```

Java admite el uso de "/" o "\" para rutas de acceso en cualquier plataforma, lo admita ésta o no (Solaris y Mac, por ejemplo, sólo usan "/"). Aunque Java se encarga de realizar la conversión necesaria, lo más recomendable es usar `File.separatorChar` en lugar de "/" o "\"; `File.separatorChar` es una variable estática que devuelve un valor `String` que corresponde al separador de archivos en el sistema utilizado.

Advertencia: Java interpreta "\" dentro de un `String` como un carácter de escape. Por ello, si se quiere indicar la ruta de un archivo Windows situado en C:/temporal/temp.txt, será necesario usar un `String` "C:\\temporal\\\\temp.txt". También podrían usarse las siguientes opciones:

```
File archivo = new File("C:/temporal/temp.txt");  
File archivo = new File("C:" + File.separatorChar + "temporal" +  
                      File.separatorChar + "temp.txt");
```

El método `public boolean exists() throws SecurityException` permite saber si el archivo o directorio representado por un objeto `File` existe. Como un objeto `File` puede referirse a un archivo o a un directorio, se proporcionan métodos

para saber si corresponde a un archivo (public boolean isFile()) o a un directorio (public boolean isDirectory()). Cuando está enlazado a un directorio, puede obtenerse la lista de los archivos en el directorio con los métodos public String[] list() throws SecurityException o public File[] listFiles() throws SecurityException. Veamos un ejemplo:

```
File directorio = new File("TEMP");
String nombresArchivos[] = directorio.list();
File archivos[] = directorio.listFiles();
```

Si el directorio TEMP, relativo respecto al directorio donde se encuentra el archivo que llamó a la MVJ, existe, el array nombresArchivos contendrá los nombres de todos los archivos del directorio, y el array archivos contendrá todos los objetos File correspondientes a aquél.

El siguiente programa admite como argumento el nombre de un directorio y muestra, si existe, los nombres de todos sus archivos y subdirectorios (no se tratan las posibles excepciones):

Ejemplo 3: ListadoDirectorio.java

```
import java.io.*;
public class ListadoDirectorio {
    public static void main(String args[]) throws IOException {
        File directorio = new File(args[0]);
        if ( (directorio.exists()) && (directorio.isDirectory()) ) {
            String[] lista = directorio.list();
            for (int i = 0; i < lista.length; i++ )
                System.out.println(lista[i]);
        } else {
            System.out.println("El directorio no existe");
        }
    }
}
```

El siguiente programa admite como argumento en la compilación el nombre de un archivo y muestra, si existe, todas sus características (no se tratan las posibles excepciones):

Ejemplo 4: InformacionArchivo.java

```
import java.io.*;
import java.util.*;
public class InformacionArchivo {
    public static void main(String args[]) throws IOException {
```

```
File archivo = new File(args[0]);

if ( (archivo.exists()) && (archivo.isFile()) ) {
    System.out.println("Nombre: " + archivo.getName());
    System.out.println("Ruta absoluta " + archivo.getAbsolutePath());
    System.out.println("Ruta: " + archivo.getPath());
    System.out.println("Padre: " + archivo.getParent());
    System.out.println("¿Admite lectura? " + archivo.canRead());
    System.out.println("¿Admite escritura? " + archivo.canWrite());
    System.out.println("Tamaño en bytes: " + archivo.length());
    System.out.println("Fecha de la última modificación: " +
        new Date(archivo.lastModified()));
} else
    System.out.println("No existe ningún archivo con esa ruta");
}

}
```

En un ejercicio de metaprogramación, cuando pido al programa que me diga cuáles son sus características (en mi sistema, con javac C:/InformacionArchivo.java) obtengo esto:

```
Nombre: InformacionArchivo.java
Ruta absoluta c:\InformacionArchivo.java
Ruta: c:\InformacionArchivo.java
Padre: c:\

¿Admite lectura? true
¿Admite escritura? true
Tamaño en bytes: 1187
Fecha de la última modificación: Sun Apr 18 12:42:16 CEST 2004
Exit code: 0
```

La clase `File` no sólo es una representación abstracta de un archivo o de un directorio: permite crear y eliminar tanto archivos como directorios (siempre que se cuenten con los permisos necesarios). El método `public boolean createNewFile() throws IOException, SecurityException` permite crear un nuevo archivo vacío (sólo si no existía antes). Si el método devuelve el valor `true` es que el archivo fue creado. Este método apenas suele usarse; en su lugar, se prefiere usar clases como `FileOutputStream` o `FileWriter`, que se verán más adelante. El método `public boolean mkdir() throws SecurityException` permite crear el directorio representado por un objeto `File`; devuelve `true` si pudo crearse. Por último, el método `public boolean delete() throws SecurityException` puede usarse para borrar un archivo o un directorio; si es un directorio, debe estar vacío para que la operación resulte exitosa.

Veamos ahora un ejemplo en el que se crea un archivo en un directorio especificado; si el directorio no existe, se crea. El camino del directorio y del archivo se pasan como argumentos en la compilación; si no hay argumentos, se toman `DIR` y `ARCHIVO` como caminos por defecto.

Ejemplo 5: CrearDirectorioYArchivo.java

```
import java.io.*;  
  
public class CrearDirectorioYArchivo {  
  
    private static String DIR = "c:/temporal"; // directorio en formato Windows  
    private static String ARCHIVO = "temporal.txt";  
  
    private static void crearArchivoYDirectorio (String dir, String archivo) {  
        boolean dirCreado = false;  
        File temp = new File(dir);  
  
        if ( (temp.exists()) && (temp.isDirectory()) ) {  
            crearArchivo(dir, archivo);  
        } else {  
            dirCreado = crearDirectorio(dir);  
            if (dirCreado) {  
                crearArchivo(dir, archivo);  
            }  
        }  
    }  
  
    private static boolean crearDirectorio (String dir) {  
        boolean dirCreado = false;  
  
        dirCreado = (new File(dir)).mkdir();  
        if (dirCreado) {  
            System.out.println("Directorio creado.");  
            return true;  
        } else {  
            System.out.println("No pudo crearse el directorio.");  
            return false;  
        }  
    }  
  
    private static void crearArchivo(String dir, String archivo) {  
        boolean archCreado = false;  
  
        try {  
            archCreado = new File(dir, archivo).createNewFile();  
        }  
        catch (IOException e) {  
            System.out.println("No pudo crearse el archivo.");  
        }  
        if (archCreado) {  
            System.out.println("Archivo creado.");  
        } else {  
            System.out.println("No pudo crearse el archivo porque ya existe.");  
        }  
    }  
  
    public static void main(String args[]) throws Exception {  
        // Si no se le dan argumentos, toma valores por defecto  
    }  
}
```

```
    if ((args.length == 2) && ( (args[0] != "") && (args[1] != "") ) ) {
        crearArchivoYDirectorio(args[0], args[1]);
    } else {
        crearArchivoYDirectorio(DIR, ARCHIVO);
    }
}
```

File también nos ofrece el método public URL toURL() throws IOException, que convierte el camino representado por el objeto File en un objeto **java.net.URL** (el paquete **java.net** se verá en apartado 8). Podemos usar el archivo creado antes para ver su URL:

```
System.out.println(archivo.toURL().toString());
```

En mi sistema, el resultado es la cadena file:/C:/temporal/tmp.txt.

3.2. La jerarquía de clases **java.io.InputStream**

La superclase raíz **java.io.InputStream** proporciona los métodos básicos para leer bytes de un flujo de entrada basado en bytes:

```
public abstract int read() throws IOException
public int read(byte[] datos) throws IOException
public int read(byte[] datos, int offset, int longitud)
throws IOException
```

El primer método lee un byte sin signo del flujo de entrada y devuelve el valor int del byte sin signo. En el caso de que no haya más datos que leer porque se ha alcanzado el final del flujo, devuelve -1.

El segundo intenta leer del flujo de entrada los bytes suficientes para llenar el array **datos** y devuelve el número de bytes leídos. Si encuentra el final del flujo, devuelve -1.

El tercero lee hasta **longitud** bytes del flujo de entrada y los almacena en el array **datos**, comenzando en la posición indicada por el entero **offset**. Devuelve el número de bytes leídos, ó -1 si se encuentra ante un final de flujo.

Los tres métodos son bloqueantes, pues bloquean la E/S del programa hasta que se dé una de estas tres situaciones: a) disponibilidad de datos de lectura; b) conclusión del flujo; c) lanzamiento de una excepción. Este carácter bloqueante tendrá importantes consecuencias para desarrollar programas con el paquete **java.net**.

El método public void close() throws IOException se encarga de cerrar el flujo de entrada y de liberar los recursos del sistema usados por el flujo. Esta clase proporciona un método public int available() throws IOException que devuelve el número de bytes que pueden leerse sin bloqueo del flujo de entrada.

Veamos un ejemplo muy sencillo del funcionamiento de la clase `InputStream` (se ha omitido el tratamiento de las excepciones):

Ejemplo 6: Entrada.java

```
import java.io.*;  
  
public class Entrada {  
  
    public static void main (String args[]) throws IOException {  
        int byteLeido;  
        InputStream entrada = (System.in); // System.in es una referencia a un  
                                         // objeto InputStream definido por el sistema  
        while (true) {  
            byteLeido = entrada.read();  
            if (byteLeido == -1) {  
                System.out.println("Se leyó todo el flujo. FIN DEL PROGRAMA.");  
                System.exit(0);  
            }  
            char caracter = (char) byteLeido;  
            System.out.print(caracter);  
            System.out.print(" ");  
        }  
    }  
}
```

Nota: Cuando se dice que una clase –o un método– utiliza objetos `InputStream` (que es una clase abstracta), no quiere decirse que use una instancia de la clase (por definición, una clase abstracta no es instanciable); sino que usa instancias de las subclases no abstractas de `InputStream`. Un método que tenga como argumento un `InputStream` aceptará cualquier objeto que sea instancia de una subclase no abstracta de `InputStream`. Una clase como `InputStream` (u `OutputStream`) proporciona una interfaz común para todas sus subclases.

El programa lee los datos de la entrada, y usa los valores `int` devueltos para obtener los caracteres de la entrada, que se presentan separados por espacios en blanco. Java ofrece un objeto `InputStream` (cuya referencia es `System.in`) ya definido y abierto por el sistema y que representa a un flujo de entrada que viene por el teclado. Este código es tremadamente ineficaz, pues lee byte a byte; podría mejorarse la eficiencia usando el método `public int read(byte[] datos)`, lo que permitiría leer de golpe un número máximo de bytes igual al tamaño del array `datos`; pero `java.io` ofrece mejores soluciones, tal como veremos enseguida.

Advertencia: Pensando en el rendimiento, podría pensarse que se puede usar el método `avaliable()` en combinación con el uso de un array de bytes, de un modo similar a éste:

```
// entrada es un objeto del tipo InputStream
int capacidad = entrada.avaliable();
if (capacidad > 0) {
    byte datos[] = new byte(capacidad);
    entrada.read(datos);
}
```

Aunque la documentación de Sun para esta clase no lo explica, usar código similar al anterior no es conveniente. Para algunas MVJ y algunos flujos de E/S, `avaliable()` siempre devuelve cero. Uno puede llevarse la desagradable sorpresa de comprobar que el código de E/S que funciona bien en una plataforma falla estrepitosamente en otra.

Antes de continuar con la clase `InputStream`, me adelanto a una objeción que se me puede hacer, resumible en dos preguntas: **¿qué tienen que ver clases como ésta con las comunicaciones en red?**, **¿por qué se explican aquí?** Se me puede decir, con toda razón, que al fin y al cabo son las clases de E/S de toda la vida (al menos, de la vida del lenguaje Java). Sí, así es: no hay más cera que la que arde; pero Java no necesita más cera para alumbrar las tenebrosas catacumbas de las comunicaciones en red. Tal como escribí ya en la introducción, se cumple que el proceso de transferir información a través de redes es tan sencillo –o tan difícil– como leer ficheros o escribir en ellos. Recibir datos a través de una red (sea Internet, una extranet o una intranet) apenas difiere de leerlos de un archivo sito en la máquina local; lo mismo vale para enviarlos. Así pues, no me parece mala idea hacer un rápido repaso de las clases más usadas de la E/S estándar de Java.

Como prueba fidedigna de lo dicho, reproduzco a renglón seguido la versión en red, cliente-servidor, de la clase `Entrada`. La clase `OutputStream` se verá en el siguiente subapartado; pero por su nombre se puede intuir que es la complementaria de `InputStream`; `ServerSocket` y `Socket` se verán en el apartado 8.

Ejemplo 7a: EntradaRed.java

```
import java.io.*;
import java.net.*;

public class EntradaRed {
    //Versión para red de la clase Entrada

    public static void main (String args[]) throws IOException {
        int byteLeido;
        InputStream entrada = (System.in);
        Socket socket = new Socket("localhost", 9000);
        OutputStream salida = socket.getOutputStream();

        while (true) {
            byteLeido = entrada.read();
```

```
        if (byteLeido == -1) {
            System.out.println("Se leyó todo el flujo. FIN DEL PROGRAMA.");
            System.exit(0);
        }
        salida.write(byteLeido);
    }
}
```

Téngase muy en cuenta la advertencia de la página 111

Ejemplo 7b: ServidorRed.java

```
import java.io.*;
import java.net.*;

public class ServidorRed {
    //Servidor para la aplicación EntradaRed

    public static void main(String args[]) throws IOException {
        int byteLeido;
        ServerSocket socketServidor = new ServerSocket(9000);
        Socket socketCliente = socketServidor.accept();
        InputStream entrada = socketCliente.getInputStream();
        while (true) {
            byteLeido = entrada.read();
            if (byteLeido == -1) {
                System.out.println("Se leyó todo el flujo. FIN DEL PROGRAMA.");
                System.exit(0);
            }
            char caracter = (char) byteLeido;
            System.out.print(caracter);
            System.out.print(" ");
        }
    }
}
```

En esta aplicación, el texto escrito por el cliente se manda al servidor, que lo muestra por pantalla. Si se desea ejecutar el cliente en una máquina distinta de aquella donde se ejecuta el servidor, habrá que modificar la línea

```
Socket socket = new Socket("localhost", 9000);
```

y escribir, en lugar de localhost, el nombre de la máquina donde se ejecute la aplicación de servidor. Exceptuando el uso de sockets, el código de E/S es idéntico al que aparecía en la clase Entrada. Por regla general, las clases del paquete `java.net` ofrecen al programador un `InputStream` o un `OutputStream` para poder enviar o recibir datos. En aplicaciones más sofisticadas que la anterior no se utilizan directamente las clases `InputStream` y `OutputStream`, pues se limitan a permitir la lectura y escritura de bytes. Como veremos en este apartado, hay muchas más clases en `java.io` que permiten “envolver” a `InputStream` o a `OutputStream` y ofrecer más posibilidades al programador.

```

1 package javahispano2;
2
3 // Author: Miguel Ángel Abián
4
5 import java.io.*;
6 import java.net.*;
7
8 public class EntradaRed {
9     //Versión para red de la clase Entrada
10
11    public static void main (String args[]) throws IOException {
12        int byteLeido;
13        InputStream entrada = (System.in);
14        Socket socket = new Socket("localhost", 9000);
15        OutputStream salida = socket.getOutputStream();
16
17        while (true) {
18            byteLeido = entrada.read();
19            if (byteLeido == -1) {
20                System.out.println("Se leyó todo el flujo. FIN DEL PROGRAMA.");
21                System.exit(0);
22            }
23        }
24    }
25}

```

Messages

F:\JBuilderX\jdk1.4\bin\javaw -classpath "C:\Documents and Settings\miguel angel\jbproject\javaHispano2\classes;I
Enviando mensaje al servidor

Figura 49. El cliente del ejemplo 7

```

1 package javahispano2;
2
3 // Author: Miguel Ángel Abián
4
5 import java.io.*;
6 import java.net.*;
7
8 public class ServidorRed {
9     //Servidor para la aplicación EntradaRed
10
11    public static void main(String args[]) throws IOException {
12        int byteLeido;
13        ServerSocket socketServidor = new ServerSocket(9000);
14        Socket socketCliente = socketServidor.accept();
15        InputStream entrada = socketCliente.getInputStream();
16
17        while (true) {
18            byteLeido = entrada.read();
19            if (byteLeido == -1) {
20                System.out.println("Se leyó todo el flujo. FIN DEL PROGRAMA.");
21                System.exit(0);
22            }
23        }
24    }
25}

```

Messages

F:\JBuilderX\jdk1.4\bin\javaw -classpath "C:\Documents and Settings\miguel angel\jbproject\javaHispano2\classes;I
Enviando mensaje al servidor

Figura 50. El servidor del ejemplo 7

Advertencia: El código de la parte a) del ejemplo 7 sólo debe tenerse en cuenta como ejemplo. Ninguna aplicación que vaya a funcionar en el mundo real debería considerar el envío de caracteres (o de bytes) individuales. Enviar un solo carácter cada vez comporta mandar un datagrama completo a través de la red para cada carácter, lo cual es tremadamente ineficaz. En estas ocasiones, el uso de *buffers* se hace imprescindible.

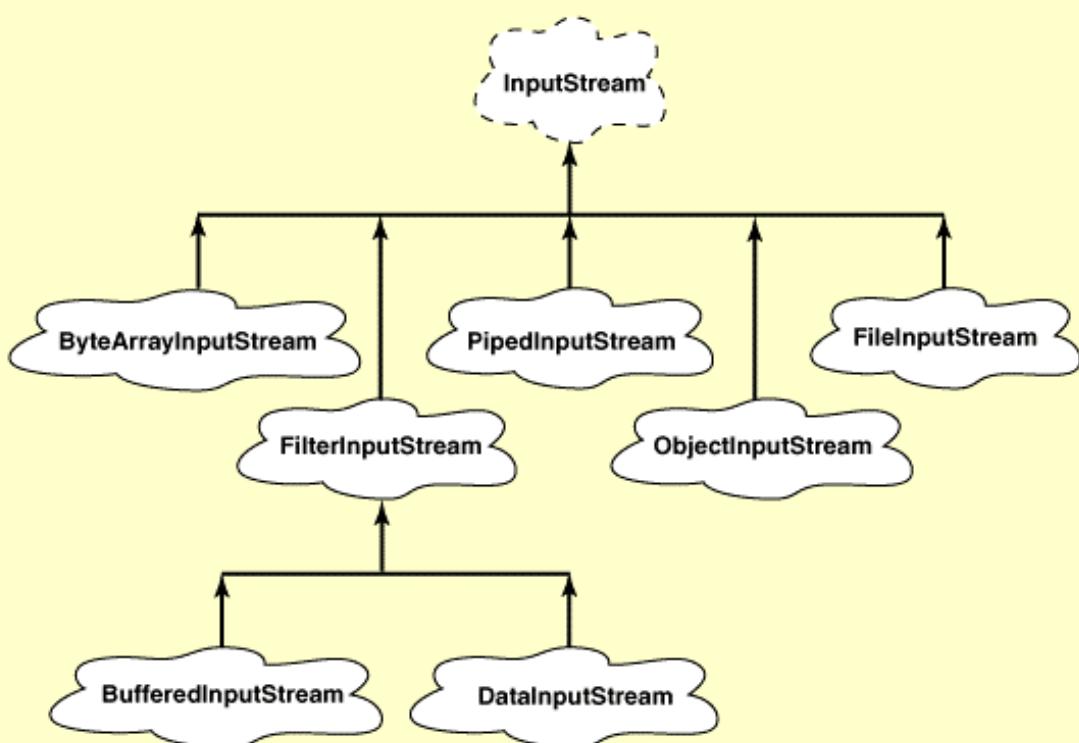


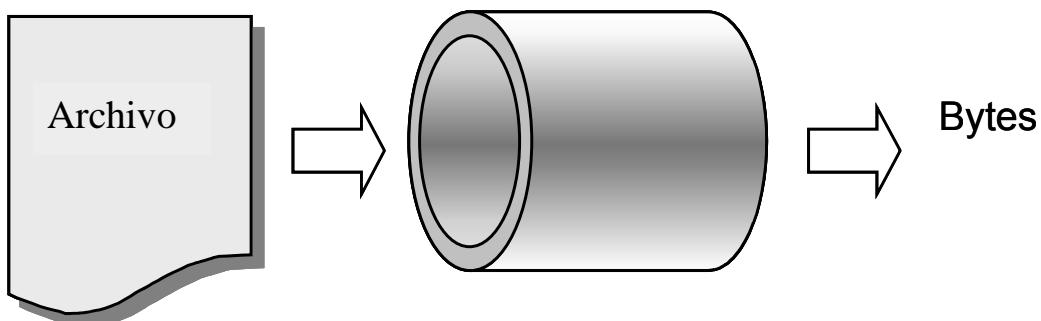
Figura 51. La jerarquía de clases `InputStream`

En la jerarquía de clases de la figura aparecen varias clases (faltan algunas). Las más importantes son `java.io.FileInputStream`, `java.io.ObjectInputStream` y `java.io.FilterInputStream`. La primera permite leer bytes a partir del flujo de entrada obtenido de un archivo; la segunda permite deserializar datos de tipos primitivos y objetos serializados antes usando `java.io.ObjectOutputStream`. `java.io.FilterInputStream` es una clase mucho más misteriosa que las otras: es un **decorador**. Para no interrumpir la exposición de las clases elementales de E/S, el patrón observador se explicará más adelante. Por ahora, es suficiente con saber que las subclases de `FilterInputStream` proporcionan nuevas funciones a las clases a las cuales "decoran" o "envuelven".

Las instancias de las subclases de `FilterInputStream` (en la figura faltan `java.io.LineNumberInputStream` y `java.io.PushbackInputStream`) permiten la lectura de un flujo y la escritura en otro, alterando los datos en el paso de

un flujo a otro. Pueden usarse para almacenar datos en *buffers*, leer tipos primitivos retroceder hacia atrás en un flujo, etc; además, pueden combinarse de modo que la salida de una instancia sea la entrada de otra.

LECTURA DE LOS BYTES DE UN ARCHIVO



```
FileInputStream(String file)  
read(byte []);  
close();
```

Manera más simple de
leer los bytes de un
archivo con la clase
FileInputStream

Figura 52. Lectura de un archivo mediante la clase **FileInputStream**

Un [BufferedInputStream](#) usa un array interno de almacenamiento temporal (*buffer*) para el flujo de entrada. Cuando se crea un [BufferedInputStream](#), se crea también un array interno de almacenamiento de bytes. Cada vez que se llama a un método `read()`, los bytes se leen del array de almacenamiento; según se van leyendo bytes, el array se vuelve a llenar con bytes del flujo de entrada, leídos de golpe. Así se evita tener que leer y almacenar byte a byte (y llamar cada vez a los métodos nativos del sistema operativo sobre el cual trabaja la máquina virtual Java), lo cual mejora mucho el rendimiento de la E/S. Siempre que se pueda, interesa manejar clases de E/S que trabajen con *buffers*. Analizemos, como ejemplo, el siguiente código:

```
BufferedInputStream bis = new  
    BufferedInputStream(objetoInputStream, 1024)  
bis.read()
```

Con el constructor, se ha creado un *buffer* de 1024 bytes de tamaño. Al llamar por primera vez a `read()`, se intentará llenar por completo el buffer a partir del flujo de entrada. En las siguientes llamadas, se leerá directamente del *buffer*, que se irá llenando, cuando convenga, con los bytes del flujo de entrada.

`DataInputStream` incorpora métodos para leer bytes y arrays de bytes, al igual que `InputStream`; pero además incluye métodos para leer caracteres, `Strings`, números (enteros, de punto flotante...), etc. Todos los métodos empiezan con `read`: `readBoolean()`, `readByte()`, `readChar()`..., excepto `skipBytes()`.

```
public class java.io.DataInputStream {  
  
    // Constructor  
    public DataInputStream( InputStream is );  
  
    public final int read( byte b[] );  
    public final boolean readBoolean();  
    public final byte readByte();  
    public final char readChar();  
    public final short readShort();  
    public final long readLong();  
    public final int readInt();  
    public final float readFloat();  
    public final double readDouble();  
    public final String readLine();  
    public final String readUTF();  
    public final int skipBytes( int n );  
  
}
```

Figura 53. La interfaz de la clase DataInputStream

Un objeto `FileInputStream` (que representa un flujo de entrada que proviene de un archivo) puede ser decorado por otro `BufferedInputStream` para proporcionar la capacidad de admitir *buffers* (memorias de almacenamiento temporal) y añadir dos nuevos métodos (`public void mark(int limitelectura)` y `public void reset()`). Al crear un objeto de esta clase, la máquina virtual Java (MVJ) intenta abrir el archivo; en el caso de que no exista, se lanza una excepción `FileNotFoundException`. Si no se puede acceder al archivo por motivos de seguridad, se arroja una excepción `SecurityException`.

En el subapartado dedicado a la clase `OutputStream` se pondrán ejemplos del uso de los objetos `ObjectInputStream`.

Advertencia: Cuando se cierra mediante el método `close()` un objeto que envuelve a otros, se producen dos acciones:

- El primer objeto envía al objeto que envuelve la información que pudiera tener almacenada.
- Se llama al método `close()` del objeto envuelto.

A su vez, si el segundo objeto envuelve a otros, se continúa el proceso arriba descrito hasta que se llega al último.

Veamos qué sucede cuando se ejecuta el siguiente código:

```
File archivo = new File("temporal.txt");
FileOutputStream fos = new FileOutputStream(archivo);
BufferedOutputStream bos = new BufferedOutputStream(fis);
... // Operaciones de escritura
bos.close();
```

Al llegar a la última línea, se produce la siguiente secuencia de acciones:

- Cualquier dato que aún esté en el objeto `bos` se envía a `fos`.
- Se llama al método `close()` de `fos`.
- Cualquier dato que aún permanezca en `fos` se envía a `archivo`, donde se escribirá.
- Se cierra `fos`.
- Se cierra `bos`.

Si en lugar de llamar primero al método `close()` de `bos` se llamara al `close()` de `archivo`, los datos que todavía quedaran en `fos` o `bos` (por el uso de *buffers* internos) no podrían escribirse ya en el archivo, pues el objeto que hace referencia a él se habría cerrado antes.

Por lo tanto, **es recomendable cerrar siempre sólo el objeto más decorado** (el último o más externo).

A continuación se expone un programa de ejemplo del uso combinado de `BufferedInputStream` y `FileInputStream`, el cual permite leer un archivo de texto (situado en el directorio donde se ejecuta la clase) y mostrar su contenido en pantalla:

Ejemplo 8: LeerCaracter.java

```
import java.io.*;
public class LeerCaracter {
    public static void main(String[] args) {
```

```
BufferedInputStream entrada = null;

try {
    // Si el archivo de texto temporal.txt no existe hay que crearlo
    // previamente o cambiar el camino para que se refiera a un archivo
    // existente.
    File archivo = new File("temporal.txt");
    entrada = new BufferedInputStream(new FileInputStream(archivo));
    while (true) {
        int temp = entrada.read();
        if (temp == -1) {
            System.out.println("");
            System.out.println("Se leyó todo el archivo.");
            break;
        }
        char caracter = (char) temp;
        System.out.print(caracter);
    }
}
catch(IOException e1) {
    e1.printStackTrace();
}
finally {
    try {
        entrada.close();
    }
    catch (Exception e2) {
        System.out.println("No pudo cerrarse el flujo.");
    }
}
}
```

Es necesaria una excepción distinta a *IOException*, pues se podría arrojar una excepción del tipo *NullPointerException*

Si el lector prescinde del decorador *BufferedOutputStream* y decide usar directamente el método *write()* de la clase *FileOutputStream*, notará –cuando use valores elevados de N– la diferencia de rendimiento ocasionada por no usar *buffers*.

Consejo: Es recomendable cerrar los flujos derivados de archivos (ya sean de entrada o salida) de la manera que aparece en el código anterior.

Si se cerraran de esta manera:

```
try {
    ...// Resto código
    entrada = new BufferedInputStream(
        new FileInputStream(archivo));
    ...// Código tratamiento de la E/S
    entrada.close();
}
catch (IOException e) {
    // Tratamiento del error
}
```

aparecería el problema de que podría lanzarse una excepción antes del `entrada.close()`, con lo que no se liberarían los recursos destinados al flujo. Con los ejemplos de este apartado no es importante el consumo de recursos; pero es algo que debe tenerse muy en cuenta cuando se programan aplicaciones que hacen un uso intensivo de la E/S.

Para hacer un **tratamiento exhaustivo** de cualquier posible excepción debería usarse código similar a éste (por motivos de espacio, coloco en la misma línea el corchete de cierre de `try` y la sentencia `catch` asociada):

```
try {
    File archivo = new File(...);
} catch (IOException e1) {
    ...// Se tratan las excepciones derivadas de File
}
try {
    FileInputStream fis = new FileInputStream(archivo);
} catch (IOException e2) {
    ...// Se tratan las excepciones derivadas de FileInputStream
}
try {
    BufferedInputStream entrada = new BufferedInputStream(fis);
} catch (IOException e3) {
    ...// Se tratan las excepciones derivadas de BufferedInputStream
}
try {
    ...// Código de manipulación del objeto entrada
} catch (IOException e4) {
    ...// Se tratan las excepciones del código que manipula a entrada
}
finally {
    entrada.close();
} catch (Exception e4) {
    ...// Se tratan las excepciones al intentar cerrar entrada
}
```

Así nos aseguraríamos que ningún recurso se quedara en el aire, sucediera lo que sucediera. De la manera que aparece el tratamiento de las excepciones en el listado de la página anterior, algún recurso (un objeto `File`) podría quedarse sin limpiar si no se pudiera crear un objeto `BufferedReader` (por falta de memoria, por ejemplo). Recomiendo esta práctica para cualquier aplicación que abuse de la E/S; pero no la sigo aquí porque alargaría en exceso el código de los ejemplos, sin facilitar su comprensión.

3.3. La jerarquía de clases `java.io.OutputStream`

La superclase raíz `java.io.OutputStream` proporciona los métodos básicos para escribir bytes en un flujo de salida basado en bytes (todos ellos son bloqueantes):

```
public abstract void write(int byte) throws IOException  
public void write(byte[] datos) throws IOException  
public void write(byte[] datos, int offset, int longitud)  
throws IOException
```

El primer método escribe un único byte en un flujo de salida. El segundo escribe un array de bytes, y el tercero escribe `longitud` bytes del array `datos`, comenzando por la posición indicada por el entero `offset`.

Los sistemas operativos utilizan *buffers* internos para evitar tener que escribir los bytes de uno en uno. Así pueden escribir decenas o cientos de bytes de golpe, lo cual redonda en un mejor rendimiento del sistema. Esta clase dispone de un método (`public void flush() throws IOException`) que obliga a escribir todos los bytes que haya en el *buffer*, esté lleno o no. El tamaño exacto de cada *buffer* depende del sistema operativo usado y de la implementación de la máquina virtual Java.

`OutputStream` también dispone de un método `public void close() throws IOException`, que se encarga de cerrar el flujo de salida y de liberar los recursos del sistema usados por el flujo.

`System.out` (el flujo de entrada estándar) y `System.err` (el flujo de salida estándar de los errores) son objetos `OutputStream`. Más específicamente, son objetos `PrintStream`.

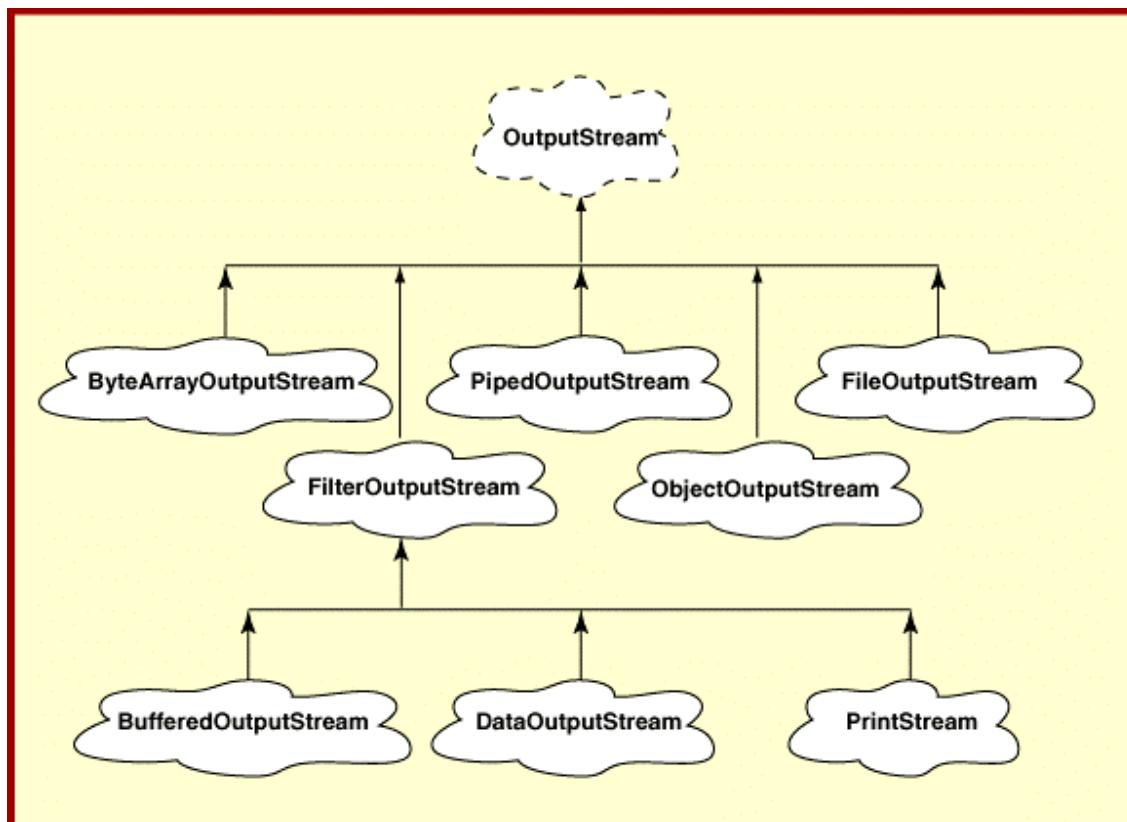


Figura 54. La jerarquía de clases `OutputStream`

En la jerarquía de clases derivadas de la superclase base `OutputStream` que aparece en la figura aparecen varias clases (faltan algunas). Las más importantes son `java.io.FileOutputStream`, `java.io.ObjectOutputStream` y `java.io.FilterOutputStream`. La primera permite escribir bytes en el flujo de salida asociado a un archivo; la segunda permite serializar datos de tipos primitivos y objetos. Como era de esperar, `FilterOutputStream` es un decorador. Las instancias de las subclases de `FilterOutputStream` (en la figura faltan `java.io.LineNumberOutputStream` y `java.io.PushbackOutputStream`) permiten decorar los objetos a los que envuelven.

`DataOutputStream` incorpora métodos para escribir bytes y arrays de bytes, al igual que `OutputStream`; pero además incluye métodos para escribir caracteres, `Strings`, números (enteros, de punto flotante...), etc. Todos los métodos empiezan con `write`: `writeBoolean()`, `writeByte()`, `writeChar()`..., excepto `flush()`.

```
public class java.io.DataOutputStream {  
  
    // Constructor  
    public DataOutputStream( OutputStream os );  
  
    public void flush();  
    public void write( byte b[], int off, int len);  
    public final void writeBoolean( boolean v );  
    public final void writeByte( int v );  
    public final void writeBytes( String s );  
    public final void writeChar( int v );  
    public final void writeChars( String s );  
    public final void writeShort( int v );  
    public final void writeLong( long v );  
    public final void writeInt( int v );  
    public final void writeFloat( float v );  
    public final void writeDouble( double v );  
    public final void writeUTF( String str );  
  
}
```

Figura 55. La interfaz de la clase DataInputStream

`FileOutputStream` complementa a `FileInputStream`. Con respecto a esta última, presenta una diferencia: sus constructores, a diferencia de los de `FileInputStream`, no arrojan excepciones del tipo `FileNotFoundException`; si el archivo no existe, `FileOutputStream` lo crea. En caso de que el archivo sí exista y se utilice el constructor que aparece en la siguiente línea de código:

```
FileOutputStream fos = new FileOutputStream(fichero);
```

hay que tener en cuenta que, con la primera llamada a `write()`, los nuevos datos se escribirán sobre los que ya tenía el archivo, con su consiguiente pérdida.

Si se desea que los nuevos datos se añadan tras los ya existentes, se deberá usar este constructor:

```
FileOutputStream fos = new FileOutputStream(fichero, true);
```

La clase `PrintStream` incluye varios métodos `public void print()` y `public void println()` para imprimir (en el sentido de mostrar al usuario por la salida estándar) cualquier valor de un objeto o un tipo primitivo.

Los métodos `print()` convierten el argumento en un `String` y luego lo transforman en bytes de acuerdo con la codificación por defecto del sistema; después estos bytes se escriben del modo descrito para los métodos `write()`.

Los métodos `println()` hacen exactamente lo mismo que los `print()`, pero incluyen un carácter de nueva línea ("`\n`", "`r`" o "`r\n`"; depende de la plataforma).

Los objetos `PrintStream` presentan una curiosa propiedad con respecto a las demás clases de E/S: no lanzan excepciones. El motivo para este comportamiento reside, probablemente, en que –tal como ya se dijo–, `System.out` (el flujo de entrada estándar) y `System.err` (el flujo de salida estándar de los errores) son objetos `PrintStream`. Sería bastante incómodo para un programador tener que escribir código para manejar las excepciones cada vez que escriba líneas inofensivas como `System.out.println("JAVA")`. Que esta clase no lance excepciones no quiere decir que no las pueda producir; lo que ocurre es que las gestiona internamente. Para comprobar si se ha producido algún error se llama al método `public boolean checkError()`.

Un objeto `BufferedOutputStream` puede decorar a un `OutputStream`, dándole la posibilidad de que los bytes leídos se almacenen en un *buffer* y se escriban cuando el *buffer* esté lleno (salvo que se use `flush()`, que fuerza a escribir, esté o no lleno). Su funcionamiento no difiere mucho del correspondiente a la clase `BufferedInputStream`, vista en el subapartado anterior: las llamadas a `write()` van almacenando los datos en el *buffer*, que sólo se escribirá cuando esté lleno (o se llame a `flush()`). A continuación, se muestra un ejemplo del uso combinado de las clases `BufferedOutputStream` y `FileOutputStream`, el cual escribe N veces las letras *A* y *B* en un archivo.

Ejemplo 9: EscribirCaracter.java

```
import java.io.*;

public class EscribirCaracter {

    private final static int N=200; // Número de escrituras.

    public static void main(String[] args) {
        // Se crea un archivo y se escribe en él N veces la letra A.
        // Luego, se escribe N veces la letra B.
        BufferedOutputStream salida = null;
```

```

try { // Si el archivo no existe, lo crea (si puede).
    File archivo = new File("temporal.txt");
    salida = new BufferedOutputStream(new FileOutputStream(archivo));
    for (int i=0; i<N; i++) {
        salida.write( (int) 'A'); // Escribe el byte correspondiente al carácter 'A'
    }
    salida.flush(); // Se fuerza a vaciar el buffer
    salida.close(); // Se cierra el flujo de salida
    // Ahora se añade N veces la letra B; usando el constructor con true se
    // se consigue que se añada después de las letras A, sin eliminarlas
    salida = new BufferedOutputStream(new FileOutputStream(archivo, true));
    salida.write ( (int) 't'); //Se añade un tabulador
    for (int i=0; i<N; i++) {
        salida.write( (int) 'B'); // Escribe el byte correspondiente al carácter 'B'
    }
    salida.flush();
}
catch(IOException e1) {
    e1.printStackTrace();
}
finally {
    try {
        salida.close();
    }
    catch (Exception e2) {
        System.out.println("No se pudo cerrar el flujo");
    }
} // fin del bloque try-catch-finally
}

}

```

La clase ObjectOutputStream es una clase muy importante para las comunicaciones en red: permite **serializar objetos** (ObjectInputStream permite deserializarlos). Serializar es la acción de codificar un objeto en un flujo de bytes; deserializar es la acción de decodificar un flujo de bytes para reconstruir una copia del objeto original. Esencialmente, serializar un objeto equivale a guardar (y luego poder cargar) el estado de un objeto. La serialización es un mecanismo de implementación de la persistencia de objetos. Uso **persistencia** (de un objeto) en el sentido de capacidad de un objeto para persistir **en el tiempo y en el espacio**, independientemente de la MVJ que lo creó. El flujo de bytes que produce la serialización de un objeto se puede enviar a máquinas remotas mediante sockets o se puede guardar en un archivo. Para poder reconstruir correctamente un objeto, el proceso de serialización también almacena en el flujo de bytes la descripción de la clase a la que pertenece el objeto.

Bruce Eckel explica así la serialización en Java (*Thinking in Java 3rd Edition*):

La serialización de objetos en Java le permite tomar cualquier objeto que implemente la interfaz Serializable y convertirlo en una secuencia de bytes que puede restaurarse completamente más tarde para regenerar el objeto original. Esto es cierto incluso a través de una red, lo que significa que el mecanismo de la serialización compensa automáticamente las diferencias entre sistemas operativos. Esto es, puede crear un objeto en una máquina Windows, serializarlo y enviarlo a través de la red a una máquina Unix donde

será reconstruido correctamente. No tiene que preocuparse sobre las representaciones de los datos en las diferentes máquinas, el orden de los bytes o cualquier otro detalle.

Por sí misma, la serialización de objetos es interesante porque le permite implementar persistencia *ligera*. Recuerde que la persistencia significa que el tiempo de vida de un objeto no está determinado por si un programa se está ejecutando; el objeto vive *entre* las invocaciones del programa. Tomando un objeto serializable y escribiéndolo en el disco, y entonces restaurando ese objeto cuando el programa es reinvocado, puede producir el efecto de persistencia. El motivo por el que se llama “ligera” es que no puede simplemente definir un objeto y usar alguna clase de palabra reservada “persistent” y dejar que el sistema se preocupe de los detalles (aunque esto podría ocurrir en el futuro). En lugar de eso, debe explícitamente serializar y deserializar los objetos en su programa.

Una característica sumamente interesante de la serialización estriba en su capacidad de congelar objetos vinculados y de hacer que retornen a su estado original, aunque la máquina de destino esté a miles de kilómetros de la maquina donde se crearon originalmente los objetos. Cuando se serializa un objeto, se serializan también todos los objetos a los que tenga referencias (los cuales deben, por tanto, implementar también la interfaz `Serializable`); como es lógico, al deserializarlo se reconstruyen también los objetos vinculados. Esta propiedad abre posibilidades muy interesantes para los programadores: mareas enteras de objetos interconectados de muchas maneras pueden almacenarse y recuperarse cuando se necesiten. Si se intentará simular la serialización de un objeto mediante la clase `DataOutputStream`, se precisaría guardar cada dato de tipos simples (`int`, `double`, `float...`) contenido en el objeto, así como los datos de tipos simples que contuvieran los objetos a los cuales contiene referencias. Se puede trabajar así, pero sería tarea muy tediosa y propensa a errores.

Cuando se deserializa un flujo de bytes, se llama al método `protected Class resolveClass(ObjectStreamClass descripcion) throws IOException, ClassNotFoundException` de la clase `ObjectInputStream` para que cargue dinámicamente los *bytecodes* de las clases cuyas descripciones encuentra en el flujo (suponiendo que no los hubiera cargado antes). Este método llama al cargador de clases de Java, el cual es el verdadero encargado de buscar los archivos `.class` necesarios y de cargar dinámicamente sus *bytecodes*. El cargador de clases busca, en primer lugar, en el directorio actual (aquel desde el cual se ejecutó la aplicación), y si no los encuentra sigue buscando en los directorios indicados en el `CLASSPATH`. Si finalmente no encuentra los `.class` que se necesitan, lanza una excepción `java.lang.ClassNotFoundException`. Siempre interesa configurar el `CLASSPATH` local para que incluya todos los directorios donde estén los archivos `.class` de las clases que se necesitarán durante el proceso de deserialización, pues no es habitual que estén todos en el directorio desde el cual se lanza la aplicación.

La relación entre la serialización de objetos y la RMI (*Remote Method Invocation*: ejecución remota de métodos) se detallará en el apartado 5.

La mejor manera de ver cómo funciona la serialización es mediante un ejemplo completo. En él veremos cómo se graban objetos Persona en un archivo llamado personas.txt (si no existe, se crea), ubicado en el directorio donde se ejecuta EscribirPersona.

Ejemplo 10a: EscribirPersona.java

```
import java.io.*;

public class EscribirPersona {

    public static void main(String args[]) {
        FileOutputStream archivo = null;
        ObjectOutputStream salida = null;

        Persona p1= new Persona("Louis Ferdinand Céline", 67);
        Persona p2= new Persona("André Breton", 69);
        Persona p3= new Persona("André Malraux", 71);
        try {
            archivo = new FileOutputStream("personas.txt");
            salida = new ObjectOutputStream(archivo);
            salida.writeObject(p1);
            salida.writeObject(p2);
            salida.writeObject(p3);
            salida.flush();
        }
        catch (IOException e1) {
            System.out.println("Imposible crear el archivo o escribir en él.");
            e1.printStackTrace();
        }
        finally {
            try {
                salida.close();
            }
            catch (Exception e2) {
                System.out.println("No pudo cerrarse el flujo.");
            }
        } // fin del bloque try-catch-finally
    }

    class Persona implements Serializable {

        private String nombre;
        private int edad;

        public Persona (String nombre, int edad) {
            this.nombre = nombre;
            this.edad = edad;
        }
    }
}
```

```
public String toString() {
    return ("Nombre: " + nombre +"; Edad: " + edad);
}

}
```

Persona implementa la interfaz Serializable porque, tal como se dijo, cualquier objeto que pueda ser serializado (y deserializado) debe implementarla.

Ejemplo 10b: LeerPersona.java

```
import java.io.*;

public class LeerPersona {

    public static void main(String args[]) {
        FileInputStream archivo = null;
        ObjectInputStream entrada = null;
        try {
            archivo = new FileInputStream("personas.txt");
            entrada = new ObjectInputStream(archivo);
            Object tmp = entrada.readObject();
            while (tmp != null) {
                Persona persona = (Persona) tmp;
                System.out.println (persona.toString());
                tmp = entrada.readObject();
            }
        }
        catch (EOFException e1) {
            System.out.println("Archivo leído");
        }
        catch (Exception e2) {
            System.out.println("Imposible abrir el archivo o leer de él.");
            e2.printStackTrace();
        }
        finally {
            try {
                entrada.close();
            }
            catch (Exception e3) {
                System.out.println("No pudo cerrarse el flujo");
            }
        } //fin del bloque try-catch-finally
    }
}
```

Para que el código funcione correctamente, se necesita ejecutar LeerPersona desde el mismo directorio donde se ejecuta EscribirPersona o configurar el CLASSPATH para que LeerPersona tenga acceso al archivo Persona.class (generado al compilar la clase EscribirPersona), y después de ejecutar esta última al menos una vez. Hecho esto, veremos que los escritores han resucitado.

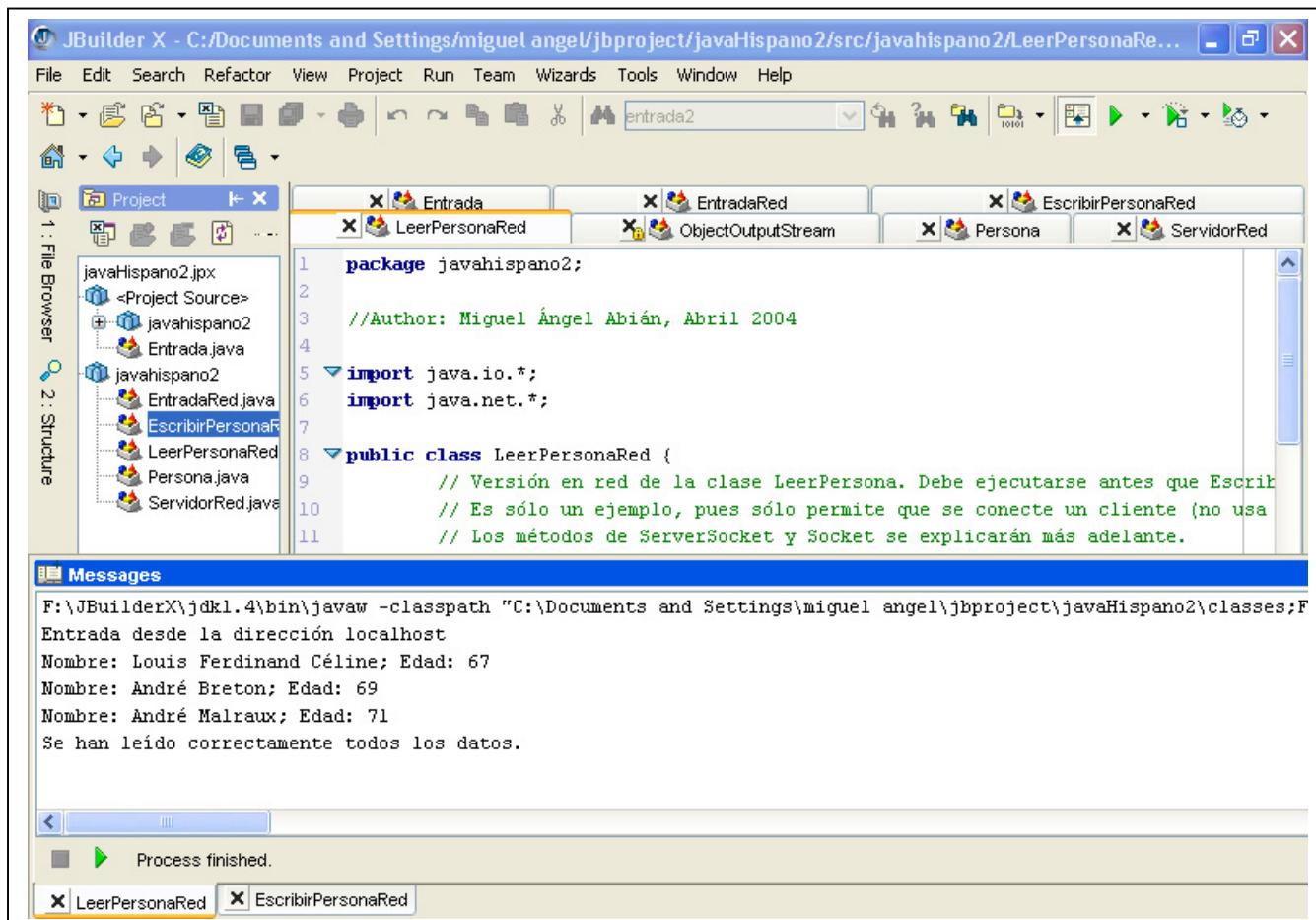


Figura 56. El ejemplo 10 en funcionamiento

No hay ningún obstáculo para serializar a través de redes. A continuación incluyó el código de la versión de red correspondiente al ejemplo anterior para que el lector vea su forma. En el apartado 8 se explicarán los métodos correspondientes a clases del paquete **java.net**.

Tal y como está escrito, se considera que el cliente y el servidor se ejecutan en la misma máquina. Si se desea ejecutar **EscribirPersonaRed** en una máquina distinta de aquella donde se ejecuta **LeerPersonaRed**, habrá que modificar la línea

```
Socket socket = new Socket("localhost", 9000);
```

y escribir, en lugar de **localhost**, el nombre de la máquina donde se vaya a ejecutar **LeerPersonaRed**. Además, será necesario colocar el archivo **Persona.class** en esta última máquina de manera que **LeerPersonaRed** pueda acceder a ella cuando se ejecute.

Ejemplo 11a: EscribirPersonaRed.java

```
import java.io.*;
import java.net.*;

public class EscribirPersonaRed {
    // Versión en red de la clase EscribirPersona.
    // Los métodos de ServerSocket y Socket se explicarán más adelante.

    public static void main(String args[]) {
        OutputStream salida = null;
        ObjectOutputStream salidaObjeto = null;
        Socket socket = null;

        Persona p1= new Persona("Louis Ferdinand Céline", 67);
        Persona p2= new Persona("André Breton", 69);
        Persona p3= new Persona("André Malraux", 71);
        try {
            socket = new Socket("localhost", 9000);
            salida = socket.getOutputStream();
        }
        catch (IOException e1) {
            System.out.println("No fue posible establecer la conexión con el servidor." +
                               "Error fatal.");
            System.exit(-1);
        }
        try {
            salidaObjeto = new ObjectOutputStream(salida);
            salidaObjeto.writeObject(p1);
            salidaObjeto.writeObject(p2);
            salidaObjeto.writeObject(p3);
            salidaObjeto.flush();
        }
        catch (IOException e2) {
            System.out.println("Imposible escribir los objetos en el servidor.");
            e2.printStackTrace();
        }
        finally {
            try {
                salidaObjeto.close();
                socket.close();
            }
            catch (Exception e3) {
                System.out.println("No pudo cerrarse el flujo o el socket, o ninguno.");
            }
        } // fin del bloque try-catch-finally
    }
}
```

Ejemplo 11b: Persona.java

```
import java.io.*;  
  
class Persona implements Serializable {  
    // Esta clase tiene que estar en el mismo directorio que EscribirPersonaRed y  
    // LeerPersonaRed o debe configurarse el CLASSPATH para incluirla.  
  
    private String nombre;  
    private int edad;  
  
    public Persona (String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    public String toString() {  
        return ("Nombre: " + nombre +"; Edad: " + edad);  
    }  
  
}
```

Ejemplo 11c: LeerPersonaRed.java

```
import java.io.*;  
import java.net.*;  
  
public class LeerPersonaRed {  
    // Versión en red de la clase LeerPersona. Debe ejecutarse antes que EscribirPersonaRed.  
    // Es un mero ejemplo, pues sólo permite que se conecte un cliente (no usa hilos).  
    // Los métodos de ServerSocket y Socket se explicarán más adelante.  
  
    public static void main(String args[]) {  
        InputStream entrada = null;  
        ObjectInputStream entradaObjeto = null;  
        ServerSocket socketServidor = null;  
        Socket socketCliente = null;  
  
        try {  
            socketServidor = new ServerSocket(9000);  
            socketCliente = socketServidor.accept();  
            entrada = socketCliente.getInputStream();  
            System.out.println("Entrada desde la dirección " + socketCliente.getInetAddress());  
        }  
        catch (IOException e1) {  
            System.out.println("No fue posible arrancar el servidor. Error fatal.");  
            System.exit(-1);  
        }  
        try {  
            entradaObjeto = new ObjectInputStream(entrada);  
            Object tmp = entradaObjeto.readObject();  
            while (tmp != null) {  
                System.out.println(tmp);  
            }  
        }  
    }  
}
```

```
        Persona persona = (Persona) tmp;
        System.out.println (persona.toString());
        tmp = entradaObjeto.readObject();
    }
}
catch (EOFException e2) {
    System.out.println("Se han leído correctamente todos los datos.");
}
catch (Exception e3) {
    System.out.println("Imposible leer del cliente.");
    e3.printStackTrace();
}
finally {
    try {
        entradaObjeto.close();
        socketCliente.close();
        socketServidor.close();
    }
    catch (Exception e4) {
        System.out.println("No pudo cerrarse el flujo o el socket, o ninguno.");
    }
} //fin del bloque try-catch-finally
}

}
```

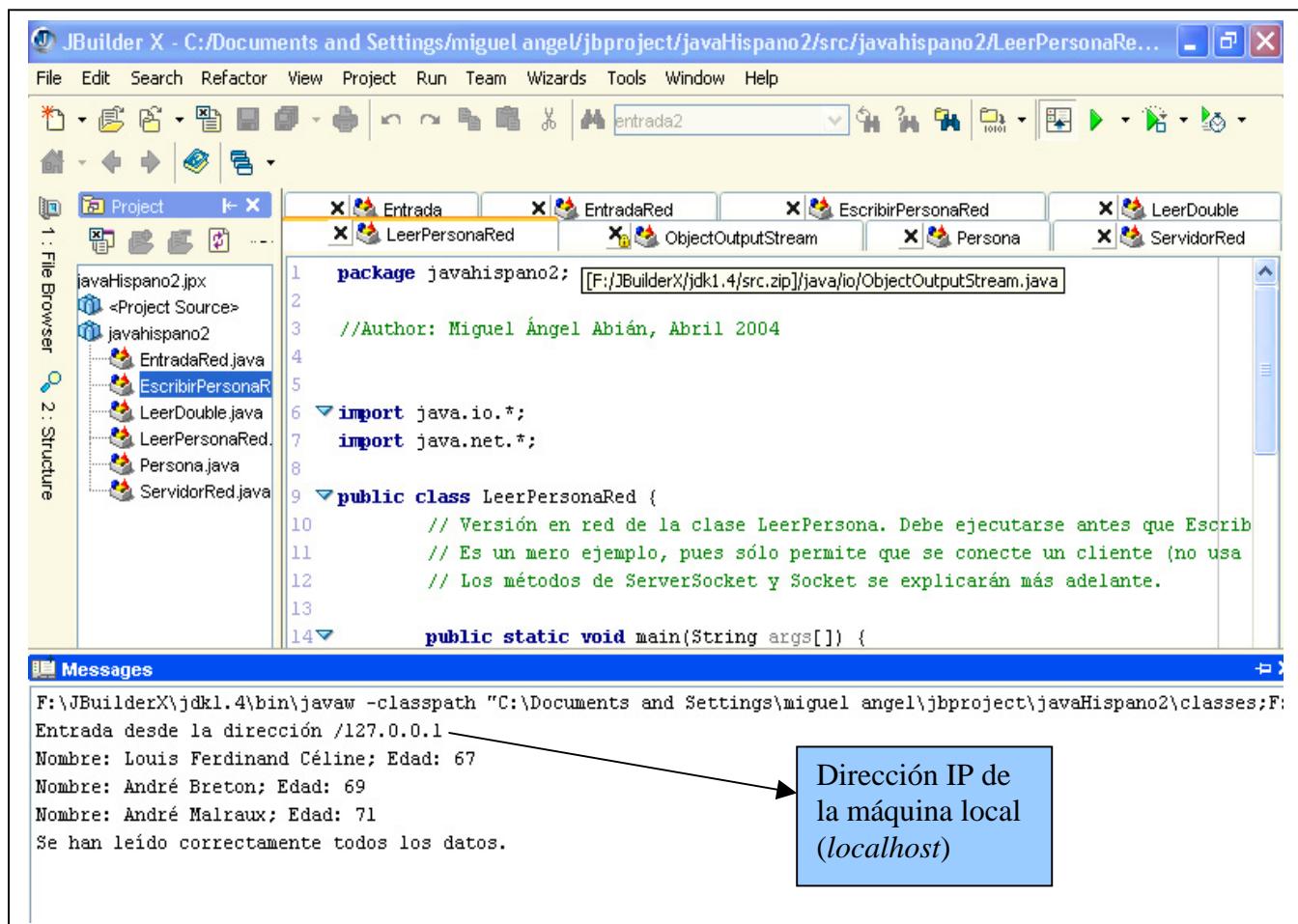


Figura 57. El ejemplo 11 en funcionamiento

Otra vez han vuelto a resucitar los escritores, de una manera que nunca hubieran imaginado.

Como resulta muy útil la propiedad de serializar a la vez conjuntos de objetos vinculados, creo pertinente incluir un ejemplo, muy sencillo, en el cual se serializa un objeto `HashMap` y luego se recuperan sus elementos, que son de tipos distintos.

Ejemplo 12a: SerializarHashMap.java

```
import java.io.*;
import java.util.*;

public class SerializarHashMap {

    private static Map hashmap = null;
    private static FileOutputStream archivo = null;
    private static ObjectOutputStream salida = null;
    // Objetos con los que se rellena la lista
    private static Persona p1 = null;
    private static Persona p2 = null;
    private static Persona p3 = null;
    private static String s1 = null;
    private static String s2 = null;
    private static Integer i1 = null;
    private static Double d1 = null;

    public static void inicializarElementos() {
        p1= new Persona("Louis Ferdinand Céline", 67);
        p2= new Persona("André Breton", 69);
        p3= new Persona("André Malraux", 71);
        s1 = "Yo no soy escritor";
        s2 = "Se acabó este conjunto tan heterogéneo";
        i1 = new Integer(100);
        d1 = new Double (3456.3879563);
    }

    public static void cargarElementos() {
        hashmap = new HashMap();

        hashmap.put("uno", p1);
        hashmap.put("dos", p2);
        hashmap.put("tres", p3);
        hashmap.put("cuatro", s1);
        hashmap.put("cinco", i1);
        hashmap.put("seis", d1);
        hashmap.put("siete", s2);
    }

    public static void serializar() {
        try {
            archivo = new FileOutputStream("hashmapserializado.txt");
        }
    }
}
```

```
salida = new ObjectOutputStream(archivo);
salida.writeObject(hashmap);
salida.flush();
}
catch (IOException e1) {
    System.out.println("Imposible crear el archivo o escribir en él.");
    e1.printStackTrace();
}
finally {
    try {
        salida.close();
    }
    catch (Exception e2) {
        System.out.println("No pudo cerrarse el flujo.");
    }
} // fin del bloque try-catch-finally
}

public static void main(String args[]) {
    inicializarElementos();
    cargarElementos();
    serializar();
}
}

class Persona implements Serializable {

    private String nombre;
    private int edad;

    public Persona (String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public String toString() {
        return ("Nombre: " + nombre + "; Edad: " + edad);
    }
}
```

Ejemplo 12b: DeserializarHashMap.java

```
import java.io.*;
import java.util.*;

public class DeserializarHashMap {
    // Esta clase debe estar en el mismo directorio que SerializarHashMap.java
    // y debe ejecutarse después de ella.

    private static Map hashmap = null;
    private static FileInputStream fis = null;
    private static ObjectInputStream entrada = null;

    public static void deserializar() {
        try {
            fis = new FileInputStream("hashmapserializado.txt");
            entrada = new ObjectInputStream(fis);
            hashmap = (HashMap) entrada.readObject();
            System.out.println(hashmap.toString());
        }
        catch (IOException e1) {
            System.out.println("Imposible abrir el archivo o leer de él.");
            e1.printStackTrace();
        }
        catch (ClassNotFoundException e2) {
            System.out.println("Imposible convertir a un HashMap.");
            e2.printStackTrace();
        }
        finally {
            try {
                entrada.close();
            }
            catch (Exception e3) {
                System.out.println("No pudo cerrarse el flujo.");
            }
        } // fin del bloque try-catch-finally
    }

    public static void main(String args[]) {
        deserializar();
    }
}
```

3.4. Cuando un byte no basta: comunicaciones en un mundo plurilingüe.

Las dos superclases raíz vistas hasta ahora y sus subclasses trabajan directamente con bytes. A menudo se precisa utilizar caracteres en lugar de bytes. Cuando un carácter corresponde a un byte, es decir, cuando se almacena un carácter en un solo byte, leer bytes corresponde a leer caracteres. Eso ocurre, por ejemplo en las codificaciones US-ASCII e ISO Latin-1. Sin embargo, cuando un carácter requiere más de un byte para ser almacenado (como sucede con los caracteres asiáticos, por ejemplo), las clases anteriores son inútiles: desconocen cómo codificar o decodificar caracteres que ocupen más de un byte. Todos los métodos de las clases anteriores que manipulan texto a partir de flujos de bytes asumen una codificación ISO Latin 1 (US-ASCII es un subconjunto suyo).

Las subclasses de las clases abstractas `java.io.Reader` y `java.io.Writer` permiten trabajar directamente con flujos de caracteres Unicode, esto es, con flujos de datos basados en caracteres Unicode.

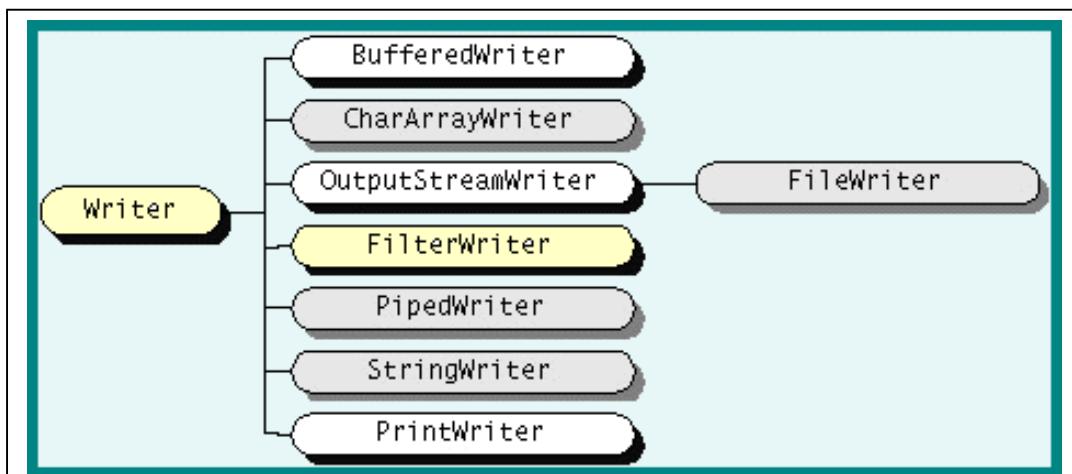


Figura 58. La jerarquía de clases `Writer`. Extraída de la documentación oficial de Sun

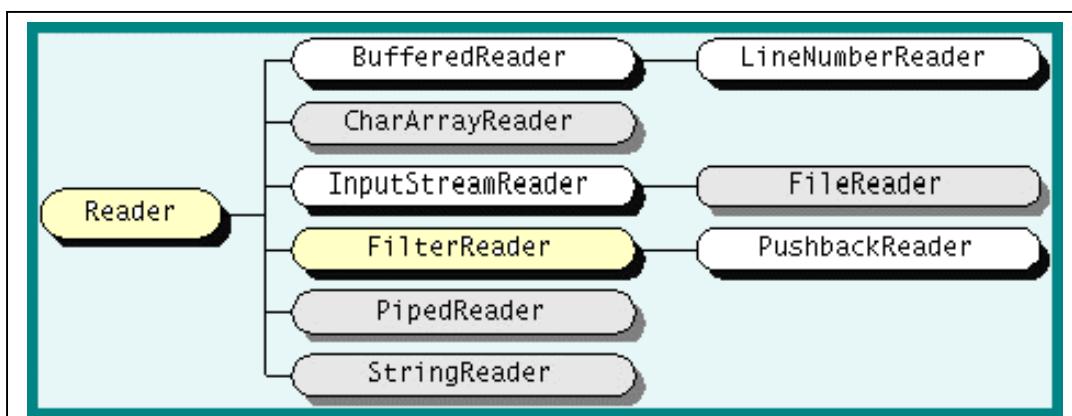


Figura 59. La jerarquía de clases `Reader`. Extraída de la documentación oficial de Sun

Nota: Las clases **Reader** y **Writer**, así como sus subclases, se introdujeron en el JDK 1.1 (no en el 1.2). Algunas de las clases de Java que se introdujeron en la versión 1.0 se han quedado obsoletas (oficialmente, *depredated*, perdón, *deprecated*). Así, la clase **LineNumberReader**. ¿Qué sentido tiene una clase que asocia un número de línea a cada línea de texto cuando se dispone de clases que manejan directamente líneas de caracteres? Algo similar ocurre con las clases **PrintStream** y **StringBufferInputStream**.

Nota: El lector que no esté familiarizado con los métodos para internacionalizar aplicaciones deberá tener en cuenta el sentido en que uso estos términos en el tutorial:

- **Carácter:** Es la unidad mínima atómica de texto. Todo texto está compuesto por una cadena de caracteres.
- **Conjunto, juego o repertorio de caracteres:** Conjunto de caracteres.
- **Conjunto de caracteres codificados:** Es un conjunto ordenado de caracteres en el que se asigna a cada carácter un número entero no negativo. Los enteros no negativos se conocen como *puntos de código*.
- **Codificación de caracteres:** Es el sistema por el cual los caracteres de un repertorio de caracteres se representan de forma binaria en un archivo o en memoria.
- **Charset:** Conjunto de caracteres que se ha codificado mediante una codificación de caracteres.

Conviene tener en cuenta que un carácter no es una cadena de ceros y unos: es un concepto teórico. Cualquier persona considera que la letra Z de la fuente Times New Roman y la letra Z de la fuente Comic Sans MS corresponden al carácter zeta, si bien sus representaciones en bytes son completamente diferentes y sus representaciones gráficas no coinciden exactamente.

Un carácter tampoco corresponde necesariamente a una letra: existen caracteres sin correspondencia con letras (los caracteres LF y ESC en US-ASCII, por ejemplo), pues existen caracteres de control, que se reservan para tareas de control (marcar el fin de una línea, del proceso, etc.). Incluso puede suceder que distintos caracteres correspondan a una misma letra.

La primera codificación oficial de caracteres para ordenadores fue la codificación US-ASCII. Es una codificación de 7 bits, aunque se use un byte u octeto para representar cada carácter. De los ocho bits que forman un octeto, se usa uno como dígito de control (por ejemplo, para comprobar posibles errores en las comunicaciones). Esta codificación permite sólo 128 caracteres (de 0 a 127), suficientes para el inglés; pero insuficientes para el español o el alemán, y completamente insuficientes para idiomas como chino, coreano o japonés. Incluso codificaciones que usan un byte completo para cada carácter (como la ISO 8859-1 o ISO Latin 1) son incapaces de representar alfabetos asiáticos, hebreos o cirílicos.

ISO 8859-1

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2	sp	!	"	#	\$	%	&	,	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	y	v	w	x	y	z	{		}	~	DE L
8																
9																
A	^{NBS} P	í	¢	£	¤	¥	¦	§	„	©	ª	«	¬	-	®	-
B	°	±	²	³	‘	µ	¶	·	,	¹	º	»	¼	½	¾	¸
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Figura 60. Juego de caracteres ISO 8859-1 (inglés, alemán, francés, italiano...)

Los juegos de caracteres ISO 8859

ISO 8859-1	Latin-1	Lenguas de Europa occidental (español, italiano, francés, italiano, etc.)
ISO 8859-2	Latin-2	Lenguas de Europa oriental (polaco, checo, húngaro, etc.)
ISO 8859-3	Latin-3	Lenguajes del sur de Europa (turco y maltés)
ISO 8859-4	Latin-4	Lenguajes del norte de europa (lituano, etc.)
ISO 8859-5	Cirílico	Ruso, ucraniano, búlgaro, serbio, etc.
ISO 8859-6	Árabe	Árabe
ISO 8859-7	Griego	Griego
ISO 8859-8	Hebreo	Hebreo
ISO 8859-9	Latin-5	Turco (Substituto de Latin-3)
ISO 8859-10	Latin-6	Lenguas del norte de Europa (unifica Latin-1 y Latin-4)
ISO 8859-11	Tailandés	Tailandés
ISO 8859-13	Latin-7	Lenguajes bálticos (substituto de Latin-4)
ISO 8859-14	Latin-8	Lenguas célticas
ISO 8859-15	Latin-9	Lenguas de Europa occidental (substituto de Latin-1)
ISO 8859-16	Latin-10	Lenguas de Europa oriental (substituto de Latin-2)

Figura 61. Juegos de caracteres ISO 8859-1

El estándar Unicode (la última versión es la 4.0) añadió más variedad lingüística a las comunicaciones mediante ordenadores. El tutorial de Java de Sun afirma erróneamente que "Unicode es una codificación de 16 bits que incluye los lenguajes más extendidos del mundo". Por bien que suene, está francamente **equivocado**. Hoy día, Unicode es un estándar cuya meta es asignar un número entero no negativo (punto de código) a cada carácter de cada lenguaje humano escrito que exista o haya existido. Por consiguiente, no constituye una codificación (en el sentido definido hace dos páginas) ni un *charset*. Asimismo, **no especifica obligatoriamente** ninguna representación binaria de los caracteres ni asigna dos bytes a cada carácter, aunque sí define algunas codificaciones *posibles* (y optativas) para los caracteres Unicode.

Este estándar sólo recopila los caracteres existentes en casi todas las lenguas usadas por los bípedos implumes del tercer planeta de un pequeño sistema solar y asigna a cada carácter un número entero único; además, reserva un cierto número de puntos de código para futuros nuevos caracteres. Por ahora, Unicode reserva 1114112 ($2^{20} + 2^{16}$) enteros no negativos o puntos de código; de ellos, sólo unos 96.000 tienen asignados caracteres: el resto permanece vacío, a la espera de la introducción de nuevos caracteres.

Desde luego, Unicode no obliga a ninguna representación computacional de los caracteres o de los números que los representan. Tomemos como ejemplo la cadena "Hello". En Unicode, esta cadena se representa así:

U+0048 U+0065 U+006C U+006C U+006F

Cada carácter se ha representado con un número entero no negativo (que aquí aparece en hexadecimal); pero no hay ningún tipo de representación asociada a un ordenador. Lo único que hace Unicode es asociar un entero no negativo a cada carácter.

Unicode fue diseñado de manera que sus 256 primeros puntos de código coinciden exactamente con los de ISO 8859-1 o ISO Latin 1 (es decir, un punto de código X en Unicode tiene asociado el mismo carácter que el punto de código X en ISO Latin 1). Como los 128 primeros puntos de código de ISO Latin 1 coinciden con los de US-ASCII, los 128 puntos de código de Unicode también coinciden con los de US-ASCII.

Java usa una codificación de tipo UTF-16 (*Universal character set Transformation Format*). Las codificaciones más comunes de Unicode son UTF-8, UTF-16 y UTF-32 (las tres se definen en el estándar Unicode). Echemos un rápido vistazo a sus características:

- **UTF-8:** Es la codificación Unicode más usada. En ella, cada carácter se almacena en uno, dos, tres o cuatro bytes. Los caracteres US-ASCII, por ejemplo, comunes en las lenguas de Europa occidental, se almacenan en un solo byte (los caracteres en las posiciones 0-127 se corresponden –en el mismo orden– con los caracteres US-ASCII). Los datos en archivos suelen almacenarse según esta codificación.
- **UTF-16:** Almacena la mayor parte de los caracteres en dos bytes, si bien se usan cuatro bytes para algunos caracteres muy poco comunes (como algunos caracteres chinos tradicionales, por ejemplo)
- **UTF-32:** Usa cuatro bytes para almacenar cada carácter. Esta manera es la más simple posible de almacenar caracteres Unicode. El principal inconveniente radica en el espacio: un carácter representable mediante un byte con US-ASCII o ISO Latin-1 ocupará siempre cuatro bytes en esta codificación.

Las dos primeras codificaciones pueden usar de uno a cuatro bytes por carácter, dependiendo del carácter; la última siempre usa cuatro bytes, sea cual sea el carácter codificado.

La codificación que usa Java para representar internamente los caracteres Unicode (presentes en datos del tipo `char` y `String`) se llama UCS2. Es una variante de UTF-16, en la que se han eliminado algunos caracteres (los que precisan más de dos bytes) para que los restantes se puedan almacenar en dos bytes.

Aunque la internacionalización es una materia muy interesante, no me propongo perderme en ella. Con lo expuesto es suficiente para entender el porqué de las clases `java.io.Reader` y `java.io.Writer`. Como ya se mencionó, las subclases derivadas de `java.io.InputStream` y `java.io.OutputStream` asumen que un carácter puede almacenarse en un byte u octeto. Este occidental planteamiento sólo resulta correcto para el juego de caracteres ISO Latin 1 (estándar ISO 8859-1), que incluye los caracteres correspondientes a muchas lenguas europeas (español, francés, alemán, etc., pero no para caracteres asiáticos, árabes, cirílicos, hebreos o griegos).

En resumen, las clases Reader y Writer proporcionan métodos similares a los de InputStream y OutputStream; pero orientados a caracteres, en lugar de a bytes. Especificando la codificación deseada, estas clases saben en qué caracteres deben convertir los bytes, y viceversa.

Para concretar ideas, supongamos que se quiere leer un archivo grabado usando la codificación ISO 8859-5 (que incluye el alfabeto cirílico) y que se desea grabarlo, mediante Internet, en un sistema ruso que trabaja con UTF-8. El código necesario se muestra aquí:

```
// Se lee del archivo de entrada
InputStreamReader entradaArchivo = new InputStreamReader(new
    FileInputStream("archivo.txt"), "ISO-8859-5");
BufferedReader entrada = new BufferedReader(entradaArchivo);
String linea = entrada.readLine(); // Se lee la primera línea
//del archivo de entrada
... // Se procesa la entrada.
... // Se escribe en el archivo de salida
OutputStreamWriter salidaArchivo = new OutputStreamWriter(
    new FileOutputStream("ruso.txt"), "UTF-8");
BufferedWriter salida = new BufferedWriter(salidaArchivo);
// Se escribe la primera linea en el archivo de salida
salida.write(linea);
...
```

Se codifica la
salida

Se decodifica
la entrada

Lo que hace exactamente las clases no resulta relevante para el ejemplo (se explicarán más adelante); pero sí la estructura del código:

- Lectura y decodificación
- Codificación y escritura

Si el sistema que lee el archivo estuviera configurado como ASCII o ISO Latin 1 y no se especificara la codificación que debe usarse para la lectura, todos los caracteres que no correspondieran a caracteres ASCII o ISO Latin 1 aparecerían equivocados. En consecuencia, los datos que se grabarían en el archivo del sistema ruso carecerían de sentido.

El resultado de leer, usando la configuración por defecto de mi sistema (ISO Latin 1), un archivo con caracteres cirílicos codificado con ISO 8859-5 y de transformarlo en un archivo UTF-8 se muestra a continuación. La primera captura de pantalla corresponde al archivo original; la segunda, al transformado.

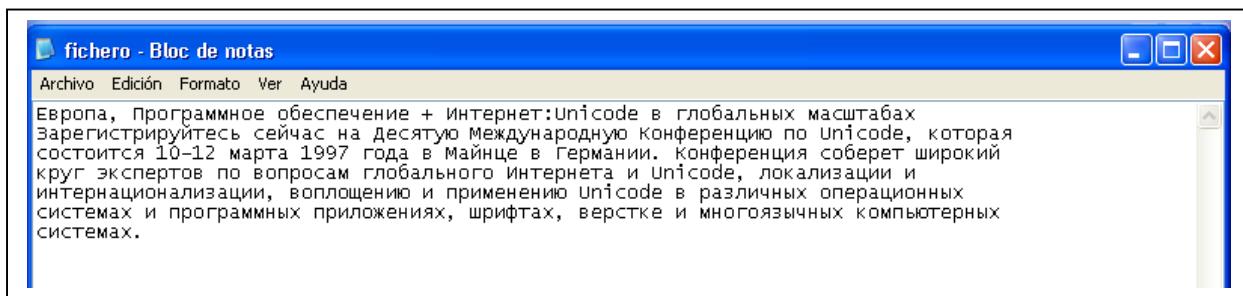


Figura 62. Archivo escrito en ruso (texto sacado de una página rusa sobre Unicode)

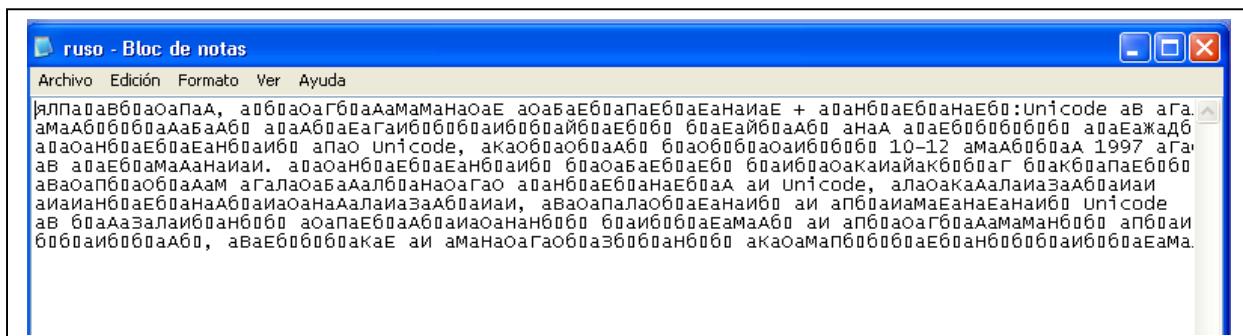


Figura 63. El archivo anterior, transformado en un galimatías.

Como puede verse, los únicos caracteres que se han conservado correctamente han sido los US-ASCII (los números y la palabra Unicode en algunas líneas). Desde luego, los angloparlantes siempre parten con ventaja: cualquier codificación respeta los caracteres US-ASCII, que permiten representar cualquier texto escrito en inglés. Dicho de otra manera: los 128 primeros caracteres de todas las codificaciones actuales siempre coinciden con los caracteres US-ASCII, en el mismo orden que tienen en el estándar norteamericano (de estos 128 caracteres, sólo aquellos con posiciones entre la 32 y la 127, ambas inclusive, corresponden a caracteres mostrables). Toda persona que trabaje siempre con información en inglés no deberá preocuparse de la configuración de su sistema: un texto en inglés siempre puede leerse correctamente, sea cual sea la codificación usada para guardar o para leerlo (US-ASCII, ISO Latin 1, UTF-8, UTF-16, etc.); lo único que varía en cada codificación es el espacio necesario para almacenarlo.

Las moralejas resultan claras:

- **Para enviar datos a través de una red o almacenarlos en un archivo hay que manejar una codificación que el receptor sepa interpretar.** De nada sirve Internet si los receptores reciben garabatos. Usar siempre la codificación por defecto del sistema es incorrecto: las aplicaciones así escritas se comportarán de distinto modo en distintos sistemas.
- **Para conseguir que un lenguaje sea respetado por todos los estándares hay que hacer los primeros basándose en ese lenguaje.** En esto de los estándares, como en tantas cosas de la vida, gana quien llega primero.

Advertencia: La codificación por defecto de un programa Java, es decir, la codificación en la que leerá y escribirá caracteres Unicode, depende de varios factores. A saber: la MVJ, el sistema operativo bajo la MVJ y los parámetros de configuración del sistema operativo.

El lector que trabaje con versiones de Java anteriores a la 1.4 debe tener en cuenta que Sun usó en ellas nombres no oficiales para casi todos las codificaciones. Es posible que para que funcionen los ejemplos sea necesario cambiar "ISO-8859-5" por "ISO8859_5", etc.

Recomiendo consultar la documentación de Java para saber cómo referirse a cada codificación.

3.5. La clase `java.io.InputStreamReader`

Esta subclase de `Reader` modela un flujo de entrada basado en bytes como un flujo de caracteres, también de entrada. Los bytes se leen de un `InputStream` y se convierten en caracteres, de acuerdo con la codificación especificada en el constructor. Proporciona métodos de lectura (`public int read()`, `public int read(char[] array, int offset, int longitud)`) que permiten leer bytes y convertirlos en caracteres, según la codificación usada (si no se especifica ninguna, se toma por defecto la que corresponde al sistema operativo). Cada vez que se llama a uno de los dos métodos de lectura, se lee uno o más octetos del flujo de bytes sobre el cual se ha construido.

Los dos constructores más usados de esta clase son

```
public InputStreamReader(InputStream entrada)
public InputStreamReader(InputStream entrada, String codificacion) throws UnsupportedEncodingException
```

Para leer caracteres se usan dos métodos `read()`:

```
public int read() throws IOException
public int read(char[] buffer, int offset, int longitud)
throws IOException
```

La siguiente línea lee la entrada estándar y la convierte en caracteres Unicode:

```
InputStreamReader entradaCaracteres = new InputStreamReader(
System.in);
```

Al haberse usado el primer constructor, se toma como codificación la usada en la máquina. En un ordenador europeo o estadounidense, lo normal es que sea Cp1252 o ISO 8859-1 (ISO Latin 1). En una máquina ubicada en China, por ejemplo, lo lógico es que se use MS936, GB18030, EUC_CN o GBK. En una máquina china cabe esperar que se manipulen archivos con datos escritos en chino mandarín y que se use un sistema operativo que permita trabajar con caracteres chinos; si se intentara usar una codificación ISO Latin 1, los bytes de los archivos (codificados para representar caracteres chinos) se intentarían convertir en caracteres europeos. Resultado: el programador recibiría, días después, comentarios como "Creo que he cometido algún error al manejar su excelente programa. ¿Sería tan amable de echarle un vistazo cuando pueda? No es urgente: habré cometido algún error" (a los chinos no occidentalizados les es casi imposible manifestar abiertamente desacuerdo con un interlocutor). El lector puede imaginarse los comentarios que sufriría el programador en cualquier país menos educado (o más franco, según se mire).

En el siguiente código se lee la entrada estándar y se convierte en caracteres Unicode, según la codificación internacional para caracteres griegos:

```
InputStreamReader entradaCaracteres = new InputStreamReader(
System.in, "ISO-8859-7");
```

Cada vez que se llame a un método `read()`, se leerán los caracteres que correspondan –según la codificación ISO 8859-7– a los bytes introducidos desde el teclado.

Otro ejemplo: si se quiere convertir un archivo escrito con la codificación ISO 8859-5 (usada para caracteres cirílicos) en caracteres Unicode, de modo que sean manipulables por los programas Java, habrá que usar código similar a éste:

```
FileInputStream archivo = new FileInputStream("ruso.txt");
InputStreamReader entradaCaracteres = new InputStreamReader(
    archivo, "ISO-8859-5");
```

Para leer el primer carácter cirílico de ruso.txt y mostrarlo en pantalla, se podría escribir esto:

```
char c = (char) entradaCaracteres.read();
System.out.println(c);
```

En un sistema con fuentes cirílicas, se mostraría el primer carácter (por ejemplo, Ђ).

Nota: Java usa internamente el formato UCS2 (un tipo de UFT-16) para almacenar cadenas de texto y caracteres; pero la capacidad de una máquina para mostrarlos reside en las fuentes que tenga instalada en su sistema operativo. Por ejemplo, si en un sistema no se han instalado fuentes chinas no se podrán mostrar correctamente caracteres chinos.

Dado un objeto `InputStreamReader`, puede obtenerse el nombre de la codificación de caracteres que se está usando mediante el método `public String getEncoding()`. Veamos un ejemplo:

```
InputStreamReader entradaCaracteres = new
    InputStreamReader(System.in);
System.out.println("Codificación " + 
    entradaCaracteres.getEncoding());
```

En mi sistema, este código muestra en pantalla la cadena "Codificación: ISO8859_1", que corresponde a ISO Latin 1.

La única subclase de `InputStreamReader` es `FileReader`, que permite leer archivos de texto usando la codificación por defecto del sistema. Esta clase proporciona una interfaz de flujos de caracteres para leer archivos de texto usando la codificación por defecto. Uno de sus constructores es éste:

```
public FileReader(String nombreArchivo) throws FileNotFoundException
```

Este constructor crea un objeto `FileReader` que lee, usando la codificación por defecto del sistema, del archivo denotado por `nombreArchivo`.

3.6. La clase `java.io.OutputStreamWriter`

Esta subclase de `Writer` modela un flujo de salida basado en bytes como un flujo de caracteres, también de salida. Proporciona métodos de escritura (`public void write();` `public void write(char[] array, int offset, int longitud);` `public void write(String cadena, int offset, int longitud)`) que permiten escribir en un `OutputStream` los bytes que resultan de codificar los caracteres, de acuerdo con la codificación especificada en el constructor (si no se especifica ninguna, se toma por defecto la del sistema).

Cada vez que se llama a uno de los tres métodos de escritura, se obtienen los bytes (o el byte) que corresponden al carácter o al grupo de caracteres que se introduce como argumento; pero los bytes no se escriben inmediatamente en el flujo de salida: permanecen en un *buffer*. El método `public void flush()` se encarga de hacer que se escriban cuando se le llama, este lleno o no el *buffer*.

Los dos constructores más usados de esta clase son

```
public OutputStreamWriter(OutputStream salida)
public OutputStreamWriter(OutputStream salida, String
codificacion) throws UnsupportedEncodingException
```

En el siguiente código:

```
OutputStreamWriter salidaCaracteres = new OutputStreamWriter(
System.out, "ASCII");
```

cada vez que se llame a un método `write()`, seguido de un `flush()`, se escribirán en el flujo estándar de salida los bytes que correspondan, según la privilegiada codificación US-ASCII, a los caracteres introducidos dentro del argumento de `write()`.

Si, por ejemplo, se quiere escribir en un archivo codificado con el juego de caracteres ISO-8859-5 (cirílico), se puede usar

```
FileOutputStream fis = new FileOutputStream("ruso.txt");
OutputStreamWriter salidaCaracteres = new OutputStreamWriter(
archivo, "ISO-8859-5");
```

Para escribir la letra rusa І en el flujo de salida se necesita este código:

```
char c = (char) 1174; // también se podría usar el código
// Unicode del carácter
salidaCaracteres.write(c);
```

Para obtener el nombre de la codificación usada en un `OutputStreamWriter`, se usa el método `public String getEncoding()`.

La única subclase de `OutputStreamWriter` es `FileWriter`, que permite escribir en archivos de texto mediante la codificación por defecto del sistema. Esta clase proporciona una interfaz de flujos de caracteres para escribir en archivos de texto mediante la codificación por defecto. Uno de sus constructores tiene esta forma:

```
public FileWriter(String nombreArchivo, boolean anyadir)  
throws IOException
```

Este constructor crea un objeto `FileWriter` que escribe en el archivo al que se refiere `nombreArchivo` usando la codificación por defecto del sistema. El parámetro `anyadir` indica si los datos deben añadirse a un archivo ya existente (`true`) o si debe eliminarse el archivo que ya existía (`false`).

3.7. La clase java.io.BufferedReader

Esta subclase de la superclase raíz Reader decora los objetos Reader añadiéndoles la posibilidad de usar buffers, lo que mejora la eficiencia.

Esta clase incorpora un nuevo método: public String readLine() throws IOException, que permite leer líneas de texto desde un archivo de texto.

Nota: En Java una línea de texto acaba en \r, \n o \r\n, según el sistema que se esté usando (Windows, Mac, Solaris, etc).

Siempre que se vaya a procesar el contenido de archivos de texto es conveniente usar esta clase. Para ver un ejemplo de su uso, vamos a considerar un archivo sagan.txt, ubicado en el directorio donde se ejecutará la clase LeerArchivo. Su contenido se muestra a continuación:

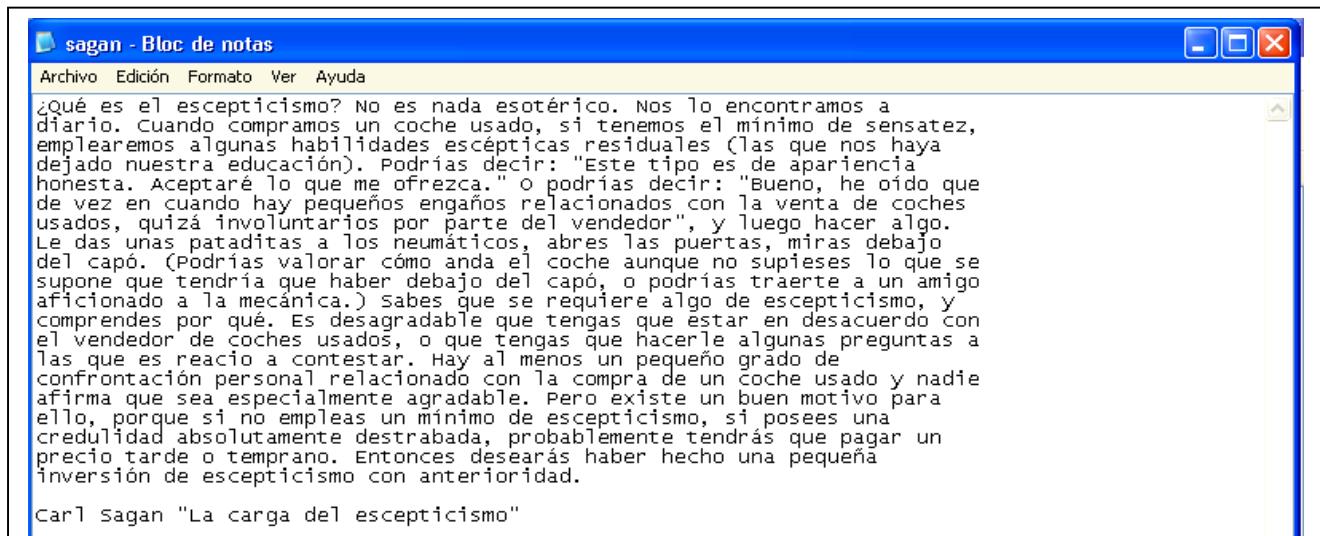


Figura 64. El archivo sagan.txt

Para mostrar por pantalla su contenido y contar las líneas que tiene, se va a usar este programa (el resultado se muestra en la figura 65):

Ejemplo 13: LeerArchivo.java

```
import java.io.*;  
  
public class LeerArchivo {  
  
    public static void main(String args[]) {  
        BufferedReader br = null;  
        String linea;  
        int contador = 0;  
  
        try {  
            br = new BufferedReader(new FileReader("sagan.txt"));  
            ...  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        ...  
    }  
}
```

```
        while ( (linea = br.readLine()) != null) {
            contador++;
            System.out.println ("Linea " + contador + ": " + linea);
        }
    }
catch (IOException e1) {
    System.out.println( "No se ha podido leer del archivo");
    e1.printStackTrace();
}
finally {
    try {
        br.close();
    }
    catch (Exception e2) {
        System.out.println("No se ha podido cerrar el flujo");
    }
} // fin del bloque try-catch-finally
}

}
```

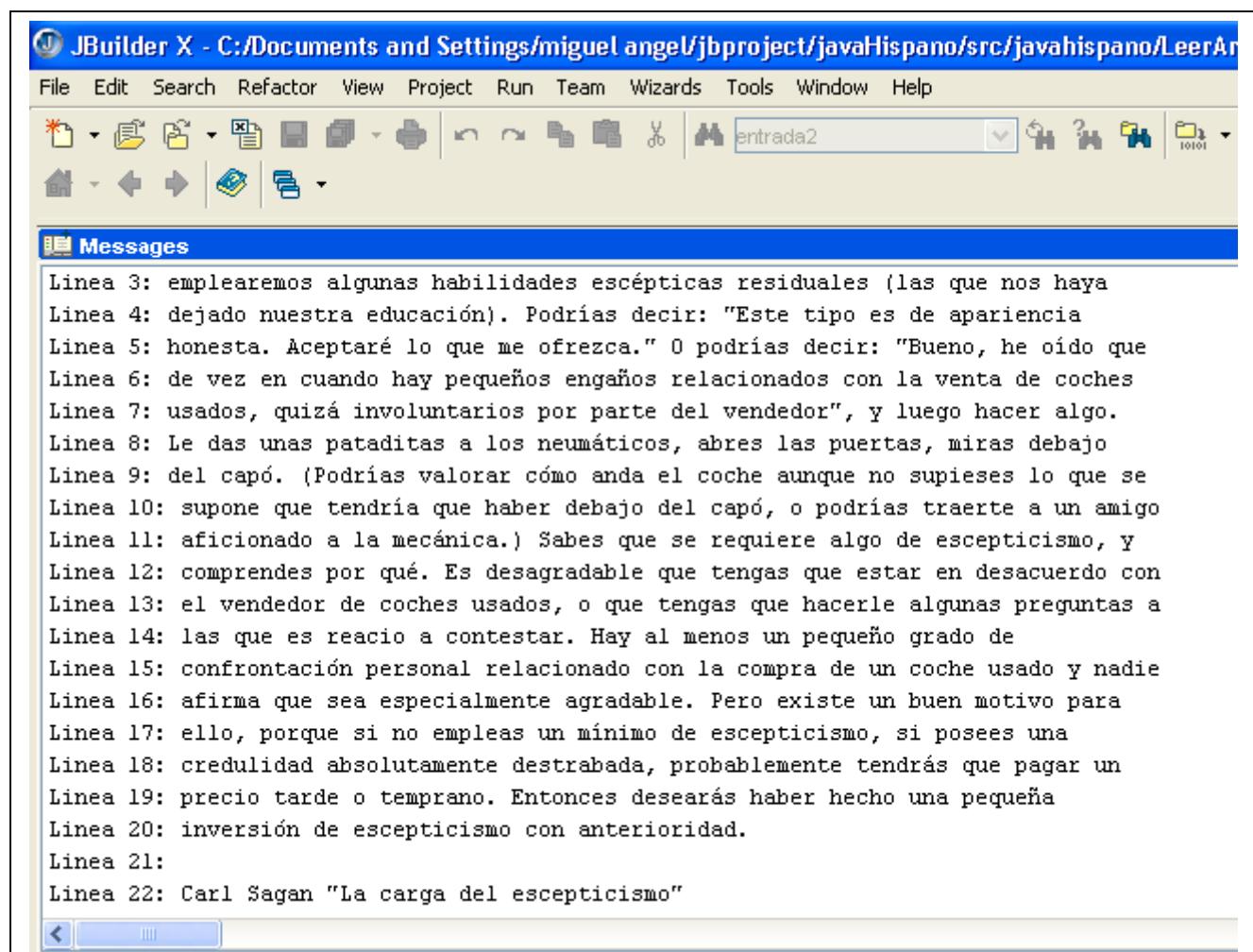


Figura 65. Salida del programa LeerArchivo

Una manera muy visual de entender las clases de E/S de Java es imaginar que son ríos (la palabra *filtro* de las clases FilterXXX sólo me hace pensar en filtros de café y en filtros pasabanda, pasabaja, etc). Un objeto como BufferedReader es como un río con un cauce más grueso que el de un objeto InputStreamReader. Así pues, el *río* InputStreamReader cabe dentro del *río* BufferedReader. Cada vez que un río es engullido dentro de otro, este último le proporciona más caudal (es decir, más funciones o funciones más especializadas), pero sigue usando el caudal del engullido (sus métodos).

3.8. La clase java.io.BufferedWriter

Esta subclase de la superclase raíz `Writer` decora los objetos `Writer` añadiéndoles la posibilidad de usar *buffers*, lo que mejora la eficiencia.

Esta clase incorpora un nuevo método: `public String newLine() throws IOException`, que permite introducir saltos de línea (en cada plataforma se traducen a lo que corresponda: `\n`, etc.)

Como ejemplo de su uso, se presenta un programa en que se emplea para insertar una línea en blanco entre cada línea del texto recogido en `sagan.txt`; el texto resultante se almacena en un archivo llamado `sagan2.txt`.

Ejemplo 14: EscribirArchivo.java

```
import java.io.*;

public class EscribirArchivo {

    public static void main(String args[]) {
        String linea = null;
        BufferedReader br = null;
        BufferedWriter bw = null;
        try {
            br = new BufferedReader(new FileReader("sagan.txt"));
            bw = new BufferedWriter(new FileWriter("sagan2.txt"));
            while ( (linea = br.readLine()) != null) {
                bw.write(linea);
                bw.flush();
                bw.newLine();
                bw.newLine();
            }
        } catch (IOException e1) {
            System.out.println( "No se ha podido leer o escribir.");
            e1.printStackTrace();
        }
        finally {
            try {
                bw.close();
                br.close();
            }
            catch (Exception e2) {
                System.out.println("No se pudo cerrar el flujo.");
            }
        } // fin del bloque try-catch-finally
    }
}
```

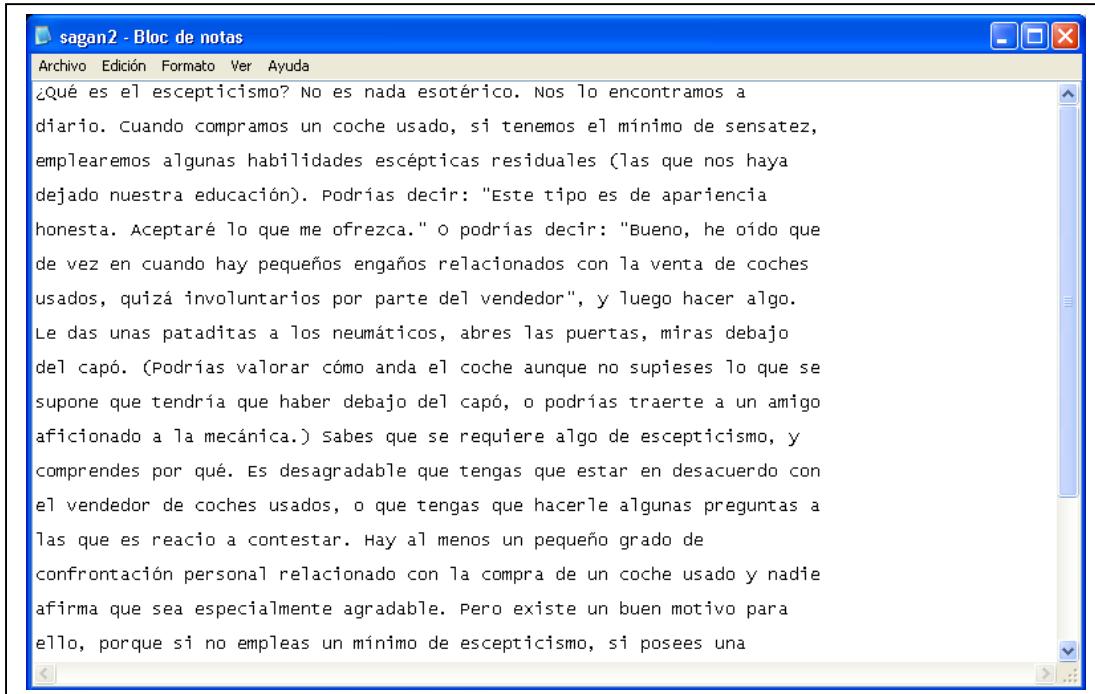


Figura 66. Contenido del archivo sagan2.txt

Consejo: A menudo resulta útil decorar un objeto `FileWriter` con un `BufferedWriter`, aunque no se necesite usar *buffers*, pues así se puede emplear el método `newLine()`.

3.9. La clase `java.io.PrintWriter`

Esta subclase de la superclase raíz `Writer` permite usar los métodos `print()` y `println()`. Sus constructores son

```
public PrintWriter(Writer out)
public PrintWriter(Writer out, boolean autoFlush)
public PrintWriter(OutputStream out)
public PrintWriter(OutputStream out, boolean autoFlush)
```

Cuando el argumento `autoFlush` se establece igual a `true`, el objeto `PrintWriter` creado vacía automáticamente su *buffer* cada vez que se llama a `println()`. A diferencia de la antigua clase `PrintStream`, esta clase usa un separador de líneas dependiente de la plataforma, en lugar de un carácter de nueva línea, y emplea la codificación de caracteres establecida en el sistema, sea ISO Latin 1 o no. `PrintWriter` permite escribir tipos primitivos, arrays de caracteres, cadenas y objetos. En el siguiente ejemplo se usa dicha clase para escribir en un archivo el alfabeto ruso (por brevedad, se ha omitido la gestión de las posibles excepciones):

Ejemplo 15: AlfabetoRuso.java

```
import java.io.*;

public class AlfabetoRuso {

    public static void main(String args[]) throws IOException {
        FileOutputStream archivoSalida = new FileOutputStream( "AlfabetoRuso.txt");
        BufferedOutputStream salidaBuffer = new BufferedOutputStream(archivoSalida);
        PrintWriter salida = new PrintWriter(new OutputStreamWriter(salidaBuffer, "UTF-8" ));

        //Se escribe el alfabeto cirílico
        for (char i = 0x0401; i < 0x0460; i++) {
            salida.print(i);
        }
        salida.close();
    }
}
```

El archivo resultante se muestra en la página siguiente (se incluye también la codificación ISO 8859-5 para que el lector pueda comprobar que está todo el alfabeto ruso).

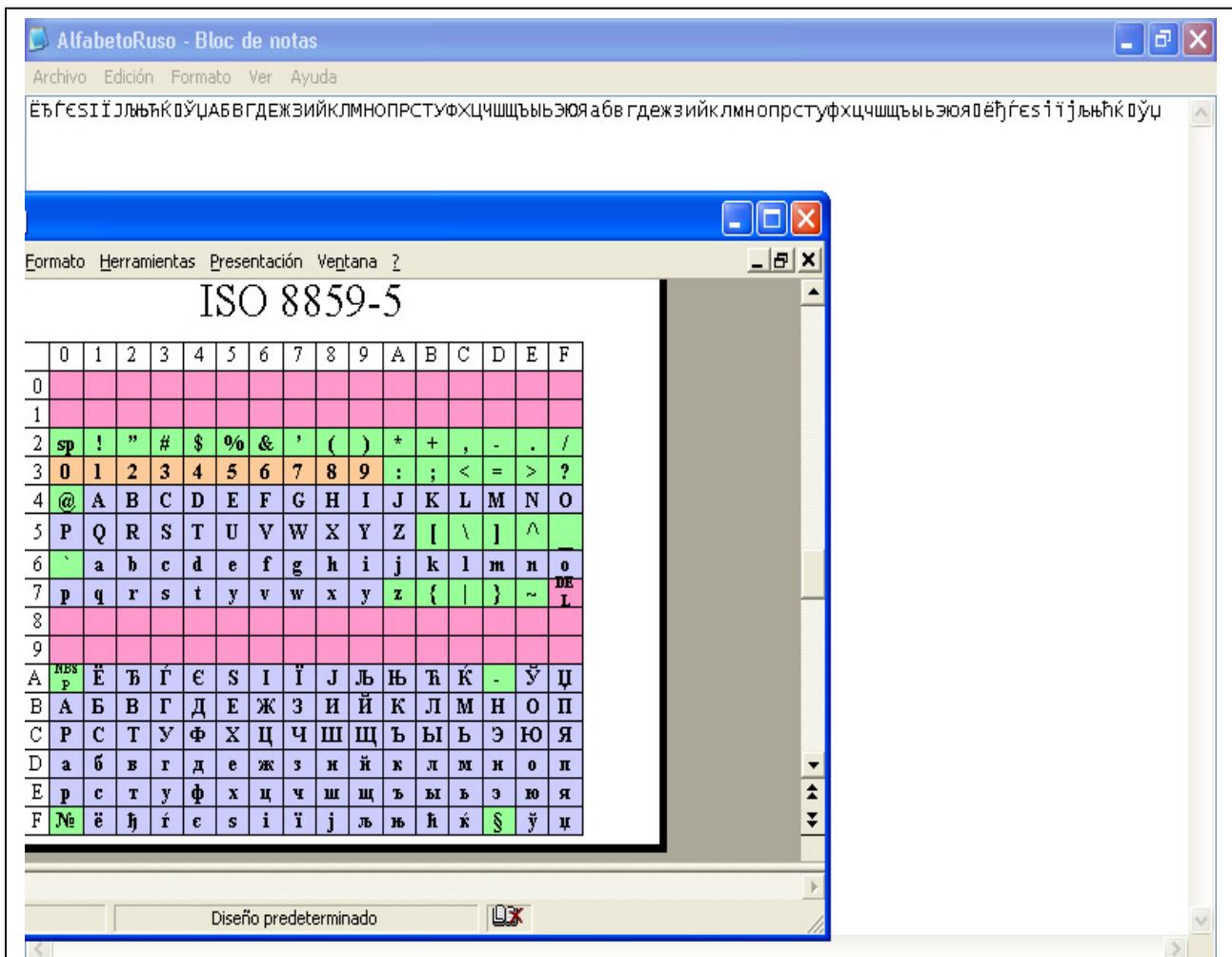


Figura 67. El resultado del programa 15

3.10. El patrón decorador y el paquete java.io

En el subapartado 3.2 se mencionó el patrón decorador. Este patrón se ha usado en la elaboración de todas las jerarquías de clases de `java.io`; por tanto, conocer cómo funciona ayuda a comprender por qué las clases de E/S de Java –se usen o no para comunicaciones en red– se crearon así. El lector sin interés en él puede pasar directamente al apartado 4.

Para entender por qué `java.io` usa el patrón decorador, no hay nada más fácil que suponer que no se hubiera usado. Consideremos, por caso, la jerarquía de clases de `java.io.InputStream`.

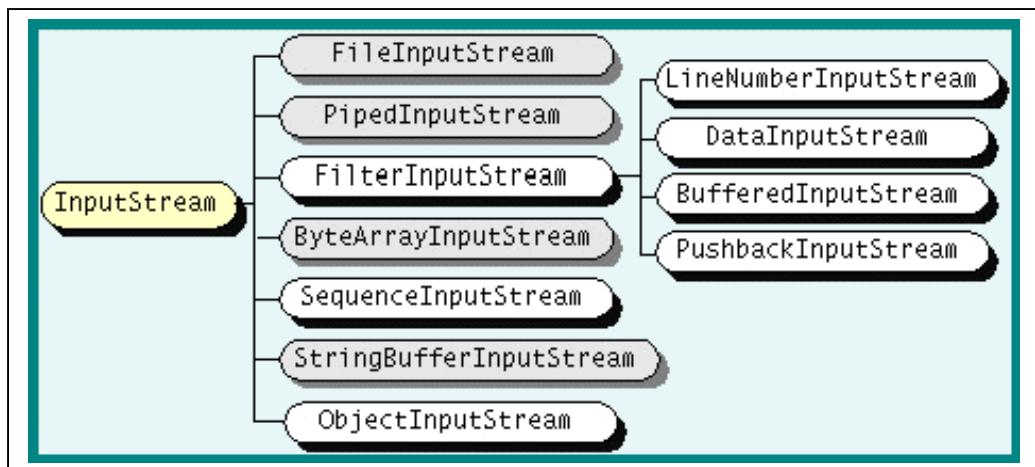


Figura 68. La jerarquía de clases InputStream. Extraída de la documentación oficial de Sun

Supongamos que no existiera la clase abstracta `FilterInputStream`. Para que clases como `FileInputStream`, `PipedInputStream`, etc., tuvieran disponibles las funciones adicionales que aportan `LineNumberInputStream`, `DataInputStream`, `BufferedInputStream` y `PushbackInputStream` sería necesario tener clases como

- `LineNumberFileStream`
- `LineNumberPipedInputStream`
- `LineNumberStringBufferInputStream`
- `DataFileInputStream`
- `DataPipedInputStream`,
- `BufferedPipedInputStream`
- `BufferedFileInputStream`
- `BufferedPipedInputStream`
- `BufferedStringBufferInputStream`
- ...

Estas clases no existen: gracias al patrón decorador no son necesarias

Algo falla, ¿verdad? El número de clases necesarias se dispararía. El problema radica en que cualquier combinación de los siguientes factores es posible cuando se trabaja con E/S:

- Entrada o salida.
- Fuente o destino: archivos, arrays de Strings, sockets, etc.
- Uso de *buffers* o ausencia de ellos.
- Formato de bytes o de caracteres Unicode.
- Tipos de operaciones: acceso secuencial, aleatorio, por línea, por palabra, etc.

El patrón decorador, cuyo esquema se representa en la figura 69; pone coto a estos crecimientos cancerosos de las jerarquías de clases.

Este patrón permite añadir de modo dinámico nuevas funciones a objetos individuales (no a clases completas). En vez de usar la herencia tradicional, este patrón encapsula un objeto dentro de un objeto decorador, que se encarga de proporcionar las nuevas funciones.

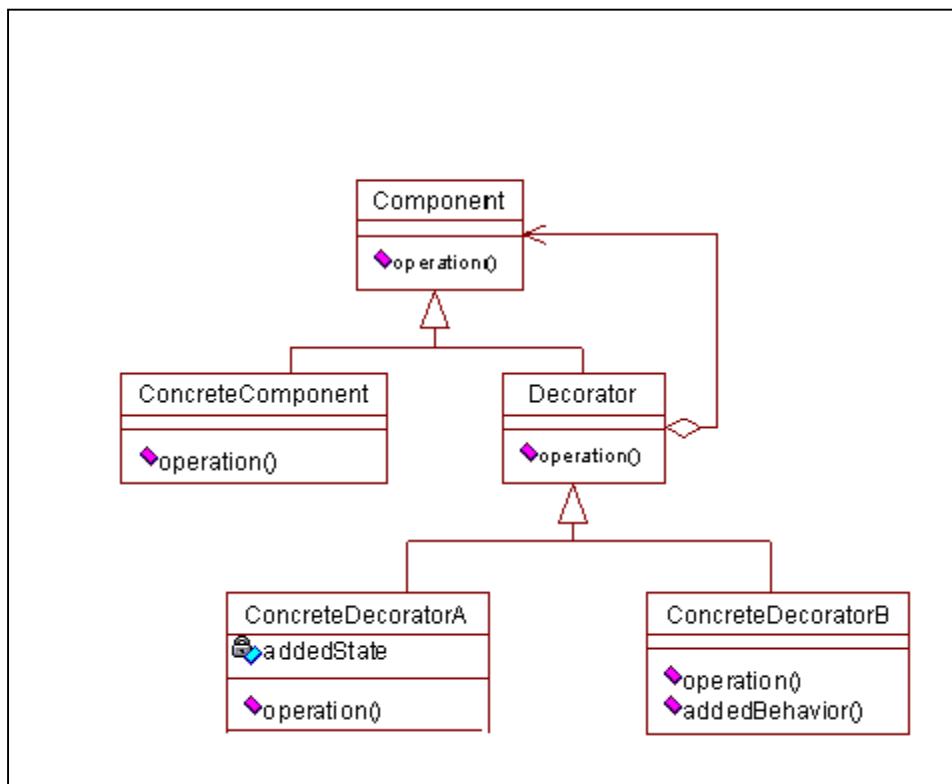


Figura 69. Esquema del patrón decorador.

En este esquema,

- El elemento **Componente** define la interfaz para los objetos a los que se les pueden añadir funciones. En Java, puede ser una clase abstracta o una interfaz.
- El elemento **ComponenteConcreto** define un objeto al cual se pueden añadir funciones.
- El elemento **Decorador** mantiene una referencia al objeto **Componente** (composición) y define una interfaz conforme a la de **Componente**. Puede ser una clase abstracta o una concreta.

- Los elementos del tipo `DecoradorConcreto` añaden funciones específicas al objeto `Componente` (o modifican las que ya tenía).

Como una imagen vale más que mil palabras, voy a abordar un ejemplo donde la decoración es visible. Para ello, abandono momentáneamente el paquete `java.io` y planteo este problema: ¿cómo se podrían incorporar bordes rectangulares y redondeados a los elementos gráficos (*widgets*) de Swing? (es éste un ejemplo donde la decoración es gráfica)

Una solución sería extender cada componente gráfico de Swing. Se tendrían clases como

```
public class JButtonBordeRecto extends JButton { ... }
public class JCheckBoxRecto extends JCheckBox { ... }
public class JComboBoxRecto extends JComboBox { ... }
...
public class JButtonBordeRedondo extends JButton { ... }
public class JCheckBoxRedondo extends JCheckBox { ... }
public class JComboBoxRedondo extends JComboBox { ... }
...
```

Esta aproximación es pésima por cuanto aumenta el número de clases que se necesita manejar. Si usamos el patrón decorador, el código que deberíamos escribir para responder a la pregunta anterior sería similar a éste:

Ejemplo 16a: DecoradorBorde.java

```
import javax.swing.*;
import java.awt.*;

public class DecoradorBorde extends JComponent {

    protected JComponent componenteHijo;
    private int forma;
    public final static int RECTANGULO = 0;
    public final static int REDONDO = 1;

    public DecoradorBorde(JComponent componente, int forma) {
        componenteHijo = componente;
        this.setLayout(new BorderLayout());
        this.add(componenteHijo);
        this.forma = forma;
    }

    public void paint(Graphics g) {
        super.paint(g);
        int alto = this.getHeight();
        int ancho = this.getWidth();
        if (forma == RECTANGULO) {
            g.drawRect(0, 0, ancho - 1, alto - 1);
        } else if (forma == REDONDO) {
            g.drawRoundRect(0, 0, ancho - 1, alto - 1, 75, 75);
        }
    }
}
```

```
    }
}

}
```

Para conseguir que un componente de Swing adquiera un borde rectangular bastará con escribir líneas como ésta:

```
DecoradorBorde db = new DecoradorBorde(new componente (
    "Decorated JLabel") , DecoradorBorde.RECTANGULO);
```

Para conseguir componentes con bordes redondos bastará con usar líneas así:

```
DecoradorBorde db = new DecoradorBorde(new componente (
    "Decorated JLabel") , DecoradorBorde.REDONDO);
```

Veamos un ejemplo:

Ejemplo 16b: Frame1.java

```
import java.awt.*;
import javax.swing.*;

public class Frame1 extends JFrame {

    // Componentes
    DecoradorBorde etiqueta = new DecoradorBorde(new JLabel("JLabel decorado"),
                                                DecoradorBorde.RECTANGULO);
    DecoradorBorde checkBox = new DecoradorBorde(new JCheckBox(
                                                "JCheckbox decorado"), DecoradorBorde.REDONDO);
    DecoradorBorde boton = new DecoradorBorde(new JButton("JButton decorado"),
                                              DecoradorBorde.REDONDO);
    DecoradorBorde comboBox = new DecoradorBorde(new JComboBox(),
                                                DecoradorBorde.RECTANGULO);

    public Frame1() {
        try {
            this.setDefaultCloseOperation(EXIT_ON_CLOSE);
            getContentPane().setLayout(null);
            etiqueta.setBounds(new Rectangle(30, 20, 140, 30));
            checkBox.setBounds(new Rectangle(30, 120, 140, 30));
            boton.setBounds(new Rectangle(30, 220, 140, 30));
            comboBox.setBounds(new Rectangle(30, 320, 140, 30));
            this.getContentPane().add(etiqueta, null);
            this.getContentPane().add(checkBox, null);
            this.getContentPane().add(boton, null);
            this.getContentPane().add(comboBox, null);

        }
        catch(Exception e) { e.printStackTrace();}
    }

    public static void main(String[] args) {
```

```
        Frame1 frame1 = new Frame1();
        frame1.setBounds(0, 0, 400, 600);
        frame1.setVisible(true);
    }

}
```

El resultado se muestra aquí:

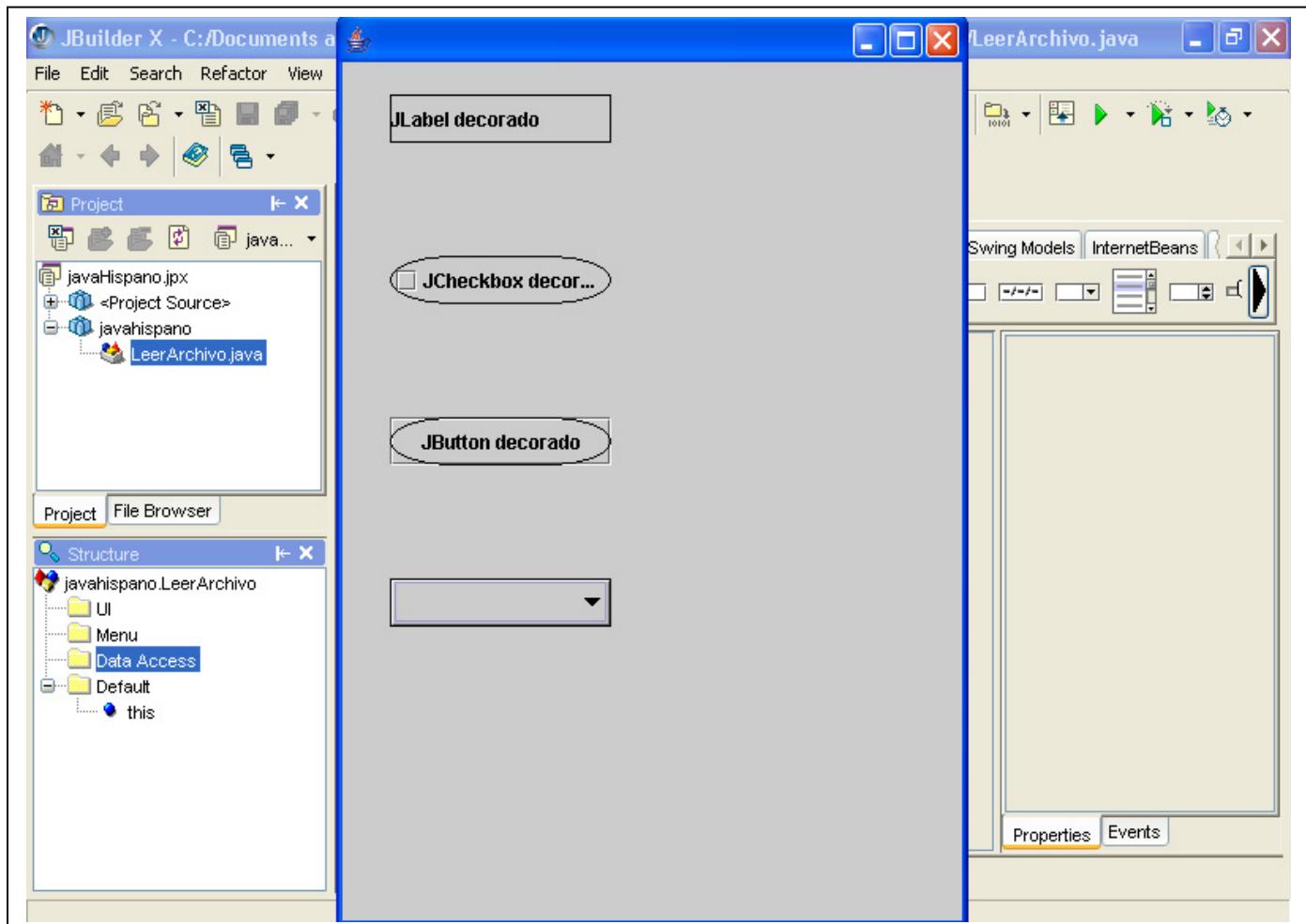


Figura 70. Ejemplo gráfico del patrón decorador

Comprendiendo este ejemplo gráfico, podemos retornar al paquete `java.io`. La misteriosa clase `FilterInputStream` del subapartado 3.2. es un decorador abstracto. `InputStream` es el componente abstracto raíz del patrón decorador, y clases como `DataInputStream`, `BufferedInputStream`, etc., son decoradores concretos (recuérdese la figura 68). Lo mismo puede decirse, *mutatis mutandis*, para la jerarquía `java.io.OutputStream`.

Si consideramos la jerarquía `java.io.Reader` (véase la figura 59), `Reader` es el componente abstracto raíz en el patrón decorador; `FilterReader`, el decorador abstracto; y `BufferedReader`, `CharArrayReader`, `PushBackReader`, `StringReader`, etc., decoradores concretos. Algo similar puede decirse para la jerarquía `java.io.Writer`, *mutatis mutandis*.

A la luz del patrón decorador, podemos profundizar en lo que significan líneas como éstas:

```
FileInputStream fis = new FileInputStream( "historial.txt" );
InputStreamReader isr = new InputStreamReader(fis);
BufferedReader br = new BufferedReader(isr);
```

El objeto `fis` se decora con la clase `InputStreamReader` y adopta nuevas funciones. Los métodos `read()` de `FileInputStream` permiten leer bytes del flujo de entrada del archivo; ahora; los métodos `read()` de `InputStreamReader` permiten leer caracteres del flujo de entrada de caracteres asociado al archivo. No es que el objeto `isr` no use los métodos `read()` de la clase a la que pertenece el objeto que encapsula: sí los usa, pero luego añade su propio código (que convierte los bytes en caracteres mediante el juego de caracteres especificado o el que esté por defecto en el sistema). Externamente, la interfaz del objeto `isr` es compatible con la de `fis` (el decorador tiene, al menos, la misma interfaz que el objeto que decora o encapsula).

`BufferedInputStream` añade funciones al objeto `isr`: le proporciona capacidad para usar *buffers* con la entrada y le añade los métodos `mark()` y `reset()`. El primer método registra un punto en el flujo de entrada, y el segundo causa que todos los bytes leídos desde la última llamada a `mark()` sean releídos antes de que se vuelvan a leer nuevos bytes del flujo de entrada.

4. Aplicaciones y sistemas distribuidos

4.1. Introducción. De los mainframes a los sistemas distribuidos

Cuando los primeros dinosaurios informáticos poblaban este planeta, el *mainframe* (macroordenador, ordenador central) era el *Tyrannosaurus Rex*. La configuración más habitual en los sistemas bancarios de los años sesenta y setenta consistía en disponer de un *mainframe* con una base de datos de tipo jerárquico y en el que se ejecutaba una aplicación COBOL. Incluso hoy día, esta configuración persiste en algunos sistemas bancarios. Por lo general, los clientes eran “terminales tontos”: máquinas sin procesador que se limitaban a enviar peticiones al *mainframe* y a mostrar las respuestas al usuario; sus funciones eran simplonas: recoger las señales eléctricas que se producían al presionar las teclas, enviarlas al *mainframe* y mostrar por pantalla los caracteres que envíaba éste. La “inteligencia”, si así se puede llamar, residía en otra parte: en el *mainframe*.

Una aplicación que se ejecutaba en un entorno de *mainframes* y clientes tontos era lo que hoy conocemos como “aplicación monolítica” (en aquellos tiempos, desde luego, las cosas se veían de otro modo). Estas aplicaciones siguen al pie de la letra estos versos (espero que Tolkien, o su espíritu, no se moleste: son tantos los que lo nombran sin reparos en estos tiempos):

One ring to rule them all,
One ring to find them,
One ring to bring them all
and in the Darkness bind them.

En este apartado uso **arquitectura de un sistema (informático)** o, simplemente, **arquitectura** en el sentido de una representación conceptual o lógica del sistema que incluya lo siguiente:

- La identificación de todos los componentes del sistema.
- Las funciones de cada componente.
- Las relaciones e interacciones entre los componentes.

En general, las funciones desempeñadas por cualquier aplicación pueden dividirse en tres funciones o componentes generales (en otras palabras: una aplicación tiene estas partes):

- Lógica de acceso a datos.
- Lógica de la aplicación.
- Lógica de la presentación.

La **Lógica de acceso a datos** se encarga del almacenamiento de los datos, así como de mantenerlos actualizados. Como casi todas las aplicaciones usan bases de datos, la lógica de acceso a datos suele implementarse mediante un sistema de gestión de bases de datos (*Database Management System*, DBMS): Oracle, MySQL, SQLServer, Access, etc. El sistema de gestión también puede corresponder a un mecanismo de gestión de archivos de texto, binarios, etc, si bien no es lo habitual en las aplicaciones empresariales. Esta capa se encarga de las interacciones con la base de datos para modificar, añadir o borrar registros, así como para efectuar consultas. La base de datos es la verdadera encargada de almacenar los datos en un medio físico

(un disco, p. ej.) y de recuperarlos. Esta capa no es responsable de manipular o procesar los datos. En ocasiones, se separa esta lógica en dos: lógica de acceso a datos y lógica del almacenamiento de datos.

La **Lógica de la aplicación** es responsable de manejar la lógica del procesado de los datos (validación e identificación de los errores de procesado), las reglas de negocio y la lógica de la gestión de datos (identificación de los datos necesarios para procesar las transacciones y consultas). Si uno quiere recurrir a palabras sencillas y claras, puede identificar esta parte de la aplicación con el código: los `if`, `while`, `do...`

La **Lógica de la presentación** se encarga de dar formato a los datos, de presentarlos a los usuarios y de gestionar las entradas de éstos (pulsaciones de teclas, del ratón, etc.).

En el caso de una aplicación monolítica o centralizada, las tres funciones o componentes generales están mezcladas en la aplicación. La situación se muestra en la siguiente figura.

ESTRUCTURA DE UNA APLICACIÓN MONOLÍTICA

Mainframe

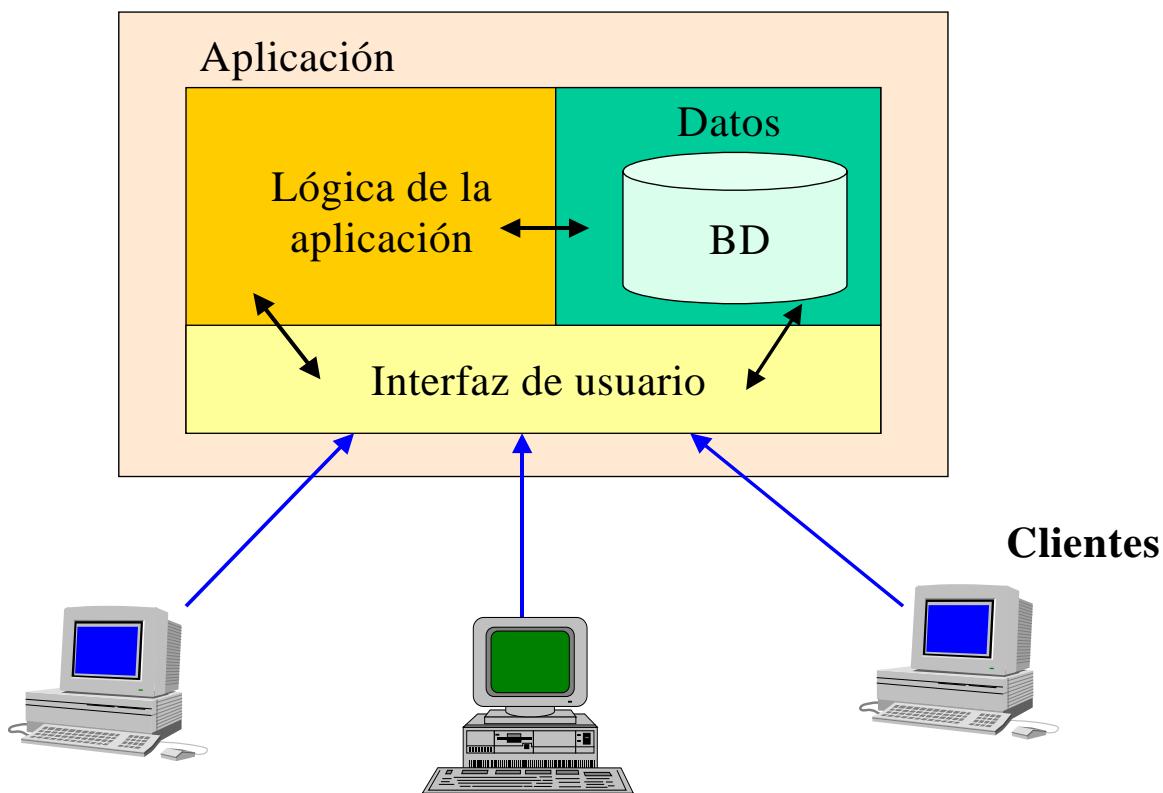


Figura 71. Cuando los mainframes dominaban la Tierra...

Las aplicaciones monolíticas tenían sentido cuando los clientes eran poco más que pantallas verdes, pero tenían numerosos problemas. No quiero hacer leña del árbol caído (aunque éste no ha caído del todo), pero citaré los tres más relevantes:

- El coste de los *mainframes* era muy elevado. Pocas pequeñas y medianas empresas podían permitirse uno.
- Como las tres funciones de la aplicación estaban entremezcladas, el código de la interfaz gráfica se mezclaba con el que implementaba la lógica de la aplicación o con el de acceso a datos. En consecuencia, a) cualquier modificación de la aplicación resultaba difícil; y b) el código se deslizaba a marchas forzadas hacia la ilegibilidad.
- La escalabilidad y eficacia de las aplicaciones estaba muy limitada. Las mejoras solo se producían aumentando la memoria del *mainframe* o añadiéndole procesadores, lo cual era muy costoso e implicaba cambios en el hardware. Cada vez que se introducía un nuevo terminal, se tornaban más lentas las contestaciones a cada usuario.

Los *mainframes* llevaron a que se plantearan las preguntas que subyacen tras los sistemas distribuidos. Cuando una organización se veía forzada a comprar un segundo *mainframe* (para repartir la carga de trabajo, para otras oficinas, etc.), tenía que plantearse estas preguntas: ¿cómo se puede repartir el código de la aplicación entre dos máquinas?, ¿cómo se pueden repartir las tareas del sistema entre dos *mainframes*?, ¿cómo puede hacerse que dos máquinas trabajen de forma sinérgica, de modo que el rendimiento de las dos sea superior a la suma de los rendimientos individuales? Estas preguntas, muy difíciles de contestar en un entorno de *mainframes* y terminales “tontos”, deben ser respondidas por todo sistema distribuido.

Con la llegada de los primeros PC, se dispuso de ordenadores en los que se podían colocar parte de las funciones desempeñadas por los *mainframes*. Por ejemplo, se podía trasladar a los clientes la lógica de la presentación, así como parte de la lógica de la aplicación.

La redistribución de las funciones tuvo importantes consecuencias económicas: como los PC quitaban trabajo a los *mainframes*, se hizo posible poder sustituirlos por máquinas más baratas, como servidores UNIX. Estos servidores no tenían la potencia o capacidad de un *mainframe*, pero tampoco las necesitaban: ya no tenían la necesidad de mantener a decenas o cientos de terminales sin capacidades de cálculo o de procesado de datos. La era de los grandes dinosaurios tocaba a su fin (bueno, algunos aún sobreviven escondidos tras CORBA).

A las aplicaciones cuyas funciones se repartían entre un servidor y los PC se las llamó de tipo cliente-servidor (ahora se llaman aplicaciones cliente-servidor de dos capas). El término **cliente-servidor**, explicado en 2.3, apareció al principio de la década de los ochenta, y se hizo popular en la industria informática a finales de esa década. En el sentido más general posible, se usa el término para designar a una aplicación susceptible de ser dividida lógicamente en dos o más procesos donde cada uno es cliente o servidor. Los clientes hacen peticiones y los servidores las atienden.

En las arquitecturas cliente-servidor de dos capas, el cliente (primera capa) se comunica directamente con los servidores (segunda capa). Uso el término *capa* en el mismo sentido en que se usó para la arquitectura TCP/IP: una capa es una abstracción donde se agrupa a un conjunto de subproblemas relacionados entre sí, relativos a

algún aspecto de las comunicaciones en red. La ventaja de trabajar con capas es que cada una puede desarrollarse e implementarse de forma independiente de las otras. El cliente (primera capa) se encarga de todos los problemas asociados con hacer peticiones; el servidor (segunda capa) trata las cuestiones vinculadas a contestar las peticiones del cliente. En las aplicaciones en red, suele identificarse –explícita o implícitamente– los términos *capa* e *implementación de la capa*.

Las separación en capas es una división lógica, y no conlleva necesariamente ninguna opción de distribución física de las capas (el cliente y el servidor podrían ejecutarse como procesos independientes en una misma máquina). **Lo importante en una capa es que esté estructurada de manera que su implementación pueda desarrollarse y mantenerse con independencia de las otras, no su ubicación física.** No obstante lo dicho, en el caso de aplicaciones en red, lo usual es que las capas se implementen en máquinas o anfitriones distintos. Muchos autores de textos sobre redes suelen dar por sentado que las capas lógicas corresponden a la separación física entre anfitriones o dispositivos. Esta correspondencia no es del todo cierta, pero resulta admisible si sólo nos referimos a entornos de red.

En la figura 72 se muestra un ejemplo de la arquitectura c-s de dos capas, correspondiente a una situación muy común: el cliente se encarga de la interfaz gráfica y de parte de la lógica de la aplicación; el servidor se encarga de la lógica de acceso a datos y de parte de la lógica de la aplicación.

En la figura 73 se muestran los cuatro tipos de arquitecturas c-s de dos capas. El caso extremo de las aplicaciones de dos capas donde el servidor realiza todas las tareas nos conduce de vuelta a las aplicaciones monolíticas o centralizadas, de una sola capa (el servidor se encarga de todo; no hay cliente, pues el terminal “tonto” no ejecuta procesos).

La figura 74 muestra un ejemplo de una aplicación de dos capas con clientes “gordos”. Un cliente de este tipo se encarga de la interfaz de usuario, de la lógica de la aplicación y de parte de la lógica de acceso a datos. En el ejemplo concreto de la figura, se encarga de enviar las consultas a la base de datos.

La figura 75 se muestra una aplicación basada en la estructura mostrada en la figura 72.

ESTRUCTURA DE UNA APLICACIÓN C-S DE 2 CAPAS

Miguel Ángel Abián Julio 2004

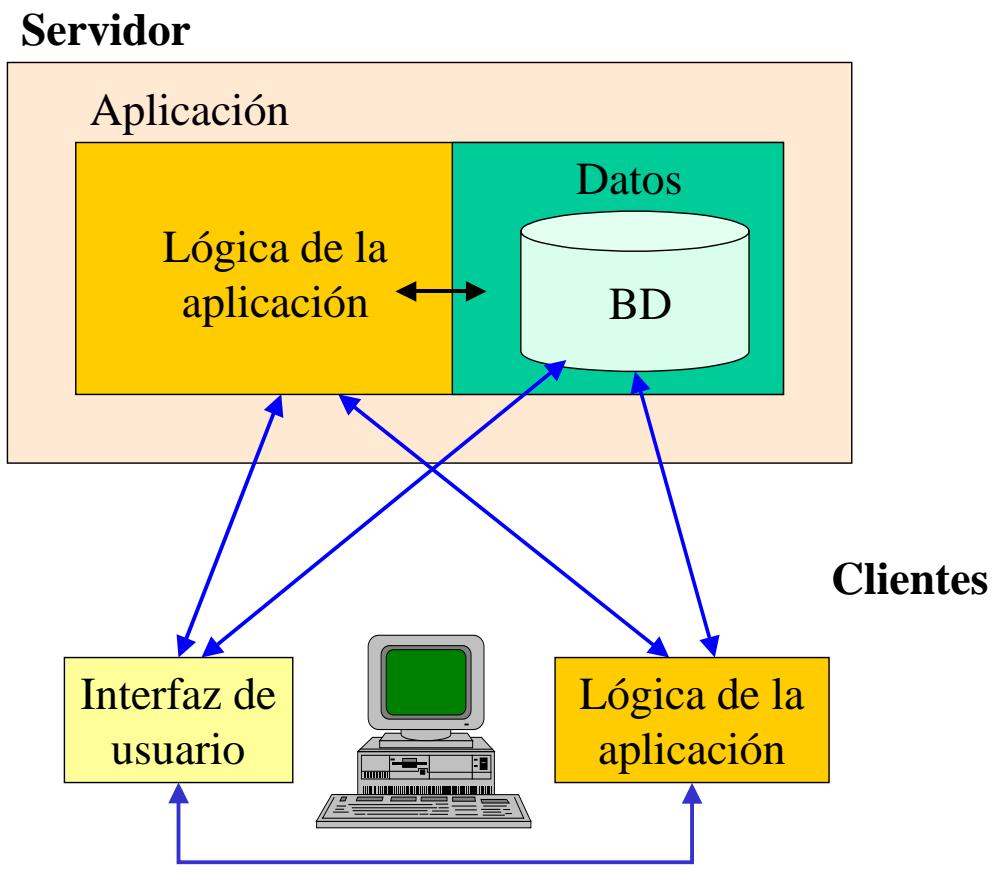
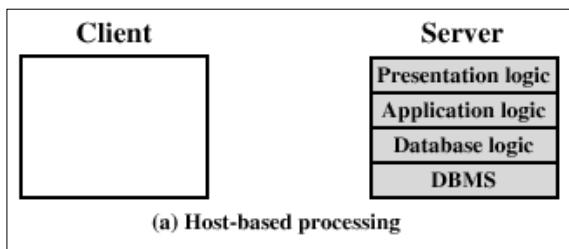


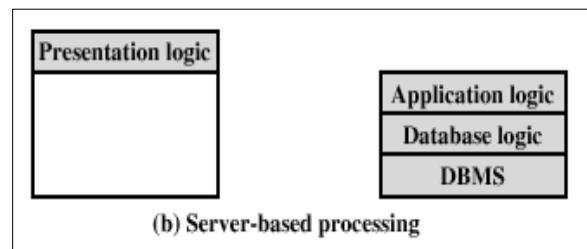
Figura 72. Estructura de una aplicación cliente-servidor de dos capas

En su momento, las aplicaciones cliente-servidor de dos capas resultaron de una novedad espeluznante. Hoy día, con los continuos avances (y retrocesos) informáticos, nos parecen demasiado simples. Para apreciar lo que supusieron hay que verlas con los ojos de las personas que estaban acostumbradas a ver gigantescos *mainframes* y pantallas verdes o grisáceas. En la actualidad se continúan usando las aplicaciones de dos capas, sobre todo en aplicaciones para intranets de tamaño pequeño o mediano.

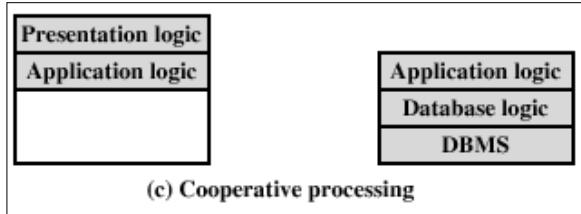
TIPOS DE ARQUITECTURAS CLIENTE-SERVIDOR DE DOS CAPAS



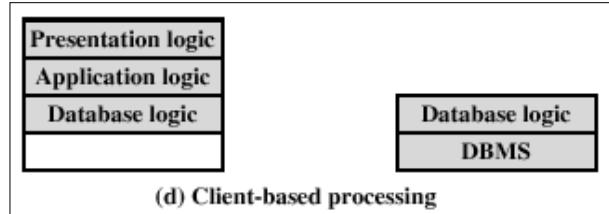
Centralizada o monolítica



Basada en el servidor (“cliente delgado”)



Cooperativa



Basada en el cliente (“cliente gordo”)

DBMS: Sistema de gestión de la base de datos

Figura 73. Tipos de arquitecturas cliente-servidor de dos capas. El sistema de gestión de la base de datos se representa separado de la lógica de acceso a datos

EJEMPLO DE LA ARQUITECTURA C-S DE DOS CAPAS CON CLIENTES “GORDOS”

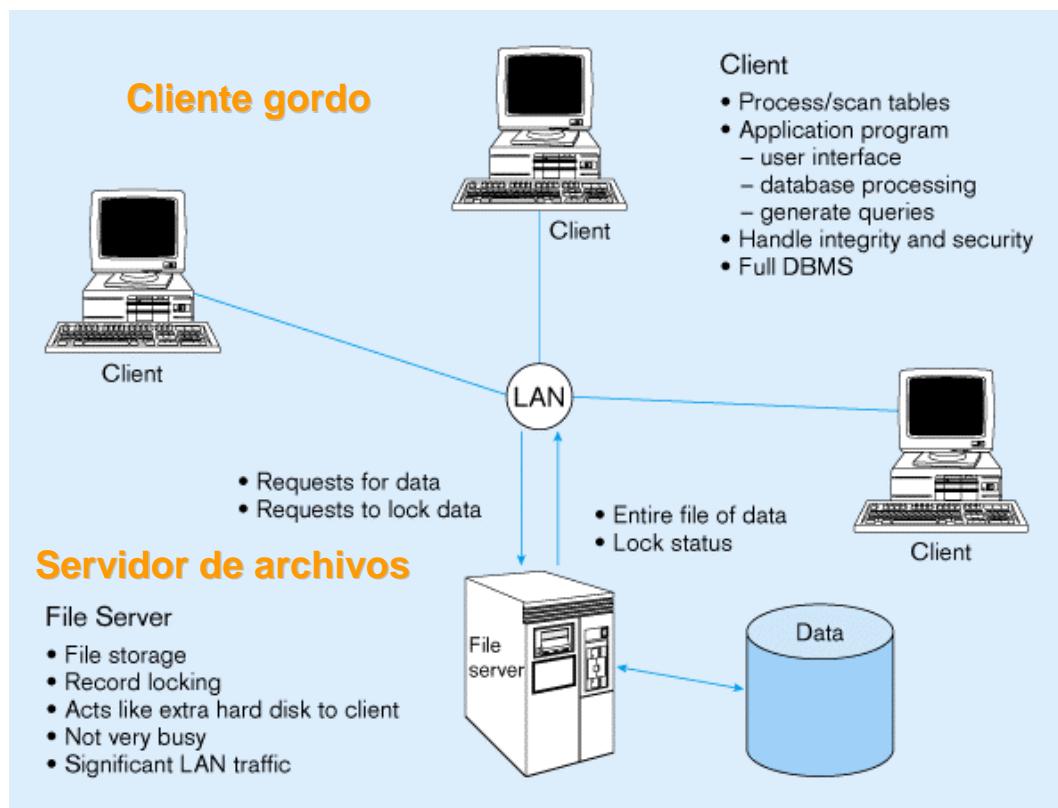


Figura 74. Un ejemplo del tipo d) de la figura 73. Extraído de una propaganda comercial



Figura 75. Un ejemplo del tipo c) de la figura 73. La aplicación SIGEF ha sido desarrollada por el Ministerio de Economía y Finanzas de Ecuador

A pesar de las ventajas con respecto a las aplicaciones monolíticas, las aplicaciones basadas en las arquitecturas c-s de dos capas también tienen sus problemas. A menudo, se encapsulan en el cliente funciones como el acceso a los datos y la lógica de la aplicación. Este enfoque tiene aparejado un importante problema: el de la distribución de la aplicación del cliente. Supongamos, por ejemplo, que la aplicación en el lado del cliente tiene código de acceso a una base de datos (mediante consultas SQL, p. ej.) mezclado con el de la lógica de la aplicación. Cualquier cambio en el código de acceso a los datos o en la lógica de aplicación hará que la aplicación del cliente tenga que ser recompilada y, luego, reinstalada en cada uno de los clientes. En un entorno en que las aplicaciones se modifican a menudo y donde hay muchos clientes, el proceso de distribución puede ser costoso o imposible.

Otro problema lo constituye la falta de escalabilidad: las aplicaciones c-s de dos capas no son escalables. Conforme aumenta el número de usuarios, la red se va saturando, porque los clientes y el servidor intercambian mensajes de control continuamente, aun cuando no se estén atendiendo peticiones. Este tipo de aplicaciones no resultan prácticas para Internet, donde un servidor puede recibir cientos o miles de peticiones simultáneas. En general, no son recomendables para más de 100 ó 150 clientes.

La solución para los problemas anteriores vino de mano de las aplicaciones cliente-servidor de tres capas, que aparecieron a principios de los años noventa y se hicieron populares a partir de 1995 (las figuras 76, 77, 78, 79 y 80 muestran varios ejemplos). En una aplicación de esta clase existe tres capas:

- La capa del cliente, asociada a la lógica de la presentación. Se encarga de la presentación de los datos, de darles formato, de recibir las peticiones de los usuarios y de controlar la interfaz gráfica.
- La capa intermedia, asociada a la lógica de la aplicación. Esta capa no existe como tal en las arquitecturas de dos capas. La capa se encarga de lo que se conoce como reglas de negocio: aplicar un IVA del 16% a las compras de combustible, retirar de una cuenta la cantidad solicitada, comprobar que un camión no sale sin carga, etc.
- La capa de almacenamiento de datos o capa de persistencia, también llamada capa de datos, asociada a la lógica de acceso a datos. La capa de datos administra y maneja la información de la aplicación, lo cual incluye el almacenamiento, mantenimiento y consulta de los datos.

Cada capa se implementa como una aplicación bien definida y separada. Además, en un contexto de red, estas aplicaciones suelen ejecutarse en anfitriones distintos. A continuación expongo los nombres típicos que se dan a esas aplicaciones y anfitriones.

- La aplicación de interfaz de usuario o aplicación GUI, que se ejecuta en el ordenador del usuario (cliente)
- La aplicación asociada a la capa intermedia, que se ejecuta en un anfitrión llamado servidor de aplicaciones (suele usarse este último término también para la propia aplicación).
- El sistema de gestión de bases de datos o aplicación de persistencia, que se ejecuta en un segundo anfitrión llamado servidor de bases de datos.

Aunque la situación anterior es la más común, no hay motivo teórico para que las aplicaciones que implementen las tres capas no puedan ejecutarse en un mismo anfitrión.

Las arquitecturas de dos capas se pueden ver como arquitecturas de tres capas donde no existe explícitamente la capa intermedia: sus funciones se hallan distribuidas entre el cliente y el servidor. La separación en tres capas permite desarrollar independientemente los tres grandes componentes de una aplicación, y que los programadores puedan dedicarse a implementar cada componente por separado. Dicha situación resulta imposible en una arquitectura de dos capas, donde siempre hay capas donde se mezclan los componentes. En el caso límite de las arquitecturas centralizadas, los tres componentes están en una sola capa: el servidor.

EJEMPLO DE UNA APLICACIÓN DE 3 CAPAS

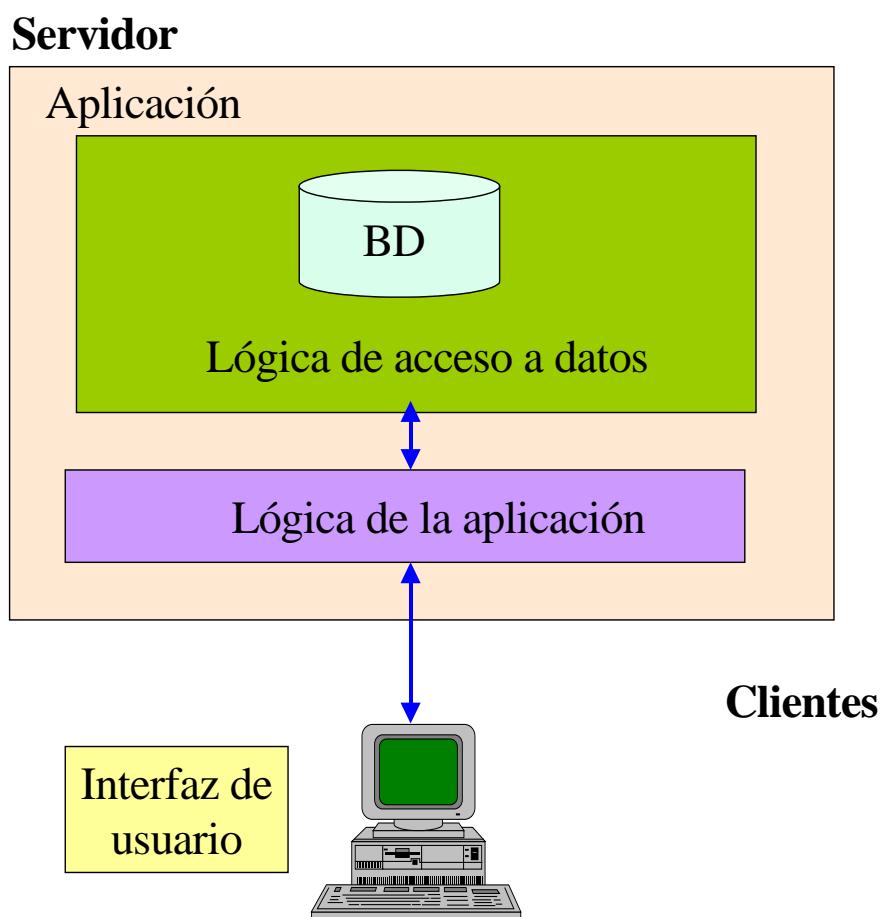


Figura 76. Ejemplo de una aplicación de tres capas que se ejecuta en dos máquinas

EJEMPLO DE LA ARQUITECTURA C-S DE TRES CAPAS CON CLIENTES “DELGADOS”

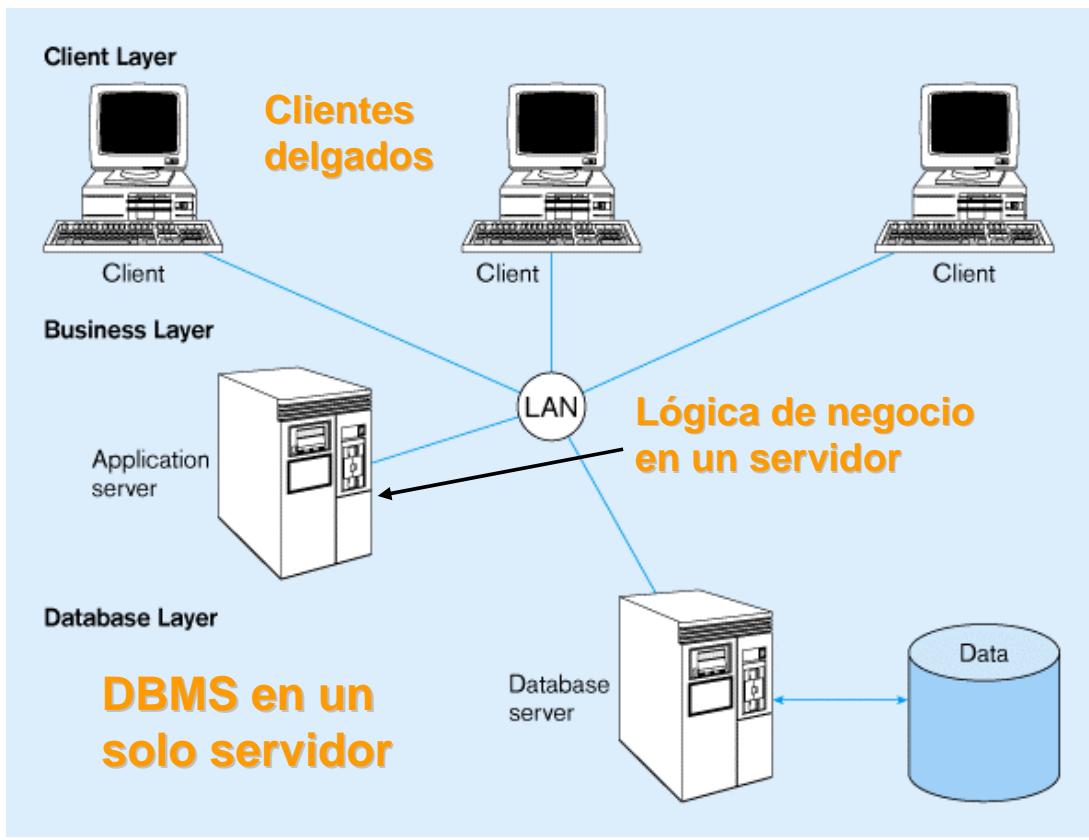


Figura 77. Ejemplo de una aplicación de tres capas que se ejecuta en tres máquinas. Extraído de una propaganda comercial

EJEMPLO DE LA ARQUITECTURA C-S DE TRES CAPAS

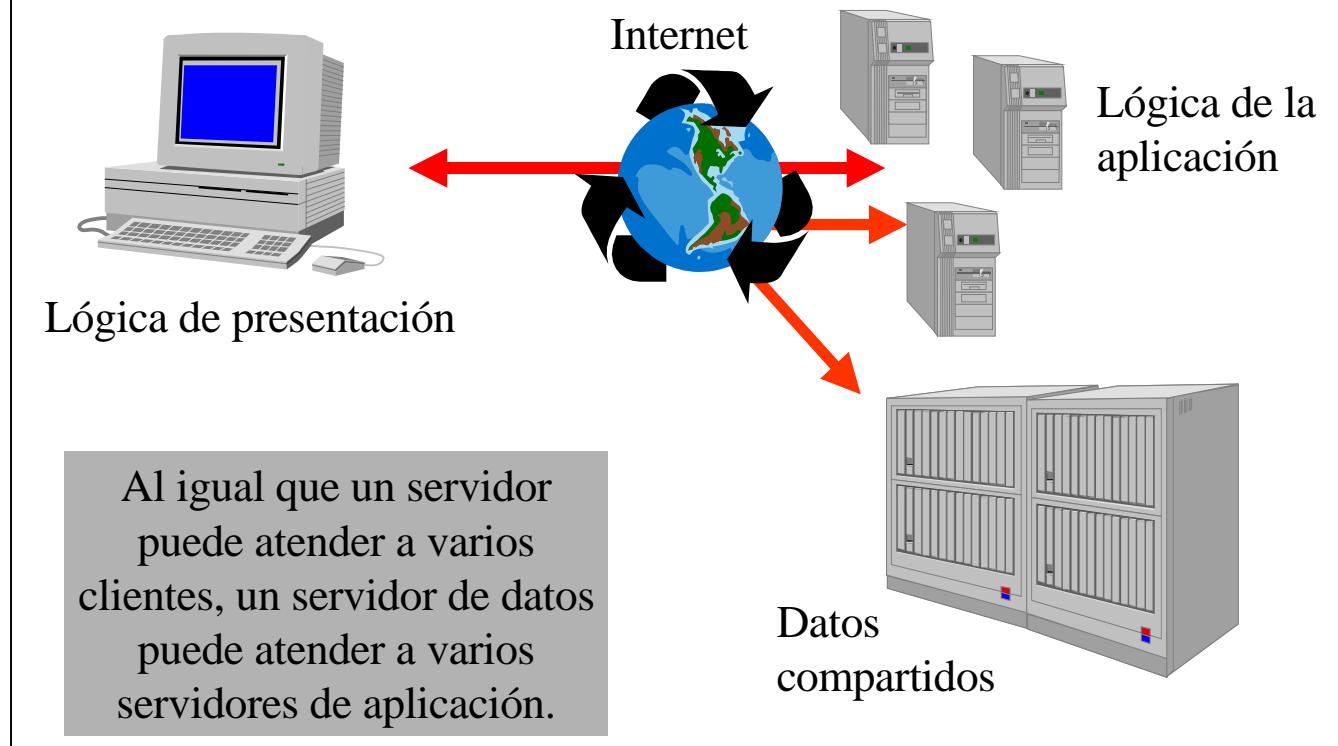


Figura 78. Ejemplo de una aplicación de tres capas que se ejecuta en N máquinas. La lógica de la aplicación puede repartirse entre varios servidores.

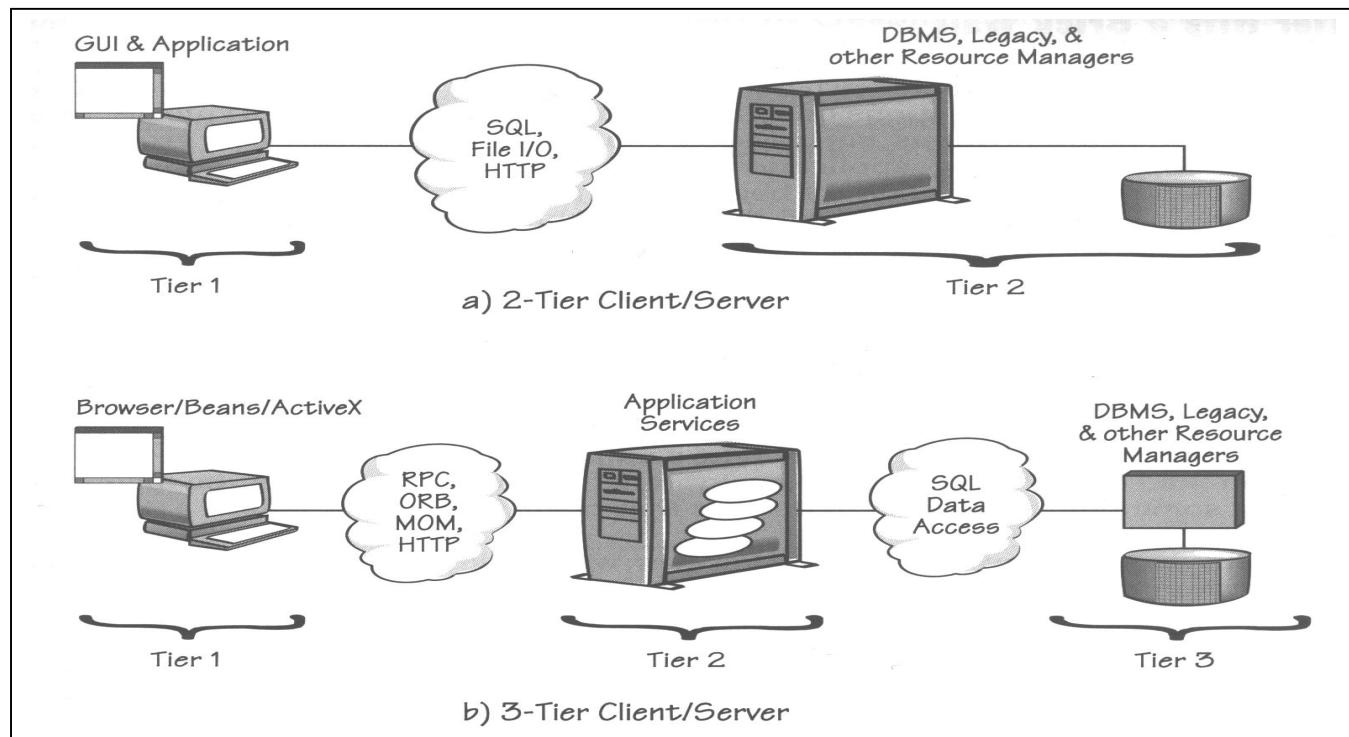
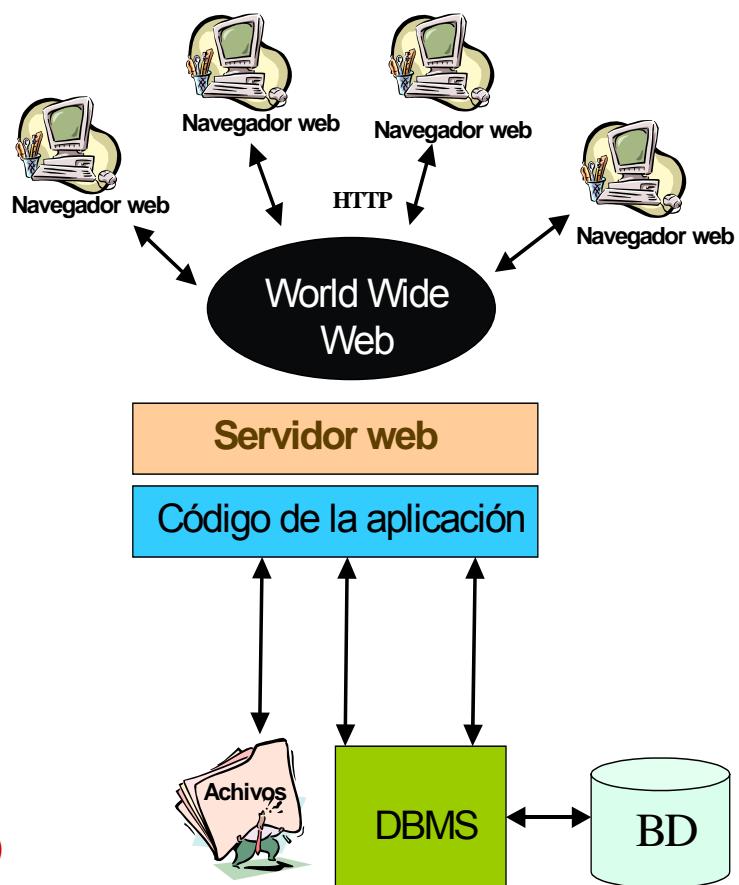


Figura 79. Comparación entre una aplicación c-s de dos capas y una de tres capas

LA WEB: UN SISTEMA DE TRES CAPAS

**Capa del cliente
(Lógica de la presentación)**



**Capa intermedia
(Lógica de la aplicación)**

**Capa de datos
(Lógica de acceso a datos)**

Figura 80. La WWW como un sistema de tres capas. La realidad es más complicada, pero vale como una primera aproximación

A diferencia de lo que sucede en las arquitecturas de dos capas, en las de tres no hay comunicación directa entre el cliente y el servidor de bases de datos: la primera hace sus peticiones a una capa intermedia (el servidor de aplicaciones), que determina qué datos se necesitan, dónde están localizados y los solicita al servidor de bases de datos. En consecuencia, la capa intermedia es cliente del servidor de bases de datos.

La división de las aplicaciones es más de dos capas favorece el reparto de la carga de trabajo entre varias máquinas. Si el servidor de la figura 76 recibiera demasiadas peticiones, una de las capas (lógica de acceso a datos o lógica de la aplicación) podría ubicarse en otro anfitrión. Si esa solución no fuera suficiente, incluso se podría dividir cada capa en subcapas que se repartirían entre varios anfitriones.

Esta división también simplifica el proceso de distribución de las aplicaciones. Como la lógica de la aplicación (el código, en definitiva) está ubicada en un único lugar (el servidor de aplicaciones), cuando hay que cambiarla basta con hacer los cambios en el servidor de aplicaciones para que todos los clientes accedan a la nueva lógica. Asimismo, si hay cambios en la lógica de acceso a datos, los clientes permanecen aislados de ellos gracias a la capa intermedia del servidor de aplicaciones.

EVOLUCIÓN DE LAS ARQUITECTURAS INFORMÁTICAS

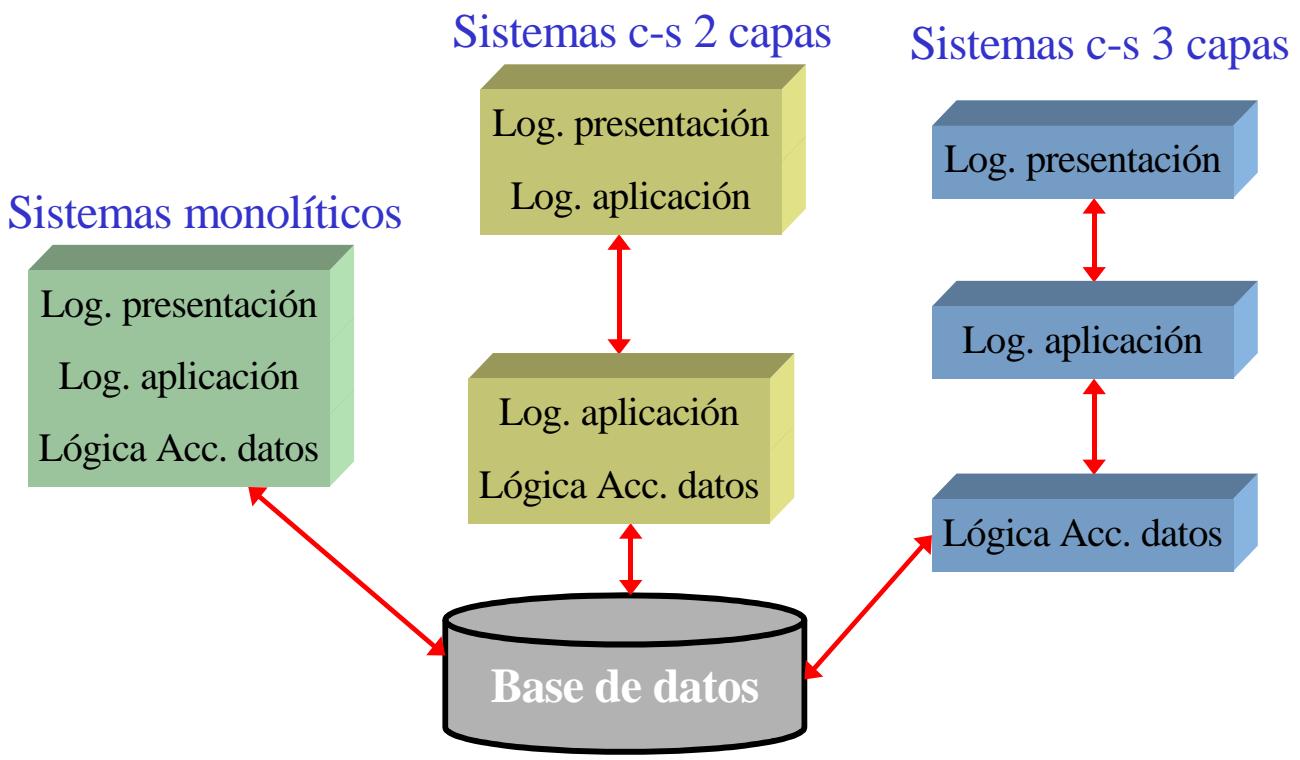


Figura 81. Evolución de las arquitecturas de las aplicaciones informáticas

Las arquitecturas c-s de capas no terminan con las de tres capas. Se pueden tener arquitecturas de cuatro capas, de cinco, etc. En general, de N capas. La división en más de tres capas resulta provechosa cuando interesa distribuir entre distintas máquinas la carga de trabajo. Más que estudiar cada una de las situaciones en las que interesa trabajar con cuatro o más capas, comentaré a continuación dos escenarios comunes.

El primer escenario corresponde al caso en que se trabajan con varias bases de datos, cada una con su DBMS. En este caso, puede que interese dividir la capa de datos o de persistencia en varias subcapas, asociando una a cada base de datos. La ventaja de esta aproximación consiste en que puede ubicarse cada subcapa de datos (mejor dicho, su implementación) en un nodo de la red, con lo cual la aplicación será fácilmente escalable cuando aumente el número de peticiones.

El segundo corresponde al caso de las aplicaciones de Internet. En ellas, se suele trabajar con navegadores *web* que procesan etiquetas HTML y con servidores intermedios o de aplicaciones escritos en C/C++ o Java. En estos casos, el “salto” entre el HTML y un lenguaje como Java o C/C++ es muy grande: estos últimos lenguajes pueden usarse para generar documentos HTML o para procesar peticiones HTTP de los clientes, pero entonces se mezcla la lógica de la aplicación y la de la interfaz. Una solución habitual para reducir el “salto” estriba en dividir la capa intermedia en dos subcapas: una es implementada por el servidor de aplicaciones; la otra, por uno o

varios programas CGI (véase 2.8.5) que reciben peticiones de los clientes (navegadores) y generan páginas HTML a partir de las respuestas que obtienen del servidor de aplicaciones, encargado de la lógica de la aplicación. En la figura 47b ya vimos un ejemplo de esta arquitectura de cuatro capas.

Las arquitecturas de N capas aparecen con cierta frecuencia en las tecnologías basadas en Java. Por ejemplo, en una aplicación JSP/Servlet bien diseñada suele descomponerse en tres partes (subcapas) el código que reside en el servidor de aplicaciones:

- Páginas JSP, encargadas de crear el HTML para las páginas web que actúan como interfaz gráfica para el usuario.
- Servlets o JavaBeans encargados de la lógica de la aplicación.
- Servlets, JavaBeans o clases Java que se encargan del acceso a los datos (suelen usar JDBC para extraer registros de las bases de datos).

En una aplicación EJB (*Enterprise JavaBeans*) sencilla, el código que reside en el servidor de aplicaciones suele dividirse en estas tres partes.

- Páginas JSP, servlets o aplicaciones Java de tipo cliente, que se encargan de la interfaz gráfica para el usuario.
- Beans de sesión (*Session Beans*) o de entidad (*Entity Beans*) que implementan la lógica de la aplicación.
- Beans de entidad cuyos campos representan datos. La persistencia de estos campos se consigue mediante los propios Beans de entidad o mediante un servidor EJB.

Las ventajas de las arquitecturas c-s multicapa (esto es, de tres o más capas) sobre las arquitecturas anteriores son muchas:

- La separación entre la lógica de la aplicación y la de presentación permite modificar de forma sencilla las aplicaciones. Cuando cambia la lógica de negocio, sólo hay que modificarla en un único lugar.
- Se reduce el tráfico de datos por las redes, pues la capa intermedia sólo transmite los datos imprescindibles para las tareas de la aplicación.
- El cliente se mantiene separado de las bases de datos y de los detalles de las redes intermedias. No necesita conocer dónde están los datos (pero sí dónde se localizan los servidores de aplicaciones).
- Las conexiones a las bases de datos, que son costosas en cuanto a recursos, pueden repartirse entre varios anfitriones.
- La administración de las aplicaciones es menos compleja, pues se reparte entre varias capas.

- La seguridad puede controlarse de manera precisa porque se pueden hacer comprobaciones en cada capa o subcapa. Además, los clientes no interactúan directamente con los datos, sino con la lógica de la implementación (que puede estar dividida en varias subcapas).

El esquema más general posible de las arquitecturas cliente-servidor multicapa se muestra en la siguiente figura. Cuando las subcapas de cada nivel se funden en una sola capa, tenemos el caso particular de las arquitecturas de tres capas.

ARQUITECTURA C-S MULTICAPA

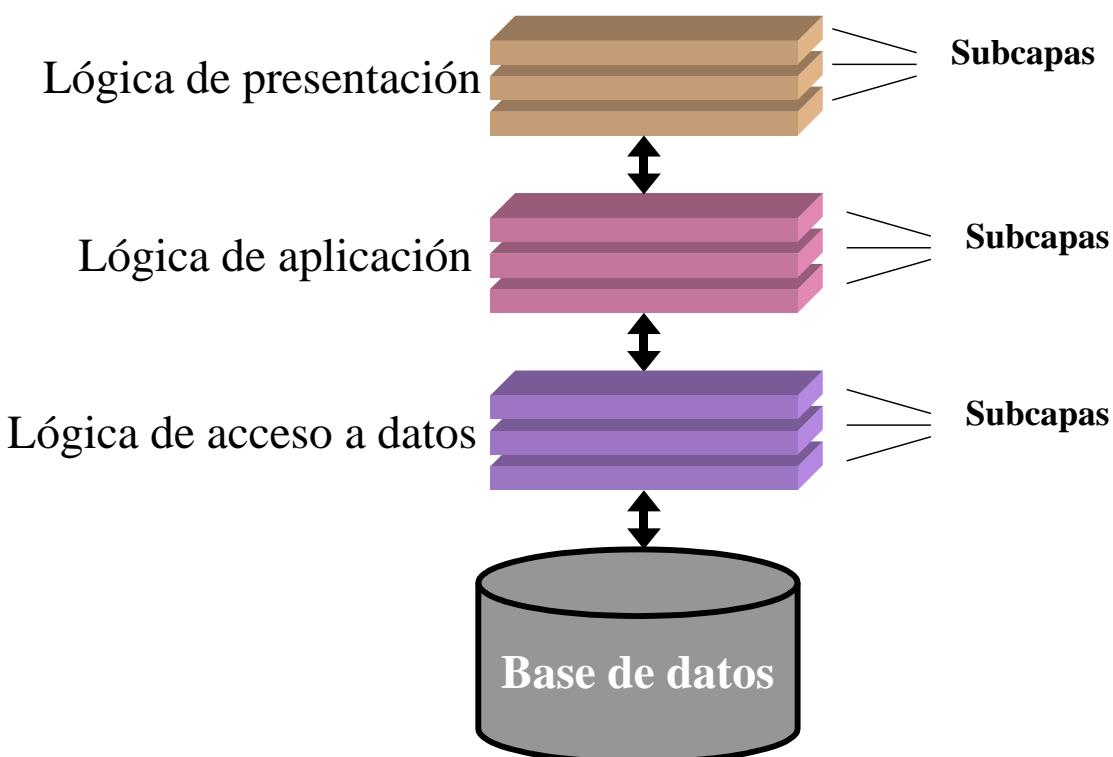


Figura 82a. Esquema general de las arquitecturas c-s de N capas

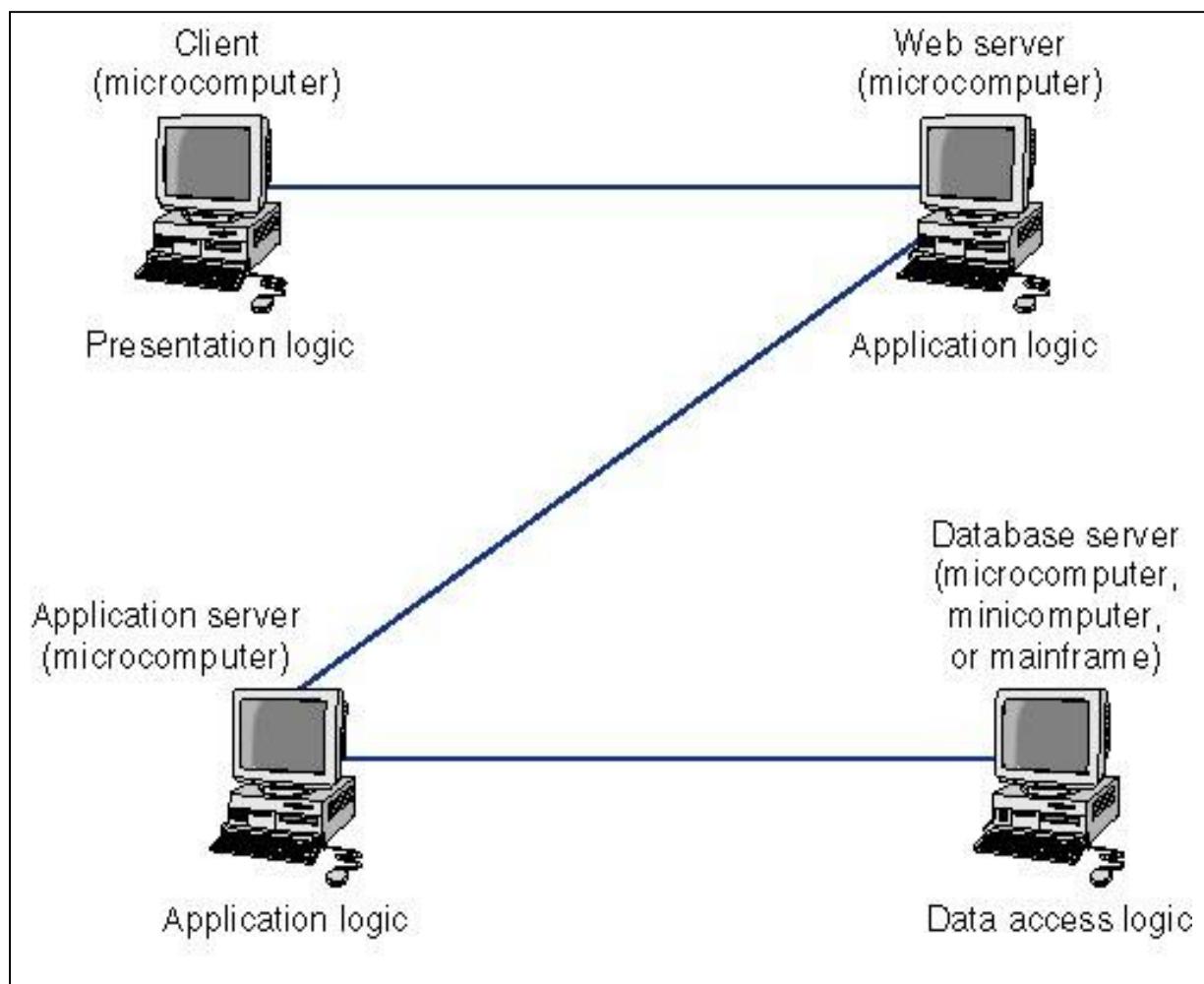


Figura 82b. Ejemplo concreto de arquitectura de N capas. Corresponde al caso en que la figura 82a tiene una capa de lógica de la presentación, dos subcapas de lógica de la aplicación y una capa de acceso a datos.

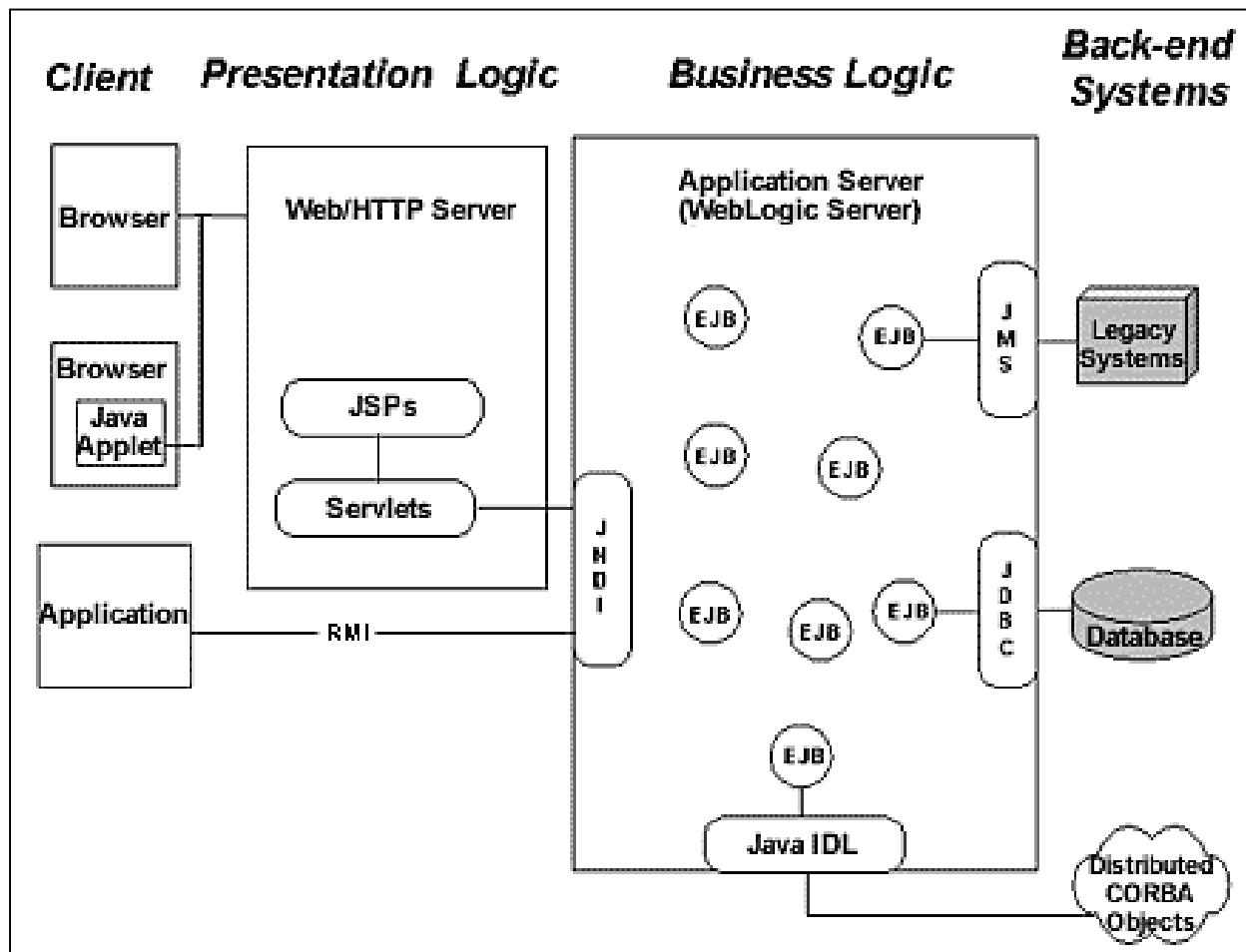


Figura 82c. Ejemplo concreto de arquitectura J2EE de N capas. Corresponde al caso en que la figura 82a tiene dos subcapas de lógica de la presentación, una capa de lógica de la aplicación, una capa de acceso a datos y una capa de almacenamiento de datos. Extraído de la documentación de Sun sobre J2EE

Los **sistemas distribuidos** nacieron a partir de las arquitecturas cliente-servidor multicapa. En esencia, un sistema distribuido consiste en un sistema cliente-servidor de N capas en el que hay un gran número de clientes y servidores, ubicados en distintos anfitriones.

La principal diferencia entre un sistema distribuido y uno de N capas estriba que en el primero no existen necesariamente los papeles explícitos de cliente o servidor. Cada elemento o componente de un sistema distribuido es, o puede ser, cliente y servidor. Debo aclarar que la falta de estos papeles explícitos no significa que un elemento distribuido sea cliente para unos elementos y servidor para otros, sino que puede ser cliente para un elemento dado y, después, servidor para ese mismo elemento (en lo dicho, se puede cambiar “cliente” por “servidor”). En un sistema distribuido pueden existir, según las necesidades del sistema, elementos que siempre actúen como clientes o servidores en relación con otros; pero si fuera preciso se podrían alternar en el papel de cliente y en el de servidor. En un sistema de N capas resulta imposible esa flexibilidad: el papel de una capa con respecto a otra (cliente, servidor) no puede variar.

A parte de la diferencia anterior, existen otras importantes diferencias entre los sistemas distribuidos y los basados en arquitecturas multicapa. Los sistemas

distribuidos se construyen para tratar con aplicaciones que quizás se ejecuten en miles o millones de nodos, correspondientes a distintas plataformas (*mainframes*, servidores UNIX, ordenadores personales de distintos fabricantes, teléfonos móviles de distintas empresas de telecomunicaciones, agendas electrónicas, electrodomésticos “inteligentes”, etc.). Para permitir el trabajo colaborativo entre nodos de tan variada naturaleza, se deben incorporar mecanismos ausentes en las aplicaciones multicapa. Por ejemplo, en un sistema distribuido no resulta razonable que el cliente sepa explícitamente en qué anfitrión se ejecuta el servidor al que debe dirigir sus peticiones (los servidores pueden cambiar de ubicación continuamente; pueden “morir” y “renacer” en otro nodo). En una aplicación de N capas, si un proceso servidor cambia de ubicación (pasa de un anfitrión a otro), los clientes deben ser “avisados”; en un sistema distribuido, los clientes jamás saben dónde se ejecutan los servidores (es más, puede que éstos no existan hasta que algún cliente los llame).

Otro ejemplo: en una aplicación multicapa puede aceptarse que los nodos usarán una misma representación externa de los datos cuando los transmitan a través de las redes, pues siempre puede elegirse el hardware para que así sea. En un sistema distribuido, la situación cambia radicalmente: hay involucrados tantos tipos de hardware que resulta absurdo suponer que todos los nodos usarán una misma representación externa para los datos.

Así las cosas, un sistema distribuido tiene que incorporar servicios o funciones innecesarias en un sistema multicapa. La lista de servicios puede ser tan larga como uno quiera (como demuestra el caso de CORBA), pero hay tres fundamentales:

- Un servicio de nombres, encargado de permitir que un elemento de la aplicación encuentre de manera dinámica a otros.
- Un servicio de representación común de los datos que son transmitidos a las redes, de manera independiente del hardware y del sistema operativo usados por cada nodo.
- Un servicio de control de las transacciones. Una **transacción** es una operación que debe ser realizada atómicamente: o se ejecuta correctamente cada paso de la operación, o se anula la operación en su conjunto. Por motivos evidentes, los sistemas bancarios son muy cuidadosos con las transacciones: a los bancos les disgusta sobremanera ver cuentas con -1.500.000 € por un fallo en el suministro eléctrico, o descubrir que una hipoteca en vigor consta como cancelada en 1910 porque un servidor ha fallado en mitad de un pago de la hipoteca. En los sistemas de N capas, los sistemas de gestión de bases de datos implementan mecanismos para el control de las transacciones que afectan a los datos; pero este control es parcial: sólo afecta a la capa de persistencia. En un sistema distribuido, todo elemento que opere con otros debe contar con un mecanismo que le permita volver a su estado inicial si hay problemas en la operación. Un solo servidor que quedara en un estado inválido o inconsistente podría dar respuestas absurdas a millones de clientes (la situación sería el equivalente distribuido a preguntar “¿Quién descubrió América?” y recibir la respuesta “La manzana es buena para la dentadura”; es decir, un diálogo de besugos distribuidos).

Internet (un sistema distribuido, a fin de cuentas) ha influido mucho en la popularidad de los sistemas distribuidos. Internet ofrece una base física y lógica para el desarrollo de aplicaciones distribuidas. La familia de protocolos TCP/IP permite edificar aplicaciones sobre la capa de transporte; el protocolo HTTP puede usarse para los mensajes entre elementos o componentes distribuidos, etc.

En el caso de muchas empresas, la posibilidad de contar con un medio como Internet como base para sus aplicaciones distribuidas ha ayudado a que estas empresas hayan optado por soluciones distribuidas, pues no necesitan disponer de redes propias (WAN) para hacerlas funcionar entre oficinas, sucursales, departamentos, etc.

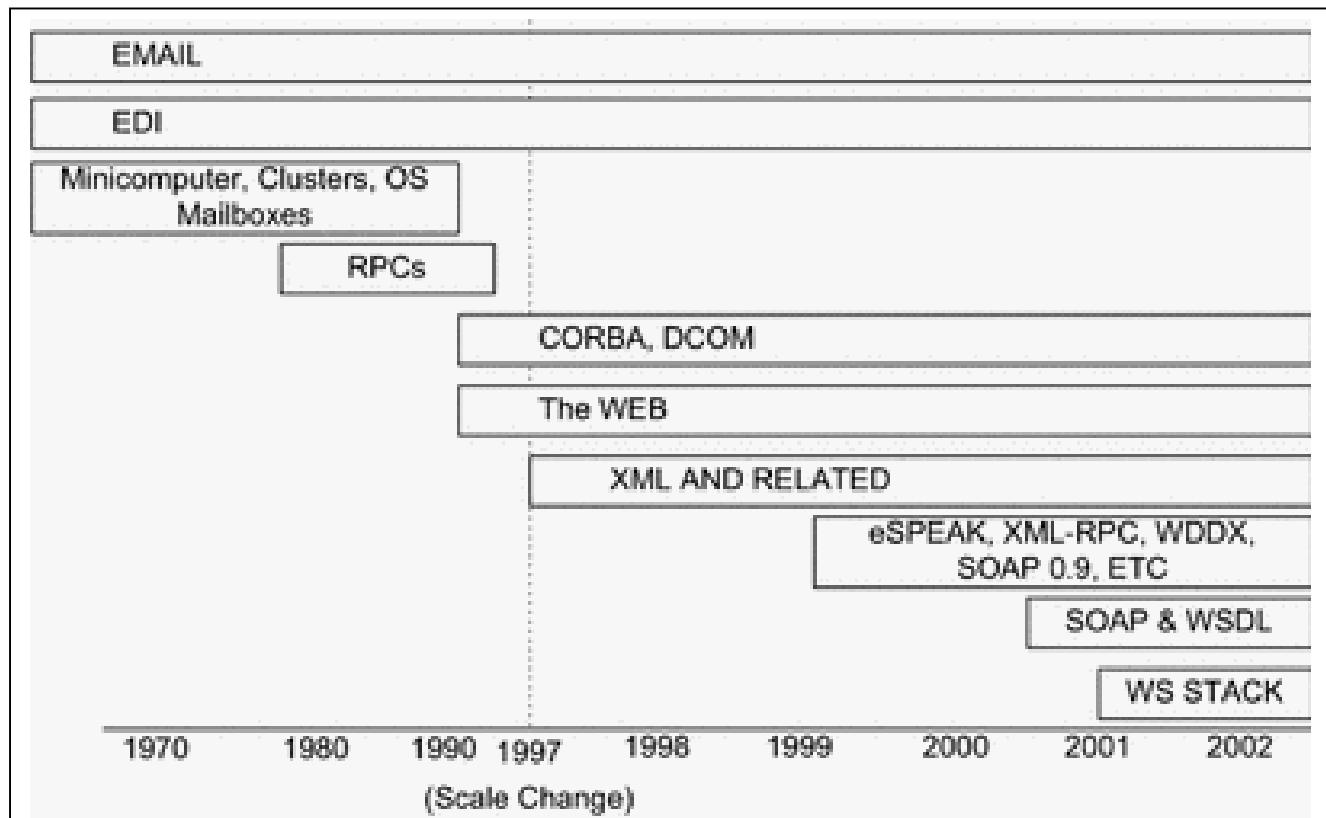


Figura 83. Evolución de los sistemas distribuidos

Arriba se muestra un esquema muy simplificado de la evolución de los sistemas distribuidos. En 2004 todavía conviven muchas de las tecnologías de la figura. Pese a todas las inversiones en tecnologías distribuidas enfocadas al comercio electrónico, el correo electrónico viene a ser el sistema más usado para encargar pedidos, enviar facturas, enviar albaranes, etc.

La tecnología RPC fue el antecesor directo de CORBA y se podría considerar como una tecnología distribuida procedural, no orientada a objetos. Actualmente ha caído en desuso.

4.2. Aplicaciones y sistemas distribuidos

Se llama **sistema distribuido** a aquel cuyos componentes de hardware y software, ubicados en diferentes máquinas conectadas en red, se comunican sólo mediante el intercambio de mensajes. Internet es un buen ejemplo de sistema distribuido, con una más que aceptable tolerancia a fallos de las redes que la componen.

Se llama **aplicación distribuida** a la que se ejecuta en un sistema distribuido; en consecuencia, tiene repartidas sus funciones o servicios entre varias máquinas de una red. En una aplicación así, el código y los datos se reparten entre distintas máquinas. Muchas aplicaciones distribuidas se usan para lograr dos metas: a) compartir recursos (disco duro, RAM, impresoras, etc.); y b) repartir la carga de trabajo entre varias máquinas, en función de la capacidad de cada una.

De acuerdo con la *European Conference on Object Oriented Programming* (ECOOP) de 1996, un componente se define como “una unidad de composición con interfaces contractuales específicas y dependencias de contexto explícitas”. De acuerdo con *Component Software: Beyond Object Oriented Programming* [C. Szyperski, 1998], “Un componente de software puede desarrollarse de manera independiente y puede ser usado por terceras partes para integrarlo mediante composición a sus sistemas”. Un componente distribuido es una unidad binaria que puede instalarse y usarse en cualquier máquina de una red. No es, por tanto, una biblioteca de clases, una biblioteca dinámica o un conjunto de código.



Figura 84. Algun día, algún día desarrollar aplicaciones será como ensamblar componentes electrónicos.

Los componentes distribuidos favorecen la reutilización del software, tal y como los circuitos integrados y *chips* favorecen la reutilización de los circuitos electrónicos. Un componente distribuido puede ser usado por otros componentes y aplicaciones, distribuidos o no. Un ejemplo aclarará esto: supongamos que hemos desarrollado un componente de cálculo matemático CalcMat que necesita ejecutarse sobre máquinas con muchos recursos (RAM, procesadores, disco duro, etc.). Si ese componente se instalara en cada máquina de una red, muchas no cumplirían los requisitos para su uso: en ellas, el componente no podría ejecutarse o su funcionamiento sería pésimo.

En cambio, si CalcMat se creara como un componente distribuido, podría ser usado – mediante los protocolos pertinentes– por muchas máquinas, cumplieran o no los requisitos para ejecutarlo en modo local. En este caso, CalcMat vendría a ser un servicio de red, como el de transferencia de archivos (basado en FTP) o el de acceso a páginas web (basado en HTTP). Un usuario podría llamar al servicio CalcMat desde un ordenador doméstico y enviar sus cálculos. CalcMat repartiría el esfuerzo de cálculo necesario entre las máquinas donde se ejecuta el componente y devolvería los cálculos al usuario. Sería equivalente, en cuanto a proceso, a llamar a una página web desde un navegador.

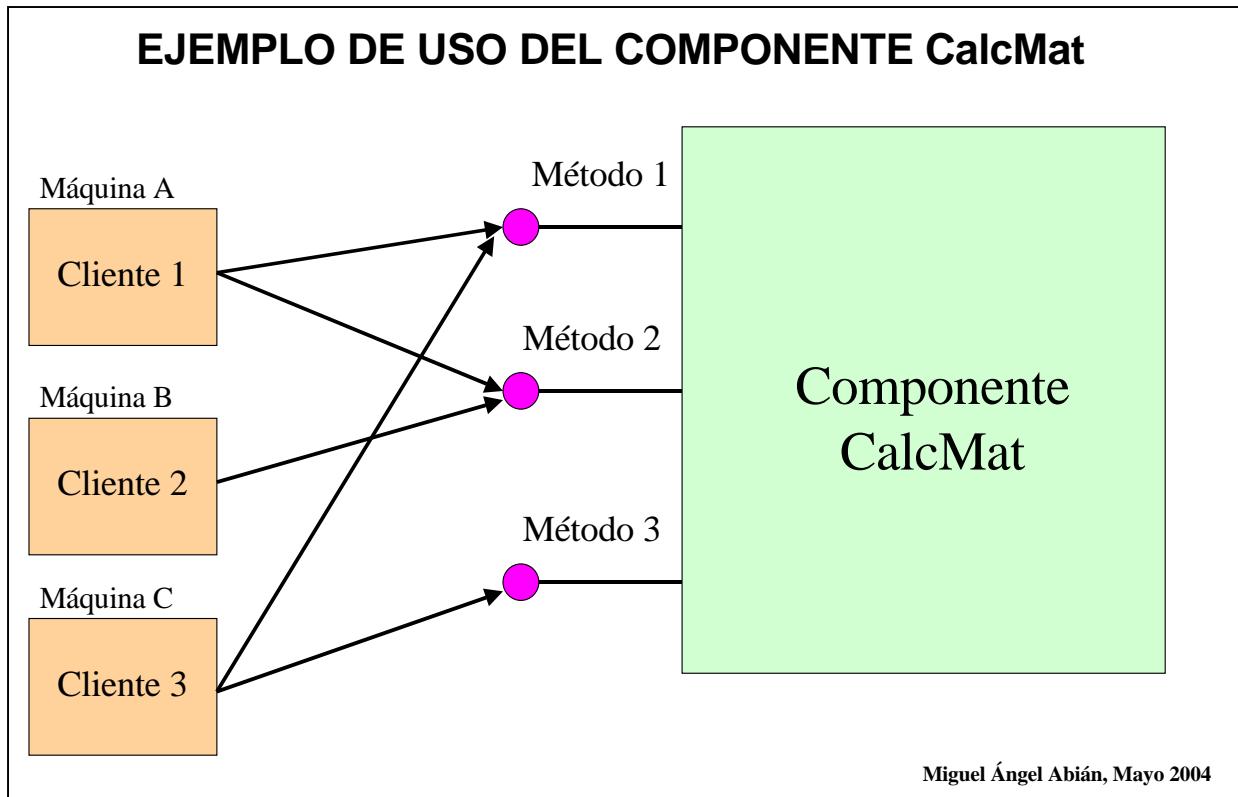


Figura 85. El componente CalcMat en funcionamiento

Las posibilidades de las aplicaciones distribuidas son infinitas: uno puede imaginar servicios financieros, médicos, de meteorología, de comercio electrónico... Imaginemos, por poner un solo ejemplo, que una empresa desarrolla un componente distribuido para llevar el seguimiento del estado actual de un pedido, de modo que en cada momento se puede saber el lugar donde se encuentra, el medio de transporte en que viaja, si se esperan retrasos, etc. Esa empresa puede ofrecer el componente a las tiendas virtuales de Internet para que no tengan que preocuparse de llevar el control de los envíos. Si un usuario quisiera saber el estado de su pedido, se conectaría a la tienda virtual donde lo hizo, se identificaría y solicitaría la información concerniente a su pedido; la tienda virtual llamaría, mediante los protocolos adecuados, al componente distribuido y mostraría al usuario la información que aquél le devolviera. Miles de compradores podrían estar usando a la vez el componente distribuido, aunque no lo supiesen. Por lo que sé, UPS usa un sistema similar al descrito.

Diseñar e implementar sistemas distribuidos obliga a considerar muchos factores que se obvian cuando se hace lo propio con aplicaciones que se ejecutan en una sola máquina. Éstos son algunos factores que obligatoriamente deben tenerse en cuenta a la hora de diseñar sistemas distribuidos:

- La heterogeneidad de los elementos usados: hardware, sistemas operativos, redes, protocolos...
- La ejecución de aplicaciones o procesos concurrentes.
- El tratamiento de los fallos: cada componente, sea de hardware o de software, puede fallar, y se necesita que cada componente conozca todos los posibles fallos que pueden suceder en los otros. Por añadidura, se presenta el problema de que muchos fallos no son reproducibles.
- La escalabilidad del sistema, entendiendo como tal la capacidad de que el coste de añadir nuevos usuarios al sistema sea constante en relación con los recursos que requeriría su incorporación. Un sistema escalable se adaptará con facilidad a futuros aumentos de su carga de trabajo.
- La seguridad del sistema. Lo usual es que exista información que deba transmitirse codificada, así como servicios restringidos a ciertos usuarios.
- El grado de apertura del sistema. Interesa que cada parte sea lo más estándar posible y que su documentación sea clara.
- La localización de los servicios ofrecidos. Debe saberse cómo localizarlos.
- La persistencia de los datos. Debe especificarse dónde y cómo se guardan.

Muchas aplicaciones distribuidas, si bien no todas, pueden diseñarse e implementarse siguiendo el modelo cliente-servidor. Deben tomarse los términos *cliente* y *servidor* en el sentido general dado en el apartado 2.3.

La programación orientada a objetos (POO) simplifica el desarrollo de software mediante la agrupación –en clases– de datos y operaciones relacionados, y mediante la separación clara entre interfaz e implementación. En general, los programas OO muestran estructuras recurrentes que promueven la abstracción, el encapsulado, la modularidad, la flexibilidad. La motivación para usar la POO en el diseño e implementación de aplicaciones distribuidas reside en lo decisivas que resultan esas características para crear sistemas robustos y flexibles. Si bien pueden escribirse componentes no basados en objetos, la tendencia dominante hoy día consiste en construir los componentes mediante POO.

En las aplicaciones distribuidas que usan objetos, los objetos que “viven” en un anfitrión pueden llamar a métodos de objetos residentes en otros anfitriones, ya sea para trabajar con ellos interactivamente o para delegar en ellos parte de sus tareas. Los objetos que residen en un cierto espacio de direcciones de memoria (conjunto de posiciones de memoria asociadas a un proceso) de un anfitrión son objetos *locales*; los que residen en otros espacios de direcciones son objetos *remotos*. Estos últimos espacios de direcciones pueden ubicarse en distintos anfitriones; es más, ésa constituye la situación habitual en las aplicaciones distribuidas. *Remoto* y *local* no son términos absolutos, pues un objeto remoto siempre es local para el espacio de direcciones que lo alberga. Siempre deben interpretarse estos términos como relativos a una determinada comunicación entre objetos, no como asociados permanentemente a la ubicación de los anfitriones o de los objetos.

Las llamadas a métodos de objetos remotos se conocen como *llamadas a métodos remotos* o *llamadas remotas*. En ellas, un objeto almacenado en un espacio de direcciones de un anfitrión llama a métodos de objetos almacenados en otros espacios de direcciones (estén o no en el mismo anfitrión). Por contraposición, las *llamadas a métodos locales* o *llamadas locales* son llamadas a métodos entre objetos que “viven” en un mismo espacio de direcciones (es decir, que forman parte de un mismo proceso).

Se denomina *objeto cliente* al objeto que llama a un método remoto, y *objeto servidor* a aquel cuyos métodos son invocados. En este apartado usaré de forma intercambiable *objeto cliente* y *objeto local*, así como *objeto remoto* y *objeto servidor*, verdad es que un objeto local puede no ser cliente y que uno remoto puede no ser servidor, pero entonces carecerían de interés para el estudio de las aplicaciones distribuidas. Asimismo, en lo que sigue usaré indistintamente *objeto cliente* y *cliente*, así como *objeto servidor* y *servidor*, cuando me interese llamar la atención sobre los procesos y no sobre los objetos, antepondré la palabra *proceso* (*proceso servidor*).

De modo general, en una aplicación distribuida en ejecución, la parte que actúa como cliente estará formada por un conjunto de objetos (donde estarán los objetos cliente); y la parte que actúa como servidor, por otro conjunto de objetos (donde estarán los objetos servidor).

En la figura 86 se muestra un ejemplo de integración de las tecnologías de objetos distribuidos con una aplicación cliente-servidor web.

EJEMPLO DEL USO DE OBJETOS DISTRIBUIDOS EN UNA APLICACIÓN NAVEGADOR-SERVIDOR WEB

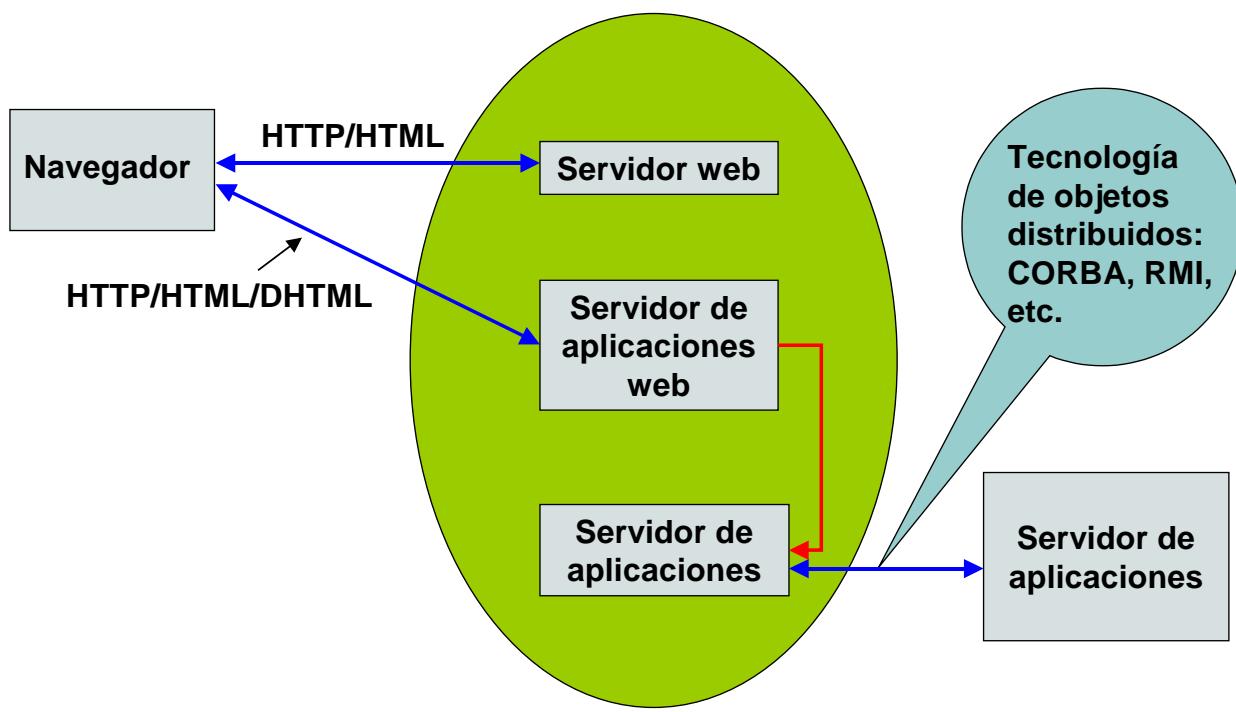


Figura 86. Ejemplo de integración entre las tecnologías distribuidas y la navegación web

Al lector interesado en conocer a fondo los sistemas distribuidos le recomiendo estos libros: *Distributed Systems, Concepts and Design 3rd Edition* [G. Coulouris et al., 2001], *Advanced CORBA Programming with C++* [M. Henning, 2001] y *UNIX Distributed Programming* [M. Henning y S. Vinoski, 1999]. Son libros densos, que requieren paciencia y un buen conocimiento de C/C++ y de la estructura de los sistemas operativos actuales; con todo, vale la pena leerlos si uno quiere manejar con soltura sistemas distribuidos. En cuanto a libros del estilo *Aprenda CORBA en 24 horas*, le aseguro que no podrán cumplir la promesa del título salvo que usted conozca muy bien la estructura de los sistemas distribuidos y sepa programar más que competentemente en C/C++; y, en ese caso, ¿para qué los necesita?

Hace poco (en abril de 2004), se publicó el libro *Network Distributed Computing: Fitscapes and Fallacies* [Max K. Goff, 2004]. Es este un libro muy recomendable para cualquiera que desee conocer el estado actual de los sistemas distribuidos y sus expectativas. Dedica bastante espacio a las tecnologías de Sun, en parte porque la actividad profesional de Goff se ha centrado en ellas y en parte porque “las observaciones y principios contenidos de aquí en adelante trascienden cualquier agenda o aproximación específica de una empresa; la tesis de Church-Turing se aplica también a la computación distribuida en redes”.

4.3. Dos ejemplos de arquitecturas distribuidas: CORBA y los servicios web

He escogido dos ejemplos de arquitecturas distribuidas: CORBA y los servicios web. Ambas son las más populares y extendidas, y proporcionan un buen comienzo para ver cómo pueden solucionarse los problemas inherentes a las aplicaciones distribuidas.

Los servicios web han sufrido tal exceso de promoción y publicidad que se hace necesario entender qué aportan con respecto a tecnologías anteriores como CORBA. Los ingenieros del software suelen decantarse por CORBA, mientras que los programadores prefieren usar servicios web. Los motivos que suelen dar unos y otros para su predisposición hacia una u otra tecnología corresponden más a las respuestas a un cuestionario religioso que a otra cosa.

Dado que no tengo dios ni amo en cuanto a tecnologías informáticas, daré al final del subapartado mis propias opiniones sobre los servicios web. Con ellas, no trato de dar un trato de favor a CORBA, como comprobará si continúa leyendo (mis objeciones a esta tecnología aparecen en los siguientes párrafos y en el apartado 6); sino señalar los fallos y carencias que los comerciales y publicistas de los servicios web parecen olvidar y callar, en una amnesia y un mutismo cuando menos sospechosos. Como CORBA se explicará con mucho más detalle en el apartado 6, si el lector desea comparar ambas tecnologías disponiendo de más criterios de juicio puede volver aquí cuando termine dicho apartado.

CORBA (*Common Object Request Broker Architecture*: arquitectura común de intermediación de solicitudes de objetos) es una especificación abierta e independiente del vendedor, desarrollada por el OMG (*Object Management Group*: grupo de gestión de objetos) para diseñar aplicaciones distribuidas mediante objetos.

El OMG es un consorcio internacional con más de 850 miembros, entre los cuales se encuentran empresas como IBM, Sun, Boeing, Alcatel, e instituciones y universidades como la NASA, INRIA y LIFL. Este consorcio funciona como una organización no comercial. Se fundó en 1989 como una organización estadounidense sin ánimo de lucro, con representación en todo el mundo. El OMG cuenta con una plantilla bastante reducida porque no se dedica a construir o vender software, sino a publicar especificaciones o normas (como el Centro Europeo de Normalización o CEN). Las especificaciones publicadas por el OMG pueden ser implementadas por cualquier empresa u organización, sin pagar al OMG derechos de autor o licencias. Las empresas fabricantes de software tienen el control de sus implementaciones basadas en especificaciones del OMG, pero carecen de derechos sobre las especificaciones. Sólo el OMG puede modificar las especificaciones que define o decidir cuál será su evolución. Si una empresa miembro del consorcio hace una propuesta para una especificación y es aceptada, la propuesta queda como propiedad del consorcio, no de la empresa que la ha hecho.

El objetivo último de CORBA consiste en dar a quienes sigan la especificación la capacidad de que un proceso pueda llamar a procesos que se ejecuten en otros anfitriones, con independencia de las plataformas y lenguajes usados. La versión 1.1 de CORBA se lanzó en 1991.

CORBA es independiente del lenguaje o lenguajes utilizados para implementar las aplicaciones distribuidas; y, por ende, permite ahora trabajar con código que ya existía y permitirá en el futuro comunicarse con lenguajes que todavía no existen (siempre que fueren compatibles con CORBA). CORBA se usa en empresas químicas, de comercio

electrónico, aeroespaciales, de finanzas, de recursos humanos, de investigación, de telecomunicaciones, de defensa, etc. Entre las grandes empresas que usan CORBA están AT&T, Lucent, Nokia y Boeing.

Con todo, la compleja estructura interna de CORBA y el elevado coste de sus implementaciones han supuesto un pesado lastre para su aceptación en el mercado. Pese a sus indudables cualidades y pese a contar con el respaldo del OMG, apenas ha conseguido introducirse en las pequeñas y medianas empresas. Otros dos problemas que han obstaculizado la introducción de CORBA en el mundo empresarial han sido sus deficientes primeras implementaciones, que daban lugar a problemas de interoperabilidad entre productos de distintos vendedores, y la nula integración de los primeros productos CORBA con los cortafuegos, presentes en cualquier red empresarial o corporativa. Existen desde hace tiempo implementaciones que solventan los dos problemas; pero nada ha podido remediar la desconfianza inicial de muchas empresas hacia esta tecnología, consecuencia de implementar especificaciones en las que se debería haber trabajado más antes de ponerlas a disposición de los fabricantes de software.

En el apartado 6 se verá una exposición de la estructura de CORBA y se explicará cómo puede usarse desde Java. Por ello, prefiero no alargarme aquí en torno a CORBA.

Los **servicios web**, tan en boga estos días, son componentes de arquitecturas distribuidas que usan sus propias interfaces entre programas y sus propios protocolos y servicios de registro para que cualquier aplicación de una plataforma pueda emplear servicios ofrecidos por otras plataformas. Se usa XML (*Extensible Markup Language*: lenguaje extensible de formato) para describir el servicio, para escribir los mensajes que genera o recibe y para registrarlos (una vez registrado, estará a disposición de quien desee usarlo).

Un servicio web es un componente distribuido que ejecuta procesos y ofrece a sus clientes una interfaz bien definitiva y accesible mediante protocolos de Internet. Las peticiones y las respuestas son mensajes XML (véase la figura 87).

Para emplear un servicio web se necesita hacer pública la interfaz que ofrece a los clientes (descrita en WSDL: *Web Service Description Language*), la cual incluye los argumentos y el tipo de retorno de cada método, y dar una manera de localizar el servicio y su interfaz (mediante UDDI: *Universal Description Discovery and Integration*). Los mensajes siguen un protocolo basado en XML: el SOAP (*Simple Object Access Protocol*), que define la sintaxis de los mensajes (envoltura, cabecera y cuerpo), la codificación de los datos y las convenciones para representar llamadas remotas.

Los servicios web simplifican mucho el desarrollo de sistemas distribuidos: cada componente del sistema puede desarrollarse con el lenguaje y la plataforma que uno desee, y luego se componen mediante servicios web.



Figura 87. Arquitectura de los servicios web. Figura de Sandra Aguirre

En la figura anterior hay tres elementos:

- **UDDI:** Permite obtener listas de los servicios disponibles y localizarlos de manera rápida. Una vez localizado un servicio mediante UDDI, puede usarse la interfaz pública del servicio. UDDI es un servicio web que se usa para encontrar dinámicamente otros servicios web.
- **WDSL:** Permite describir las interfaces de los servicios, de manera que las aplicaciones puedan utilizarlas para saber cómo interoperar con los servicios (qué campos XML deben tener las peticiones, etc.). Las interfaces descritas por WDSL tienen una apariencia muy similar a las interfaces de Java.
- **SOAP:** Incluye los mecanismos para la ejecución de llamadas remotas entre aplicaciones.

LLAMADA Y RESPUESTA A UN SERVICIO WEB MEDIANTE SOAP

Llamada a un servicio web (SOAP)

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<SOAP-ENV:Envelope ... >
  <SOAP-ENV:Body ... >
    <VerPrecioProducto ...>
      <codigo>GRP0112</codigo>
    </VerPrecioProducto>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Argumento de la llamada

Respuesta del servicio (SOAP)

```
<?xml version="1.0" encoding="iso-8859-1" ?>
<SOAP-ENV:Envelope ... >
  <SOAP-ENV:Body ... >
    <VerPrecioProductoRespuesta ...>
      <precio divisa="euro">62800</precio>
    </VerPrecioProductoRespuesta>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Valor devuelto por la llamada

Figura 88. Llamadas y respuestas con SOAP. Figura de Sandra Aguirre

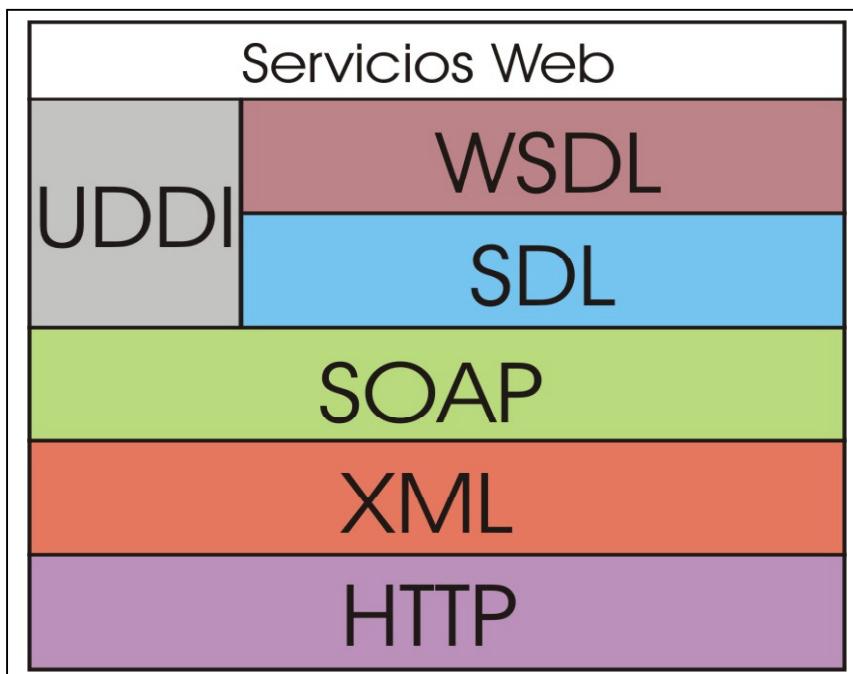


Figura 89. Los servicios web se construyen sobre HTTP

La principal diferencia entre CORBA y los servicios *web* estriba en que los últimos no trabajan con objetos, sino con mensajes. Ignoro por qué SOAP significa “protocolo sencillo de acceso a objetos”, cuando no trabaja con objetos (en las últimas especificaciones de este protocolo no aparece ya el significado de las siglas). A diferencia de CORBA, los servicios *web* no definen un mecanismo de persistencia: se deben usar los mecanismos que proporcionan los lenguajes con los que se escriben las aplicaciones (C++, Java, C#, etc.).

Desde una perspectiva crítica, no se puede decir que los servicios *web* sean muy originales: obedecen al modelo cliente-servidor más simple que pueda haber (un cliente y un servidor, sin cambio de papeles). Su única novedad consiste en usar XML. Aunque se les ha visto como los sucesores de CORBA (y a veces como sus enterradores), la realidad es que ambas tecnologías son muy distintas: CORBA está orientada al desarrollo de aplicaciones seguras y escalables, mientras que los servicios *web* están orientados a ofrecer acceso a aplicaciones CORBA, J2EE, .Net, RMI.

Los servicios *web* permiten usar aplicaciones ya escritas, para simplificar el acceso de los clientes a ellas; pero no se encargan de la implementación de los servicios, que se hace con los lenguajes tradicionales (C, C++, Java, etc.). Se puede acceder a las aplicaciones CORBA mediante un servicio *web*, como hace AT&T para dar un punto de entrada cómodo a los usuarios; pero no se pueden escribir aplicaciones CORBA mediante los protocolos de los servicios *web*.

Si el lector ha trabajado con C#, el lenguaje nativo de la plataforma .Net, quizá se pregunte qué sentido tiene usar CORBA o RMI (esta última tecnología se verá en el siguiente subapartado), cuando uno dispone en C# de la palabra clave `[WebMethod]`. Esta palabra, delante de un método, permite exponer un servicio *web* basado en dicho método. Para llamar al método desde una máquina remota, Visual Studio .Net genera un código que actúa de intermediario, el cual debe colocarse en el anfitrión remoto. Más sencillo imposible.

Pero que algo sea sencillo no quiere decir que sea bueno: con `[WebMethod]` no se pueden construir aplicaciones que sobrevivan a una buena dosis de realidad. CORBA (y, en menor medida, RMI) es complicado porque se creó para fabricar aplicaciones industriales, donde palabras como seguridad, escalabilidad, velocidad y eficacia definen las santas virtudes de los grandes sistemas empresariales. Si la especificación de CORBA es tan larga (más de 1.000 páginas en su última versión) no es por casualidad: define muchas formas de funcionamiento que pueden marcar la diferencia entre que una aplicación sea capaz o no de manejar miles de transacciones por segundo.

Trabajar con C# y `[WebMethod]` tiene sentido si uno quiere construir pequeñas aplicaciones o si quiere ver cómo se registra y se llama a un servicio *web* (con Visual Studio .Net es casi trivial hacerlo); pero no si uno quiere construir aplicaciones que atiendan muchas peticiones, que admitan modificaciones dinámicas o que deban cumplir unos requisitos mínimos de fiabilidad y seguridad. Para esas aplicaciones, CORBA sigue siendo una opción muy válida y de mucha solera.

No negaré que CORBA es complicado, pero su complejidad no es innecesaria: las aplicaciones distribuidas no son sencillas (véase la página 177). Las personas del OMG que trabajan en CORBA no son tecnócratas con tendencias sádicas hacia los desarrolladores ni creen en el interés técnico e intelectual de la complejidad. Resulta divertido imaginárselos como personas de ojos vidriosos que escuchan *Kiss the boot of shiny, shiny leather / Shiny leather in the dark* mientras se devanan los sesos intentando complicar un poco más la vida de los programadores; pero las cosas no son

así: tienen que considerar todas las situaciones en las que puede verse envuelta una aplicación distribuida de tipo empresarial.

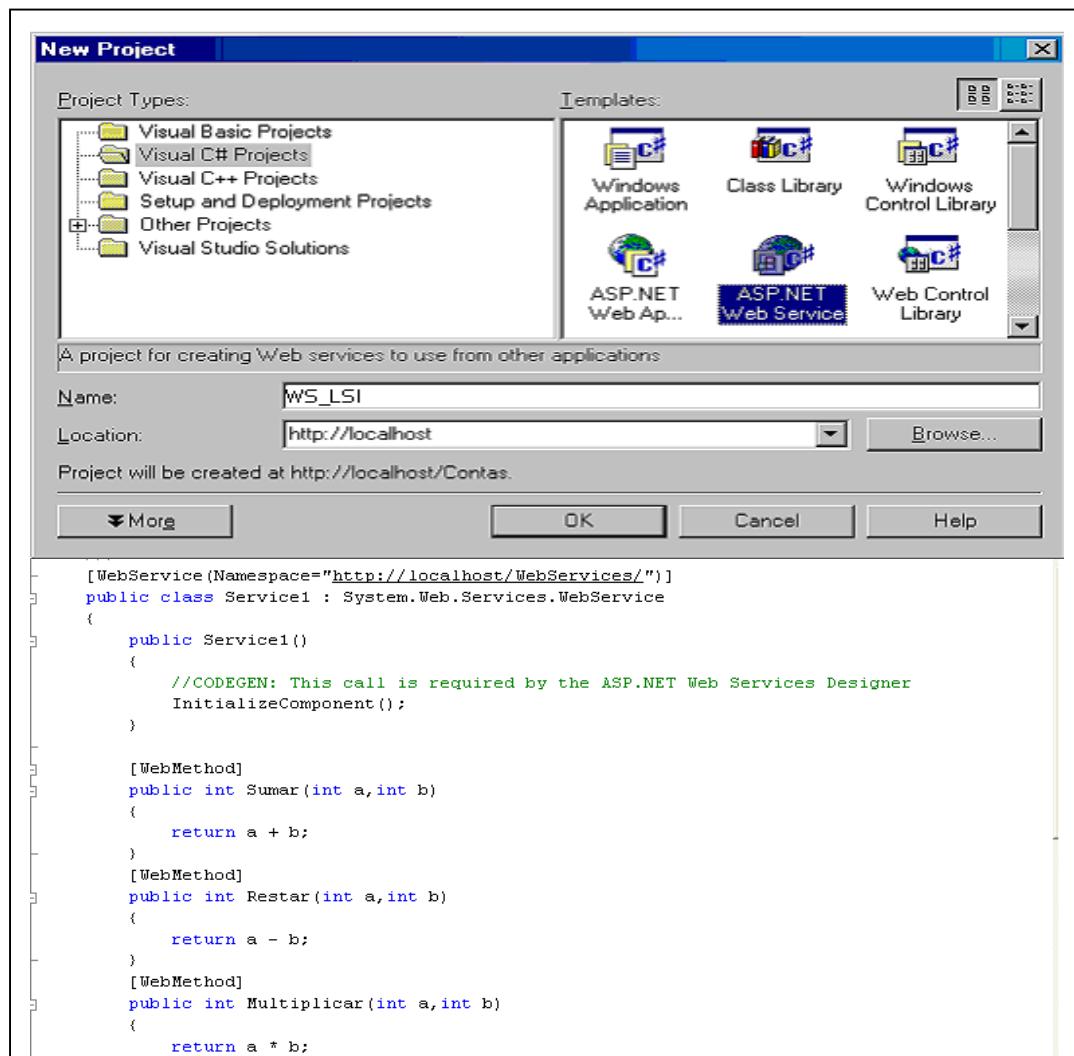


Figura 90. Código en C#. No intente usarlo para escribir aplicaciones empresariales. Como puede suponer, los ingenieros de CORBA, de Sun y de muchas empresas no han gastado años de trabajo y millones de dólares para que todos los problemas de las llamadas remotas se resuelvan con [WebMethod]

Imagino que CORBA sobrevivirá en la parte de servidor de las aplicaciones distribuidas, en *mainframes* y servidores UNIX, donde la escalabilidad y la eficacia son imprescindibles; y que los servicios *web* se usarán en el lado del cliente, donde la configuración de CORBA nunca ha sido tan fácil como debería haber sido. No creo que los servicios *web* sean un punto de partida para construir nuevas aplicaciones, mejores y más eficaces. Más bien considero que definen una aproximación sencilla para resolver los problemas de interoperabilidad entre aplicaciones escritas en distintos lenguajes y que se ejecutan en diferentes plataformas. CORBA tiene también soluciones para esos problemas (comunes a todos los sistemas distribuidos), pero no son sencillas.

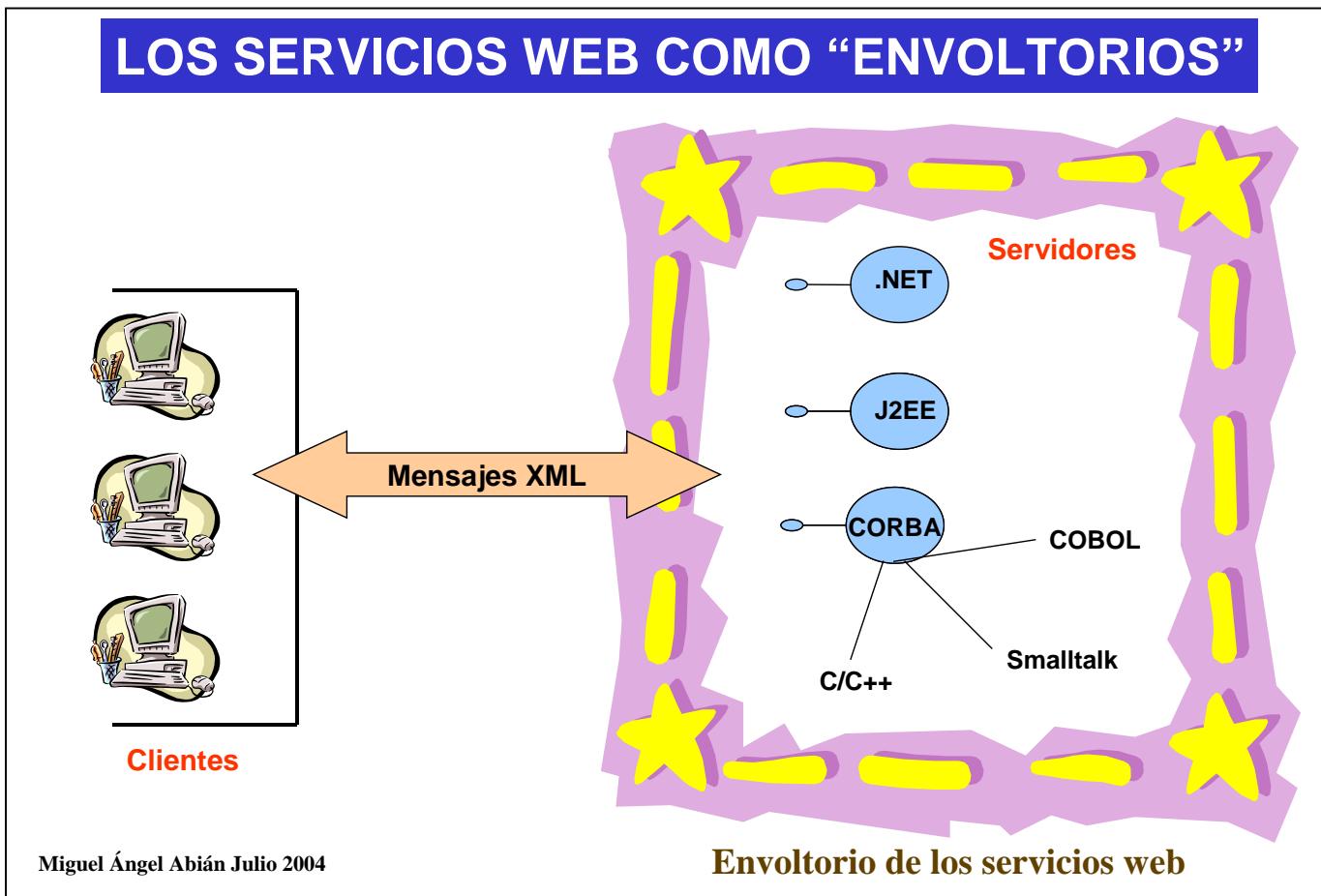


Figura 91. Integración entre sistemas heterogéneos mediante servicios web

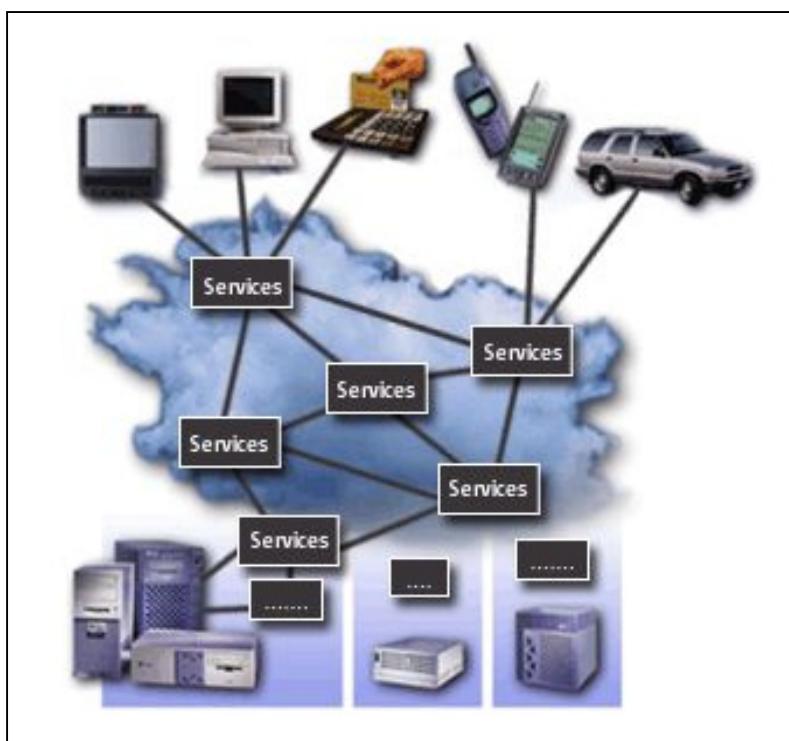


Figura 92. Más integración entre sistemas heterogéneos mediante servicios web

Cuando me siento pesimista, pienso que los servicios *web* constituyen –con muchas matizaciones, desde luego– un paso hacia atrás con respecto a CORBA o a RMI. La falta de orientación a objetos recuerda a las RPC (*Remote Procedure Call*: llamada a procedimientos remotos), que eran comunes cuando los lenguajes procedurales dominaban la programación. Al no existir polimorfismo, no se pueden hacer comprobaciones o conversiones dinámicas de tipos. De hecho, tampoco se pueden hacer comprobaciones estáticas: un cliente SOAP sólo averigua que ha enviado un argumento de un tipo incorrecto cuando el servidor SOAP que lo recibe lo rechaza. Enviar datos a través de redes para que el servidor descubra que no son del tipo adecuado es un desperdicio de tiempo y de ancho de banda, desperdicio acrecentado porque los datos en un mensaje XML no constituyen más que una pequeña parte del mensaje (el resto corresponde a información XML).

Ni siquiera el protocolo HTTP es el más adecuado para los procesos síncronos de tipo petición-respuesta: este protocolo fue creado para abrir una conexión, intercambiar datos y cerrar la conexión; no para esperar pasivamente –quizás durante largos períodos– respuestas. HTTP dista mucho de ser una solución óptima para dichos procesos, pero se usa porque es simple. Aparte de la falta de seguridad, fiabilidad y persistencia de soluciones como [WebMethod], también hay que señalar que consumen muchos recursos al usar un protocolo que, en definitiva, no es el más apropiado. Mantener una conexión HTTP durante milisegundos o segundos no es problema para ninguna máquina, pero mantener cientos o miles de conexiones simultáneas durante minutos u horas sí lo es.

Si piensa que esconde algún motivo poco confesable en contra de C# o de los servicios web, le sugiero que haga la siguiente prueba: escriba un método remoto con [WebMethod] que permanezca varios minutos haciendo cálculos antes de devolver una respuesta (con un bucle dentro del método, por ejemplo). Después, haga que unos cuantos clientes (diez, pongamos por caso) accedan a él simultáneamente (mediante hilos, por ejemplo). Por último, vaya aumentando el número de clientes: cien, doscientos, mil, dos mil, etc. Si usa un PC normal y corriente, estoy seguro de que le sorprenderán los resultados. Como puede intuirse, la situación empeora si se consideran los tiempos de latencia de las redes y los retrasos por reenvíos de paquetes defectuosos.

Si escucha que los servicios *web* pueden abrir de forma automática los sistemas empresariales a otros sistemas (de clientes, de socios, etc.), tal y como dos personas aprenden conversando lo que una puede ofrecer a otra, tenga por seguro que no le están hablando con la voz de la razón. Para entender una interfaz, hay que saber cuáles son sus significados para las partes implicadas y si son coincidentes (véase la nota sobre ontologías en la página 25). Por ahora, sólo las personas pueden adquirir ese tipo de conocimiento.

Un ejemplo muy sencillo bastará para poner los puntos sobre las íes. Suponga que un servicio *web* descubre automáticamente que una organización ofrece un servicio *web* con una interfaz del tipo double calcularAmortizacion1987 (double importe). ¿De qué le servirá al usuario final esa información si no sabe qué clase de amortización se calcula (puede ser financiera o de un bien) o que algo especial pasó en 1987? Mientras un ser humano no conozca el significado de la interfaz, de bien poco servirá ésta. Por ahora, el uso automático de agentes que descubran y “comprendan” las interfaces de los sistemas empresariales es químérico.

Suele argumentarse que CORBA es una tecnología complicada (lo cual es innegable); y que los servicios *web* son mucho más simples porque se basan en XML. Ante esa afirmación, una balanza que midiera la simplicidad caería del lado de los servicios *web*; pero otra que midiera la veracidad de la afirmación se rompería: los

servicios web usan XML, pero también usan muchas tecnologías asociadas, como XSL/XSLT ¿Acaso es sencillo trabajar con XSL/XSLT?

Algunos de los errores que se cometieron con CORBA se están repitiendo con los servicios *web*, en lo que parece otra confirmación de que lo único que nos enseña la Historia es que la Historia no nos enseña nada. Por ejemplo, las API disponibles para SOAP carecen de interoperabilidad entre sí (como sucedía con los primeros ORB de CORBA). Un programador que desarrolle un servicio *web* con una herramienta basada en una determinada implementación de SOAP no podrá llevarse el código a otra herramienta basada en otra implementación de SOAP (aunque el lenguaje de programación no varíe): se verá obligado a volver a escribir el código en la segunda herramienta, porque los métodos de cada API son propios de ella. Todo parece interoperable cuando las cosas se mantienen en el aire; pero cuando se ponen en el suelo y llega la hora de escribir código, aparecen problemas de interoperabilidad y de portabilidad del código entre los productos, bibliotecas y *frameworks* SOAP.

Así las cosas, me pregunto qué habría sucedido si Microsoft e IBM hubieran apoyado a CORBA con la misma energía con que promueven los servicios *web*. Es una pena que el presupuesto del OMG para mercadotecnia fuera tan reducido.

5. RMI: Llamando desde lugares remotos

5.1. Fundamentos de la RMI: el modelo de objetos distribuidos de Java

Los *sockets* de Java, basados en TCP/IP, se usan para programar aplicaciones distribuidas (en el apartado 2 vimos varios ejemplos simples, y veremos más en próximos apartados); pero a veces se necesitan enfoques más complejos, más cercanos a CORBA. En esos casos, la API RMI de Java (*Remote Method Invocation*: invocación de métodos remotos o ejecución de métodos remotos) es la mejor opción, salvo que se quiere programar con *sockets* gran parte de las funciones que la interfaz de métodos remotos ya incluye. Internamente, esta interfaz utiliza *sockets* TCP por defecto. Como los *sockets* están más cercanos al sistema operativo que la RMI, consumen menos recursos y son más fáciles de integrar con redes protegidas por cortafuegos. En contrapartida, la RMI es más fácil de integrar con otras API de Java (JDBC, por ejemplo) y con sistemas heredados.

Con la RMI, un objeto de Java puede señalarse como remoto, de forma que los procesos remotos de las aplicaciones distribuidas puedan acceder a él como si fuera un objeto de Java normal.

De acuerdo con David Curtis, director de tecnologías de plataforma en el OMG, la RMI de Java es una *tecnología de programación*, mientras que CORBA es una *tecnología de integración*. Ambas llevan grabadas en sus objetivos una misma leyenda: “**INTEROPERABILIDAD**”, si bien RMI es una tecnología monógama en cuanto al lenguaje y CORBA es polígama.

La API RMI, incluida en todos los JDK de Java desde la versión 1.1 (1995), incluye su propio modelo de objetos distribuidos, optimizado para las características de Java. De este hecho se derivan tres importantes consecuencias:

- El modelo de objetos distribuidos de Java no coincide con el de CORBA, que es independiente del lenguaje de programación usado. CORBA se puede usar desde Java, si bien se hace necesario traducir los objetos de un modelo a otro, y viceversa.
- Cualquier plataforma para la que exista un JDK (1.1 o posterior) puede usar la RMI.
- La RMI es mucho más sencilla y eficaz que CORBA si se va a usar Java como lenguaje de desarrollo, pues se adapta como un guante al modelo de objetos de Java (mucho más simple que el de CORBA) y saca partido de todas las cualidades de Java. Si en una aplicación distribuida se va a emplear sólo Java, la RMI tiene muchas ventajas sobre CORBA. Si hay partes escritas en C o C++, debe valorarse la opción de CORBA, siempre que la aplicación sea lo bastante compleja.

En el modelo de objetos distribuidos de Java, un *objeto remoto* es aquel cuyos métodos pueden llamarse desde otra máquina virtual Java (MVJ), la cual puede ejecutarse en otro anfitrión o nodo. En consecuencia, un objeto remoto ubicado en un proceso puede recibir llamadas desde otros procesos (distintos procesos se ejecutan en distintos espacios de direcciones). Una *clase remota* es cualquier clase cuyas instancias son objetos remotos. Dentro del espacio de direcciones de la MVJ donde se crea un objeto remoto, éste es un objeto normal y corriente: puede usarse como cualquier objeto de Java.

Una *llamada a un método remoto* o una *llamada remota* es una llamada a un método de un objeto remoto desde un espacio de direcciones donde éste no reside. Dos objetos pertenecientes a distintas MVJ pueden interaccionar mediante llamadas remotas. Las *llamadas a métodos locales* o *llamadas locales* son llamadas entre objetos que residen en una misma MVJ. Todo clase remota en Java implementa a una o más interfaces remotas en las que se declaran los métodos remotos (uso aquí *interfaces* en el sentido de construcciones del lenguaje Java definidas mediante la palabra reservada *interface*).

Java permite trabajar con objetos situados en anfitriones remotos como si estuvieran en el local, con la misma sintaxis que tienen las llamadas locales y de un modo absolutamente transparente para el programador. La palabra clave es *como*: tras las bambalinas sólo se oye el zumbido sordo y constante de los datagramas que van y vienen, sabedores de que el tiempo juega en contra de ellos; delante de ellas, RMI representa una obra teatral en la que todo ese vaivén de datagramas se muestra en forma de límpidas llamadas a interfaces remotas. Incluso hace posible que en un anfitrión se puedan crear objetos de forma dinámica y que los demás anfitriones puedan usar los métodos de los nuevos objetos, aun cuando la aplicación distribuida nunca hubiera tratado antes con esos objetos.

Otra característica del modelo de objetos distribuidos de Java es que los objetos locales no llaman directamente a los métodos de los objetos remotos: **utilizan las interfaces remotas de estos últimos**. Los clientes no necesitan más que la interfaz remota para llamar a los métodos de los objetos remotos. La interfaz local de un objeto puede no coincidir con la interfaz remota. Asimismo, un objeto remoto puede presentar distintas interfaces remotas, dependiendo del modo de acceso. Así, un objeto remoto puede ofrecer distintas interfaces remotas a los objetos, dependiendo de si se ejecutan con unos permisos u otros (en un sistema suelen convivir en paz y armonía usuarios normales, usuarios con privilegios, administradores de red, etc.).

El uso de las interfaces remotas proporciona varias ventajas a RMI:

- Las implementaciones de los métodos quedan a salvo de las miradas de los clientes.
- Si hay modificaciones en las implementaciones de los métodos remotos, no necesitan ser comunicadas a los clientes (siempre que se respete la interfaz remota).

Para ilustrar lo expuesto, consideraré el ejemplo del componente CalcMat del subapartado 4.2. En Java se implementaría mediante una clase y una interfaz:

```
import java.rmi.*;  
  
public interface CalcMat extends Remóte {  
  
    double hacerCalculoMuyComplicado(double a, double b)  
        throws remoteException;  
    ... // Resto de declaraciones de métodos  
}  
  
import java.rmi.*;  
import java.rmi.server.*;  
  
public class CalcMatImp extends UnicastRemoteObject  
    implements CalcMat {  
  
    public double hacerCalculoMuyComplicado(double a, double b)  
        throws remoteException {  
        ... // Cuerpo del método  
    }  
    ... // Resto de declaraciones de métodos y método main  
}
```

Toda interfaz remota debe extender la interfaz `java.rmi.Remote`

Es conveniente que extienda a `UnicastRemoteObject`

Toda método remoto debe declarar la excepción `java.rmi.RemoteException`

Un cliente que desea acceder al servicio de cálculo matemático se conectaría así:

```
CalcMat cm = new Registro("anfitrion", "nombreObjeto");  
cm.hacerCalculoMuyComplicado(2.5656, 3.14159265358972);
```

Registro sería una clase encargada de conectar con el anfitrión donde se ejecute CalcMat, de localizar el objeto remoto nombreObjeto, instancia de la clase CalcMatImp, y de instanciar en el anfitrión cliente un objeto representante (*proxy*) del objeto remoto en el espacio de direcciones del objeto local (cm). La necesidad de que se realicen estos tres pasos y la manera como se realizan se irán explicando en el resto de este apartado. No obstante, adelanto ya que no es necesario programar la clase Registro: RMI proporciona todo lo necesario.

A veces se dice que RMI (o CORBA) es **middleware**. Esta término es una de esas entrañables palabras anglosajonas que significan lo que a uno le conviene que signifiquen (*scattering* es mi favorita). Según el libro *Client / Server Survival Guide Third Edition [R. Orfali, D. Harkey y J. Edwards, 1999]*,

Middleware es un término indefinido que abarca a todo el software distribuido necesario para el soporte de interacciones entre clientes y servidores. Imagínelo como el software que ocupa la parte intermedia del sistema de cliente/servidor. Es el enlace que permite que un cliente obtenga un servicio de un servidor. ¿Dónde empieza y dónde acaba el middleware? Empieza en el módulo de la API en la parte del cliente que se emplea para invocar un servicio y comprende la

transmisión de la solicitud por la red y la respuesta resultante. Pero no incluye al software que presta el servicio real; esto pertenece a los dominios del servidor. Tampoco a la interfaz del usuario ni a la lógica de la aplicación, en los dominios del cliente.

Sin pecar de puristas, se puede aceptar que el *middleware* es una capa de software que media entre clientes y servidores, y que separa las comunicaciones cliente-servidor de los protocolos de red y de los mecanismos de comunicación entre procesos (como *sockets*). Con el *middleware* se ocultan a los programadores los orígenes reales de los datos de las aplicaciones distribuidas y los detalles de las redes que median entre los anfitriones. Gracias a él, todas las aplicaciones pueden trabajar con una API común, independiente de la plataforma.

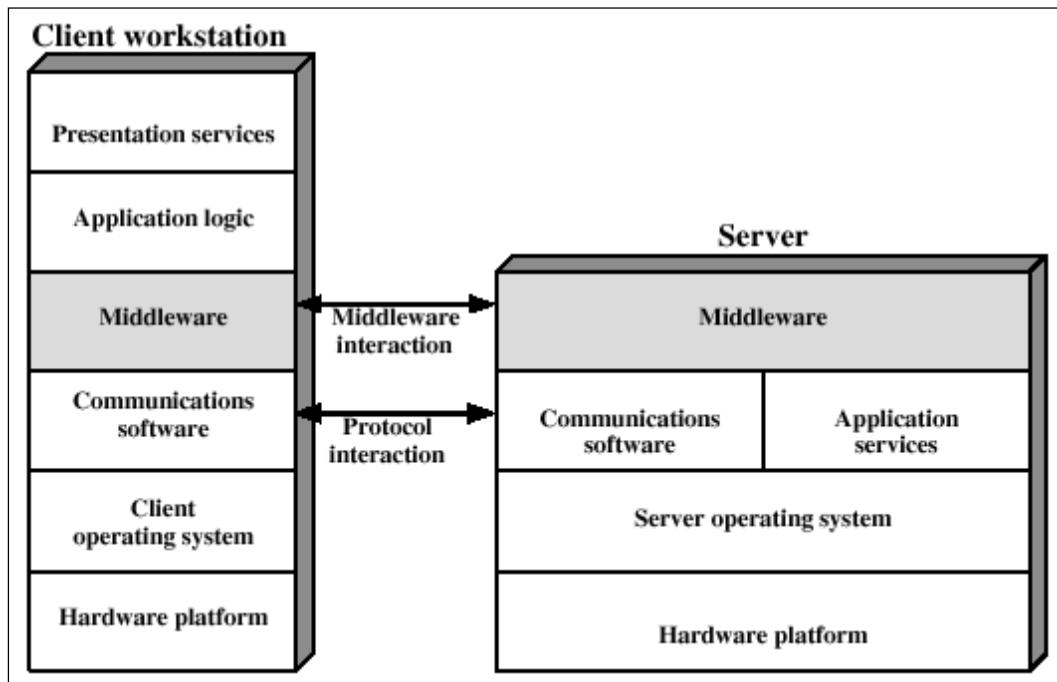


Figura 93. Ubicación del middleware en una arquitectura c-s de dos capas

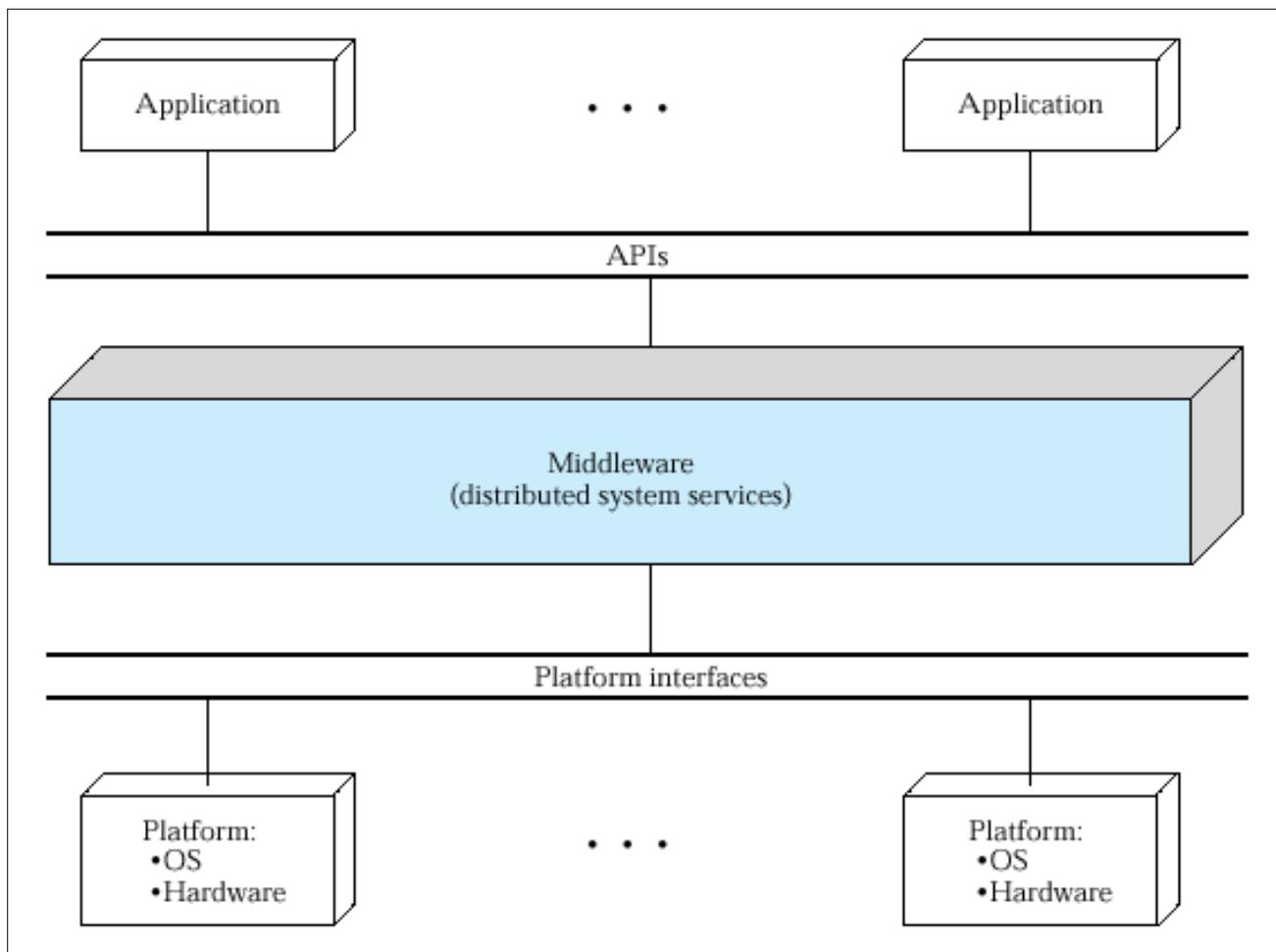


Figura 94. Representación lógica del middleware

5.2. Anatomía de las aplicaciones RMI: adaptadores y esqueletos. La arquitectura RMI. El servicio de registro remoto RMI

5.2.1. Anatomía de las aplicaciones RMI: adaptadores y esqueletos

Una aplicación RMI admite la descomposición en dos aplicaciones separadas: un servidor y un cliente. El servidor se encarga de crear los objetos remotos, los hace accesibles a los otros objetos de la aplicación y permanece a la espera de peticiones para dichos objetos remotos (llamadas). El cliente consigue referencias remotas a uno o más objetos remotos y usa estas referencias para hacer llamadas remotas.

Aunque toda la estructura de la RMI de Java corresponde al modelo cliente-servidor, está adornada con sus propios ornamentos. En ella, un objeto cliente nunca accede directamente a los servicios o métodos de un objeto remoto: entre uno y otro siempre median un adaptador y un esqueleto.

Un **adaptador** (*stub*; mi traducción es, en realidad, una adaptación: *stub* significa cabo o resguardo) es un objeto que actúa como representante local (*proxy*) de un objeto remoto. Una clase adaptadora implementa exactamente el conjunto de interfaces remotas de la clase remota a la cual aparece vinculada. Cuando un objeto local llama a cualquier método de la interfaz remota de un objeto remoto, en verdad llama a los métodos del adaptador local (los cuales, como ya he avanzado, coinciden fielmente con los declarados remotos), que se encargan de transmitir su llamada al remoto objeto. En consecuencia, no hay comunicación directa entre los objetos locales y los remotos: cuando un objeto local llama a un método de un objeto remoto, la llamada pasa al adaptador asociado a este último. Al momento, el adaptador o *stub* inicia una conexión con la MVJ donde reside el objeto remoto, envía los argumentos del método a esa MVJ, espera el resultado de la llamada, lee el valor devuelto por ésta (suponiendo que el método no sea `void` y que no se haya lanzado ninguna excepción) y lo devuelve al objeto local que comenzó la llamada remota.

Un **esqueleto** (*skeleton*) es un objeto que actúa como representante remoto (*proxy*) del objeto remoto; reside, pues, en la MVJ remota. Es la contrapartida remota del adaptador. El esqueleto se encarga de transmitir las llamadas que vienen desde fuera al objeto remoto. Cuando recibe una llamada entrante desde una MVJ foránea, lee los argumentos enviados, llama localmente al método correspondiente del objeto remoto y envía su respuesta a la MVJ que hizo la petición.

Las clases de los adaptadores y los esqueletos no se programan, sino que las genera RMI, si bien se necesita compilarlas manualmente (salvo en el JDK 1.5, que permite la compilación dinámica de las primeras; esta propiedad no se explorará aquí). En el subapartado 5.5 veremos cómo generarlas.

Advertencia: En la versión 1.2 del JDK, se desarrolló una implementación de la RMI que no necesita esqueletos (los suple con el uso de la reflexión). Por lo tanto, sólo es imprescindible usar esqueletos con el JDK 1.1. En este texto seguiré usando esqueletos por compatibilidad, pese a su fúnebre nombre y destino; pero no son ya necesarios. Si el lector sólo trabaja con el JDK 1.2 o posterior, puede leer el texto olvidándose de los esqueletos. Con ese nombre, su destino estaba escrito desde el principio.

IMPLEMENTACIÓN DE LA INTERFAZ REMOTA

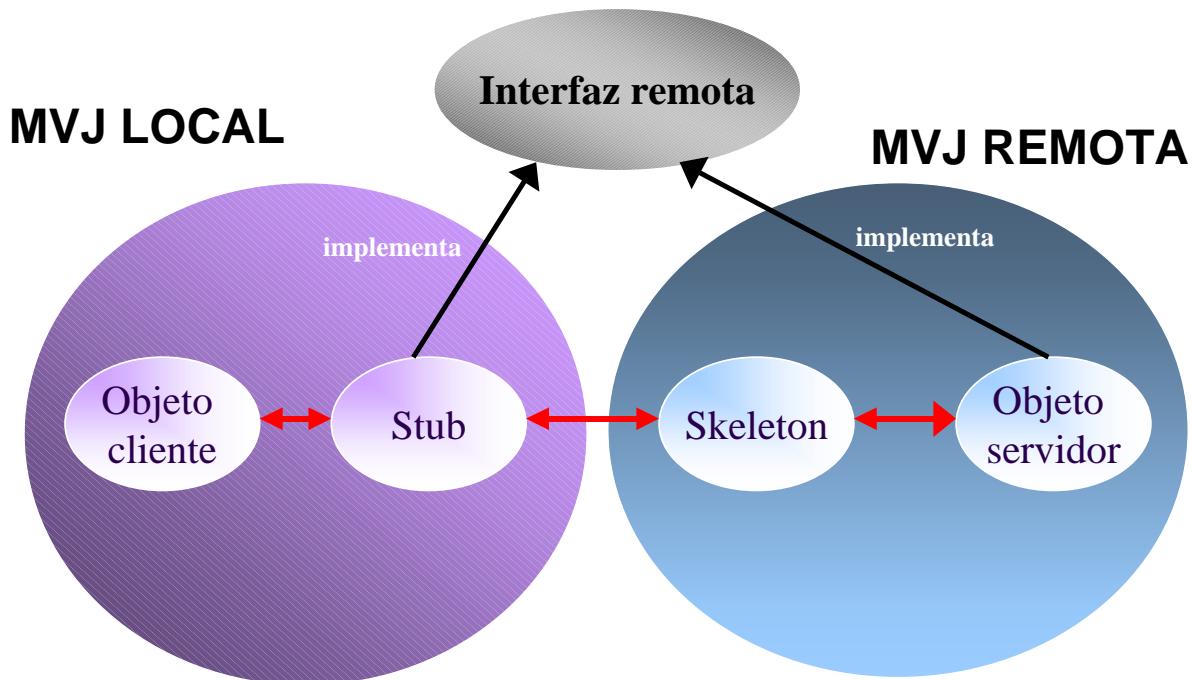


Figura 95. Relaciones entre varios elementos de RMI

Los aficionados a la navaja de Occam, ilustre navajista del siglo XIV, harán muy bien preguntándose por qué se necesitan dos entidades más o si son realmente imprescindibles. Pues bien: sí lo son. Un objeto remoto y uno local “viven” en máquinas virtuales Java distintas. Directamente, un objeto local no puede pasarse por referencia a un método de un objeto remoto, pues las direcciones locales de memoria no tienen sentido para las MVJ remotas. Una dirección de memoria que corresponda a un objeto en la pila de la MVJ remota puede corresponder a cualquier otro objeto (o a ninguno) en otra MVJ. Dado que la comunicación directa es imposible, deben existir intermediarios que operen de puente entre unas MVJ y otras. Precisamente, esos intermediarios son los adaptadores y esqueletos.

Cuando un objeto local llama a un método remoto, su llamada pasa al adaptador correspondiente. Un adaptador es, a fin de cuentas, una referencia al objeto remoto con que se halla vinculado o, si se prefiere, **una referencia remota**. Esta referencia no es una referencia de memoria o un puntero, pues no tendría sentido en un espacio de direcciones que no fuera aquel donde se creó. Una referencia remota contiene una dirección de Internet (la del anfitrión donde está el objeto remoto), un número de puerto (por donde el objeto remoto espera peticiones), un número único con respecto a un anfitrión, que identifica al objeto remoto, y una interfaz remota.

Tanto los adaptadores como los esqueletos implementan las interfaces remotas de los objetos a los cuales están asociados. Examinemos lo que sucede cuando un adaptador recibe una llamada de un objeto local (para los esqueletos el proceso es similar): mediante la **serialización de objetos**, su implementación del método llamado produce un flujo ordenado de bytes, independiente de cualquier plataforma, donde graba las descripciones de las clases de los objetos que se pasan como argumento, las secuencias de bytes que representan a los objetos y la información sobre los lugares desde donde se pueden cargar los *bytecodes* de dichas clases.

En el subapartado 3.3 se escribió que en el proceso de serialización se guarda la descripción de las clases de los objetos serializados, además de la información correspondiente al estado de los objetos; pero no se mencionó que se grabase información sobre la ubicación de las clases. Antes al contrario: se afirmó que había que configurar el CLASSPATH local para que hiciera referencia a los archivos .class necesarios. Esta aparente discrepancia conduce a dos preguntas: ¿qué clases se usan en la RMI para serializar y deserializar objetos?; ¿por qué se guarda información sobre la localización de los .class?

La contestación a la primera es que se utilizan unas subclases de `java.io.ObjectOutputStream` y de `java.io.ObjectInputStream`, no estas clases. Por ejemplo, se usa la subclase de `ObjectOutputStream` `sun.rmi.server.MarshalOutputStream`. `ObjectOutputStream` define un método `protected void annotateClass (Class clase) throws IOException` que no hace nada (no está implementado). Pese a ello, siempre que `ObjectOutputStream` escribe las descripciones de las clases, llama a `annotateClass()`. Si se incluye este método es con vistas a que los programadores y la RMI puedan implementarlo en las subclases de `ObjectOutputStream`. La infraestructura de la RMI, al redefinir este método –y otros–, adapta a sus necesidades el proceso de serialización de objetos, incluyendo en el flujo información (anotaciones) sobre el lugar o lugares desde donde pueden cargarse los *bytecodes* necesarios. Con respecto a `ObjectInputStream`, la subclase encargada del proceso de deserialización redefine el método `protected Class resolveClass(ObjectStreamClass descripción) throws IOException, ClassNotFoundException`, que por defecto carga las clases locales cuyas descripciones aparecen en el flujo, y permite que las clases puedan ser cargadas desde cualquier otra fuente. Internamente, la RMI crea y maneja las subclases derivadas de `ObjectInputStream` y `ObjectOutputStream`; el programador no necesita preocuparse por serializar y deserializar explícitamente.

La respuesta a la segunda deriva de que la RMI trabaja con entornos distribuidos. En una aplicación que se ejecuta en una sola máquina, cuando se deserializa el flujo se busca, para cada descripción de clase que se encuentra en él, su archivo .class en el directorio actual y en los incluidos en el CLASSPATH. Como todos las clases se compilan en la misma máquina, lo lógico es que se encuentren en ella. Si algún archivo no se encuentra es porque no ha sido compilado o porque el CLASSPATH está mal configurado. En una aplicación distribuida no podemos esperar que todos los archivos .class de la aplicación estén en los sistemas locales de archivos de todos los anfitriones donde se ejecuta. Por ello, las subclases de `ObjectOutputStream` que usa RMI redefinen el método `annotateClass()` para que permita grabar en el flujo de bytes información sobre la codebase (especificada en la propiedad

`java.rmi.server.codebase`, que veremos un poco más adelante). Una `codebase` no es más que un lugar (o varios) desde donde se pueden cargar clases en una MVJ, en forma de URL. Dicho de otro modo, no es más que un URL que especifica una localización en la red desde la cual pueden cargarse los *bytecodes* de los ficheros `.class` que se necesitan para deserializar los objetos dentro del flujo. Veamos un ejemplo de `codebase` (considero que las clases están en un fichero `misclases.jar`):

```
http://www.uv.es:9000/directorioclasses/misclases.jar
```

Usando `codebases`, la RMI puede cargar dinámicamente nuevas clases basándose en la información sobre ellas que obtenga del flujo; para ello usa la clase `java.rmi.server.RMIClassLoader`, que veremos más adelante. Resulta lógico que `codebase` tenga la forma de un URL: si al serializar se grabara información sobre la localización de las clases en el sistema local de archivos, la MVJ que deserializara instancias de estas clases se encontraría con que esas localizaciones no existen en el sistema de archivos de su anfitrión (la probabilidad de tener dos máquinas con idénticas estructuras de directorios es muy reducida).

Usando la versión redefinida de `annotateClass()`, la RMI graba para cada objeto que serializa (no olvidemos que todos los argumentos de métodos remotos y sus valores de retorno se pasan serializados) la propiedad `java.rmi.server.codebase`, que contiene las `codebases` y debe ser establecida al compilar la clase del objeto. Los objetos que deserialicen el flujo leerán esta propiedad mediante la versión redefinida del método `resolveClass()` y sabrán de dónde cargar las clases que necesiten para reconstruir los objetos dentro del flujo. Por ejemplo, consideramos la ejecución de una clase `Nomina` que deserialice instancias de una clase `Empleado` (reduzco el tipo de letra para que el comando quepa en una línea):

```
java -Djava.rmi.server.codebase=http://www.uv.es/directorioclasses/ Nomina
```

Cualquier objeto remoto `Nomina` que trate de deserializar en tiempo de ejecución una instancia de `Empleado` buscará –por medio de `java.rmi.server.RMIClassLoader`– en el URL especificado el archivo `.class` (`Empleado.class`) que necesita para cargar dinámicamente la clase `Empleado`. Eso sí, antes de buscar en el URL, buscará en el directorio actual y en el `CLASSPATH`, y si la encuentra ignorará la propiedad `java.rmi.server.codebase`. Si la propiedad `java.rmi.server.codebase` no se especifica, y los archivos `.class` que necesita un objeto que deserializa no están en su directorio actual ni aparecen en su `CLASSPATH` local, la MVJ lanzará una excepción `java.lang.ClassNotFoundException`.

En general, **si para ejecutar una clase se establece la propiedad `java.rmi.server.codebase`, cualquier proceso remoto que necesite cargar archivos `.class` para objetos recibidos durante una sesión RMI usará ese URL para encontrar los `.class` (siempre que no pudiera hallarlos antes en el `CLASSPATH` local)**. Veamos un ejemplo:

```
java -Djava.rmi.server.codebase=http://www.uv.es/misclases/ CalcMat
```

La MVJ donde se ejecute `CalcMat` intentará buscar todos los archivos `.class` que necesite (si no los encuentra antes en su directorio actual ni en su `CLASSPATH` local) en el URL `http://www.uv.es/misclases`. Si los encuentra, los descargará.

Descargar clases implica considerar ciertas cuestiones relativas a la seguridad, que se verán en el subapartado 5.5.

Nota: En los URL especificados en la propiedad `java.rmi.server.codebase` también puede usarse `ftp` o `file` en lugar de `http`. Si esta propiedad especifica un directorio debe incluirse la barra “/” al final del URL; si especifica un archivo, no.

Cuando un esqueleto recibe a través de la red un flujo de bytes enviado por un adaptador, flujo donde se encuentran incrustados los argumentos de la llamada remota, para deserializarlo lee las descripciones de las clases de los objetos incluidos en el flujo y busca los archivos `.class` correspondientes. Primero busca en su directorio, luego en su `CLASSPATH`; si no los encuentra, continúa buscando en el `codebase` o los `codebases` que figuren en el flujo (suponiendo que se especificara la propiedad `java.rmi.server.codebase`). Si los encuentra, crea las instancias de los objetos en su espacio de direcciones. En caso contrario (el URL puede ser incorrecto o estar fuera de servicio), lanza una excepción `java.lang.ClassNotFoundException`.

En el caso de que se devuelva algún objeto como valor de retorno, el esqueleto lo envía (serializado) al adaptador, que se encarga de deserializarlo. La situación se representa en la siguiente figura.

UNA LLAMADA REMOTA EN RMI (1)

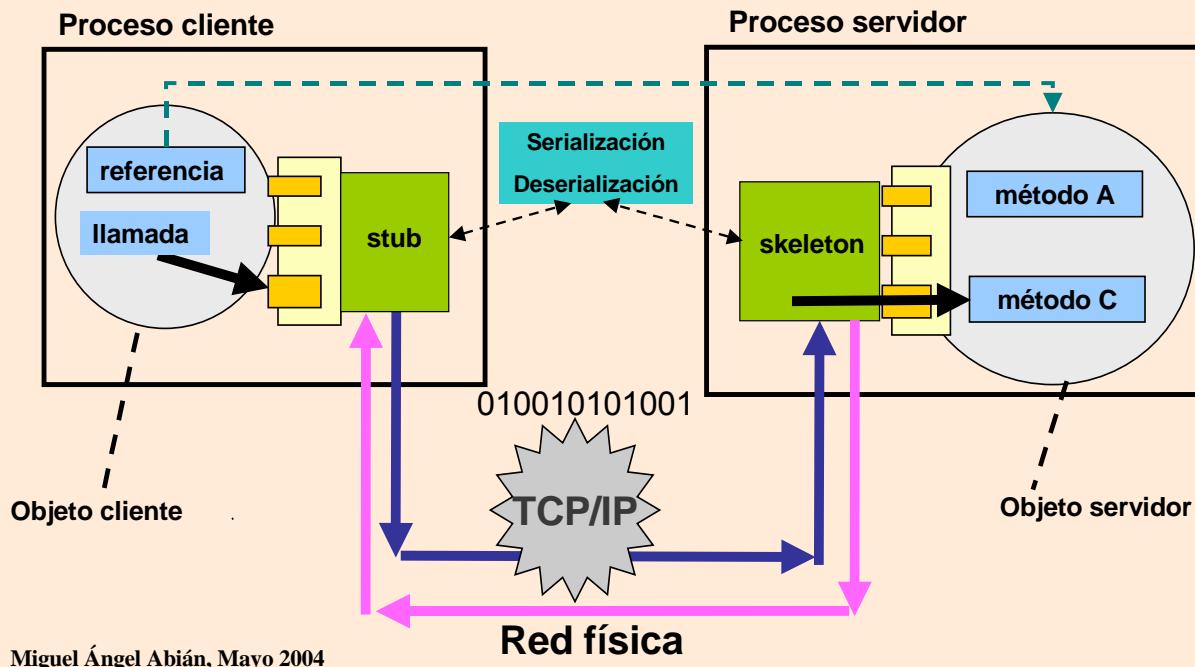
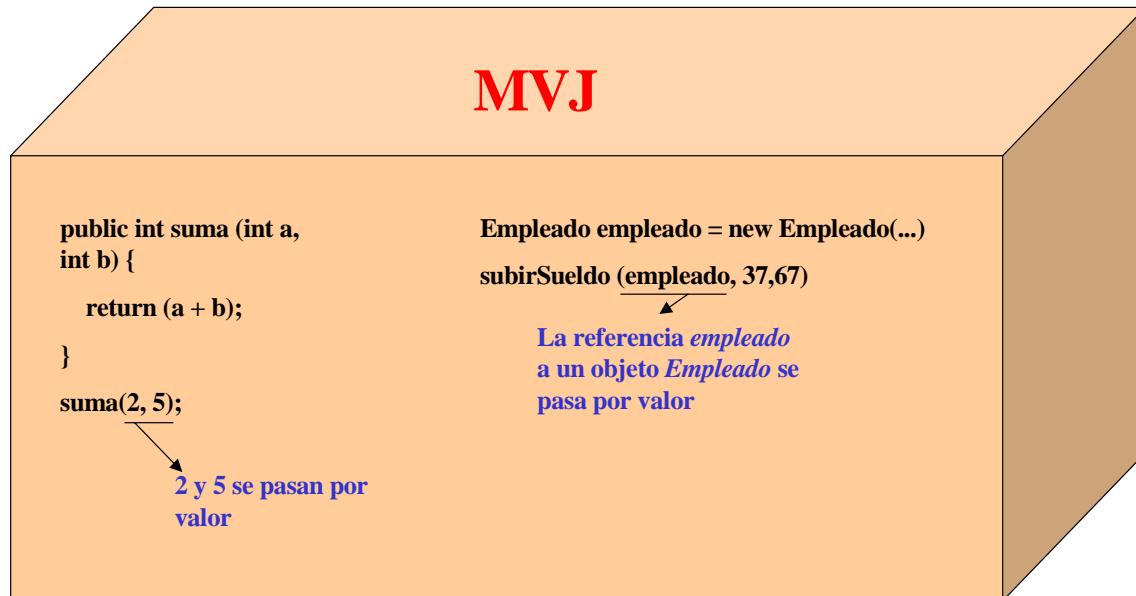


Figura 96. Tras una llamada remota hay muchos subprocessos

Para saber qué argumentos y valores de retorno puede admitir un método remoto, se debe conocer cómo se transmiten los distintos tipos de objetos en las llamadas remotas (en Java, un objeto remoto debe implementar obligatoriamente la interfaz `java.rmi.Remote`, que se verá más adelante):

- Los tipos de datos primitivos (`short`, `int`, `double`, `char...`) y los objetos predefinidos en Java cuyas clases implementan la interfaz `java.io.Serializable` (`String`, etc.) se pasan por valor. Esto es, se copian del espacio de direcciones de una MVJ al de otra. Lo mismo vale para los valores de retorno del método remoto.
- Los objetos no remotos cuyas clases implementan la interfaz `java.io.Serializable` se pasan también por valor (lo mismo aplica a los valores de retorno).
- Los objetos remotos que están exportados (esto es, preparados para aceptar peticiones de los clientes por un puerto; ya veremos más adelante cómo se exportan los objetos remotos) nunca se envían, en su lugar se envían referencias a ellos (instancias de una clase adaptadora o *stub*). Lo mismo aplica para los valores de retorno. Estos objetos se pasan, pues, por referencia. Si estos objetos, aun siendo remotos, no están exportados, se pasan por valor.
- Los objetos que no son remotos ni serializables (es decir, que no implementan las respectivas interfaces) no pueden enviarse a un objeto remoto ni tampoco ser devueltos por él: la MVJ lanzará una excepción.
- Con respecto a los objetos que son a la vez remotos y serializables (esto es, que implementan las respectivas interfaces), albergo algunas dudas. Por un lado, he comprobado (en el JDK 1.2 y en el 1.4.2) que estos objetos son admitidos por el compilador y que no arrojan excepciones en tiempo de ejecución. Sin embargo, recomiendo no utilizarlos, pues me parece una posibilidad bastante confusa y que no aporta ninguna ventaja. Desconozco el mecanismo interno por el cual se transmiten las llamadas remotas cuando tienen argumentos de este tipo, aunque supongo que coincidirá con el que se usa para los adaptadores. Agradeceré cualquier sugerencia al respecto.

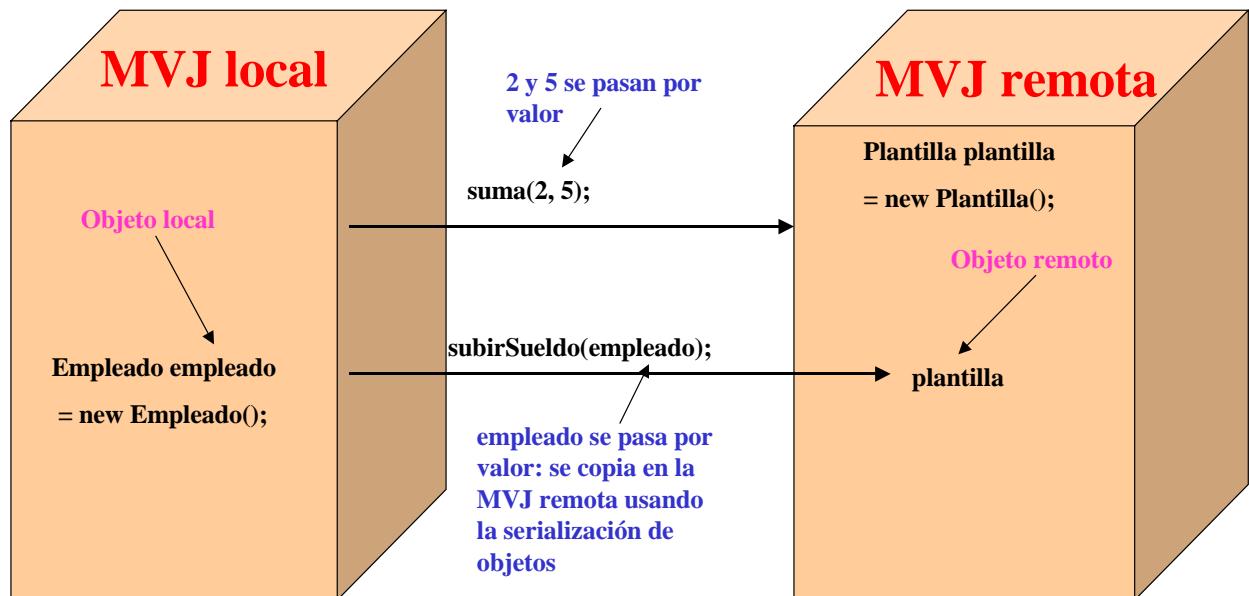
PASO DE ARGUMENTOS EN UNA ÚNICA MVJ



Miguel Ángel Abián, Mayo 2004

Figura 97. El paso de argumentos en una MVJ

PASO POR VALOR Y POR REFERENCIA EN UNA APLICACIÓN RMI (1)



Miguel Ángel Abián, Mayo 2004

Figura 98. Argumentos que se pasan por valor en una llamada remota

PASO POR VALOR Y POR REFERENCIA EN UNA APLICACIÓN RMI (2)

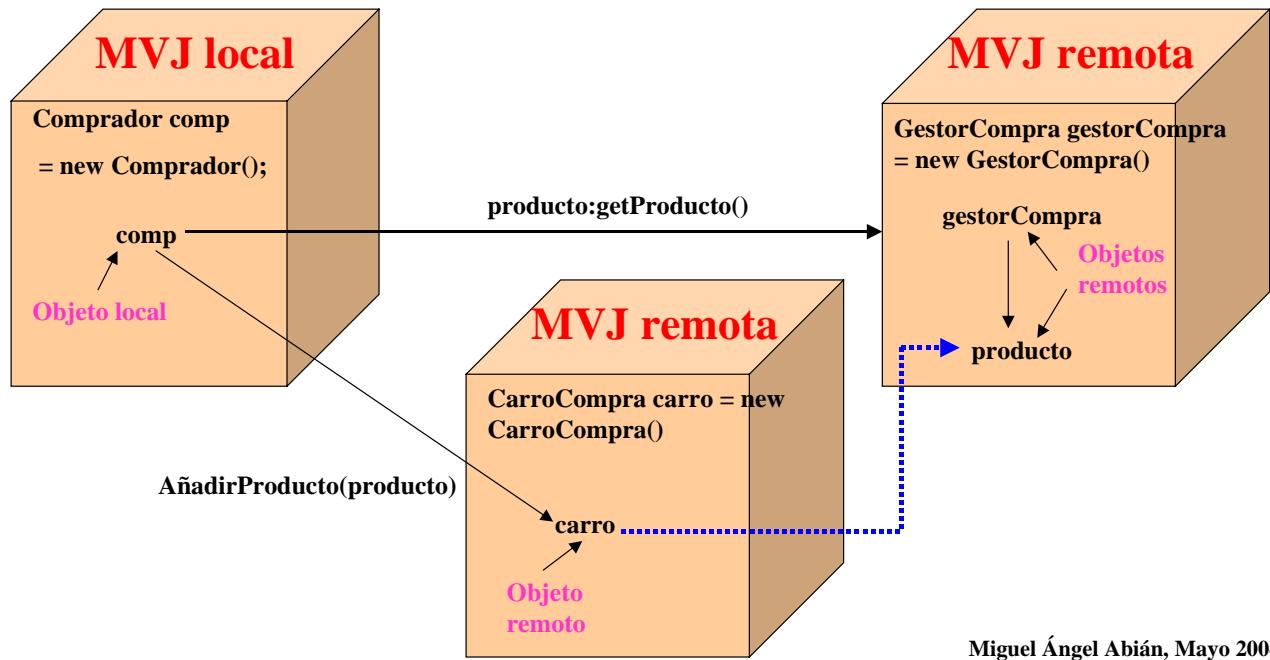


Figura 99. Argumentos que se pasan por referencia en una llamada remota

Que los objetos locales se pasen por valor resulta inevitable: si sólo se pasaran referencias a los objetos locales (adaptadores o *stubs*), los remotos tendrían que consultar a la máquina local (que para ellos sería remota, pues se ejecuta en otra MVJ).

Supongamos, para clarificar la situación, que un cliente enviara como argumento del método remoto `guardarEmpleado(Empleado empleado)` un adaptador del objeto `empleado` en lugar de una copia de `empleado`. El objeto remoto al cual va dirigida la llamada podría necesitar información sobre la instancia `empleado`: edad, sueldo, tipo de contrato, etc. Necesitaría, pues, consultar al objeto cliente y llamar a métodos como `getEdad()`, `getSueldo()`, etc.; porque, a fin de cuentas, tendría una referencia remota al objeto local, no una copia. ¿En qué se traduciría todo esto? En un continuo ir y venir de mensajes a través de la red o de las redes que mediarian entre ambos objetos. Esta estrategia resulta inviable por dos motivos; a saber: los tiempos de latencia asociados a las transmisiones en red y los fallos de las redes.

Nota: Java pasa argumentos sólo por valor, sean tipos primitivos u objetos. Muchas veces se confunde el paso por valor de referencias a objetos con el paso por referencia. El primero es el mecanismo que usa Java; C++ utiliza el segundo. Cuando se dice que los objetos remotos Java se pasan por referencia en las llamadas remotas, se busca expresar que se pasan *stubs* o adaptadores, los cuales vienen a ser referencias a los verdaderos objetos remotos. Y estas referencias se pasan por valor mediante serialización.

RMI usa el paso por valor de adaptadores para simular el paso por referencia, inexistente en Java. ¿Cómo lo hace? Cuando el cliente llama al servidor con un argumento que es un objeto remoto exportado, se envía, en lugar del objeto remoto, un adaptador. Cuando el servidor llama al cliente, en realidad llama al adaptador (que es local para él). Como el adaptador apunta al objeto remoto del cliente, existe un vínculo entre el servidor y el verdadero objeto remoto.

La confusión entre paso por valor de referencias y paso por referencia es muy frecuente, tanto entre expertos como entre neófitos. En la bibliografía sobre sistemas distribuidos, siempre se usa “paso por referencia” para designar a las dos posibilidades anteriores. Aun siendo consciente de la inexactitud, uso “paso por referencia” en ocasiones donde se trata de paso por valor de referencias. Intento así no discrepar de la terminología usada por la mayor parte de los textos.

La copia por valor se realiza mediante la serialización de objetos de Java, vista en el apartado dedicado al paquete `java.io` y que ya se ha mencionado en este subapartado. Mediante ella, los objetos pueden ser enviados por los clientes en forma de flujos de bytes, y pueden ser reconstruidos en los servidores, que crearán en sus espacios de direcciones las nuevas instancias (idénticas a las serializadas en los clientes), basándose en la información contenida en los flujos. Aparte de la serialización, Java no incorpora ningún otro mecanismo para copiar objetos de forma automática. Con esta potente herramienta, se asegura además que, si un objeto contiene referencias a otros objetos, éstas se conservarán en sus copias. Sin la serialización, una llamada a un objeto remoto en la que se llamaría internamente a un método de algún objeto al cual se tuviera una referencia, obligaría a que existiera comunicación con la MVJ donde residiera este último objeto. Se produciría, pues, un trasiego innecesario –y peligroso– de mensajes entre la red o las redes intermedias.

Cuando se pasan objetos remotos exportados (o se devuelven como valores de retorno), la copia se realiza por referencia. Internamente, se sigue usando la serialización de objetos, pero la RMI da el cambazo: incluye en el flujo objetos adaptadores o *stubs* en lugar de los verdaderos objetos remotos. Así pues, la copia por valor de los objetos remotos se sustituye por la copia por valor de sus referencias remotas (adaptadores, los cuales saben en qué máquina está el verdadero objeto remoto y mediante qué puerto pueden acceder a él). Como las clases adaptadoras implementan la interfaz `java.io.Serializable`, no hay ningún problema para que sus instancias sean serializadas y deserializadas. Para sustituir los objetos remotos exportados por sus adaptadores correspondientes se usa una redefinición del método `protected Object replaceObject(Object objeto) throws IOException` de la clase `java.io.ObjectOutputStream`.

5.2.2. La arquitectura RMI

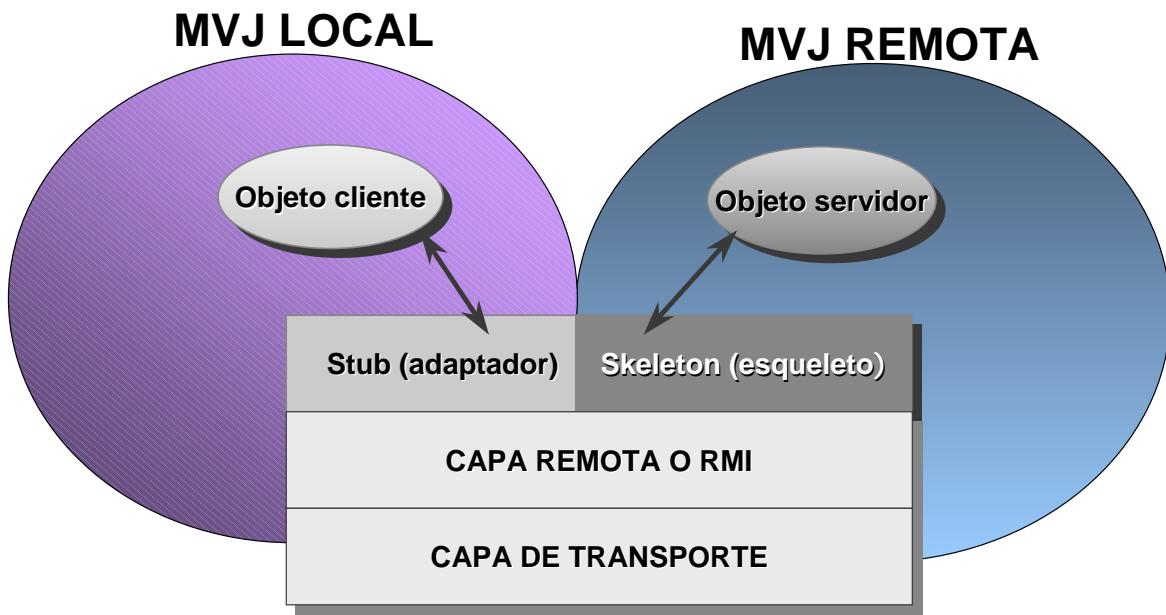
Simplificando un poco, puede decirse que la RMI presenta una arquitectura de tres capas, representada en la figura 100.

- a) La **capa adaptador-esqueleto** (*stub-skeleton*) dota a unos y a otros de una interfaz común.
- b) La **capa remota o RMI**. Se encarga de controlar la creación y gestión de las referencias a objetos remotos (mantiene una tabla de objetos distribuidos), y de convertir las llamadas remotas en peticiones a la capa de transporte. Para ello utiliza un protocolo independiente de los adaptadores y esqueletos, así como de la plataforma donde se ejecuta la MVJ. En el JDK 1.1 y 1.2, RMI usaba el protocolo JRMP (*Java Remote Method Protocol*: protocolo de métodos remotos de Java). JRMP sólo se usa con Java y no coincide con el protocolo equivalente que usa CORBA (IIOP: *Internet Inter-ORB Protocol*). Con el JDK 1.3 se añadió la posibilidad de trabajar también con IIOP, lo cual ha abierto nuevos horizontes a Java: ahora se puede acceder a los objetos remotos RMI (escritos en Java) desde clientes CORBA escritos en otros lenguajes (C/C++, Ada, etc.), y viceversa.

Las peticiones RMI, descritas según establece el protocolo JRMP, suelen ser bloqueadas por los cortafuegos, ya que se usan puertos aleatorios que pueden tener restricciones de seguridad. Cuando se trabaja con cortafuegos, RMI encapsula automáticamente las llamadas RMI dentro de peticiones POST del protocolo HTTP (esta técnica se conoce como *HTTP tunneling* o pasarela HTTP). Esta encapsulación empeora el rendimiento de las aplicaciones RMI, pero muchas veces resulta inevitable.

- c) La **capa de transporte** es la capa de transporte del conjunto de protocolos TCP/IP, ya vista en el apartado 2. Por defecto, RMI usa el protocolo de transporte TCP, pero admite otros.

ARQUITECTURA SIMPLIFICADA DE LA RMI DE JAVA



Miguel Ángel Abián, Mayo 2004

Figura 100. Esquema parcial de la arquitectura RMI

ARQUITECTURA COMPLETA DE LA RMI DE JAVA

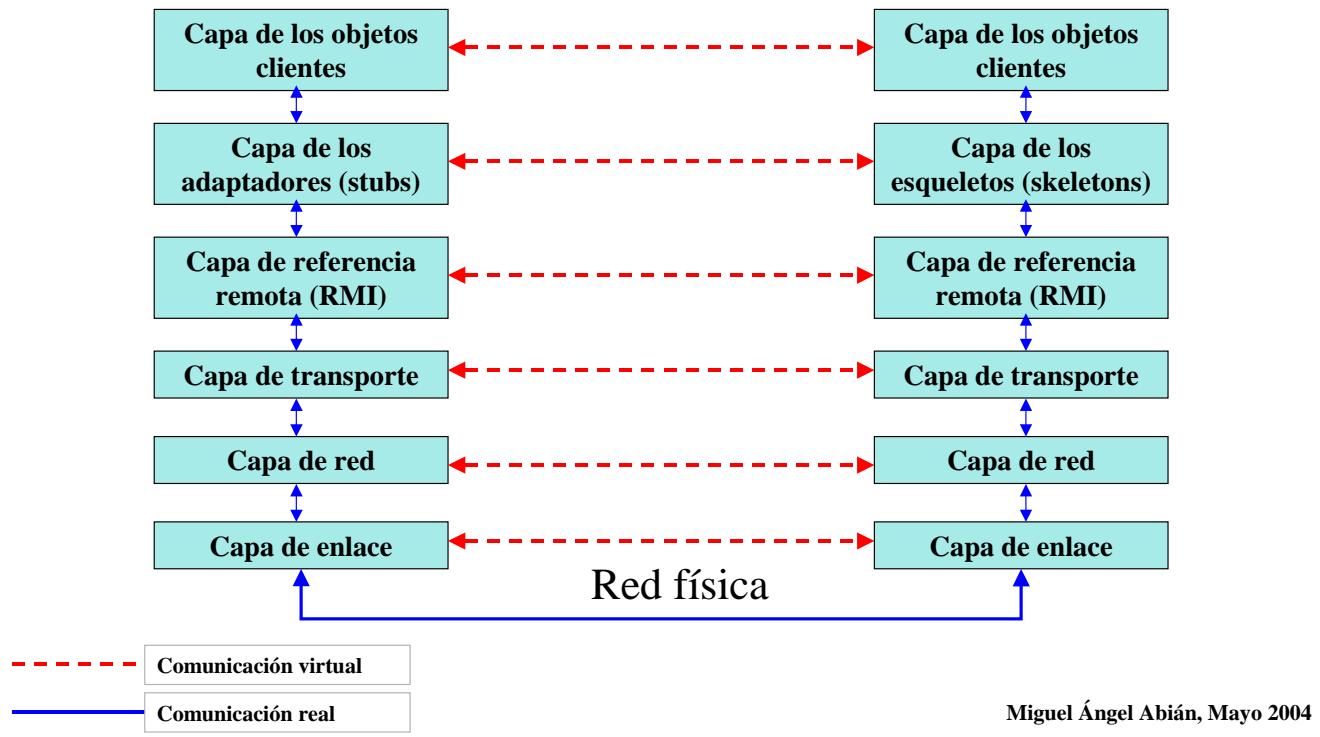


Figura 101. Esquema completo de la arquitectura RMI

5.2.3. El servicio de registro remoto RMI

La única pieza que falta para completar el puzzle de las comunicaciones con RMI se obtiene de contestar a esta pregunta: ¿cómo pueden los clientes encontrar los servicios? Pues mediante un servicio de nombres y directorios. En el apartado 2 ya vimos uno: el DNS (*Domain Name Service*: servicio de nombres de dominio), que asigna nombres de máquina a las direcciones IP. Un sistema de nombres es un mecanismo para asociar nombres con objetos o dispositivos de una red, que proporciona un medio de encontrar un objeto o dispositivo a partir de un nombre dado. El proceso de búsqueda de un objeto a partir de un nombre se llama resolución. En un sistema de nombres y directorios, un nombre de fichero –por ejemplo– está asociado con una referencia que las aplicaciones pueden usar para acceder al archivo. Todo servicio de nombres y directorios viene a ser equivalente a un listín telefónico; pero, en vez de asociar números de teléfono con direcciones y nombres de personas, asocia nombres lógicos con objetos o componentes de una red.

Por diseño, RMI puede usar como servicio de nombres y directorios la JNDI (*Java Naming and Directory Interface*: interfaz de nombres y directorios de Java) o su propio servicio: el **servicio de registro remoto de la RMI** o servicio de registro de objetos remotos de la RMI de Java (por brevedad, usaré simplemente *servicio de registro RMI*). La JNDI ofrece muchas más posibilidades que el servicio de registro RMI, pero también es de manejo mucho más complicado, amén de exigir el estudio de una nueva API. Aquí se usará solamente el servicio de registro RMI, que se implementa mediante la aplicación de servidor `rmiregistry`, incluida en el directorio `bin` de los JDK.

El servidor de registro RMI (`rmiregistry`) debe estar en ejecución antes de que lo estén los objetos que actúen como clientes y servidores en una aplicación RMI. Sin él, los clientes no pueden localizar los servicios remotos buscados (los métodos ofrecidos por los servidores) ni los servidores pueden atenderlos. Asimismo, si en una aplicación RMI falla el registro remoto, no podrá funcionar. A diferencia de la JNDI, el registro remoto de RMI no admite persistencia: cuando la aplicación acaba, se pierden para siempre los vínculos entre objetos remotos y nombres lógicos.

Cuando se ejecuta `rmiregistry` en el anfitrión donde reside el objeto remoto (servidor), se lanza un proceso que utiliza por defecto el puerto TCP 1099. En el caso de que no se pueda usar el protocolo TCP (por el uso de cortafuegos, por ejemplo), RMI es compatible también con protocolos de transporte como SSL y UDP. Para que los clientes puedan acceder remotamente a un servidor, éste debe antes ser registrado en el servicio de registro RMI (lo cual implica asociarlo con un nombre lógico). Registrar un objeto remoto no es más que asociarle un nombre, de manera que el nombre asignado pueda usarse más tarde para buscarlo y acceder a sus servicios. Un cliente que llame a un método remoto de un servidor buscará a éste por el nombre que se le diera al registrarse, obtendrá una referencia remota a él (un adaptador o *stub*) y, luego, llamará a sus métodos. La idea clave del servicio de registro RMI radica en proporcionar a los clientes referencias a objetos que residen en distintos anfitriones (o quizás en otras MVJ que se ejecutan en la misma máquina).

Personalmente, me resulta útil la idea de imaginarme el servidor de registro remoto como una centralita donde hubiera una persona con un listín de dos columnas: una para los nombres y otra para los adaptadores. Cuando se recibiera una petición referida a un nombre (`luis.aumentarSalario()`, p.ej.), la persona buscaría el nombre (`luis`) en el listín, miraría cuál es el adaptador asociado y lo enviaría al cliente. El adaptador contiene toda la información necesaria para conducir las llamadas hasta el objeto remoto asociado.

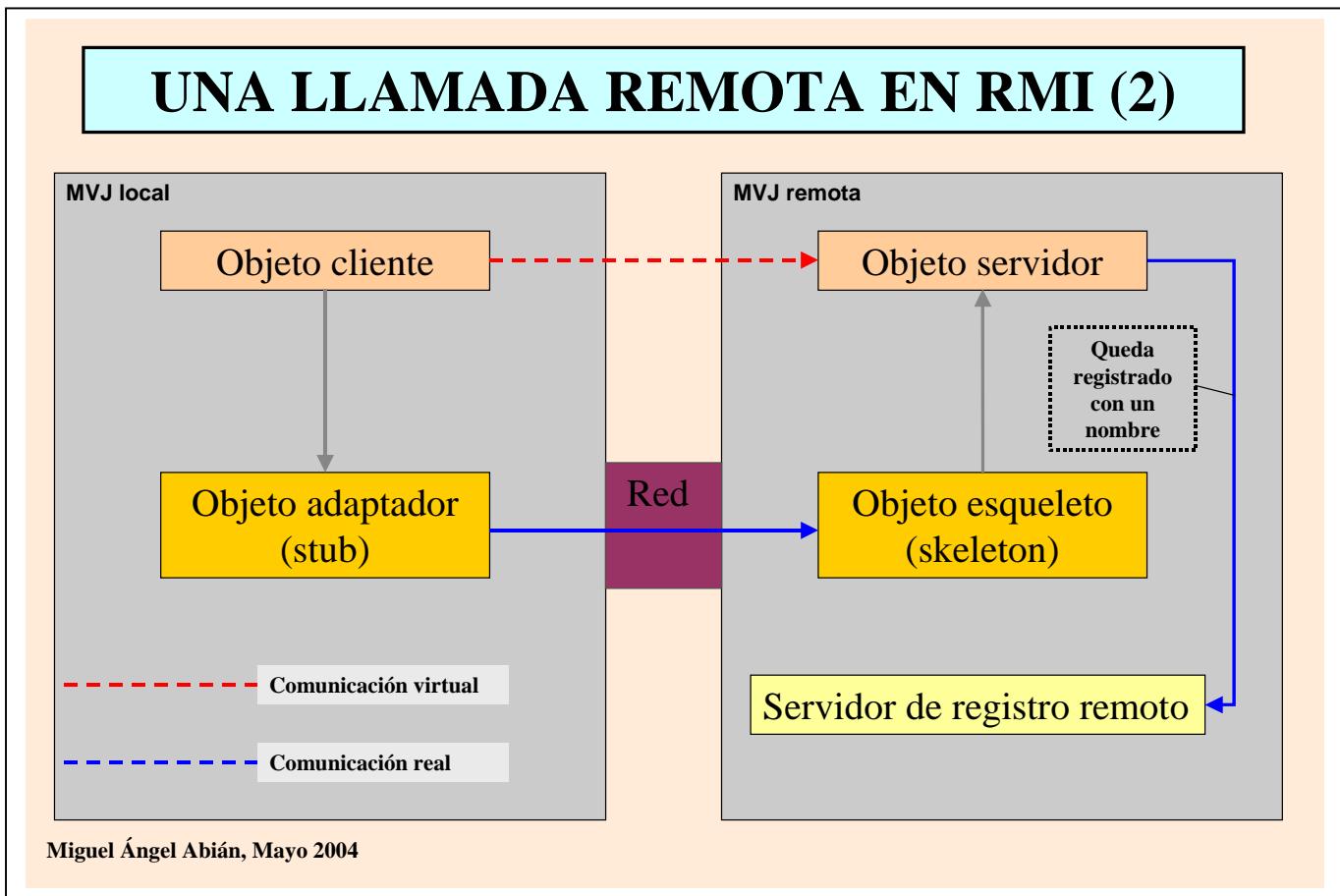


Figura 102. Vista simplificada de una llamada remota (se tiene en cuenta el servicio de registro RMI)

Con carácter general, para las comunicaciones en la arquitectura de la RMI se siguen estos pasos (algunos son realizados por los usuarios, otros por la RMI):

- 1) Se ejecuta el servidor de registro RMI en la máquina remota (debe estar en ejecución antes de que se ejecuten los demás objetos), el cual permanece a la espera de peticiones por el puerto TCP 1099 (puerto por defecto). Para ello se crea un *socket* de servidor que permanece a la espera de peticiones.
- 2) Se crea el objeto remoto, se exporta (esto es, se deja preparado para que escuche llamadas a sus métodos por un determinado puerto del anfitrión remoto o por un puerto anónimo) y se registra en el servicio de registro RMI con un nombre (pueden crearse varios, pero al menos uno debe registrarse). Al registrarse, se crea en la máquina remota una instancia de la clase esqueleto. La escucha de peticiones se hace mediante un *socket* de servidor asociado al puerto por el cual se ha exportado.
- 3) Se crea el objeto local, que llama a un método de la interfaz remota del objeto remoto, usando el nombre con que se registró este último.

- 4) El servidor de registro RMI envía a la MVJ local una referencia remota al objeto remoto (una instancia de la clase adaptadora o *stub* del objeto remoto), pero no el objeto remoto. Realmente, el servidor de registro RMI envía un flujo de bytes serializado en el que está codificado el adaptador; el cliente, al deserializarlo, crea la instancia del adaptador. El adaptador contiene la dirección IP del anfitrión donde está el objeto remoto al que hace referencia, el número del puerto que usa éste para escuchar llamadas a sus métodos y un identificador único del objeto.
- 5) El adaptador abre un flujo de salida, serializa los argumentos (si son objetos remotos exportados, serializa sus adaptadores, no los verdaderos objetos), envía a la capa remota o RMI una petición de conexión y delega en ella la llamada.
- 6) La capa remota informa a la capa de transporte de que necesita iniciar una conexión y le envía el flujo de salida.
- 7) La capa de transporte crea un *socket* de cliente y por él envía el flujo de salida.
- 8) La llamada recorre las capas por debajo de la de transporte y se transmite a la MVJ remota a través de la red.
- 9) La llamada recorre las capas TCP/IP del anfitrión remoto y llega a la capa de transporte. Allí, el *socket* de servidor asociado al objeto remoto lee el flujo entrante y lo transmite a la capa remota.
- 10) La capa remota pasa la llamada al esqueleto.
- 11) El esqueleto abre un flujo de entrada, deserializa los objetos incluidos en el flujo entrante y envía la llamada al objeto remoto.
- 12) El objeto remoto ejecuta el método llamado, con los argumentos que se le proporcionan y, si corresponde, devuelve un valor al objeto esqueleto o una excepción.
- 13) El esqueleto envía a la capa remota una petición de conexión y delega en ella el envío del valor de retorno.
- 14) La capa remota se encarga de abrir un flujo de salida, de serializar el valor de retorno (si es un objeto remoto exportado, serializa su adaptador) y de enviar el flujo a la capa de transporte, además de informar a esta última de que necesita iniciar una conexión.
- 15) La capa de transporte del anfitrión remoto abre un *socket* de cliente y lo usa para enviar el valor de retorno al anfitrión local.
- 16) El valor devuelto recorre las capas TCP/IP del anfitrión remoto, pasa por la red y llega a la capa de transporte del anfitrión que desencadenó el proceso. Allí, un *socket* de servidor lee el flujo entrante y lo transmite a la capa remota.
- 17) La capa remota abre un flujo de entrada, deserializa el valor de retorno incluido en el flujo entrante y lo envía al adaptador.
- 18) El adaptador o *stub* envía este valor al objeto local.

ESQUEMA DE LAS COMUNICACIONES CON RMI

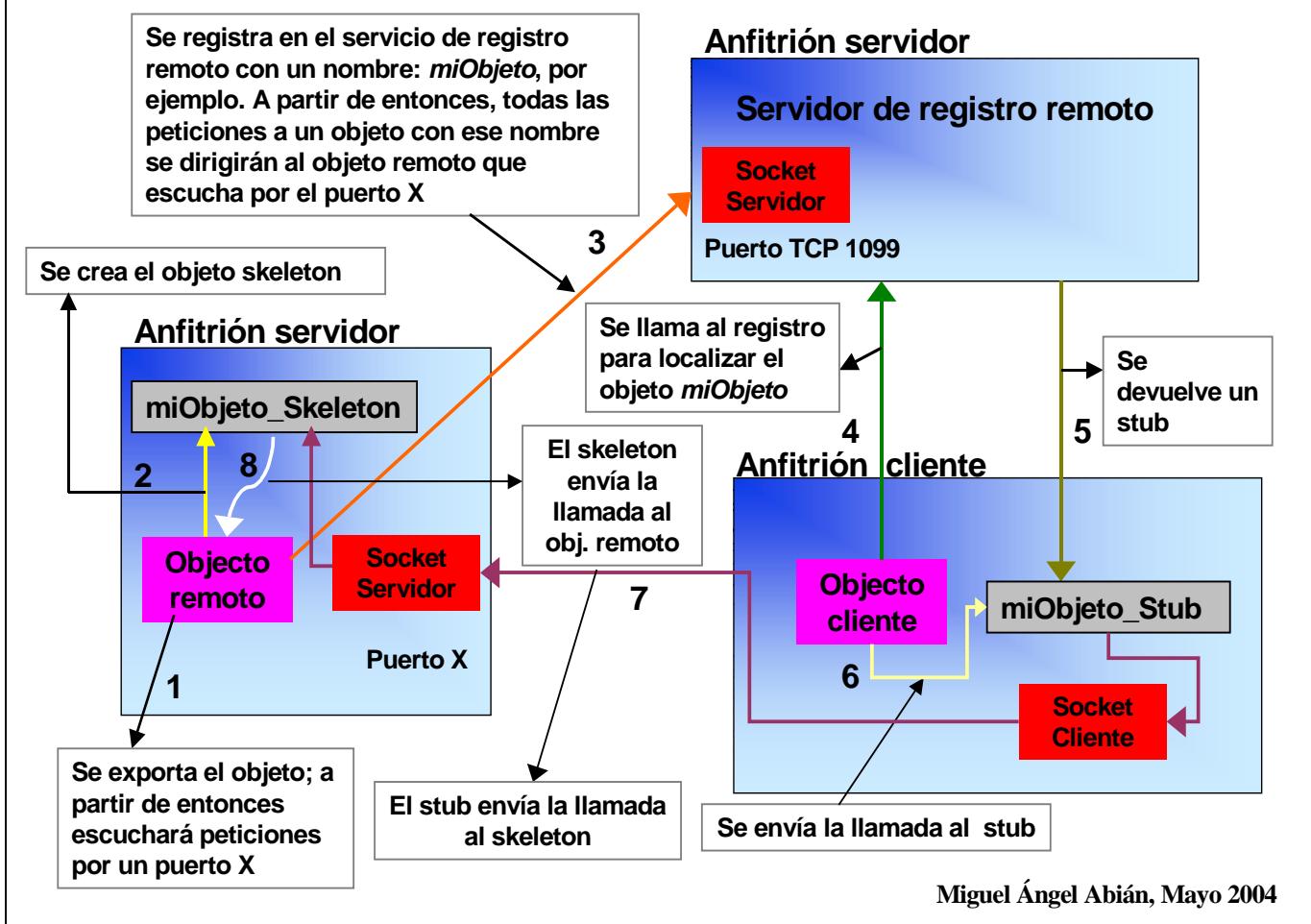


Figura 103. Procesos que subyacen bajo una llamada remota con RMI

En la figura anterior se esquematizan algunos de los pasos, no todos, por razones de espacio.

Como puede ver el lector, se ha recorrido un largo camino para llegar hasta aquí. Arquitecturas de comunicaciones, capas, protocolos, sockets, serialización de objetos: todos estos conceptos han sido necesarios para entender cómo funciona la RMI de Java. En el apartado 6, usaremos estas ideas para avanzar un poco más en la comprensión de los sistemas de objetos distribuidos.

5.3. Recorrido rápido por el paquete java.rmi

La API RMI se implementa mediante las clases pertenecientes a estos paquetes:

- **java.rmi**
- **java.rmi.registry**
- **java.rmi.server**
- **java.rmi.activation**
- **java.rmi.dgc**

Como los dos últimos paquetes trabajan con aspectos avanzados de la RMI, no desglosaré las clases e interfaces que contienen. No obstante, incluyo más adelante una breve descripción de estos paquetes.

java.rmi

El paquete **java.rmi** proporciona la interfaz **Remote** y las clases **MarshalledObject**, **Naming** y **RMISecurityManager**, así como unas cuantas excepciones.

La interfaz **Remote** carece de métodos. **Toda clase remota debe implementarla**; en caso contrario, Java no la considera como tal.

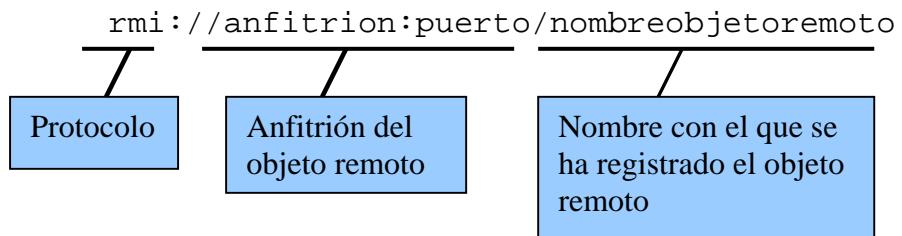
La clase **MarshalledObject** apareció por primera vez en el JDK 1.2. Una instancia de ella contiene el flujo de bytes serializado de un objeto. Sus métodos son utilizados internamente por la RMI.

La clase **Naming** incluye métodos para obtener y almacenar referencias a objetos remotos mediante el URL de la RMI. Los métodos más usados son

- `public static void bind(String nombre, Remote objeto) throws AlreadyBoundException, MalformedURLException, RemoteExceptionBinds`
- `public static void rebind(String nombre, Remote objeto) throws RemoteException, MalformedURLException`
- `public static Remote lookup(String nombre) throws NotBoundException, MalformedURLException, RemoteException.`

El método **bind()** asocia un nombre a un objeto remoto mediante un URL de la RMI (lo “registra”); así, ese nombre podrá usarse para localizar el objeto remoto. Todos los argumentos **nombre** de los métodos de **Naming** deben ser **Strings** con la forma de los URL de la RMI.

El URL de un objeto remoto registrado tiene este formato:



Por ejemplo:

```
rmi://www.uv.es:9000/ListaNotas
```

En el caso de que el URL incluya un protocolo que no sea `rmi`, se obtendrá una excepción `java.net.MalformedURLException`. El anfitrión o *host* y el puerto son opcionales. Si no se incluye el anfitrión, se toma el anfitrión local; si no se especifica el puerto, se toma el puerto TCP 1099, que está asignado por defecto al `rmiregistry` de la RMI. Una llamada típica a `bind()` tiene esta forma:

```
MiClase instancia = new MiClase(...);  
Naming.bind("rmi://anfitrion:puerto/directorio", instancia);
```

Dicho código registra una instancia de `MiClase` con el nombre `instancia` en un URL.

Para el ejemplo de `CalcMat` del subapartado 4.1, el registro, que se hace en el servidor, podría tomar esta forma:

```
CalcMatImp calculo = new CalcMatImp();  
Naming.bind("rmi://www.uv.es:9000/CentroCalculo", calculo);
```

El método `rebind()` funciona como `bind()`, pero permite volver a asociar un nombre a un objeto remoto, reemplazando al que ya tenía. Si se intenta registrar con `bind()` un objeto ya registrado, se lanzará una excepción `java.rmi.AlreadyBoundException`.

El método `lookup()` devuelve una referencia al objeto remoto especificado en el URL introducido en su argumento `nombre`. Con esa referencia, se puede llamar a sus métodos remotos. Gracias a este método, la MVJ local averigua qué MVJ proporciona o "sirve" al objeto remoto. Una vez conseguida la referencia al objeto remoto (un adaptador o *stub*), la MVJ local usará el anfitrión y el puerto incluidos en la referencia para abrir con sockets una conexión con la MVJ remota cada vez que llame a algún método remoto. Una llamada típica a `lookup()` tiene la forma

```
Naming.lookup("rmi://anfitrion:puerto/nombreobjetoremoto");
```

Para buscar la instancia `calculo` de `CalcMat` registrada pocas líneas más arriba, debería escribirse en el cliente

```
Naming.lookup("rmi://www.uv.es:9000/CentroCalculo/calculo");
```

Como `lookup()` devuelve un objeto genérico del tipo de la interfaz `java.rmi.Remote`, se hace necesaria la conversión

```
CalcMat miCalculo = (CalcMat)
Naming.lookup("rmi://www.uv.es:9000/CentroCalculo/calcu");
```

para poder usarlo. He aquí un ejemplo de uso:

```
miCalculo.hacerCalculoMuyComplicado(2.5656, 3.141592653589);
```

La clase `RMISecurityManager` proporciona un controlador de seguridad para las aplicaciones que usan código procedente de descargas. Si no se ha establecido un `RMISecurityManager`, el cargador de clases de RMI no permitirá descargar ninguna clase remota desde un anfitrión que no sea el local; esto no es válido para los *applets*, que usan otro controlador de seguridad.

java.rmi.registry

El paquete `java.rmi.registry` proporciona las interfaces `Registry` y `RegistryHandler`, así como la clase `LocateRegistry`.

La interfaz `Registry` define los métodos `bind()`, `lookup()`, `rebind()`, `unbind()` y `list()` de la clase `Naming`, y define la constante `public static final int REGISTRY_PORT`, correspondiente al puerto TCP que se usa para registrar objetos.

La interfaz `RegistryHandler` figura en la documentación de Sun como *deprecated* (censurada o desaprobada) desde la versión 1.2 del JDK y no debe utilizarse.

La clase `LocateRegistry` se usa para recuperar objetos `Registry` de un par anfitrión-puerto o para crear objetos `Registry` a partir de un número de puerto o de puertos y de factorías de sockets RMI. Los métodos `getRegistry()` se encargan de recuperar los objetos `Registry`; y los métodos `createRegistry()` de crearlos. Las factorías de sockets RMI, introducidas en JDK 1.3, permiten usar sockets que codifican o comprimen datos, y sockets que no sean TCP.

java.rmi.server

Este paquete proporciona clases (`ObjID`, `RemoteObject`, `RemoteServer`, `RemoteStub`, `RMIClassLoader`, `RMIClassLoaderSpi`, `RMI SocketFactory`, `UID` y `UnicastRemoteObject`) e interfaces (`RemoteRef`, `RMIClientSocketFactory`, `RMIFailureHandler`, `RMI ServerSocketFactory`, `ServerRef` y `Unreferenced`), así como un conjunto de excepciones, para la parte de servidor de las aplicaciones con RMI. Sólo he mencionado aquellas clases e interfaces que no figuran como *deprecated* en el JDK 1.4. Mientras escribo estas líneas, el JDK 1.5 está en versión beta (Sun ha cambiado el nombre de JDK 1.5 por JDK 5.0).

La clase `ObjID` genera identificadores de objetos que los anfitriones declaran como remotos. Proporciona métodos para crear identificadores y para leerlos de flujos de bytes o escribirlos en éstos. Vimos en el apartado anterior que las referencias remotas contienen números únicos que identifican a los objetos remotos referenciados. Pues bien, esta clase genera esos números.

La clase `RemoteObject` implementa, para objetos remotos, el comportamiento de `java.lang.Object` (superclase de todos las clases de Java; cuando se dice que la clase X hereda de Y, se sobreentiende que hereda de Y y de `java.lang.Object`), amén de implementar la interfaz `java.rmi.Remote`. Por tanto, implementa los métodos `hashCode()`, `equals()` y `toString()`.

La clase `RemoteServer` es subclase de `RemoteObject`. Constituye la superclase común para todas las implementaciones de objetos remotos. Su método `static String getClientHost()` devuelve el identificador del anfitrión que está ejecutando la invocación remota de métodos.

La clase `RemoteStub` también es subclase de `RemoteObject`. Esta clase abstracta es la superclase común para los *stubs* de los clientes.

La clase `RMIClassLoader` incluye métodos estáticos para permitir la carga dinámica de clases remotas. Si un cliente o servidor de una aplicación RMI necesita cargar una clase desde un lugar remoto, llama a `RMIClassLoader` para que lo haga. `RMIClassLoaderSpi` implementa algunos de los métodos de la anterior.

La clase `RMISocketFactory` es usada por RMI para obtener *sockets* de cliente y de servidor para las llamadas RMI.

La clase `UnicastRemoteObject` es subclase de `RemoteServer` e incluye la implementación por defecto de los objetos remotos. Con ella se puede llamar a los métodos remotos mediante conexiones TCP ligadas por defecto al puerto 1099. Por lo general, cuando se necesitan objetos que tengan un comportamiento remoto más especializado, se crean heredando de `UnicastRemoteObject`. También podrían heredar directamente de `RemoteObject`; pero entonces deberían implementarse los métodos `hashCode()`, `equals()` y `toString()`, heredados de `java.lang.Object`.

Cuando una clase hereda de `UnicastRemoteObject`, debe incluir un constructor que declare que puede arrojar excepciones del tipo `RemoteException`. El constructor, al llamar a `super()`, activa el código de `UnicastRemoteObject` encargado de enlazar el objeto con la infraestructura de la RMI e inicializa el objeto remoto. Enlazar un objeto con la infraestructura de la RMI es lo mismo que exportarlo, esto es, dejarlo disponible para que acepte peticiones remotas por un puerto.

En el caso de que se usen objetos remotos que no sean instancias de una subclase de `UnicastRemoteObject`, se deberá codificar explícitamente la exportación mediante `UnicastRemoteObject.exportObject(Remote objeto)` o `UnicastRemoteObject.exportObject(Remote objeto, int numpuerto)`. El primero exporta `objeto` de modo que permanezca a la escucha de peticiones por un puerto anónimo, establecido por la RMI; y el segundo lo exporta para que escuche por un puerto establecido por el programador.

La interfaz `RemoteRef` es usada por los objetos `RemoteStub` para referirse a objetos remotos. Incluye métodos para llamar a métodos de objetos remotos, para comparar objetos remotos y para trabajar con aquellos objetos que implementan la interfaz `RemoteCall`.

La interfaz `RMIClientSocketFactory` es usada por RMI para obtener *sockets* de cliente para las llamadas RMI.

Su único método es `public Socket SocketcreateSocket(String anfitrion, int puerto) throws IOException`, que crea un *socket* de cliente asociado al par anfitrión-puerto especificado.

La interfaz `RMIFailureHandler` especifica los métodos que se encargan de manejar los fallos derivados de los intentos de crear `ServerSockets`.

La interfaz `RMIServerSocketFactory` es usada por RMI para obtener *sockets* de servidor para las llamadas RMI.

Su único método es `public ServerSocket createServerSocket(int puerto) throws IOException`, que crea un *socket* de servidor para el puerto especificado.

La interfaz `ServerRef` extiende la interfaz `RemoteRef` y es implementada por los objetos remotos para poder acceder a sus objetos `RemoteStub`.

La interfaz `Unreferenced` se usa para que los objetos remotos puedan recibir mensajes de aviso cuando no existan más clientes que mantengan referencias a un objeto remoto.

java.rmi.activation

Añadido en el JDK 1.2, este paquete permite activar remotamente objetos, desactivarlos cuando no se esté trabajando con ellos y reactivarlos cuando se precise, conservando el estado que tenían antes de ser desactivados.

Este paquete resulta muy útil cuando se desarrollan aplicaciones distribuidas con miles de objetos distribuidos por muchas máquinas, pues permite aprovechar al máximo los recursos de las máquinas, que no tienen que cargar con objetos que temporalmente no usan.

java.rmi.dgc

Proporciona las clases e interfaces que utiliza el recolector de basura (distribuida) de la RMI. Rara vez el programador tendrá que bregar con este paquete, salvo que decida sustituir el sistema de recolección de basura de la RMI por otro propio, práctica esta desaconsejable por completo.

ALGUNAS CLASES E INTERFACES DE java.rmi

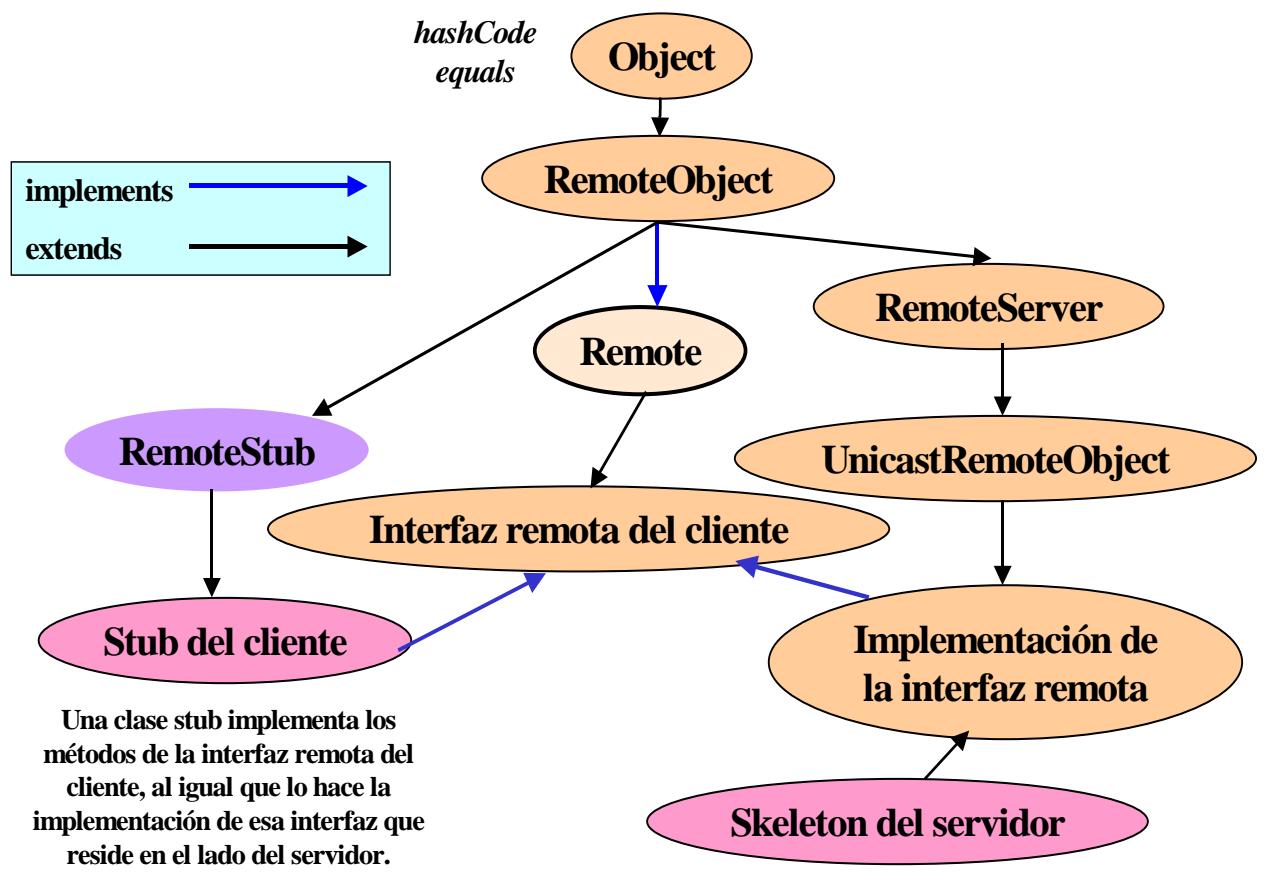


Figura 104. Algunas clases e interfaces del paquete java.rmi

5.4. Ejemplo completo del desarrollo e implementación de una aplicación con RMI

En este subapartado se explicarán los pasos que hay que seguir para construir una aplicación distribuida con la RMI de Java. Para ello, se desarrollará una aplicación de cálculo vectorial que toma un vector tridimensional y devuelve su módulo y el vector unitario asociado.

Todo el ejemplo se va a implementar considerando que el cliente y el servidor se ejecutan en el mismo anfitrión; es decir, que todos los ficheros .class de la aplicación se encuentran en una misma máquina. En el siguiente subapartado daré las directrices para distribuir los .class cuando clientes y servidores residen en anfitriones distintos.

La secuencia de pasos para desarrollar una aplicación con RMI se representa en la siguiente figura.

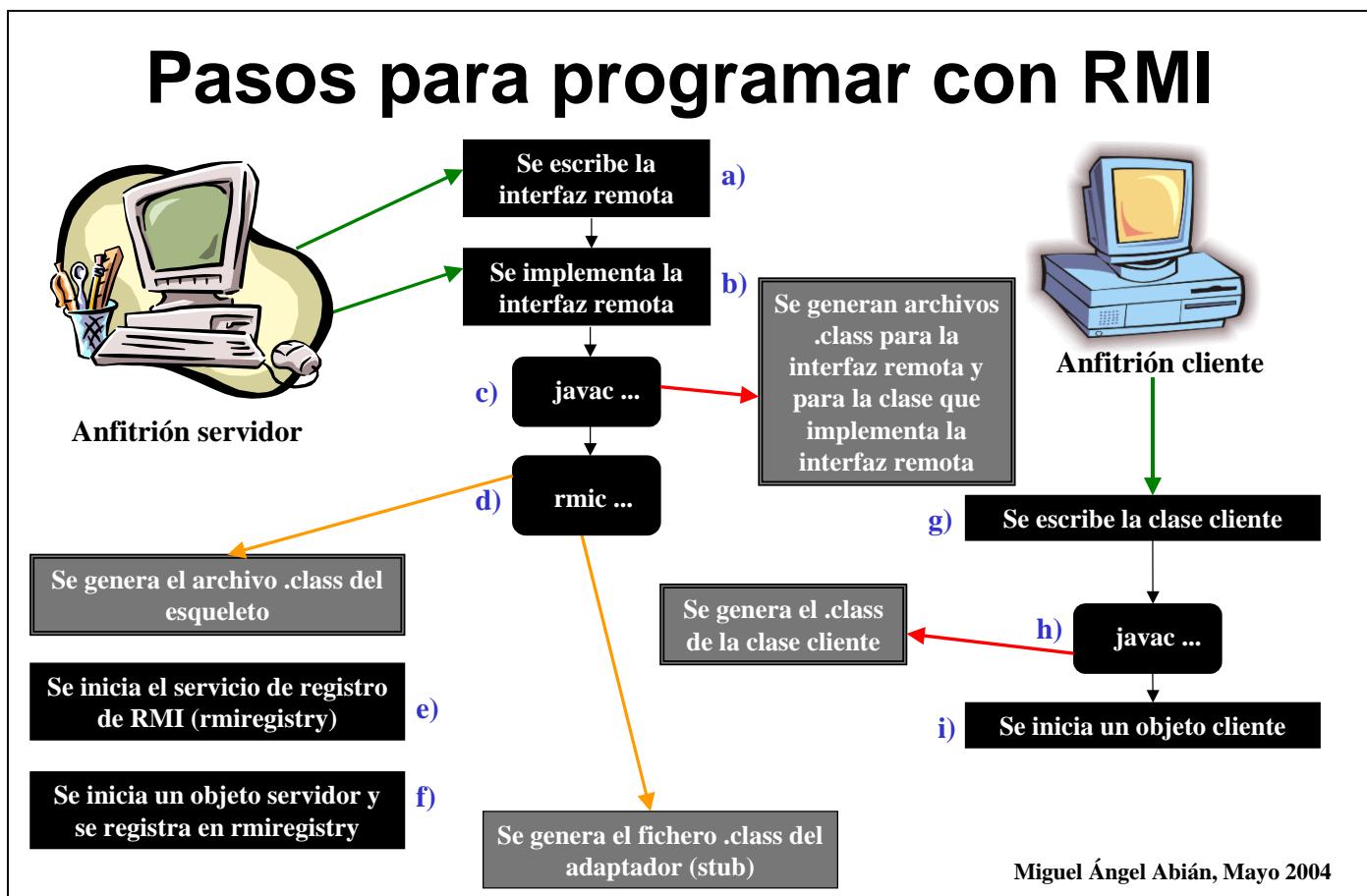


Figura 105. Secuencia general de pasos para programar aplicaciones RMI

Nota: La versión del JDK usada para la capturas de pantallas es la 1.4.2, que viene por defecto en el JBuilder X; pero también he comprobado que todo el código funcionaba en la versión 1.2 y en el JDK 1.5.0 Beta 1. Todo el código que usa RMI se ha probado con los sistemas operativos Windows XP Profesional (en un PC doméstico), Red Hat 9.0 (en un PC doméstico) y Solaris 8.0 (en una estación de trabajo SUN BLADE 150).

Paso a) En primer lugar, se escribe la interfaz remota del servidor. Toda interfaz remota debe declararse como `public` y debe extender la interfaz `java.rmi.Remote`, incluida en el paquete `java.rmi`. Dicha interfaz debe definir los métodos a los que el servidor permitirá el acceso remoto.

Cada método de la interfaz remota tiene que capturar la excepción `java.rmi.RemoteException` o declararla con `throws`.

Ejemplo 17a: CalculoVectorial.java

```
/*
 * Definición de la interfaz CalculoVectorial, que se usará
 * para permitir calcular la norma de vectores 3D y sus vectores unitarios.
 */

import java.rmi.*;

// Necesariamente, la interfaz debe extender la interfaz java.rmi.Remote

public interface CalculoVectorial extends java.rmi.Remote {

    // Devuelve el módulo del vector
    double getModulo (double x, double y, double z) throws java.rmi.RemoteException;
    // Devuelve la primera coordenada del vector unitario asociado al vector
    double getUnitarioX (double x, double y, double z) throws java.rmi.RemoteException;
    // Devuelve la segunda coordenada del vector unitario asociado al vector
    double getUnitarioY (double x, double y, double z) throws java.rmi.RemoteException;
    // Devuelve la tercera componente del vector unitario asociado al vector
    double getUnitarioZ (double x, double y, double z) throws java.rmi.RemoteException;

}
```

Paso b) Se implementa la interfaz remota. La clase remota que la implementa debe heredar de la clase `RemoteServer`. Tal como ya se dijo, lo habitual es hacer que herede de la clase `UnicastRemoteObject` del paquete `java.rmi.Server`. Si no se hiciera así, habría que exportar explícitamente los objetos remotos antes de registrarlos mediante

```
UnicastRemoteObject.exportObject(objetoremoto);
```

Ejemplo 17b: CalculoVectorialImp.java

```
/*
 * Implementación de la interfaz CalculoVectorial.
 * Esta clase hereda de UnicastRemoteObject
 * (que es el modo más simple para crear clases disponibles
 * para llamadas remotas).
 */
```

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

/** Esta clase se usará para generar (con rmic) los
 * ficheros stub y skeleton.
 */

public class CalculoVectorialImp extends UnicastRemoteObject implements
    CalculoVectorial {
    // Esta clase implementa a la interfaz CalculoVectorial.

    /** El constructor debe lanzar la excepción RemoteException.
     */

    public CalculoVectorialImp() throws RemoteException {
        super();
        try {
            // Se registra la clase con el servidor de registro de RMI en el anfitrión local
            // con el nombre "CalculosVector". Al no indicarse puerto, se usará el
            // puerto TCP 1099.
            Naming.rebind("rmi://localhost/CalculosVector", this);
        }
        catch (AccessException e1) {
            System.out.println("Se rechazó el acceso. Su sistema no tiene los permisos " +
                               "necesarios para realizar la operación remota.");
            System.exit(-1);
        }
        catch (java.net.MalformedURLException e2) {
            System.out.println("La URL para el registro no es válida.");
            System.exit(-1);
        }
    }

    /**
     * calcularModulo es un método no remoto y privado que devuelve el módulo de un vector 3D.
     *
     * @param x,y,z son las tres coordenadas (en formato double) del vector.
     * @return devuelve el módulo como un double.
     */
    private double calcularModulo (double x, double y, double z) {
        return (Math.sqrt(x*x + y*y + z*z));
    }

    /**
     * calcularUX es un método no remoto y privado que calcula la coordenada X del vector
     * unitario asociado al que se le pasa como argumento.
     * @param x,y,z son las tres coordenadas (en formato double) del vector.
     * @return devuelve la coordenada X como un double.
     */
    private double calcularUX (double x, double y, double z) {
        return (x/calcularModulo(x, y, z));
    }

    /**
     * calcularUY es un método no remoto y privado que calcula la coordenada Y del vector
     *
```

```
* unitario asociado al que se le pasa como argumento.  
* @param x,y,z son las tres coordenadas (en formato double) del vector.  
* @return devuelve la coordenada Y como un double.  
*/  
  
private double calcularUY (double x, double y, double z) {  
    return (y/calcularModulo(x, y, z));  
}  
  
/** calcularUZ es un método no remoto y privado que calcula la coordenada Z del vector  
 * unitario asociado al que se le pasa como argumento.  
 * @param x,y,z son las tres coordenadas (en formato double) del vector.  
 * @return devuelve la coordenada Z como un double.  
*/  
  
private double calcularUZ (double x, double y, double z) {  
    return (z/calcularModulo(x, y, z));  
}  
  
/** getModulo es un método remoto accesor de calcularModulo().  
 * @param x,y,z son las tres coordenadas (en formato double) del vector.  
 * @return devuelve la coordenada Z como un double.  
*/  
  
public double getModulo (double x, double y, double z) throws RemoteException {  
    return calcularModulo(x, y, z);  
}  
  
/** getUnitarioX es un método remoto accesor de calcularUX.  
 * @param x,y,z son las tres coordenadas (en formato double) del vector.  
 * @return devuelve la coordenada X como un double.  
*/  
  
public double getUnitarioX (double x, double y, double z) throws RemoteException {  
    return calcularUX(x, y, z);  
}  
  
/** getUnitarioY es un método remoto accesor de calcularUY.  
 * @param x,y,z son las tres coordenadas (en formato double) del vector.  
 * @return devuelve la coordenada Y como un double.  
*/  
  
public double getUnitarioY (double x, double y, double z) throws RemoteException {  
    return calcularUY(x, y, z);  
}  
  
/** getUnitarioZ es un método remoto accesor de calcularUZ.  
 * @param x,y,z son las tres coordenadas (en formato double) del vector.  
 * @return devuelve la coordenada Z como un double.  
*/  
  
public double getUnitarioZ (double x, double y, double z) throws RemoteException {  
    return calcularUZ(x, y, z);  
}  
}
```

Paso c) Se compilan la interfaz remota y la clase que la implementa. En nuestro caso, tendremos como resultado dos archivos: `CalculoVectorial.class` y `CalculoVectorialImp.class`.

Paso d) Se generan los archivos `.class stub` y `skeleton` a partir de la clase que implementa la interfaz remota (`CalculoVectorialImp`). Para ello se utiliza el compilador RMI `rmic`, sito en el subdirectorio `bin` del directorio donde se haya instalado el JDK. Para ejecutar `rmic` se puede escribir

```
rmic nombreclase
```

y se generarán los archivos `stub` y `skeleton` en el directorio actual. Si se usa

```
rmic -d directorio nombreclase
```

los archivos se guardarán en el directorio especificado. A continuación pongo dos ejemplos, uno para Windows y otro para UNIX:

```
> rmic -d C:\java\RMI\clases CalculoVectorial (Windows)
```

```
% rmic -d /usr/local/RMI/clases CalculoVectorial (UNIX)
```

Advertencia: La clase que se va a compilar con `rmic` debe estar en el directorio donde está `rmic`, debe figurar en el `CLASSPATH` local o debe definirse su ubicación con la opción `-classpath` del compilador. En caso contrario, el compilador no la encontrará.

Para este ejemplo, he obtenido por colocar todos los archivos `.java` y `.class` del ejemplo en el directorio `bin` del JDK. Me parece la mejor opción para que el lector o lectora siga los pasos, pues así no tengo que despistarle configurando el `CLASSPATH` con directorios que sólo funcionarán en mi máquina, mejor dicho, en una que tenga la misma estructura de directorios que la mía. Desde luego, lo normal al desarrollar una aplicación (con RMI o sin ella) **es configurar adecuadamente el `CLASSPATH` o usar la propiedad `-classpath` (o `-cp`) al compilar**, no colocar todos los archivos en el directorio `bin`; si no lo hago así en este texto es por claridad expositiva. Recomiendo usar dos directorios diferentes (con los subdirectorios que sean precisos): uno para los archivos del cliente y otro para los del servidor.

Si no se desea el archivo esqueleto (Java ya no necesita esqueletos desde la versión 1.2), basta con compilar así:

```
rmic -v1.2 nombreclase
```

Nota: Como ya avancé, el JDK 1.5 (ahora JDK 5.0) permite la generación dinámica de los ficheros *stub* (los *skeleton* ya no son necesarios); se vuelve innecesaria, pues, la compilación manual con `rmic`. Si no hago uso de esta cómoda propiedad es por compatibilidad con las versiones anteriores, que son las predominantes en el mercado.

Nota: Si se desea generar adaptadores y esqueletos que usen el protocolo IIOP en lugar de JRMP, hay que usar `rmic -iiop`. Esta opción permite construir aplicaciones RMI capaces de interoperar con objetos CORBA. El proceso exacto se explicará en el subapartado 6.5.

Teniendo en cuenta lo dicho, aquí está la captura de pantalla correspondiente a la compilación RMI de `CalculoVectorialImp`.

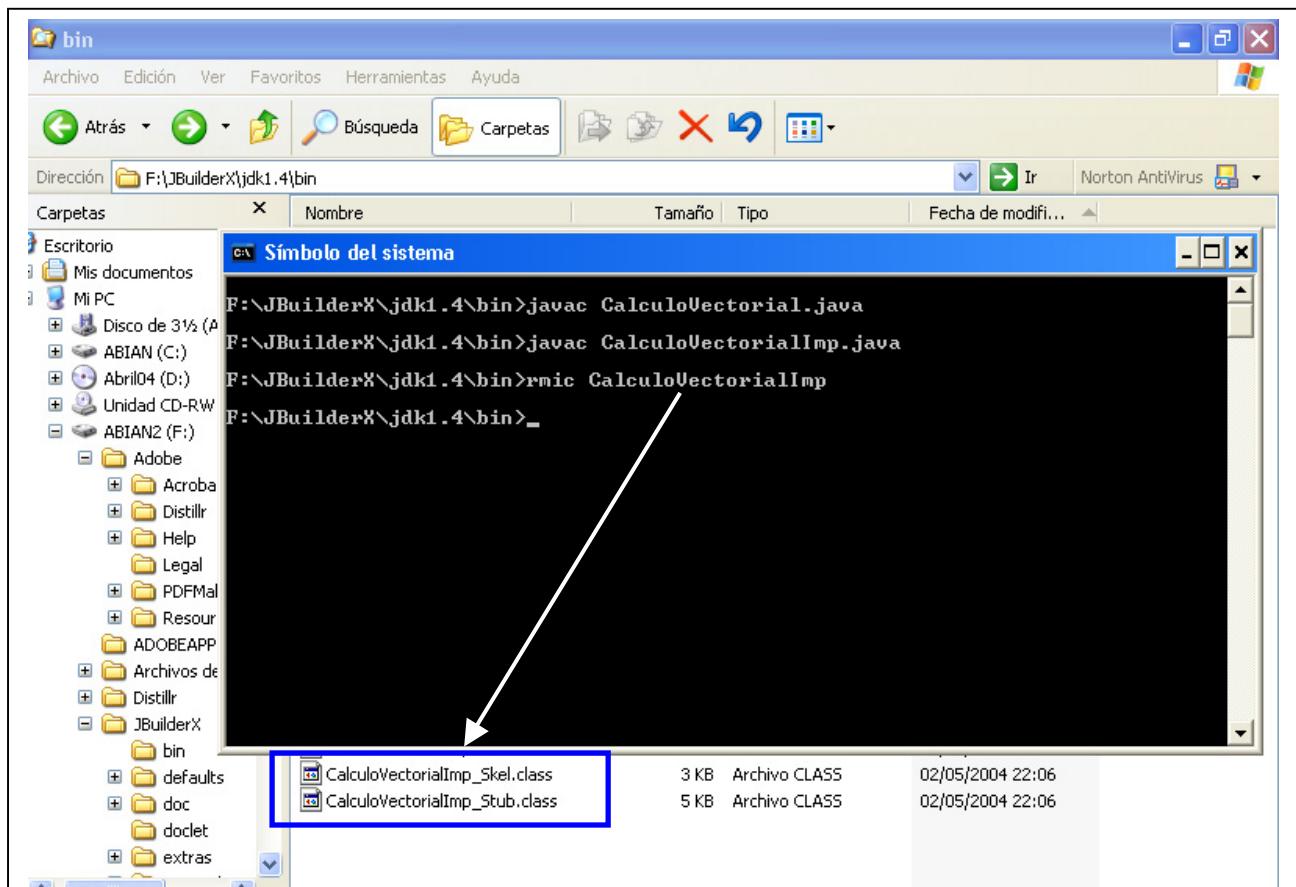


Figura 106. Resultado de compilar con rmic `CalculoVectorialImp`

Como vemos, la aplicación rmic genera dos archivos .class: CalculoVectorialImp_Skel.class y CalculoVectorialImp_Stub.class. Tal como indiqué en la página 219, mantendré CalculoVectorial.class, CalculoVectorialImp.class, CalculoVectorialImp_Skel.class, CalculoVectorialImp_Stub.class en un mismo directorio: el bin del JDK.

Por defecto, rmic no genera los archivos .java correspondientes a las clases *stub* y *skeleton*; pero pueden obtenerse mediante las opciones *-keep* y *-keepgenerated* de rmic.

Advertencia: Si se editan los archivos .java asociados a adaptadores y esqueletos, no deben modificarse. En general, nunca es necesario utilizarlos.

Éste es el código de la clase CalculoVectorialImp_Stub, generado automáticamente por la RMI de Java:

```
// Stub class generated by rmic, do not edit.  
// Contents subject to change without notice.  
  
public final class CalculoVectorialImp_Stub  
    extends java.rmi.server.RemoteStub  
    implements CalculoVectorial, java.rmi.Remote  
{  
    private static final java.rmi.server.Operation[] operations = {  
        new java.rmi.server.Operation("double getModulo(double, double, double)"),  
        new java.rmi.server.Operation("double getUnitarioX(double, double, double)"),  
        new java.rmi.server.Operation("double getUnitarioY(double, double, double)"),  
        new java.rmi.server.Operation("double getUnitarioZ(double, double, double)")  
    };  
  
    private static final long interfaceHash = -8424770855087595345L;  
  
    private static final long serialVersionUID = 2;  
  
    private static boolean useNewInvoke;  
    private static java.lang.reflect.Method $method_getModulo_0;  
    private static java.lang.reflect.Method $method_getUnitarioX_1;  
    private static java.lang.reflect.Method $method_getUnitarioY_2;  
    private static java.lang.reflect.Method $method_getUnitarioZ_3;  
  
    static {  
        try {  
            java.rmi.server.RemoteRef.class.getMethod("invoke",  
                new java.lang.Class[] {  
                    java.rmi.Remote.class,  
                    java.lang.reflect.Method.class,  
                    java.lang.Object[].class,  
                    long.class  
                });  
            useNewInvoke = true;  
            $method_getModulo_0 = CalculoVectorial.class.getMethod("getModulo", new  
java.lang.Class[] {double.class, double.class, double.class});  
        } catch (Exception e) {}  
    }  
}
```

```

        $method_getUnitarioX_1 = CalculoVectorial.class.getMethod("getUnitarioX", new
java.lang.Class[] {double.class, double.class, double.class});
        $method_getUnitarioY_2 = CalculoVectorial.class.getMethod("getUnitarioY", new
java.lang.Class[] {double.class, double.class, double.class});
        $method_getUnitarioZ_3 = CalculoVectorial.class.getMethod("getUnitarioZ", new
java.lang.Class[] {double.class, double.class, double.class});
    } catch (java.lang.NoSuchMethodException e) {
        useNewInvoke = false;
    }
}

// constructors
public CalculoVectorialImp_Stub() {
    super();
}
public CalculoVectorialImp_Stub(java.rmi.server.RemoteRef ref) {
    super(ref);
}

// methods from remote interfaces

// implementation of getModulo(double, double, double)
public double getModulo(double $param_double_1, double $param_double_2, double
$params_double_3)
    throws java.rmi.RemoteException
{
    try {
        if (useNewInvoke) {
            Object $result = ref.invoke(this, $method_getModulo_0, new java.lang.Object[]
{new java.lang.Double($param_double_1), new java.lang.Double($param_double_2), new
java.lang.Double($param_double_3)}, -1906106596166222523L);
            return ((java.lang.Double) $result).doubleValue();
        } else {
            java.rmi.server.RemoteCall call = ref.newCall((java.rmi.server.RemoteObject)
this, operations, 0, interfaceHash);
            try {
                java.io.ObjectOutput out = call.getOutputStream();
                out.writeDouble($param_double_1);
                out.writeDouble($param_double_2);
                out.writeDouble($param_double_3);
            } catch (java.io.IOException e) {
                throw new java.rmi.MarshalException("error marshalling arguments", e);
            }
            ref.invoke(call);
            double $result;
            try {
                java.io.ObjectInput in = call.getInputStream();
                $result = in.readDouble();
            } catch (java.io.IOException e) {
                throw new java.rmi.UnmarshalException("error unmarshalling return", e);
            } finally {
                ref.done(call);
            }
            return $result;
        }
    } catch (java.lang.RuntimeException e) {
}
}

```

The diagram illustrates the flow of data during a RMI call, corresponding to the annotated code above. It uses callouts to explain specific sections of the code:

- Se escriben los argumentos (codificados) en el flujo de salida**: Points to the section where arguments are written to the output stream.
- Se crea un flujo de salida**: Points to the creation of an output stream.
- Se llama al método**: Points to the call to the remote method.
- Se lee el resultado del flujo de entrada y se decodifica**: Points to the reading and decoding of the result from the input stream.
- Se devuelve el resultado**: Points to the final return statement.

```
        throw e;
    } catch (java.rmi.RemoteException e) {
        throw e;
    } catch (java.lang.Exception e) {
        throw new java.rmi.UnexpectedException("undclared checked exception", e);
    }
}

// implementation of getUnitarioX(double, double, double)
public double getUnitarioX(double $param_double_1, double $param_double_2, double
$param_double_3)
    throws java.rmi.RemoteException
{
    try {
        if (useNewInvoke) {
            Object $result = ref.invoke(this, $method_getUnitarioX_1, new
java.lang.Object[] {new java.lang.Double($param_double_1), new
java.lang.Double($param_double_2), new java.lang.Double($param_double_3)},
185093586419828030L);
            return ((java.lang.Double) $result).doubleValue();
        } else {
            java.rmi.server.RemoteCall call = ref.newCall((java.rmi.server.RemoteObject)
this, operations, 1, interfaceHash);
            try {
                java.io.ObjectOutput out = call.getOutputStream();
                out.writeDouble($param_double_1);
                out.writeDouble($param_double_2);
                out.writeDouble($param_double_3);
            } catch (java.io.IOException e) {
                throw new java.rmi.MarshalException("error marshalling arguments", e);
            }
            ref.invoke(call);
            double $result;
            try {
                java.io.ObjectInput in = call.getInputStream();
                $result = in.readDouble();
            } catch (java.io.IOException e) {
                throw new java.rmi.UnmarshalException("error unmarshalling return", e);
            } finally {
                ref.done(call);
            }
            return $result;
        }
    } catch (java.lang.RuntimeException e) {
        throw e;
    } catch (java.rmi.RemoteException e) {
        throw e;
    } catch (java.lang.Exception e) {
        throw new java.rmi.UnexpectedException("undclared checked exception", e);
    }
}
```

```
// implementation of getUnitarioY(double, double, double)
public double getUnitarioY(double $param_double_1, double $param_double_2, double
$params_double_3)
    throws java.rmi.RemoteException
{
    try {
        if (useNewInvoke) {
            Object $result = ref.invoke(this, $method_getUnitarioY_2, new
java.lang.Object[] {new java.lang.Double($param_double_1), new
java.lang.Double($param_double_2), new java.lang.Double($param_double_3)}, -
6040331168870031281L);
            return ((java.lang.Double) $result).doubleValue();
        } else {
            java.rmi.server.RemoteCall call = ref.newCall((java.rmi.server.RemoteObject)
this, operations, 2, interfaceHash);
            try {
                java.io.ObjectOutput out = call.getOutputStream();
                out.writeDouble($param_double_1);
                out.writeDouble($param_double_2);
                out.writeDouble($param_double_3);
            } catch (java.io.IOException e) {
                throw new java.rmi.MarshalException("error marshalling arguments", e);
            }
            ref.invoke(call);
            double $result;
            try {
                java.io.ObjectInput in = call.getInputStream();
                $result = in.readDouble();
            } catch (java.io.IOException e) {
                throw new java.rmi.UnmarshalException("error unmarshalling return", e);
            } finally {
                ref.done(call);
            }
            return $result;
        }
    } catch (java.lang.RuntimeException e) {
        throw e;
    } catch (java.rmi.RemoteException e) {
        throw e;
    } catch (java.lang.Exception e) {
        throw new java.rmi.UnexpectedException("undclared checked exception", e);
    }
}

// implementation of getUnitarioZ(double, double, double)
public double getUnitarioZ(double $param_double_1, double $param_double_2, double
$params_double_3)
    throws java.rmi.RemoteException
{
    try {
        if (useNewInvoke) {
            Object $result = ref.invoke(this, $method_getUnitarioZ_3, new
java.lang.Object[] {new java.lang.Double($param_double_1), new
java.lang.Double($param_double_2), new java.lang.Double($param_double_3)}, -
5640072250555346327L);
        }
    }
```

```
        return ((java.lang.Double) $result).doubleValue();
    } else {
        java.rmi.server.RemoteCall call = ref.newCall((java.rmi.server.RemoteObject)
this, operations, 3, interfaceHash);
        try {
            java.io.ObjectOutput out = call.getOutputStream();
            out.writeDouble($param_double_1);
            out.writeDouble($param_double_2);
            out.writeDouble($param_double_3);
        } catch (java.io.IOException e) {
            throw new java.rmi.MarshalException("error marshalling arguments", e);
        }
        ref.invoke(call);
        double $result;
        try {
            java.io.ObjectInput in = call.getInputStream();
            $result = in.readDouble();
        } catch (java.io.IOException e) {
            throw new java.rmi.UnmarshalException("error unmarshalling return", e);
        } finally {
            ref.done(call);
        }
        return $result;
    }
} catch (java.lang.RuntimeException e) {
    throw e;
} catch (java.rmi.RemoteException e) {
    throw e;
} catch (java.lang.Exception e) {
    throw new java.rmi.UnexpectedException("undclared checked exception", e);
}
}
}
```

Esta clase *stub* declara los métodos de la interfaz remota de *CalculoVectorialImp* y delega las llamadas de los métodos en la implementación RMI, que se encarga de serializar los argumentos y de enviar los datos al servidor.

Paso e) Se inicia el servicio de registro de RMI. Cualquier anfitrión que quiera exportar referencias remotas (adaptadores) a los objetos locales de Java, de modo que éstos puedan llamar a los métodos remotos, debe estar ejecutando un servidor de registro RMI.

En Windows, se arranca este servicio mediante

```
start rmiregistry
```

En UNIX se usa `rmiregistry &`. El proceso de usar `rmiregistry` es muy parecido a llamar al demonio `portmap` de un sistema UNIX.

Si se desea arrancar este servicio para que escuche por un puerto que no sea el estándar (1099), hay que usar

start rmiregistry numpuerto (**Windows**)

O

rmiregistry numpuerto & (**UNIX**)

Al ejecutarse, la aplicación rmiregistry crea un objeto Registry que escucha por un puerto y entra en un letargo aparente, a la espera de procesos clientes que registren objetos remotos y de procesos clientes que busquen objetos remotos en el registro RMI. Nótese que, para el servicio de registro, tanto los clientes RMI como los servidores RMI son clientes. Los procesos de registro de objetos remotos deben ejecutarse en el mismo anfitrión donde se está ejecutando rmiregistry; en caso contrario, se lanzará una excepción `java.rmi.AccessException`.

En mi sistema (Windows XP), ejecutar rmiregistry lanza una nueva ventana negra.

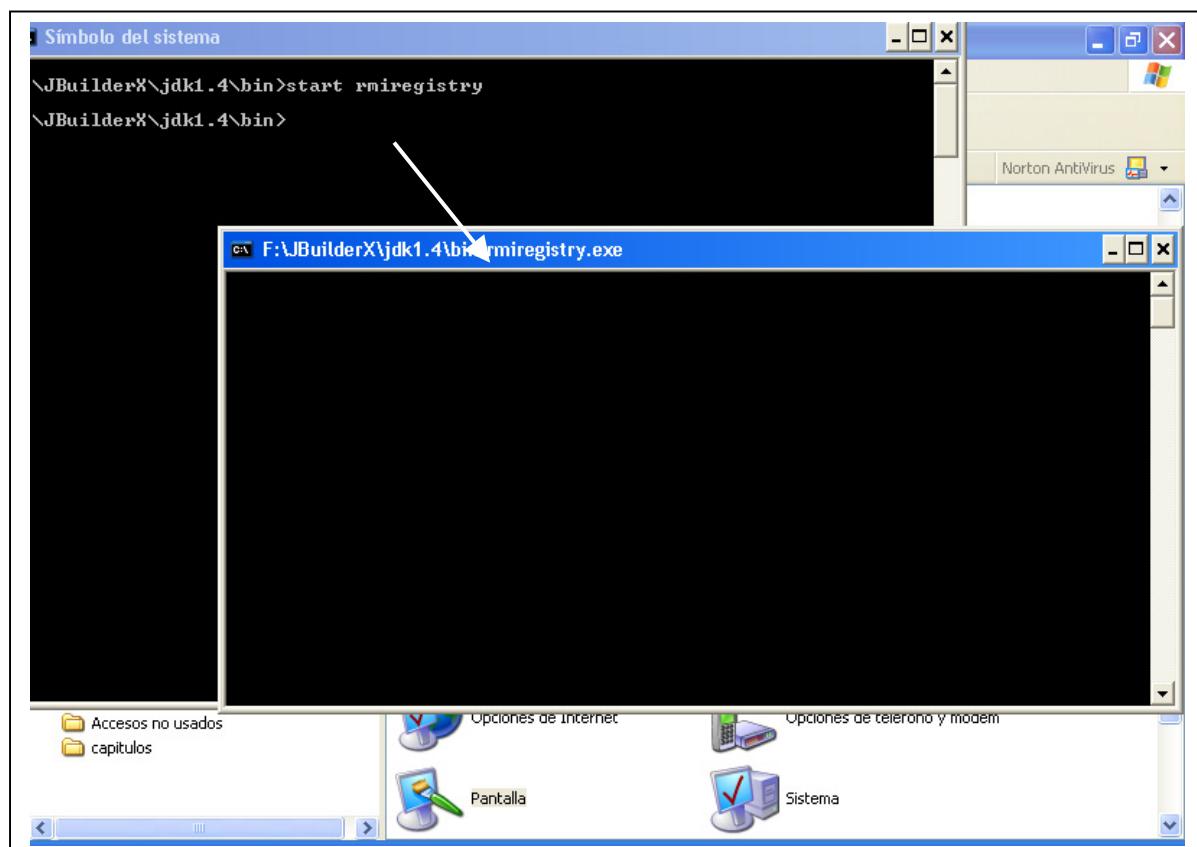


Figura 107. Ejecución de rmiregistry

Advertencia: En general, si el servidor y el cliente están en la misma máquina, el registro debe iniciarse incluyendo los archivos *stub* en el CLASSPATH. En el ejemplo no es necesario hacerlo porque todos los archivos están en un mismo directorio (bin).

Paso f) Se inicia un objeto servidor y se registra en rmiregistry. La clase que actúa como servidor en el ejemplo es ServidorCalculoVectorial, que se limita a crear una instancia de CalculoVectorialImp.

Ejemplo 17c: ServidorCalculoVectorial.java

```
/*
 * Clase que actúa como servidor. Se limita a crear una instancia de
 * CalculoVectorialImp. Al crearla, se registra en el servicio de registro RMI.
 */
import java.rmi.*;
import java.rmi.server.*;

public class ServidorCalculoVectorial {

    public static void main(String args[]) {
        try {
            new CalculoVectorialImp();
            System.out.println("Servicio de cálculo vectorial disponible.");
        } catch (Exception e) {
            System.out.println("No pudo arrancarse el servicio.");
        }
    }
}
```

Una vez compilada y ejecutada, el resultado se muestra en la siguiente captura de pantalla.

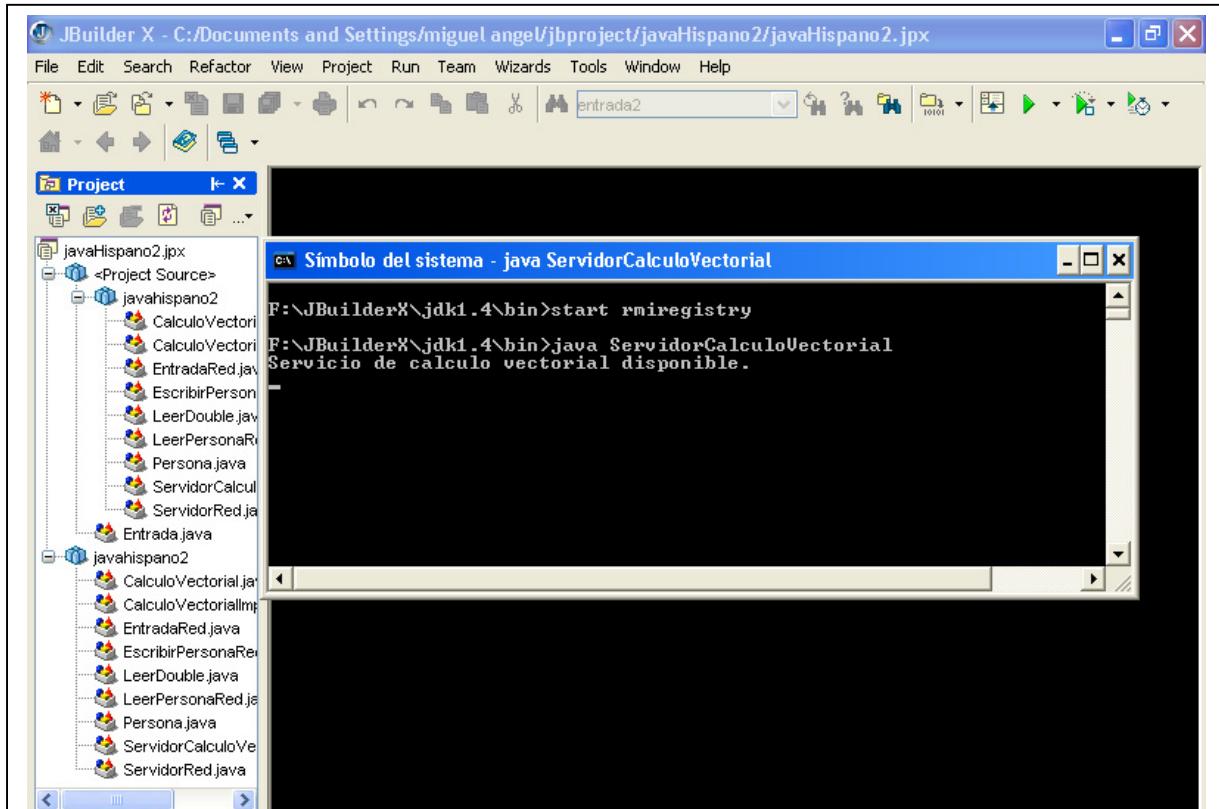


Figura 108. El servicio de cálculo vectorial en marcha

Paso q) Se escribe la clase cliente. En ella se llama a un objeto CalculoVectorial con el nombre que se le dio en CalculoVectorialImp (CalculosVector)

Ejemplo 17d: ClienteCalculoVectorial.java

```
/*
 * Cliente de ejemplo para ServidorCalculoVectorial
 *
 * Muestra por la salida estándar el módulo de un vector y las
 * coordenadas del vector unitario asociado.
 */

import java.rmi.*;
import java.rmi.registry.*;

public class ClienteCalculoVectorial {

    public static void main(String args[]) {
        try {
            // El servidor de registro de RMI devuelve un objeto genérico, que debe
            // ser convertido al tipo correspondiente. Ojo: el nombre por el que buscamos
            // debe coincidir con el que se le dio al servicio en CalculosVectorImp.
            CalculoVectorial cv =
                (CalculoVectorial)Naming.lookup("rmi://localhost/CalculosVector");
            System.out.println("El módulo es: " + cv.getModulo(7, 2, 4));
            System.out.println("La primera coordenada del vector unitario es: " +
                cv.getUnitarioX(7, 2, 4));
            System.out.println("La segunda coordenada del vector unitario es: " +
                cv.getUnitarioY(7, 2, 4));
            System.out.println("La tercera coordenada del vector unitario es: " +
                cv.getUnitarioZ(7, 2, 4));
        }
        catch (AccessException e1) {
            System.out.println("Se rechazó el acceso. Su sistema no tiene los permisos " +
                "necesarios para realizar la operación remota.");
            e1.printStackTrace();
        }
        catch (NotBoundException e2) {
            System.out.println("El nombre del objeto no está asociado a un objeto remoto " +
                "registrado.");
            e2.printStackTrace();
        }
        catch (java.net.MalformedURLException e3) {
            System.out.println("El nombre del objeto no está asociado a un objeto remoto " +
                "registrado.");
            e3.printStackTrace();
        }
        catch (RemoteException e4) {
            System.out.println("No se encontró el servicio o hubo errores al llamar a los " +
                "métodos remotos.");
        }
    }
}
```

```
        e4.printStackTrace();
    }
}

}
```

Paso h) Se compila la clase cliente. El resultado para el ejemplo es el archivo ClienteCalculoVectorial.class.

Paso i) Se inicia un objeto cliente. Tras ejecutar la clase ClienteCalculoVectorial, mi sistema muestra la imagen de la captura de pantalla siguiente.

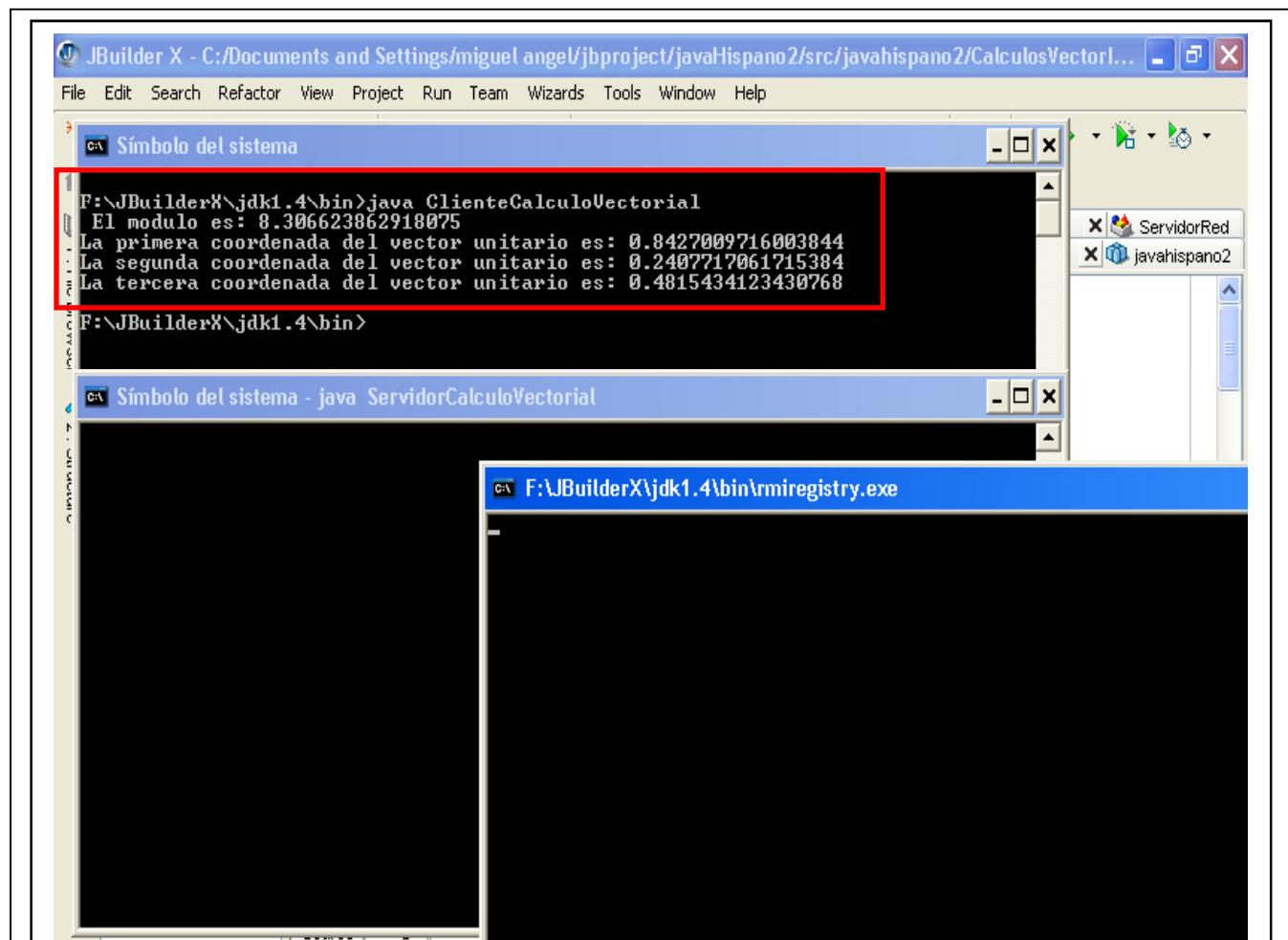


Figura 109. Resultado de una llamada al servicio de cálculo vectorial

Desde luego, si uno se toma todo este trabajo para devolver el módulo de un vector, a su jefe le acudirá enseguida a la cabeza la temida frase “Despido procedente” y alguna que otra maldición dirigida a quien le entrevistó para el puesto (aunque seguramente fue él); pero hay que tener en cuenta que es un mero ejemplo.

Un servidor RMI que lea datos de una base de datos (mediante JDBC) y que los ofrezca a los clientes para que los consulten, los modifiquen y, luego, le transmitan las modificaciones es un ejemplo típico de aplicación RMI empresarial. RMI se integra muy bien con la versión empresarial de Java (J2EE) y resulta muy sencillo integrarla con los *Enterprise Java Beans*. JINI, una tecnología Java destinada a la gestión de periféricos en red, usa RMI.

5.5. Distribución de las aplicaciones RMI: carga dinámica de clases con RMI

En el ejemplo del apartado anterior he considerado que todas las clases se hallan en una misma máquina. En una aplicación distribuida, esa situación sólo suele darse durante el proceso de depuración. En teoría, es posible colocar en cada máquina las clases de la aplicación, de manera que cada anfitrión pueda acceder a ellas configurando adecuadamente su `CLASSPATH` local. Aun cuando se puede trabajar de dicha manera, en muchos casos no constituye una solución viable. Imaginemos que tenemos una aplicación distribuida cuya implementación cambia a menudo, pero cuya interfaz no. Cada vez que se modifica la implementación, habría que distribuir los nuevos archivos entre todos los clientes. Dependiendo del número de éstos, de sus ubicaciones y de la frecuencia de las actualizaciones, la tarea de distribución podría volverse sumamente compleja y pesada.

La RMI de Java, al permitir la carga dinámica de clases, simplifica el proceso de distribución y hace posible que los clientes RMI puedan acceder a nuevas versiones de los servicios RMI sin necesidad de distribuir a los clientes los archivos `.class` del código cada vez que se modifica el código fuente. CORBA está mucho más limitado que Java en cuanto a la distribución de las aplicaciones y la carga dinámica de clases.

En una aplicación con RMI, lo habitual es distribuir los ficheros `.class` asociados a las clases e interfaces de la aplicación entre varios anfitriones, de modo que los clientes y servidores puedan acceder a ellos.

En el cliente, el cargador de clases debe poder acceder a los archivos `.class` de los siguientes elementos:

- La interfaz remota.
- La clase adaptadora o *stub* que implementa a la interfaz remota.
- Las clases del servidor cuyas instancias usa el cliente.
- Las clases propias del cliente.

En el servidor, el cargador de clases debe poder acceder a los archivos `.class` de los siguientes elementos:

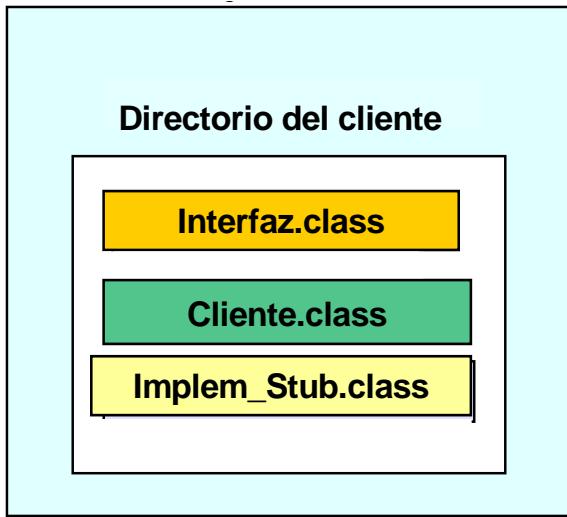
- La interfaz remota.
- La clase que implementa a la interfaz remota.
- La clase esqueleto o *skeleton* asociada a la clase que implementa la interfaz remota (sólo es obligatoria en el JDK 1.1).
- La clase adaptadora o *stub* que implementa a la interfaz remota.

- Las clases propias del servidor.

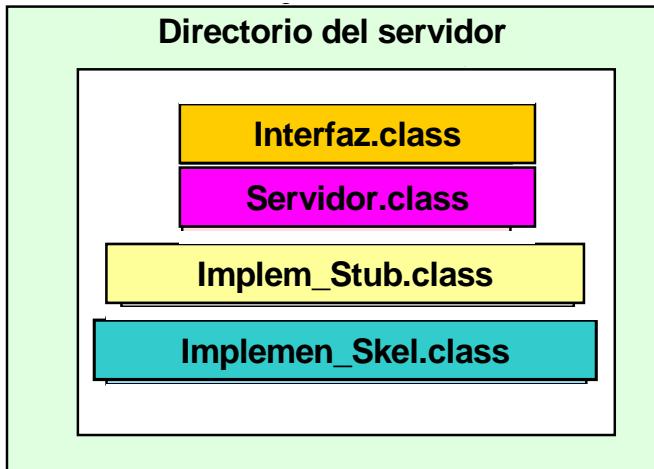
Lo dicho se representa en la siguiente figura.

DISTRIBUCIÓN DE LOS ARCHIVOS EN UNA APLICACIÓN RMI

Anfitrión del cliente



Anfitrión del servidor



Las clases skeleton no son necesarias en el JDK 1.2 y posteriores

Figura 110. Distribución de los archivos en una aplicación RMI

Gracias a la posibilidad de la carga dinámica de clases, la RMI de Java proporciona un sencillo y potente mecanismo para distribuir aplicaciones. Vimos ya en el subapartado 5.2.1 lo que sucede cuando un objeto intenta deserializar un flujo que contiene algún objeto:

- a) Se lee la descripción de la clase, incluida en el flujo.
- b) El cargador de clases de Java intenta cargar los *bytecodes* de esa clase desde el directorio actual.
- c) Si no la encuentra, continúa buscando en los directorios especificados en el `CLASSPATH` local.
- d) Si no la encuentra, recurre al `codebase` que figura en el flujo (siempre que se haya especificado la propiedad `java.rmi.server.codebase`). Esta propiedad puede especificarse al ejecutar las clases.

En consecuencia, colocando los archivos adecuados en un lugar común y especificando la propiedad `java.rmi.server.codebase`, clientes y servidores pueden acceder a ellos pese a no tenerlos en los anfitriones donde se ejecuten. Los candidatos idóneos para ser depositados en un lugar común son los archivos `.class` de las clases adaptadoras o *stub*, porque son necesarios para clientes y servidores. Las interfaces también deben estar disponibles para clientes y servidores; pero su distribución no plantea problemas, pues rara vez cambian.

La configuración más frecuente (y la que usaré más adelante) para distribuir una aplicación RMI consiste en hacer accesibles todos los archivos adaptadores mediante un servidor *web* o FTP, con lo que se evita tenerlos en los anfitriones, y en configurar adecuadamente la propiedad `java.rmi.server.codebase`. Sun proporciona un pequeño servidor HTTP para RMI, muy sencillo de usar, que incluyo en la página siguiente.

La configuración descrita sólo es una de las posibles, existen otras muchas: por ejemplo, se pueden usar clientes de tipo *applet* (que no necesitan tener ningún `.class` en el sistema local), o bien clientes o servidores que se limiten a descargar todos los *bytecodes* necesarios (interfaz remota, etc.) desde un servidor *web* o FTP.

El uso de *applets* como clientes RMI vuelve trivial la instalación y configuración de los clientes. Si se opta por una distribución mediante *applets* RMI para el lado del cliente, basta con tener un navegador compatible con Java para poder acceder siempre a las versiones más actuales de los clientes. En este caso, las peticiones de carga dinámica de los `.class` pasarán por el navegador *web*, que las traducirá a peticiones HTTP y las enviará al servidor *web* donde se almacenen los `.class`. Una vez descargados los *bytecodes*, se ejecutarán; después, las llamadas remotas serán conducidas por los objetos del cliente hasta los objetos servidores.

Nota: Si ha leído detenidamente los párrafos anteriores, habrá notado que afirmo que los archivos `.class` de las clases adaptadoras (*stubs*) son necesarias para clientes y servidores. No se trata de un error. Los adaptadores tratan con las llamadas remotas en el lado del cliente, pero también se necesitan en el lado del servidor.

En la documentación de Sun sobre RMI, la necesidad de los adaptadores en el lado del servidor no queda clara; pero la prueba del algodón nunca falla: si se intenta ejecutar un objeto servidor sin acceso (ya sea mediante `CLASSPATH` o mediante la propiedad `java.rmi.server.codebase`) a los adaptadores, se obtendrá un error “*Stub class not found*”.

El lado del servidor utiliza los adaptadores cuando registra objetos remotos en el servicio de registro RMI (este servicio viene a ser, a fin de cuentas, un almacén de adaptadores y de nombres). Luego, los clientes reciben por serialización esos adaptadores cuando llaman a un método remoto mediante algún nombre de los que figuran en el servicio de registro RMI.

Servidor web de Sun: ClassFileServer.java

```
/*
 * Copyright (c) 1996, 1996, 1997 Sun Microsystems, Inc. All Rights Reserved.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
 * SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES
 * SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING
 * THIS SOFTWARE OR ITS DERIVATIVES.
 */
package server;

import java.io.*;
import java.net.*;

/**
 * The ClassFileServer implements a ClassServer that
 * reads class files from the file system. See the
 * doc for the "Main" method for how to run this
 * server.
 */
public class ClassFileServer extends ClassServer {

    private String classpath;

    private static int DefaultServerPort = 2001;

    /**
     * Constructs a ClassFileServer.
     *
     * @param classpath the classpath where the server locates classes
     */
    public ClassFileServer(int port, String classpath) throws IOException
    {
        super(port);
        this.classpath = classpath;
    }

    /**
     * Returns an array of bytes containing the bytecodes for
     * the class represented by the argument <b>path</b>.
     * The <b>path</b> is a dot separated class name with
     * the ".class" extension removed.
     *
     * @return the bytecodes for the class
     * @exception ClassNotFoundException if the class corresponding
     * to <b>path</b> could not be loaded.
     */
    public byte[] getBytes(String path)
        throws IOException, ClassNotFoundException
    {
        System.out.println("reading: " + path);
```

```
File f = new File(classpath + File.separator +
                   path.replace('.', File.separatorChar) + ".class");
int length = (int)(f.length());
if (length == 0) {
    throw new IOException("File length is zero: " + path);
} else {
    FileInputStream fin = new FileInputStream(f);
    DataInputStream in = new DataInputStream(fin);

    byte[] bytecodes = new byte[length];
    in.readFully(bytecodes);
    return bytecodes;
}

/***
 * Main method to create the class server that reads
 * class files. This takes two command line arguments, the
 * port on which the server accepts requests and the
 * root of the classpath. To start up the server: <br><br>
 *
 * <code>  java ClassFileServer <port> <classpath>
 * </code><br><br>
 *
 * The codebase of an RMI server using this webserver would
 * simply contain a URL with the host and port of the web
 * server (if the webserver's classpath is the same as
 * the RMI server's classpath): <br><br>
 *
 * <code>  java -Djava.rmi.server.codebase=http://zaphod:2001/ RMIServer
 * </code> <br><br>
 *
 * You can create your own class server inside your RMI server
 * application instead of running one separately. In your server
 * main simply create a ClassFileServer: <br><br>
 *
 * <code>  new ClassFileServer(port, classpath);
 * </code>
 */
public static void main(String args[])
{
    int port = DefaultServerPort;
    String classpath = "";

    if (args.length >= 1) {
        port = Integer.parseInt(args[0]);
    }

    if (args.length >= 2) {
        classpath = args[1];
    }

    try {
        new ClassFileServer(port, classpath);
    } catch (IOException e) {
        System.out.println("Unable to start ClassServer: " +
```

```
        e.getMessage());
        e.printStackTrace();
    }
}
```

Servidor web de Sun: ClassServer.java

```
/*
 * Copyright (c) 1996, 1996, 1997 Sun Microsystems, Inc. All Rights Reserved.
 *
 * SUN MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
 * SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY DAMAGES
 * SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING
 * THIS SOFTWARE OR ITS DERIVATIVES.
 *
 * CopyrightVersion 1.1_beta
 */

package server;

import java.io.*;
import java.net.*;

/**
 * ClassServer is an abstract class that provides the
 * basic functionality of a mini-webserver, specialized
 * to load class files only. A ClassServer must be extended
 * and the concrete subclass should define the <b>getBytes</b>
 * method which is responsible for retrieving the bytecodes
 * for a class.<p>
 *
 * The ClassServer creates a thread that listens on a socket
 * and accepts HTTP GET requests. The HTTP response contains the
 * bytecodes for the class that requested in the GET header. <p>
 *
 * For loading remote classes, an RMI application can use a concrete
 * subclass of this server in place of an HTTP server. <p>
 *
 * @see ClassFileServer
 */
public abstract class ClassServer implements Runnable {

    private ServerSocket server = null;
    private int port;

    /**
     * Constructs a ClassServer that listens on <b>port</b> and
     * obtains a class's bytecodes using the method <b>getBytes</b>.
     *
     * @param port the port number
     * @exception IOException if the ClassServer could not listen
    
```

```

        *      on <b>port</b>.
    */
protected ClassServer(int port) throws IOException
{
    this.port = port;
    server = new ServerSocket(port);
    newListener();
}

/**
 * Returns an array of bytes containing the bytecodes for
 * the class represented by the argument <b>path</b>.
 * The <b>path</b> is a dot separated class name with
 * the ".class" extension removed.
 *
 * @return the bytecodes for the class
 * @exception ClassNotFoundException if the class corresponding
 * to <b>path</b> could not be loaded.
 * @exception IOException if error occurs reading the class
 */
public abstract byte[] getBytes(String path)
    throws IOException, ClassNotFoundException;

/**
 * The "listen" thread that accepts a connection to the
 * server, parses the header to obtain the class file name
 * and sends back the bytecodes for the class (or error
 * if the class is not found or the response was malformed).
 */
public void run()
{
    Socket socket;

    // accept a connection
    try {
        socket = server.accept();
    } catch (IOException e) {
        System.out.println("Class Server died: " + e.getMessage());
        e.printStackTrace();
        return;
    }

    // create a new thread to accept the next connection
    newListener();

    try {
        DataOutputStream out =
            new DataOutputStream(socket.getOutputStream());
        try {
            // get path to class file from header
            DataInputStream in =
                new DataInputStream(socket.getInputStream());
            String path = getPath(in);
            System.out.println(path);
            // retrieve bytecodes
            byte[] bytecodes = getBytes(path);

```

```
// send bytecodes in response (assumes HTTP/1.0 or later)
try {
    out.writeBytes("HTTP/1.0 200 OK\r\n");
    out.writeBytes("Content-Length: " + bytecodes.length +
                  "\r\n");
    out.writeBytes("Content-Type: application/java\r\n\r\n");
    out.write(bytecodes);
    out.flush();
} catch (IOException ie) {
    return;
}

} catch (Exception e) {
    // write out error response
    out.writeBytes("HTTP/1.0 400 " + e.getMessage() + "\r\n");
    out.writeBytes("Content-Type: text/html\r\n\r\n");
    out.flush();
}

} catch (IOException ex) {
// eat exception (could log error to log file, but
// write out to stdout for now).
System.out.println("error writing response: " + ex.getMessage());
ex.printStackTrace();

} finally {
try {
    socket.close();
} catch (IOException e) {
}
}
}

/***
 * Create a new thread to listen.
 */
private void newListener()
{
    (new Thread(this)).start();
}

/**
 * Returns the path to the class file obtained from
 * parsing the HTML header.
 */
private static String getPath(DataInputStream in)
    throws IOException
{
    String line = in.readLine();
    String path = "";
    System.out.println(line);

    // extract class from GET line
    if (line.startsWith("GET /")) {
        line = line.substring(5, line.length()-1).trim();
    }
}
```

```
int index = line.indexOf(".class ");
if (index != -1) {
    path = line.substring(0, index).replace('/', '.');
}
}

// eat the rest of header
do {
line = in.readLine();
System.out.println(line);
} while ((line.length() != 0) &&
        (line.charAt(0) != '\r') && (line.charAt(0) != '\n'));

if (path.length() != 0) {
return path;
} else {
throw new IOException("Malformed Header");
}
}

}
```

Para utilizar con RMI el elemental servidor *web* de Sun, hay que compilar las clases y ejecutar *ClassFileServer* especificando el puerto elegido para escuchar peticiones HTTP y el camino donde se almacenan los archivos .class que se quieran poner a disposición de clientes y servidores (*stubs*). Por ejemplo:

```
java server.ClassFileServer 8080 /home/ejemplosRMI/clases
```

hará que el servidor HTTP de Sun escuche por el puerto 8080 y que sirva los ficheros situados en la ruta de acceso (relativa) /home/ejemplosRMI/clases.

Cargar dinámicamente clases provenientes de sistemas remotos requiere algunas consideraciones en cuanto a seguridad (después de todo, se ejecuta código que proviene de otra máquina). Para que RMI permita ejecutar un .class descargado de una máquina remota, Java exige que la MVJ de destino tenga instalado un gestor de seguridad. El gestor de seguridad por defecto puede instalarse así:

```
System.setSecurityManager(new RMISecurityManager());
```

RMISecurityManager es un gestor muy restrictivo, y en general conviene definir una política de seguridad personalizada. La política se almacena en un archivo (seguridad.policy, por ejemplo) y se instala así:

```
java -Djava.rmi.server.codebase=http://www.miservidorweb.com/
-Djava.security.policy=seguridad.policy Nomina
```

Así, cualquier descarga de *bytecodes* que necesite *Nomina* se descargará de http://www.miservidorweb.com/ (si esos *bytecodes* no aparecen en el CLASSPATH local) y se tendrá en cuenta la política de seguridad definida en el archivo seguridad.policy.

A continuación pongo tres ejemplos de archivos de política de seguridad (el formato general se describe en la documentación de Sun):

```
grant{ permission java.net.SocketPermission "*:1024-65535","connect"; }

grant{ // Poco recomendable: deja al sistema desprotegido
    permission java.security.AllPermission;
};

grant {
    permission java.io.filePermission "/tmp/*", "read", "write";
    permission java.net.SocketPermission
        "anfitrion.dominio.com:1025","connect";
    permission java.net.SocketPermission "*:1024-65535",
        "connect,request";
    permission java.net.SocketPermission "*:80","connect";
};
```

El tercer ejemplo especifica lo siguiente:

- El código Java que proceda de otras máquinas puede leer cualquier archivo (y escribir en él) que esté en el directorio /tmp o en sus subdirectorios.
- Todas las clases de Java pueden establecer una conexión de red con el anfitrión anfitrion.dominio.com por el puerto TCP 1025.
- Las clases pueden conectarse o aceptar conexiones por cualquier puerto TCP superior a 1024, desde cualquier anfitrión.
- Todas las clases pueden conectarse al puerto TCP 80 (HTTP) desde cualquier anfitrión.

Para concretar todo lo expuesto, voy a considerar un ejemplo de aplicación distribuida que tiene en cuenta todo lo dicho. La aplicación mantiene un registro de empleados y permite que los clientes añadan o borren empleados, así como que consulten la información de los empleados mediante códigos alfanuméricos.

A diferencia del ejemplo del subapartado anterior, en éste se devuelven objetos remotos y se considera la sincronización de métodos. Lo último es necesario siempre que los clientes comparten información del servidor; pues RMI asigna un hilo a cada cliente que se conecta, y es obligación del programador considerar en el código las situaciones conflictivas que pudieran generarse. Si no se sincronizaran los métodos, podría darse el caso de que un cliente borrara un registro que otro intenta leer al mismo tiempo.

En el ejemplo, los archivos *stub* están en un servidor *web* imaginario al que se accede mediante el URL `http://www.miservidorweb.com/misclases/` y por el puerto HTTP por defecto (80); la aplicación del servidor reside en un anfitrión con URL `http://www.miservidorRMI.com` y escucha por el puerto TCP 9000.

Los clientes deben disponer en su anfitrión de los archivos RegistroEmpleados.class, Empleado.class (asociados a interfaces remotas) y ClienteRegistroEmpleados.class (asociado al cliente). El servidor debe tener en su sistema local de archivos RegistroEmpleados.class, Empleado.class, RegistroEmpleadosImp.class, EmpleadoImp.class, RegistroEmpleadosSkel.class, ServidorRegistroEmpleados.class, EmpleadoImp_Skel.class. El servidor *web* proporciona los ficheros EmpleadoImp_Stub.class y RegistroEmpleadosImp_Stub.class mediante el URL `http://www.miservidorweb.com/misclases/`.

La aplicación del servidor registra los objetos remotos por el puerto TCP 9000, por lo que hay que iniciar `rmiregistry` por ese puerto. La política de seguridad para el cliente y el servidor viene definida por el archivo `seguridad.policy` (se pueden usar otras extensiones: `seguridad.txt`, por ejemplo):

Ejemplo 18a: seguridad.policy

```
grant {  
    // Política de seguridad que permite que cualquiera escuche, se conecte  
    // o acepte peticiones mediante los puertos superiores al 1024  
    permission java.net.SocketPermission "localhost:1024-", "listen";  
    permission java.net.SocketPermission "localhost:1024-", "accept";  
    permission java.net.SocketPermission "localhost:1024-", "connect";  
};
```

El código de la aplicación se expone a continuación (he abreviado bastante los comentarios).

Ejemplo 18b: Empleado.java

```
import java.rmi.*;  
  
public interface Empleado extends java.rmi.Remote {  
  
    String getCodigo() throws java.rmi.RemoteException;  
  
    String getNombre() throws java.rmi.RemoteException;  
  
    double getSueldo() throws java.rmi.RemoteException;  
}
```

Ejemplo 18c: Empleadolmp.java

```
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.*;

/**
 * Implementación de la interfaz Empleado
 */

public class Empleadolmp extends UnicastRemoteObject implements Empleado {

    private String codigo = null;
    private String nombre = null;
    private double sueldo = 0.0;

    // Constructor
    public Empleadolmp(String codigo, String nombre, double sueldo) throws
        java.rmi.RemoteException{
        this.codigo = codigo;
        this.nombre = nombre;
        this.sueldo = sueldo;
    }

    // Métodos de acceso remotos
    public String getCodigo() throws java.rmi.RemoteException {
        return codigo;
    }

    public String getNombre() throws java.rmi.RemoteException {
        return nombre;
    }

    public double getSueldo() throws java.rmi.RemoteException {
        return sueldo;
    }

    // Este método no es remoto y no puede accederse a él mediante llamadas remotas.
    // Se pone como ejemplo para ver que una interfaz local y una remota no tienen por qué
    // coincidir, pero no se usará en el ejemplo.
    public String toString() {
        return ("Código: " +this.codigo+ ". Nombre: " + this.nombre + ". Sueldo: "+
            this.sueldo + " Euros.");
    }
}
```

Ejemplo 18d: RegistroEmpleados.java

```
import java.rmi.*;  
  
public interface RegistroEmpleados extends java.rmi.Remote {  
  
    // Registra un nuevo empleado  
    boolean registrar(String codigo, String nombre, double salario) throws  
        java.rmi.RemoteException;  
    // Devuelve un objeto Empleado  
    Empleado getEmpleado(String codigo) throws java.rmi.RemoteException;  
    // Borra un empleado  
    boolean borrar(String codigo) throws java.rmi.RemoteException;  
  
}
```

Ejemplo 18e: RegistroEmpleados.java

```
import java.rmi.*;  
import java.rmi.server.*;  
import java.rmi.registry.*;  
import java.util.*;  
  
/**  
 * Implementación de la interfaz RegistroEmpleados  
 */  
  
public class RegistroEmpleadosImp extends UnicastRemoteObject implements  
RegistroEmpleados {  
  
    private List lista= null;  
  
    public RegistroEmpleadosImp () throws java.rmi.RemoteException {  
        super();  
        cargarEmpleados();  
        try {  
            // Se registra la clase con el servidor de registro de RMI en el anfitrión  
            // www.miservidorRMI.com con el nombre "Plantilla".  
            // Se usará el puerto TCP 9000 (rmiregistry deberá  
            // lanzarse con ese puerto).  
            Naming.rebind("rmi://www.miservidorRMI.com:9000/Plantilla", this);  
        }  
        catch (AccessException e1){  
            System.out.println("Se rechazó el acceso. Su sistema no tiene los permisos " +  
                "necesarios para realizar la operación remota" );  
            System.exit(-1);  
        }  
        catch (java.net.MalformedURLException e2){  
            System.out.println("La URL para el registro no es válida.");  
            System.exit(-1);  
        }  
    }  
}
```

```
public void cargarEmpleados() throws java.rmi.RemoteException {
    lista = new ArrayList();

    Empleadolmp emp1 = new Empleadolmp("A0001", "Luis Monsalvez", 1219.68);
    Empleadolmp emp2 = new Empleadolmp("A0002", "Ernesto Navarro", 967.19);
    Empleadolmp emp3 = new Empleadolmp("A0003", "Manuel Soriano", 1456.34);
    lista.add(emp1);
    lista.add(emp2);
    lista.add(emp3);
}

public synchronized boolean registrar(String codigo, String nombre, double salario)
throws java.rmi.RemoteException {
    boolean temp = true;

    if ( (codigo == null) || (codigo.equals("")) )
        return false; // No se pudo registrar
    Iterator it = lista.iterator();
    while ( it.hasNext() ) {
        if ( ((Empleadolmp) it.next()).getCodigo().equals(codigo) ) {
            temp = false; // No se puede registrar: hay un empleado con ese código
            break;
        }
    }
    if (temp) {
        lista.add(new Empleadolmp(codigo, nombre, salario));
        return temp; // Registro correcto
    } else {
        return temp; // Registro incorrecto: existe un empleado con ese código
    }
}

public synchronized boolean borrar(String codigo) throws java.rmi.RemoteException {
    Iterator it = lista.iterator();
    Empleadolmp emp = null;

    if ( (codigo == null) || (codigo.equals("")) )
        return false; // No se pudo borrar

    while ( it.hasNext() ) {
        emp = (Empleadolmp) it.next();
        if ( (emp.getCodigo().equals(codigo)) ) { // Hay un empleado con ese código
            it.remove(); // Se borra el empleado
            return true; // Borrado correcto
        }
    }
    return false; // Borrado incorrecto: no existe un empleado con ese código
}

public synchronized Empleado getEmpleado(String codigo) throws
java.rmi.RemoteException {
    Empleadolmp emp = null;
    Iterator it = lista.iterator();

    if ( (codigo == null) || (codigo.equals("")) )
```

```
        return null; // Código inválido
    while ( it.hasNext() ) {
        emp = (EmpleadoImp) it.next();
        if ( (emp).getCodigo().equals(codigo) )
            return emp; // Empleado encontrado
    }
    return null; // No se encontró el empleado
}

}
```

Ejemplo 18f: ServidorRegistroEmpleados.java

```
import java.rmi.*;
import java.rmi.server.*;

public class ServidorRegistroEmpleados {

    public static void main(String args[]) {
        // Siempre conviene usar un controlador de seguridad. El controlador de
        // seguridad por defecto de RMI es demasiado estricto para permitir que este
        // ejemplo funcione, y por eso he definido una política propia de seguridad.
        // Cuando se compile esta clase debe hacerse con
        // java -Djava.security.policy = seguridad.policy ServidorRegistroEmpleados
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            new RegistroEmpleadosImpl();
            System.out.println("Servicio de registro de empleados disponible.");
            System.out.println("Puede registrar, borrar o modificar empleados.");
        }
        catch (Exception e) {
            System.out.println("No pudo arrancarse el servicio.");
            e.printStackTrace();
        }
    }
}
```

Ejemplo 18g: ClienteRegistroEmpleados.java

```
import java.rmi.*;
import java.rmi.registry.*;

public class ClienteRegistroEmpleados {

    public static void main(String args[]) throws Exception {
        // Siempre conviene usar un controlador de seguridad. El controlador de
        // seguridad por defecto de RMI es demasiado estricto para permitir que este
        // ejemplo funcione, y por eso he definido una política propia de seguridad.
        // Cuando se compile esta clase debe hacerse con
        // java -Djava.security.policy = seguridad.policy ClienteRegistroEmpleados
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try {
            // El servidor de registro de RMI devuelve un objeto genérico, que debe
            // ser convertido al tipo correspondiente. Ojo: el nombre por el que buscamos
            // debe coincidir con el que se le dio al servicio en ServidorRegistroEmpleados.
            RegistroEmpleados re =
                (RegistroEmpleados) Naming.lookup("rmi://www.miservidorRMI.com:9000/Plantilla");

            // Se intenta registrar un nuevo empleado
            if ( (re.registrar ("A0004", "Mercedes Romero", 1537.36)) ) {
                System.out.println("Registro correcto de Mercedes Romero.");
            } else
                System.out.println("El codigo ya corresponde a un empleado.");

            // Se intenta registrar un empleado con un código que ya existe
            if ( !(re.registrar("A0001", "Carlos Gandia", 1212.67)) )
                System.out.println("El codigo ya corresponde a un empleado.");

            // Se obtienen los datos de un empleado
            Empleado emp = re.getEmpleado("A0004");
            if ( (emp != null) ) {
                System.out.println("Nombre: " + emp.getNombre());
                System.out.println("Sueldo: " + emp.getSueldo());
            } else
                System.out.println("No puedo encontrarse a nadie con ese codigo.");

            // Se intenta borrar un empleado
            if ( re.borrar("A0002") ) {
                System.out.println("Registro borrado");
            } else
                System.out.println("No se pudo encontrar el registro.");
        }
        catch (Exception e) {
            System.out.println ("O bien no se encontró el servicio o no se pudo arrancarlo, o "
                               "bien hubo problemas en las llamadas a métodos remotos.");
            e.printStackTrace();
        }
    }
}
```

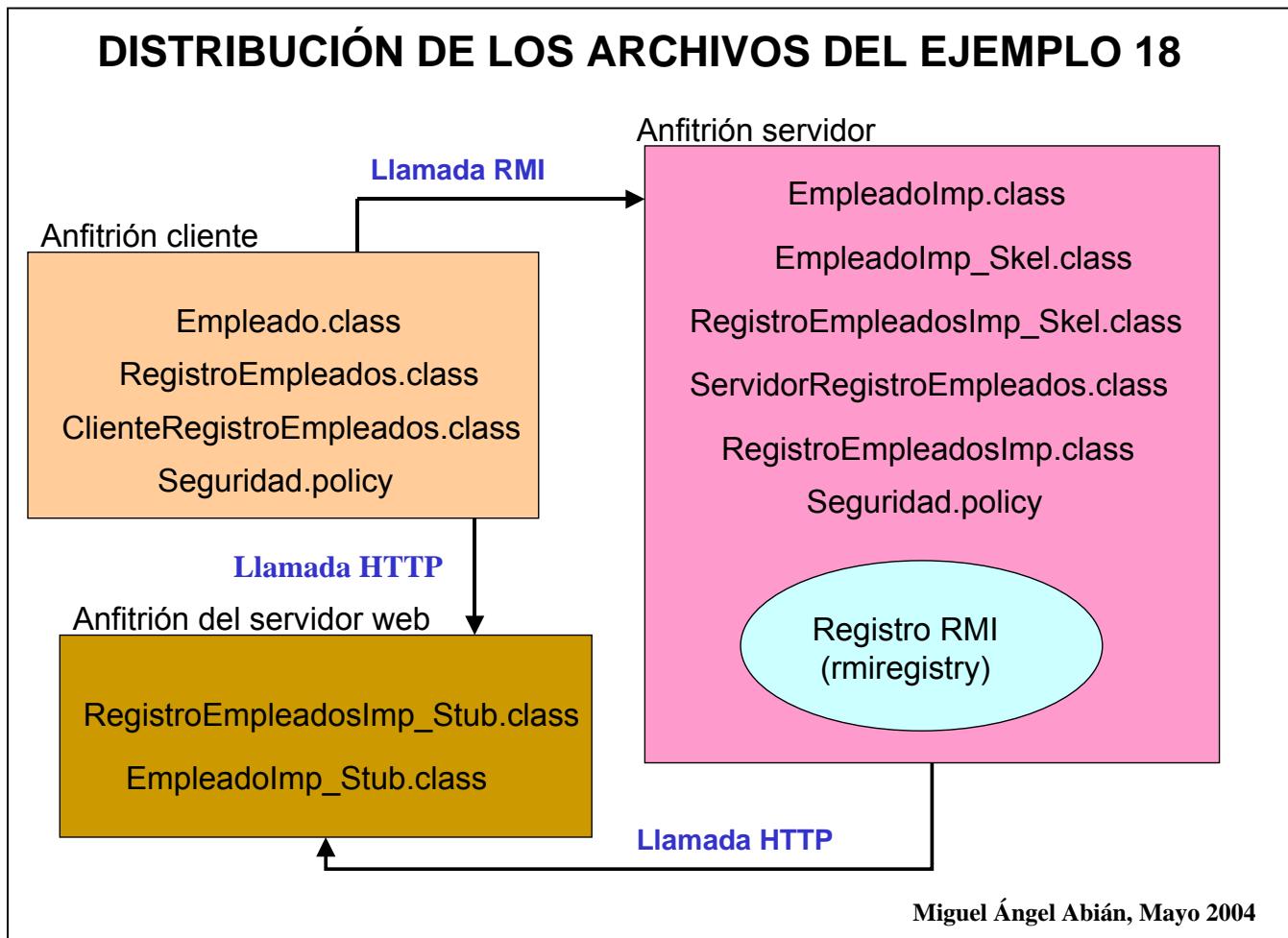


Figura 111. Distribución de los archivos del ejemplo 18

Al iniciar `rmiregistry` (en un sistema Windows, con `start rmiregistry 9000`), hay que asegurarse de que bajo ningún concepto los archivos *stub* estén disponibles localmente para `rmiregistry`, sólo mediante el servidor *web*. Dicho de otro modo: en la ventana donde se ejecute `rmiregistry`, el `CLASSPATH` no debe incluir caminos a los archivos *stub*. En caso contrario, `rmiregistry` los cargaría del sistema local y haría caso omiso de la propiedad `java.rmi.server.codebase`. En consecuencia, los archivos *stub* que devolvería a los clientes no tendrían configurada esa propiedad. Las consecuencias serían fatales para los clientes: cuando intentaran deserializar instancias de la clase adaptadora o *stub* no encontrarían nada en `java.rmi.server.codebase` y, al no poder cargar la clase, lanzarían excepciones `java.lang.ClassNotFoundException`.

Cuando se ejecute `ServidorRegistroEmpleados`, habrá que hacerlo así (supongo que el archivo de política de seguridad está en el mismo directorio que `ServidorRegistroEmpleados.class`):

```
java -Djava.rmi.server.codebase=http://www.miservidorweb.com/misclases/
-Djava.security.policy=seguridad.policy ServidorRegistroEmpleados
```

Espacio en blanco

Cuando se ejecute ClienteRegistroEmpleados, habrá que hacerlo así (supongo que el archivo de política de seguridad está en el mismo directorio que ClienteRegistroEmpleados.class):

```
java -Djava.security.policy=seguridad.policy ServidorRegistroEmpleados
```

El ejemplo ha sido probado con la siguiente configuración, en la que se han usado tres máquinas distintas, ubicadas en una LAN de tipo Ethernet:

- Anfitrión del cliente RMI: PC con Windows XP Profesional.
- Anfitrión del servidor RMI: estación de trabajo SUN BLADE 150 con Solaris 8.0.
- Anfitrión del servidor web (con los archivos *stub*): PC con Red Hat 9.0 y con el servidor Apache.

Al ejecutar el cliente desde el PC con Windows XP así:

```
F:\JbuilderX\jdk1.4\bin>java -Djava.rmi.server.codebase=
http://www.miservidorweb.com/misclases/ -Djava.security.policy= seguridad.policy
ServidorRegistroEmpleados
```

obtuve esta salida (en vez de `http://www.miservidorweb.com/misclases/` escribí el URL asociado al directorio del PC con Red Hat donde coloqué los .class *stub* y para el cual configuré Apache):

**Registro Correcto de Mercedes Romero
El código ya corresponde a un empleado.
Nombre: Mercedes Romero
Sueldo: 1537.36**

Si el lector quiere probar el ejemplo en una sola máquina (que actuará, por tanto, como servidor, cliente y servidor web), deberá cambiar en el código "`rmi://www.miservidorRMI.com:9000/Plantilla`" por "`rmi://localhost:9000/Plantilla`".

5.6. Ventajas e inconvenientes de la RMI

Éstas son algunas de las ventajas de trabajar con RMI:

- **Sencillez de uso.** Cualquier programador en java aprende enseguida a desarrollar aplicaciones RMI. En cualquier aplicación para intranets que trabajen con entornos Java, RMI puede ser una buena elección.
- **Separación entre interfaz e implementación.** En sentido general, la interfaz describe los servicios ofrecidos por un componente de un sistema de comunicaciones (una capa, por ejemplo) y los protocolos para usarlos. En los sistemas orientados a objetos, la interfaz de un objeto es el conjunto de métodos definidos para ese objeto, incluyendo los argumentos de entrada y salida. En un sistema distribuido de objetos, esta separación resulta decisiva para aprovechar las ventajas de la orientación a objetos (polimorfismo, etc.).
- **Carga dinámica de código.** Java permite la descarga automática de *bytecodes*, lo que hace que el proceso de instalación y configuración de los clientes puede ser tan sencillo como uno quiera. En el caso más simple, los clientes RMI pueden ser *applets* accesibles mediante un navegador web. Si se opta por esta solución, los usuarios no tendrán que instalar o configurar nada; bastará con que dispongan de un navegador y de acceso a la intranet o a Internet.
- **Sencillez de la localización de los servicios.** A diferencia de los *sockets*, un cliente RMI no necesita saber las direcciones explícitas de los servidores (nombre del anfitrión y número de puerto). Es suficiente con que sepa los nombres de éstos y el registro RMI donde se han registrado.
- **Seguridad.** Puede usarse con protocolos de seguridad como SSL o HTTPS.

He aquí algunos de los inconvenientes de trabajar con RMI:

- **RMI sólo usa Java.** Esta limitación era un problema al principio; pero, con el JDK 1.3, Sun incluyó la posibilidad de trabajar con el protocolo IIOP de CORBA, lo cual permite una cierta interoperabilidad de las aplicaciones RMI con las aplicaciones CORBA escritas en otros lenguajes (COBOL, C/C++, Smalltalk, etc.).
- **Falta de metainformación.** Al igual que los *sockets*, RMI no tiene un sistema de metainformación que almacene los servicios disponibles y sus API (esto es, los nombres de los métodos, los argumentos de cada uno, los valores de retorno, etc.).
- **Paso de objetos por valor.** Este mecanismo penaliza la eficacia y escalabilidad de las aplicaciones RMI. Cuanto más aumenta el tamaño de los objetos que se envían como argumentos, más datos hay que codificar en JRMP y enviar. Además, no debe olvidarse que, cuando se serializa un objeto, se serializan también todos los objetos a los cuales tiene referencias. Si el objeto corresponde a una colección de Java, puede darse el caso de que haya que serializar cientos o miles de objetos (y luego habrá que deserializarlos en el servidor).

- **Falta de control de las transacciones.** Tras una transacción, todos los objetos involucrados deben actualizar su estado (si la transacción se ha completado) o deben volver al estado anterior a la transacción (si no ha podido completarse). Así se consigue que todos los objetos estén siempre en estados válidos o consistentes. RMI carece de un mecanismo automático que controle las transacciones y que permita revertir las modificaciones si finalmente la transacción no se lleva a cabo.

6. CORBA: Llamando desde más lejos aún. CORBA y Java

6.1. Introducción. ¿Para qué se usa CORBA?

Ahora que ya hemos visto cómo funciona la RMI de Java, podemos echar un vistazo rápido a CORBA, tecnología ya mencionada en el apartado anterior. No es propósito del tutorial dar una visión completa de CORBA, ni mucho menos; pero ahora que hemos profundizado en RMI, podemos entender a la perfección los entresijos de CORBA sin necesidad de adentrarnos en los detalles más técnicos. Tenga en cuenta el lector que CORBA se creó con la idea de ser una tecnología neutral en cuanto al lenguaje utilizado para implementarla; es lógico, por tanto, que sea más detallada y compleja que RMI, pues esta última sólo trabaja con Java. Resulta meritario el esfuerzo hecho por Sun para sacar productos compatibles con CORBA o basados en su arquitectura (RMI-IIOP y Enterprise Java Beans son dos ejemplos), pues demuestran que Sun, al igual que otras muchas empresas, ha reconocido la importancia de un estándar internacional, neutral en cuanto a plataformas y vendedores. Al lector interesado en saber más sobre CORBA, le recomiendo los libros mencionados al final del subapartado 4.2.

CORBA (*Common Object Request Broker Architecture*: arquitectura común de intermediación de solicitudes de objetos) es una arquitectura abierta, desarrollada por el OMG (*Object Management Group*: grupo de gestión de objetos) para construir aplicaciones distribuidas. CORBA consiste en un conjunto de especificaciones (no de implementaciones) que, de seguirse, hacen posible la interacción entre objetos CORBA escritos en distintos lenguajes y ejecutados en diferentes plataformas.

Basándose en el modelo de comunicaciones de CORBA, se pueden construir aplicaciones distribuidas en las que clientes y servidores se implementen en una mezcla diversa de lenguajes y se ejecuten en una mezcla también diversa de sistemas operativos y hardware. Por ejemplo, un cliente CORBA puede ejecutarse en un Mac y estar escrito en Java; otro puede ejecutarse en una estación de trabajo de IBM y estar escrito en Smalltalk; mientras que el servidor puede ejecutarse en un sistema Windows y estar escrito en C++. Desarrollar con CORBA viene muy bien, e incluso puede ser la única opción, cuando se trabaja con muchas plataformas y lenguajes distintos.

CORBA también puede resultar muy útil cuando hay que tratar con aplicaciones antiguas cuyo código no se puede modernizar a lenguajes más actuales. Por ejemplo, imaginemos que debe escribir código que interaccione con una aplicación monstruosa escrita en COBOL, cuya reescritura en Java o C++ podría costar meses (o años) y millones de euros o dólares. Hoy día, resultaría bastante absurdo escribir un cliente en COBOL para esa aplicación (sí, ya sé que existe el Visual COBOL y el Object COBOL; pero sigue siendo COBOL, maldita sea). Usar CORBA nos permitiría escribir el cliente en el lenguaje que deseáramos y evitaríamos vernos atados a COBOL.

En general, CORBA resulta una opción que considerar cuando se desea dar una interfaz “moderna” a aplicaciones vetustas o que huelen a rancio. Podría parecer que este uso de CORBA es inusitado, pero aún hay mucho código COBOL o Ada que se aferra desesperadamente a la vida. Millones de líneas escritas en COBOL y Ada forman parte de los actuales sistemas bancarios y militares de muchos países (durante un tiempo, Ada era, ¡por ley!, el lenguaje que debía usarse en muchas instituciones públicas estadounidenses). De todos modos, no hay que irse a grandes sistemas para ver a COBOL en funcionamiento: me faltan dedos en las manos para contar las empresas donde me he encontrado aplicaciones de gestión escritas en COBOL. Como el hardware y los sistemas operativos donde se ejecutaban originalmente estas

aplicaciones duermen ya el sueño de los justos en los museos de informática, no queda más remedio que ejecutarlas sobre emuladores.

Debido a la gran escalabilidad de CORBA, también se usa en servidores que atienden un gran número de peticiones simultáneas: permite repartir entre distintas máquinas el trabajo de contestar las peticiones, aunque éstas sean de diferentes fabricantes (lo cual suele implicar variadas arquitecturas de hardware y sistemas operativos heterogéneos). Por ejemplo, imagínese un sistema bancario internacional que procese y registre cientos o miles de transacciones monetarias por segundo, procedentes de todas las sucursales repartidas por el mundo. CORBA podría usarse para distribuir el procesado y registro de las peticiones entre varias máquinas: *mainframes*, estaciones de trabajo UNIX, estaciones de trabajo de IBM, PCs de altas prestaciones, etc. CORBA se encargaría de repartir el trabajo entre todas las máquinas, dependiendo del volumen de peticiones, de la capacidad de cada máquina y de la carga de trabajo de cada una. Asimismo, CORBA puede usarse en aplicaciones de tipo *web* que reciban muchas peticiones simultáneas.

Cuatro son las principales características arquitectónicas de CORBA

- **Separación entre interfaz e implementación.** Como los clientes llaman a los objetos por medio de interfaces, no se ven afectados por cambios en las implementaciones de éstos. Más aún: los clientes desconocen los cambios en las implementaciones de los objetos a los que llaman. Esta característica simplifica los cambios y actualizaciones en los objetos de una aplicación distribuida.
- **Independencia de la localización.** Los clientes pueden acceder a los objetos sitos en una red sin conocer dónde residen. El cliente desconoce si los objetos a los que llama están en procesos distintos en la misma máquina, en una misma LAN, en una WAN o en una sonda espacial con una órbita geoestacionaria.
- **Independencia del vendedor.** Los productos CORBA de un fabricante pueden interoperar con los de otros.
- **Integración de sistemas mediante la interoperabilidad.** Los productos que cumplen las especificaciones de CORBA son independientes de la plataforma: cualquier cliente puede llamar a un servidor, independientemente de las plataformas donde se ejecuten uno y otro.

Los objetos CORBA se distribuyen como componentes binarios que pueden recibir llamadas remotas a sus métodos. A los clientes de un objeto CORBA les es indiferente dónde se encuentra, en qué plataforma se ejecuta o cómo se implementa. CORBA es una tecnología de integración; no se encuentra atada a ningún lenguaje o plataforma.

CORBA usa el IDL (*Interface Definition Language*: lenguaje de definición de interfaces) del OMG para definir las interfaces a las que pueden acceder los clientes. A veces, uno encuentra escrito “IDL de CORBA” en lugar de “IDL del OMG”. A mi juicio, la primera denominación frisa el desacuerdo: ni el IDL del OMG se usa sólo con CORBA ni es un lenguaje atado a CORBA. Por brevedad, usaré casi siempre “IDL” para referirme al IDL del OMG; pero el lector debe saber que existen otros IDL (por ejemplo, Microsoft tiene el suyo propio para su tecnología DCOM). El IDL del OMG es un mero lenguaje declarativo, basado en C++, no un lenguaje de implementación. Los métodos

especificados en una interfaz escrita con IDL pueden implementarse más tarde con otros lenguajes.

El OMG define traducciones o conversiones “oficiales” para COBOL, C, Ada 95, Smalltalk, C++, Java, Lisp, Python, IDLScript y PL/I (había intención de hacer lo mismo para C#, pero la traducción se ha desestimado por motivos no bien aclarados). Por lo que he podido averiguar, existen conversiones IDL “no oficiales” para otros dieciséis lenguajes, entre los que destacan Delphi, Pascal, Perl y Visual Basic. Una traducción de IDL a un lenguaje define cómo se transforman las estructuras sintácticas del IDL en las del lenguaje de destino.

Utilizar un lenguaje declarativo “neutro” añade complejidad, pero reporta una ventaja en absoluto desdeñable: clientes y servidores pueden comunicarse con independencia de los lenguajes en que estén escritos.

6.2. El modelo de objetos de CORBA

En un sistema de software, un modelo de objetos es un conjunto de especificaciones de implementación y de propiedades semánticas que define cómo se representan los objetos en un determinado lenguaje de programación y cómo se implementan las características de la orientación a objetos (abstracción, encapsulado, herencia, etc.).

Aunque el término “orientación a objetos” es una caja donde caben muchos conceptos, hay unanimidad a la hora de definir las características que debe proporcionar un modelo de objetos: **clase/tipo, objetos, identidad de los objetos, abstracción, encapsulado, herencia y polimorfismo**.

Los conceptos de **clase** y **tipo** son conocidos por cualquiera que programe en Java. En un lenguaje de programación, una clase es tanto la definición de un conjunto de objetos con propiedades y operaciones comunes como una “fábrica” de objetos (se dice que estos objetos son instancias de la clase). En tiempo de ejecución, una clase actúa como molde para un conjunto de objetos, dotándoles de interfaz (parte externa) y de implementación (parte interna). Los tipos resultan un poco más sutiles: vienen a ser representaciones software de los tipos de datos abstractos (pilas, colas, etc.). Cuando se dice que un objeto es de tipo x, se busca expresar que ese objeto tiene la interfaz definida por el tipo x.

Ambos conceptos suelen usarse de forma intercambiable, pero son distintos. Varias son las diferencias: a) *tipo* es un concepto asociado al tiempo de compilación, mientras que *clase* se vincula más con el tiempo de ejecución; b) los tipos no definen implementaciones y, por tanto, no pueden crear objetos; c) un objeto es instancia de una sola clase, pero puede ser de varios tipos a la vez.

Hay lenguajes OO basados en tipos (Smalltalk, p. ej.), en clases (C++) y basados en tipos y clases (Java, C#). En los lenguajes que permiten tipos, éstos se usan para comprobar, durante el proceso de compilación, si los programas son correctos (comprobación de tipos). En los lenguajes que no tienen tipos, un error como pasar un objeto Persona a un método que espera un argumento Coche no se advertirá hasta que se llame al método en tiempo de ejecución.

Un **objeto** es una instancia de una clase. En tiempo de ejecución, un objeto es un área de memoria dentro de un proceso.

La **identidad de un objeto** es como el DNI de una persona: permite identificarlo únicamente. La identidad de un objeto es única y no puede ser modificada una vez creado el objeto.

La **abstracción** es la capacidad de un modelo para ignorar algunos aspectos de la información con que trabaja. Cada objeto es una abstracción que puede recibir peticiones y contestarlas sin revelar cómo lo hace.

El **encapsulado** (u ocultación de la información) es el principio por el cual los objetos se esconden (se encapsulan) tras su interfaz. Un objeto bien diseñado revela lo menos posible de su funcionamiento interno (implementación). La gran ventaja del encapsulado estriba en que permite cambiar la implementación de un objeto sin que los demás objetos tengan que ser modificados o advertidos.

El **polimorfismo** es la capacidad de asociar distintos tipos (distintos comportamientos, en definitiva) a un mismo objeto.

La **herencia** es el mecanismo por el cual se pueden derivar nuevas clases de las ya existentes. Una subclase hereda las propiedades y métodos de la superclase, y puede modificarlos o incluir otros. Dentro de la herencia, hay dos variedades: la de tipos y la de implementación. En la primera, las subclases heredan la interfaz de la superclase; en la segunda, heredan de ésta los datos y el código de los métodos. Los lenguajes OO con tipos admiten ambas.

Como CORBA es una arquitectura independiente de cualquier lenguaje o plataforma, su modelo de objetos se ve obligado a ser abstracto. Un sistema de objetos CORBA es un conjunto de objetos en el que unos solicitan servicios (clientes) y otros los proporcionan (servidores). Los clientes pueden hacer peticiones sólo mediante la interfaz de los servidores y desconocen la implementación de estos últimos. Como ocurre en cualquier modelo de objetos, la comunicación entre objetos se lleva a cabo exclusivamente mediante el intercambio de mensajes (llamadas y respuestas). Los objetos CORBA pueden ser creados o destruidos; pero los clientes no tienen ningún mecanismo para crearlos o destruirlos.

El modelo de objetos de CORBA no proporciona todas las características desglosadas antes. Por ejemplo, el concepto de *clase* no existe: la estructura *class* de lenguajes como C++, Java, Smalltalk o C# brilla por su ausencia. Los objetos sí existen: son entidades identificables y encapsuladas que proporcionan al menos un servicio que puede ser solicitado por uno o más clientes. El concepto de objeto CORBA es un tanto escurridizo y lo abordaré con detalle en el siguiente subapartado.

Con respecto a los tipos, el IDL del OMG es un lenguaje basado en tipos y en el cual todo objeto debe declarar su tipo (lo que se conoce como fuertemente “tipado”). Hay dos tipos especiales en CORBA: Any y TypeCode.

El tipo Any permite la especificación de valores que permiten expresar cualquier tipo IDL (sea primitivo o definido por el usuario, siempre que haya sido definido en tiempo de compilación). Un Any contiene un TypeCode y un valor descrito por éste. En cada “traducción” del IDL a un lenguaje concreto, existen operaciones que permiten insertar un TypeCode en un Any (y extraerlo de él). Los Any resultan muy útiles para hacer comprobaciones de tipos de manera dinámica.

El tipo TypeCode es un metatipo: un TypeCode representa un tipo de datos definidos por el IDL del OMG. Es la especificación de CORBA, TypeCode se define como una interfaz con operaciones para averiguar cuál es el tipo representado por un

TypeCode y para saber si dos Typecodes son iguales o equivalentes. Este tipo resulta útil para CORBA, que lo usa para pasar entre máquinas datos que se describen a sí mismos. En general, los programadores nunca tienen que trabajar directamente con este tipo.

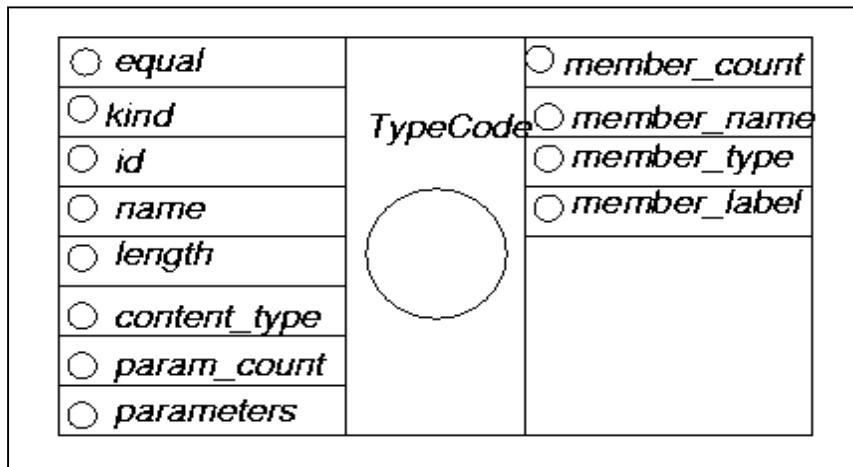
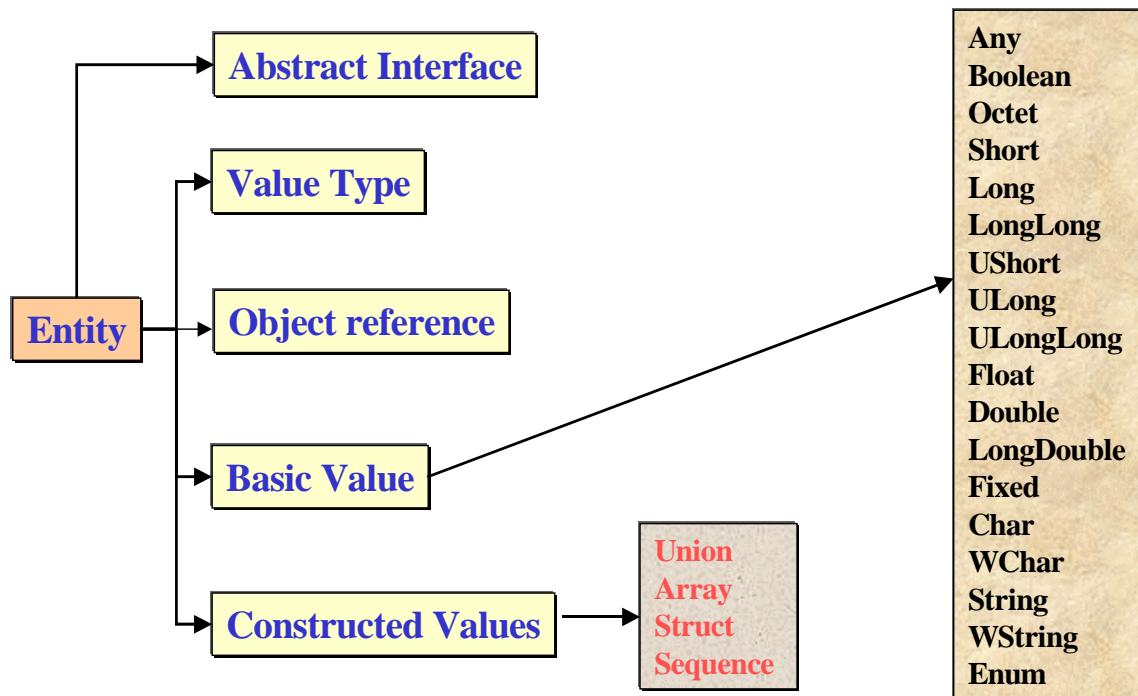


Figura 112. Representación del metatipo TypeCode. Extraído de la documentación del OMG sobre CORBA

ENTIDADES DEL IDL DEL OMG



He optado por mantener en inglés los nombres de todas las entidades que aparecen en la documentación del OMG

Miguel Ángel Abián Julio 2004

Figura 113. Entidades del IDL. Se incluyen los tipos primitivos

En el caso de otras características, el significado en CORBA varía del dado antes. Por ejemplo, CORBA separa muy claramente (al igual que RMI) interfaz e implementación, lo cual tiene consecuencias para la herencia. De hecho, la separación no es sólo conceptual: la interfaz puede estar en un anfitrión y la implementación puede estar repartida entre anfitriones a miles de kilómetros del primero.

La herencia de implementación no tiene cabida en el modelo de CORBA, pues esta herencia se basa en heredar atributos y código de métodos –lo cual está vinculado a lenguajes de programación concretos–, mientras que CORBA es independiente del lenguaje de programación. Por tanto, “herencia” en CORBA significa “herencia de tipos”. En este tipo de herencia, se heredan las interfaces (las operaciones). Así, en CORBA una clase `Caniche` puede heredar una operación `ladrar()` de una clase `Perro`, pero no puede heredar el código de `ladrar()` que haya en `Perro`. Al igual que ocurre en cualquier lenguaje OO, una interfaz CORBA puede tener muchas implementaciones (dicho de otro modo: una operación puede materializarse en muchos métodos). Ahora bien, en CORBA, estas implementaciones de una misma interfaz pueden estar escritas en distintos lenguajes: C, COBOL, C++, Java, etc. Una operación como `ladrar()` puede implementarse con C++ en una clase `Caniche`, y puede implementarse con Smalltalk en una clase `Galgo`.

El concepto de *identidad del objeto* pervive en CORBA, pero transformado en el de *referencia a objeto*. Una referencia a un objeto es una prolongación de la identidad de éste, con información adicional sobre la ubicación del objeto (anfitrión, número de puerto, protocolo de acceso, etc.). Los clientes llaman a los servidores mediante referencias a estos últimos, nunca directamente.

6.3. Vocabulario básico de CORBA

Para evitar ambigüedades, antes de proseguir expongo la definición de los siguientes términos en el sentido dado por el OMG y por la mayor parte de los textos sobre CORBA:

- **Objeto CORBA (u objeto).** Entidad virtual susceptible de ser encontrada por una aplicación CORBA y de recibir peticiones por parte de un cliente. Tiene asociada una identidad inmutable, una interfaz, una implementación (sirviente) y una localización. Los objetos CORBA pueden implementarse en lenguajes no orientados a objetos.
- **Referencia (a un objeto CORBA).** Se asigna en el momento de la creación de un objeto CORBA. Los clientes hacen sus peticiones a los objetos CORBA mediante referencias, pero no tienen acceso a ellas ni pueden modificarlas (son “opacas” para ellos). Toda referencia contiene un identificador del objeto (*object ID*), única en un proceso del servidor. Un objeto tiene un solo identificador, si bien puede tener múltiples referencias. Las referencias a objetos CORBA se asemejan a los punteros de C++: pueden apuntar a objetos inexistentes o inalcanzables, o a ningún lugar.
- **Sirviente (servant).** Entidad de un lenguaje de programación que implementa uno o más objetos CORBA. Por tanto, define los métodos especificados por la interfaz IDL del objeto u objetos. En los lenguajes OO, los sirvientes se definen con clases sirviente y, en consecuencia, los objetos CORBA se implementan como instancias de las clases sirvientes. En la documentación inicial del OMG, *objeto sirviente* o *sirviente* se usaba como sinónimo de *implementación del objeto* (*object implementation*). Cuando una clase sirviente se “activa”, genera sirvientes. En tiempo de ejecución, los sirvientes tienen asignados unos recursos de CPU y unas zonas de memoria, que se liberan cuando son destruidos.
- **Cliente (o cliente CORBA).** Entidad de programación que realiza peticiones a un objeto CORBA. No necesariamente tiene que ser instancia de una clase. Los clientes no trabajan directamente con los objetos CORBA, sino con referencias a los objetos.
- **Servidor (o servidor CORBA).** Proceso o aplicación en la que hay uno o más objetos CORBA y que atiende las peticiones de los clientes. Un servidor contiene implementaciones de uno o más interfaces IDL (en los lenguajes OO, dichas implementaciones son clases sirvientes). En general, “servidor” y “objeto CORBA” son términos intercambiables, si bien no idénticos. Antepondré a “servidor” la palabra “proceso” cuando me interese destacar esa faceta o evitar la posible confusión con la máquina o anfitrión donde se ejecuta el proceso servidor.
- **Petición.** Llamada de un cliente a una operación de un objeto CORBA por parte de un cliente.

El concepto de objeto CORBA es bastante delicado, y pocas veces se explica correctamente. A pesar de la “O” de “Object”, el OMG no clarifica mucho este concepto; quizás saben demasiado como para caer en esa trampa. El uso de “objeto” induce a confusión: un objeto CORBA es un objeto en el sentido de “una entidad con estado,

identidad e interfaz"; pero no es necesariamente un objeto en el sentido de "instancia de una clase de un lenguaje orientado a objetos". CORBA trabaja con lenguajes no OO (COBOL, C, Ada), que carecen de clases y de instancias de clases. De hecho, en el IDL del OMG prescinde del concepto de clase.

Por otro lado, los objetos CORBA son virtuales: pueden existir sin estar asociados a código que implemente las operaciones de la interfaz. Cuando se crea un objeto CORBA, se crea una referencia de objeto que encapsula la información de la interfaz del objeto junto con un identificador (generado por la aplicación CORBA). Esta referencia puede enviarse al cliente para que pueda usar hacer sus llamadas remotas; pero no está asociada a ningún proceso. **Crear un objeto CORBA** no es crear una instancia o activar un proceso con una memoria reservada y unos recursos de CPU: sólo es asignarle una referencia. **Destruir un objeto** es privarle de su referencia, la cual se vuelve inutilizable. Un objeto destruido ya no puede ser llamado por un cliente. Como leí en alguna parte, "no hay nada parecido a un desfibrilador que pueda devolver la vida a los objetos CORBA".

Cuando un cliente realiza una petición a un objeto CORBA que acaba de ser creado, suele producirse la activación de éste. La **activación** es el proceso por el cual se asigna un sirviente a un objeto CORBA. En los lenguajes OO, un sirviente se deriva de una clase; en los no OO, un sirviente se deriva de un conjunto de funciones que manipulan una estructura de datos (un *struct*, por ejemplo). En los primeros –que son los que nos interesan–, la activación consiste en asignar una instancia de una clase sirviente a un objeto CORBA (lo cual suele implicar la creación de la instancia). Dicho de otra forma: un servidor activa un objeto cuando crea un sirviente, asignándole un espacio de memoria y unos recursos de CPU, y lo asocia a un objeto CORBA. Como un sirviente deriva de una clase que implementa los métodos de la interfaz IDL del objeto CORBA, puede manejar las llamadas de los clientes al objeto.

En cierto modo, un objeto CORBA es un vestido vacío; cuando se le asigna un sirviente, se le dota de "cuerpo". Desde el punto de vista de los sirvientes, se dice que un sirviente "**encarna**" a un objeto cuando se crea un vínculo o ligadura entre ellos.

Desactivar un objeto CORBA es eliminar el vínculo entre él y su sirviente. Este proceso no implica destruir el objeto (un objeto desactivado puede ser reactivado si recibe la llamada de algún cliente), pero suele implicar la destrucción del sirviente (y la liberación de la memoria y de los recursos reservados a éste). Cuando se desactiva un objeto, se le quita el "cuerpo" del sirviente: vuelve a ser un ente fantasmal, con una referencia y una interfaz; pero sin código en ejecución que implemente a esta última.

Las maneras más frecuentes de desactivar un objeto son dos: detener el proceso de servidor que lo ha creado y destruir el sirviente al cual está asociado. Desde el momento en que el objeto queda desactivado, no puede responder a las peticiones de los clientes mientras no se le vuelva a asociar un sirviente.

En las activaciones explícitas, se crean sirvientes para todos los objetos que se van creando (y se van asociando con éstos). En las activaciones bajo demanda, sólo se activan aquellos objetos que reciben llamadas de los clientes.

Durante la vida de un objeto CORBA, éste puede estar asociado a muchos sirvientes; asimismo, un sirviente puede estar asociado, en un instante determinado, a varios objetos.

Resulta lógico que uno se pregunte el porqué de tantos términos: objeto CORBA, sirviente, creación, destrucción, activación, desactivación, "encarnación", etc. Una primera respuesta estriba en que se pretende marcar la dicotomía entre los objetos CORBA y los sirvientes, de modo que se perciban desde el principio como entidades separadas. ¿A qué obedece esta separación? ¿No sería mucho más sencillo identificar

un objeto CORBA con un sirviente, y viceversa? Sí, sí lo sería; pero esa identificación tendría graves repercusiones en la eficacia de las aplicaciones CORBA.

Veamos un ejemplo de esas consecuencias: imagine una base de datos integrada en una aplicación CORBA y que tiene miles o millones de registros. Si a cada objeto CORBA que representa a un registro se le asociara un sirviente, el servidor debería crear muchísimas instancias y mantenerlas con vida durante todo el tiempo de la vida de la aplicación, por si algún cliente llamara a alguna. Como puede suponer, el consumo de recursos (RAM, CPU) sería exagerado. En dicho caso, tampoco se podría reutilizar sirvientes (salvo que se desvistiera a un santo para vestir a otro), pues crear un objeto CORBA sería idéntico a crear un sirviente. Las dificultades no acabarían ahí: al destruirse un sirviente se destruiría también el objeto CORBA asociado, lo que impediría la persistencia de los objetos (deseable en muchas ocasiones).

La separación entre objeto y sirviente complica conceptualmente el entendimiento de CORBA; pero mejora el rendimiento y la escalabilidad de las aplicaciones CORBA. Y no olvidemos que CORBA se usa en aplicaciones industriales donde esas propiedades son tan valiosas como el agua en el desierto. Con esta distinción, un sirviente se crea sólo cuando un cliente llama a un objeto. Asimismo, un sirviente puede aprovecharse para distintos objetos, sin que haya que crear más sirvientes.

Un ejemplo valdrá para aclarar lo dicho: considere una aplicación CORBA que almacena los datos de los empleados de una empresa. Como el servidor no sabe qué datos van a consultar los clientes, lo normal es que cree un objeto CORBA para cada empleado. Tendrá, pues, que almacenar una referencia por empleado, exigencia que no resultará devastadora para sus recursos. Cuando un cliente llame al método `getNombre()` del objeto CORBA `luisNavarro`, el servidor creará un sirviente con la implementación de ese método y de los otros que pudiera tener el objeto, así como con los atributos correspondientes (edad, sueldo, etc.). Pasado un tiempo, si el servidor no recibe más peticiones, el objeto CORBA será desactivado (no destruido). Cuando esto ocurra, el sirviente afrontará una de estas suertes: será destruido o se mantendrá a la espera de nuevas peticiones. En el primer caso, una llamada a un método del objeto `juanSanchez` producirá la creación de un nuevo sirviente; en el segundo, el sirviente de `luisNavarro` pasará a ser de `juanSanchez`.

En CORBA, el cliente siempre es afortunado: ignora cualquiera de las sutilezas expuestas. Al cliente poco le importa que el objeto al cual llama se active o desactive, o que haya un cambio de sirvientes: él se limita a hacer sus peticiones. Su conveniente desinterés por la manera como se complacen sus peticiones resalta la similitud con algunos clientes de carne y hueso. Perdonen que asigne cualidades humanas a las entidades CORBA: ya sé que a ellas no les gusta ser tratadas así.

6.4. Dentro de la arquitectura de CORBA

CORBA es una especificación de una arquitectura de red (esto es, un conjunto de capas y protocolos para las comunicaciones en red) cuyas partes y conceptos más importantes vamos a ir viendo a lo largo de este subapartado.

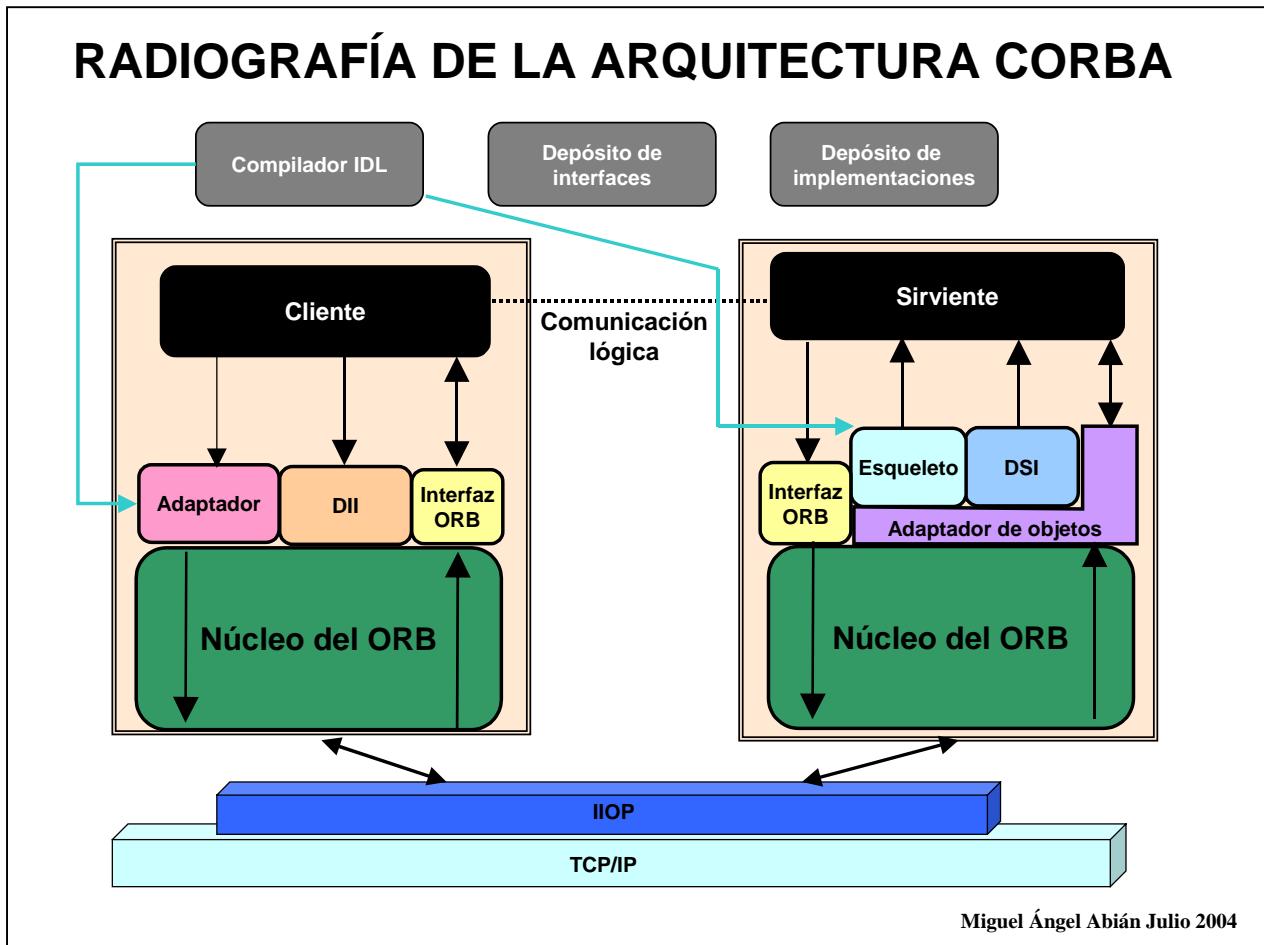


Figura 114. Lo que nos mostrarían los rayos-X si CORBA no fuera virtual

El **IDL** (*Interface Declaration Language*: lenguaje de declaración de interfaces), ya mencionado, especifica las interfaces que ofrecerán los servidores CORBA a sus clientes. En consecuencia, define los atributos y métodos de los objetos distribuidos en una aplicación CORBA. Para cada tipo de objeto distribuido de CORBA, se define una interfaz con el IDL (estas interfaces equivaldrían a las interfaces que extienden `java.rmi.Remote` en RMI). En las especificaciones de CORBA se definen “traducciones” de las interfaces IDL a construcciones de lenguajes como C, C++, Java, COBOL, etc. Como puede imaginarse el lector, las traducciones a uno u otro lenguaje poco se parecerán, pues los lenguajes que se usan con CORBA son francamente heterogéneos. Por ejemplo, COBOL o C no manejan objetos, mientras que Java o C++ sí.

Si miramos con lupa bajo el IDL, evitando tocar el polvo, notaremos que es interesante no tanto por la sintaxis como por el mecanismo de abstracción que proporciona para separar las interfaces de las implementaciones. El IDL actúa como un contrato entre clientes y servidores. Viene a decir al servidor: “Debe respetar esta interfaz”. Asimismo, viene a decir a los clientes: “Si quiere obtener respuesta, debe

acomodar sus peticiones a la manera especificada por la interfaz". Un servidor que no respete la interfaz provocará el desconcierto entre sus clientes ("¿Por qué me devuelve una patata cuando he pedido una manzana?", "¿Por qué me dice que el dólar ha subido si le pregunto por el tiempo en Ginebra?"). Un servidor que mute su interfaz repentinamente y sin avisar a los clientes enseguida descubrirá una verdad elemental del comercio, mencionada antes: se tarda una vida en conseguir un cliente, y sólo unos minutos en perderlo para siempre.

Usando IDL, se puede describir:

- Las interfaces de objetos o módulos
- Las operaciones y los atributos de un objeto o módulo
- Las excepciones lanzadas por un método.
- Los tipos de datos de los argumentos de un método, así como de sus valores de retorno.

La sintaxis del IDL es un subconjunto de la del ANSI-C++, a la que se le han añadido algunas palabras para incluir conceptos propios de las arquitecturas distribuidas. Dicho con un trabalenguas: en cuanto a sintaxis, IDL es un superconjunto de un subconjunto del ANSI-C++.

Palabras reservadas del IDL de CORBA

any	attribute	boolean	case	char
const	context	default	double	enum
exception	FALSE	fixed	float	in
inout	interface	long	module	
Object	octet	oneway	out	
raises	readonly	sequence	short	
string	struct	switch	TRUE	
typedef	unsigned	union	void	
wchar	wstring			

Figura 115. Palabras reservadas del IDL de CORBA

Vemos un ejemplo de un módulo IDL (los módulos equivaldrían a las clases de C++, Java o Smalltalk):

```
// Código IDL
module Personal {

    struct Persona {
        string nombre;
        string apellido;
        short edad;
        double salario;
    };
}
```

```
exception Expcion{};  
  
interface Salario {  
    double getSalario() raises(Expcion);  
    // salario es el argumento de setSalario  
    void setSalario (in double salario);  
};  
  
};
```

La palabra `in` indica que estamos ante un argumento de entrada; se puede usar también `out` (argumento de salida) e `inout` (argumento de entrada y salida).

SINTAXIS DEL IDL DEL OMG

```
module <identificador> {  
    <declaraciones de tipos>;  
    <declaraciones de constantes>;  
    <declaraciones de excepciones>;  
  
    interface <identificador> [::<herencia>] {  
        <declaraciones de tipos>;  
        <declaraciones de constantes>;  
        <declaraciones de atributos>;  
        <declaraciones de excepciones>;  
  
        [<tipo operacion>]<identificador>(<lista argumentos>)  
            [raises <excepciones>]  
        [<tipo operacion>]<identificador>(<lista argumentos>)  
            [raises <exceptiones>]  
        . . .  
    };  
    interface <identificador> [::<herencia>] { . . . };  
    . . .  
};
```

Miguel Ángel Abián Julio 2004

Figura 116. Estructura de un archivo IDL

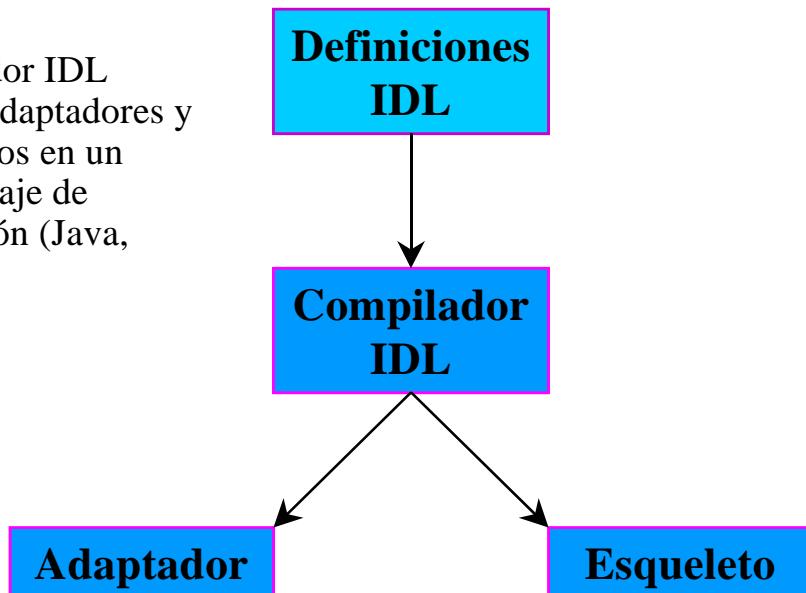
Para todos los lenguajes compatibles con CORBA, existen “traducciones” de las estructuras del IDL a los lenguajes compatibles (se pueden consultar en http://www.omg.org/technology/documents/idl2x_spec_catalog.htm). Los compiladores IDL convierten los archivos con código IDL en código escrito en el lenguaje de destino. Personalmente, preferiría llamar “traductor IDL” al compilador IDL, pues en realidad no compila a código nativo ni a *bytecode*; pero en la documentación de OMG se usa casi exclusivamente el término “compilador”.

Cuando se usa un compilador IDL con un archivo donde se especifica la interfaz de una objeto CORBA, se generan –en el lenguaje de destino: C, C++, Java, etc.– varios archivos. Dos de ellos corresponden al adaptador (*stub*) y al esqueleto (*skeleton*) para la interfaz. Como ya vimos en el apartado anterior, un adaptador es un intermediario local para el objeto remoto: su interfaz coincide con la del objeto servidor, pero se ejecuta en el anfitrión del cliente. Un esqueleto es análogo al adaptador del cliente, pero en el lado del servidor. Mediante unos y otros, las llamadas remotas se simulan como si fuesen locales: para el objeto cliente, los adaptadores simulan los métodos del objeto remoto; para el objeto servidor, los esqueletos simulan las llamadas del objeto cliente como si fueran locales.

Al generar adaptadores y esqueletos, un compilador IDL produce el código que se usará para codificar los datos que se envíen como argumentos en las llamadas remotas y para decodificarlos (lo mismo hace para RMI el compilador `rmic`). En los lenguajes OO, las clases sirviente suelen derivarse por herencia de las clases esqueleto generadas por el compilador IDL. Asimismo, las clases que actúan como clientes suelen derivarse por herencia de los adaptadores generados por el compilador IDL.

GENERACIÓN DE ADAPTADORES Y ESQUELETOS

El compilador IDL genera los adaptadores y los esqueletos en un cierto lenguaje de programación (Java, C++, etc.)



Miguel Ángel Abián Julio 2004

Figura 117. Generación de adaptadores y esqueletos en CORBA

El **ORB** (*Object Request Broker*: intermediario de peticiones de objetos) es el núcleo, el corazón de CORBA. Gracias a él, los objetos pueden intercambiar llamadas con independencia de las plataformas donde se ejecutan. El papel del ORB es de coordinador general para todos los objetos en una aplicación CORBA. Se encarga de identificar y localizar los objetos, de abrir y gestionar las conexiones en las máquinas donde residen clientes y servidores (de forma independiente de las plataformas usadas) y de la entrega y devolución de los datos. Clientes y servidores acceden a las funciones del ORB mediante la interfaz que éste ofrece. El trabajo que el ORB ahorra a los programadores es monumental: no tienen que preocuparse de la localización de los objetos, de los protocolos de red, de las implementaciones de los objetos, etc. En el lado del cliente, el ORB es responsable de

- aceptar las peticiones a los objetos remotos;
- encontrar las implementaciones de los objetos (sirvientes);
- aceptar las referencias a objetos remotos;
- encaminar las llamadas que hacen los clientes –mediante referencias a objetos– a la implementación del objeto llamado (sirviente).

En el lado del servidor, el ORB

- permite que los objetos servidores registren nuevos objetos;
- recibe peticiones de los clientes;
- usa la interfaz de los objetos esqueletos para llamar a los métodos de activación de los objetos;
- crea referencias para los nuevos objetos y se las transmite a los clientes.

La comunicación entre adaptadores y esqueletos se realiza a través del ORB. Más adelante veremos cómo lo hace.

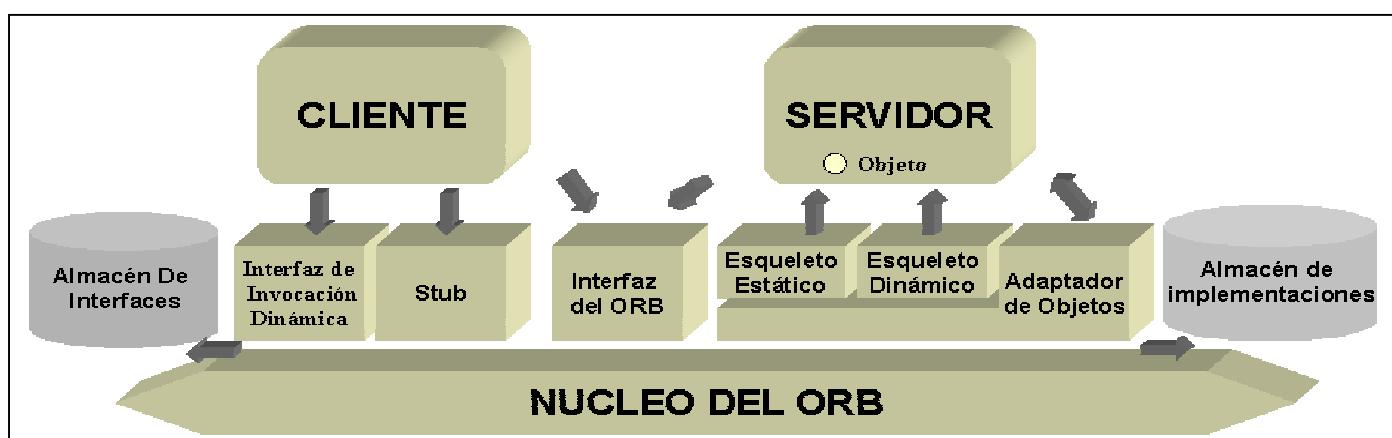
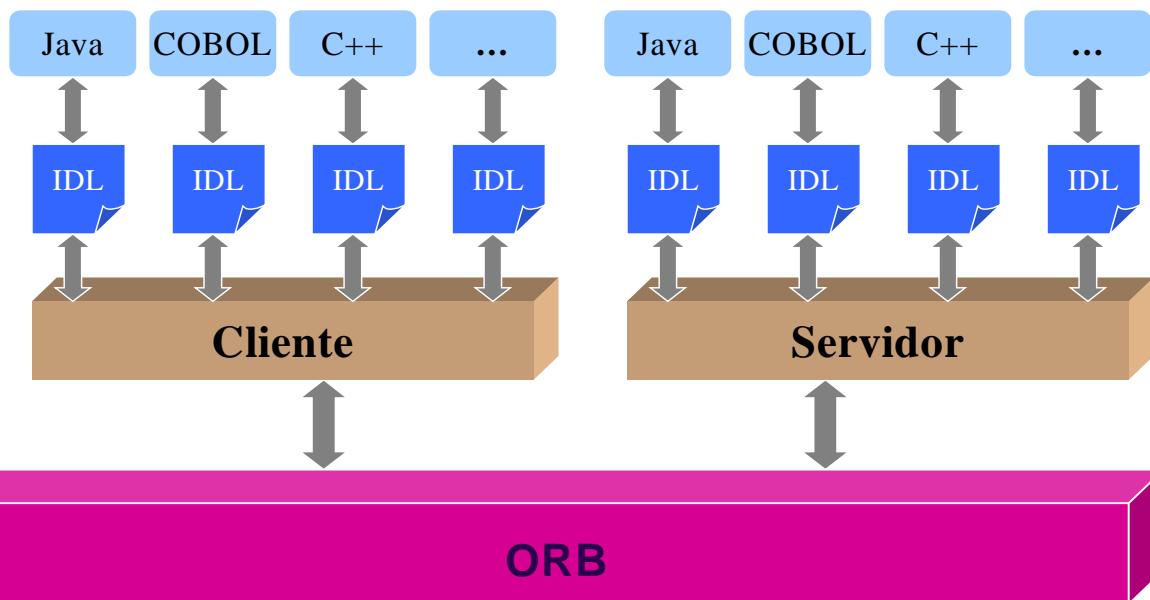


Figura 118. Otra vista de la arquitectura de CORBA

CORBA REPOSA SOBRE EL ORB



Miguel Ángel Abián Julio 2004

Figura 119. Todo pasa por el ORB

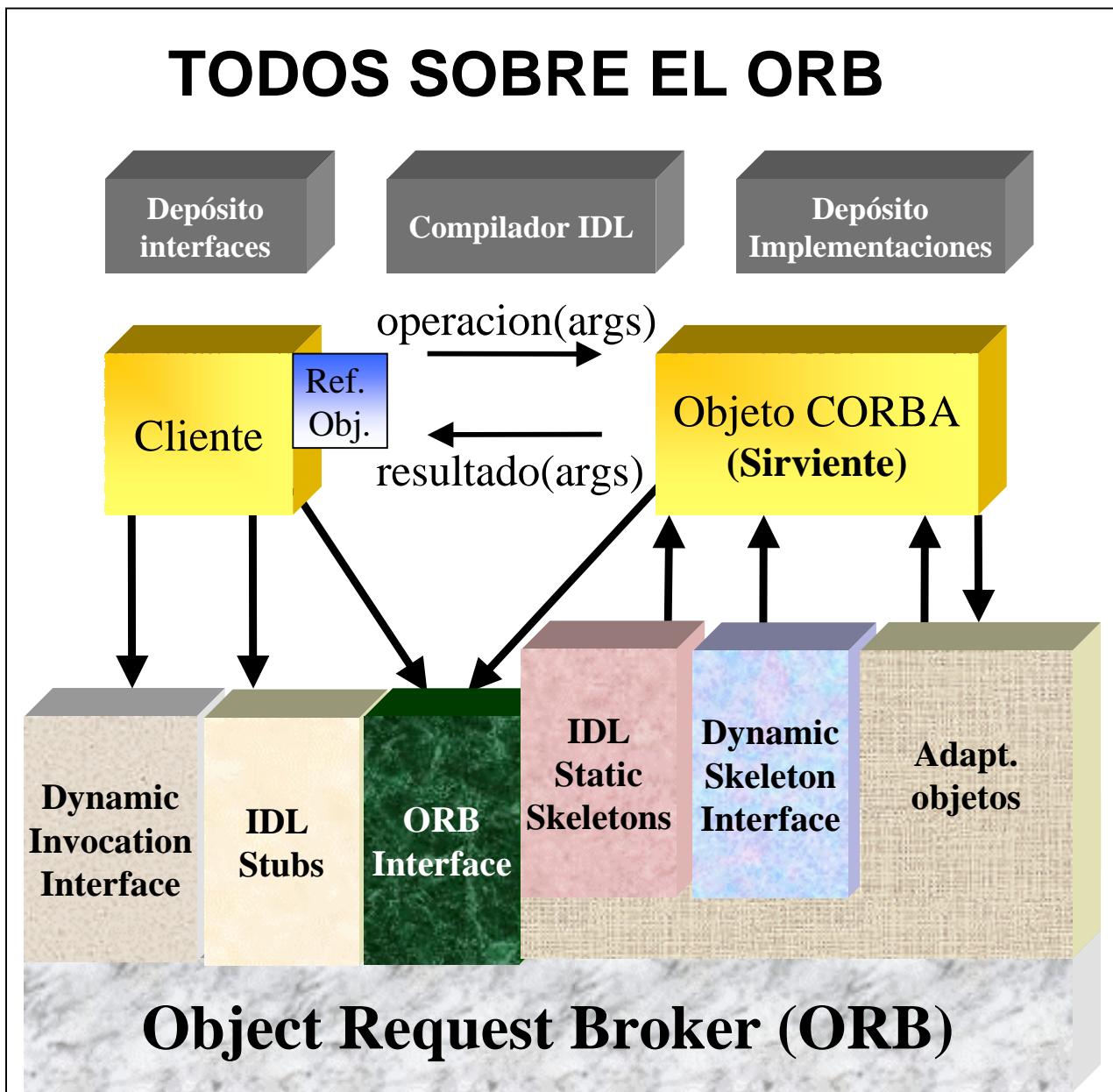


Figura 120. La figura anterior tras hacer un zoom. Cada elemento se irá viendo en este subapartado.

No debe pasarse por alto que el ORB que se representa en las figuras 118, 119 y 120 es un concepto lógico, no de implementación. El ORB se puede implementar de diversas formas:

- Mediante demonios del sistema.
- Integrado en el sistema operativo.
- Mediante código ubicado en distintas máquinas (es la más habitual: cada cliente y cada servidor ejecuta su implementación del ORB).
- Mediante código ubicado en una máquina, a la cual acceden clientes y servidores.
- Con bibliotecas.

Como el ORB es una especificación para CORBA, no una implementación, cada fabricante de software desarrolla sus propios productos ORB (por lo general, “un ORB” o “un orb” significa una implementación concreta del ORB). El OMG establece los servicios mínimos que debe ofrecer un ORB para asegurar su interoperabilidad con otros, pero no los máximos: hay fabricantes que ofrecen implementaciones del ORB con funciones adicionales (políticas de seguridad, de acceso, etc.).

Cada vez que se inicia un objeto en el lado del servidor, el ORB realiza estas tareas:

- Inicia el componente ORB.
- Obtiene una referencia al objeto mediante el servicio de nombres.
- Convierte la referencia en una cadena de texto.
- Conecta la referencia del objeto al objeto servidor y se desconecta.

El **IOP** (*Internet Inter-ORB Protocol*: protocolo de Internet entre los ORB) es el protocolo que se usa para transmitir mensajes a través del ORB (**sobre una arquitectura TCP/IP**). Este protocolo de red es una materialización del **GIOP** (*General Inter-ORB Protocol*: protocolo general entre los ORB). Éste último, más que un protocolo, es una especificación que define cómo crear protocolos concretos que funcionen en la arquitectura de CORBA. El IOP define

- los requisitos exigibles a la capa de transporte;
- la CDR (*Common Data Representation*: representación común de los datos);
- el formato de los mensajes.

La **CDR** es una codificación binaria que define la manera como los tipos de datos IDL se transforman en flujos de bytes que puedan ser enviados a la red. Mediante ella se codifican en bytes las referencias interoperables, los códigos de control, los argumentos de los métodos, los valores de retorno, las excepciones, etc.

Uno de los mayores errores de la versión 1.0 de CORBA fue no especificar ningún protocolo estándar para la comunicación entre objetos. Cada ORB podía utilizar el suyo, ya fuera libre o propietario, lo que provocaba que un cliente con un ORB fuera incapaz de comunicarse con un servidor con otro ORB. En la versión 2.0 de CORBA se introdujeron el GIOP y el IOP para permitir la interoperabilidad entre distintos ORB. El protocolo IOP se asienta sobre la capa de transporte TCP/IP, al igual que lo hace el protocolo JRMP de RMI (véase el subapartado 5.2.2). Como ya se comentó en 5.4, el compilador `rmiic` permite generar adaptadores y esqueletos que usan IOP en lugar de JRMP.

Como los cortafuegos suelen imponer restricciones al envío de datos con IOP, pues IOP puede usar cualquier puerto TCP, existen algunos ORB que permiten empaquetar los datos IOP en mensajes HTTP, lo que permite pasar a través de los cortafuegos (esta técnica se conoce como *HTTP tunneling* o pasarela HTTP).

Nota: Si un vendedor de tecnología software le dice que los servicios web se basan en protocolos de Internet y que CORBA no, absténgase de replicarle. Piense que se enfrenta con alguien cuya vida puede ser contradictoria: quizás se vea obligado a comer caviar para que sus hijos puedan comer lentejas.

No obstante, tenga claro que el protocolo IIOP es un protocolo de Internet de pleno derecho, tal y como dice su nombre.

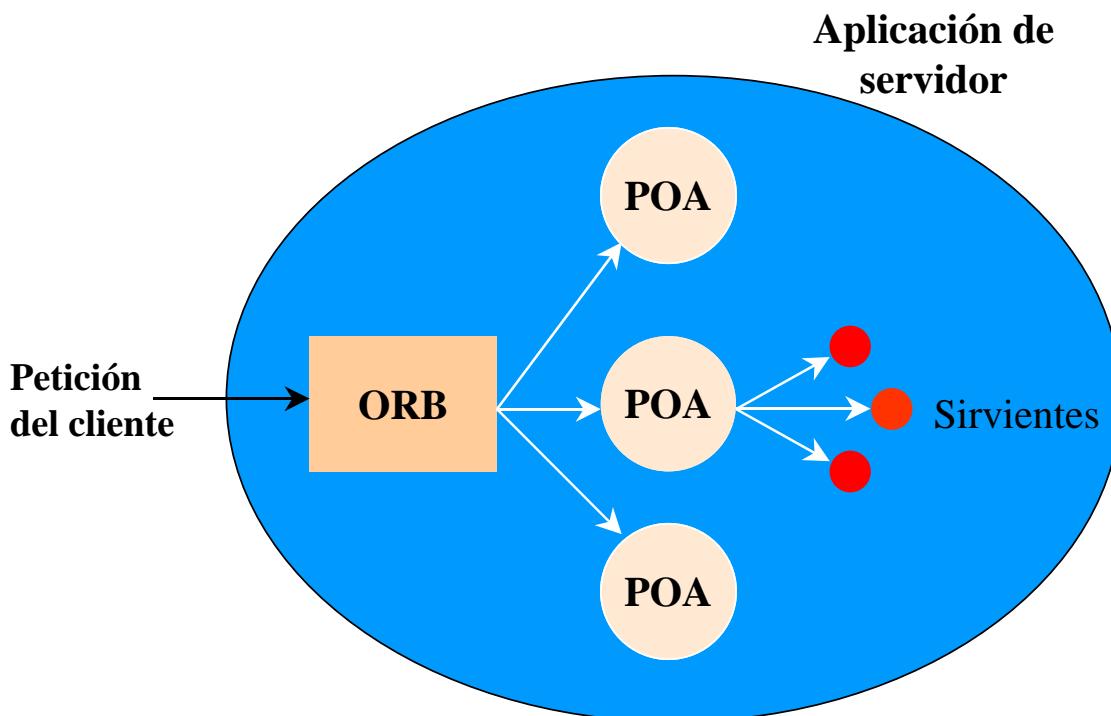
Un **OA** (*Object Adapter*: adaptador de objetos) es un elemento de la arquitectura de CORBA que se encarga de gestionar las peticiones que llegan a los servidores y de asignar referencias a los objetos CORBA cuando los crea. Mediante los adaptadores, el ORB consigue información sobre los objetos CORBA. Cuando el ORB recibe una llamada destinada a un objeto CORBA, debe encaminarla hacia la implementación del objeto (sirviente). El OA le ayuda de este modo: primero, encuentra el objeto sirviente apropiado, pues mantiene un registro de todos los sirvientes que implementan a objetos CORBA; después, pasa la llamada al esqueleto que, a su vez, se la pasa al sirviente.

Un OA tiene, entre otras, estas funciones:

- Registra objetos sirviente. Cada vez que se crea un objeto sirviente, debe registrarse con el OA para que éste tenga en cuenta que ese sirviente existe en el servidor. Para ello, el OA asigna un identificador único (*object ID*) para cada objeto sirviente (el identificador es único en el contexto de un OA).
- Envía cada llamada al código del servidor (mediante esqueletos).
- Activa y desactiva objetos. Con el proceso de activación, los clientes pueden enviar peticiones a los objetos (un objeto no activado no puede ser usado por los clientes). Activar implica asociar un objeto CORBA con un objeto sirviente. La desactivación consiste en eliminar la relación entre un objeto CORBA y su sirviente.
- Mantiene un equivalencia entre los objetos CORBA y sus sirvientes. Así se pueden localizar el sirviente adecuado cuando un cliente lo llama.
- Gestiona el depósito de implementaciones (*Implementation Repository*), el cual veremos más adelante.

Inicialmente, CORBA definió como OA el BOA (*Basic Object Adapter*: adaptador básico de objetos). Debido a las limitaciones que imponía a la interoperabilidad y a que ofrecía un conjunto de funciones muy limitado, apareció en 1998 el POA (*Portable Object Adapter*: adaptador portable de objetos), que sigue en vigor. Como un servidor puede tener múltiples POA asociados, el flujo de peticiones dentro de cada POA es controlado por un objeto *POAManager* asociado a cada POA. Un gestor POA puede dar prioridad a unas peticiones y descartar otras.

MANEJO DE LAS PETICIONES DE LOS CLIENTES



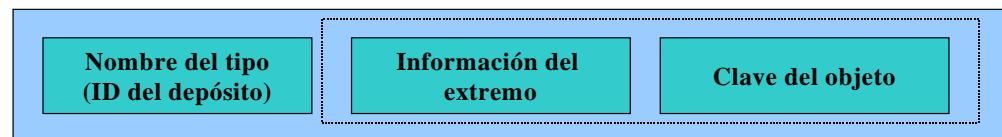
Miguel Ángel Abián Julio 2004

Figura 121 Manejo de una petición CORBA. Dentro de un proceso servidor puede haber muchos POA. Ahora bien, dado un POA, el identificador de un objeto es único. Si hay varias peticiones a un mismo POA, el POAManager (no representado en la figura) decide el orden de ejecución.

El identificador de un objeto se almacena dentro de una **IOR** (*Interoperable Object Reference*: referencia interoperable a objetos). Una IOR es una secuencia de bytes que contiene la información para localizar un objeto CORBA en una red. Como mínimo, una IOR contiene la dirección IP de un anfitrión, un número de puerto y un identificador del objeto dentro del proceso de servidor que lo creó (ya se dijo que una IOR se crea cuando se crea un objeto). Dos o más IOR pueden apuntar a un mismo objeto CORBA.

Como los adaptadores o *stubs* no pueden existir sin una IOR y como resulta infrecuente que una IOR exista fuera de un adaptador, suelen usarse de forma intercambiable los términos “referencia remota”, “IOR” y “adaptador” o “stub” (lo mismo sucedía con RMI). Siempre que un cliente consigue una IOR, el ORB genera un objeto adaptador.

REPRESENTACIÓN CONCEPTUAL DE UNA IOR



- **Nombre del tipo (Identificador del depósito)**
 - Almacena información sobre la interfaz del objeto representado por la IOR
- **Información del extremo**
 - Da información para la conexión física en red: nombre del anfitrión y número de puerto
- **Clave del objeto**
 - Clave por la que el ORB (así como el OA) identifica el objeto CORBA dentro del servidor.

Miguel Ángel Abián Julio 2004

Figura 122. Representación conceptual de una IOR.

REPRESENTACIÓN DE UNA IOR

```
IOR:01000000d00000049444c3a42616e6b3a312e3000000000200000000000000  
20000000010100000c00000726f73652e6670782e646500e830000040000004261  
6e6b01000000240000000100000010000001400000010000001000100  
0000000090101000000000
```

Representación mediante una cadena de texto de una referencia interoperable a un objeto

Repo Id: IDL:Bank:1.0

IIOP Profile

Version: 1.0
Address: inet:rose.fpx.de:12456
Location: iioploc://rose.fpx.de:12456/Bank
Key: 42 61 6e 6b

Bank

Multiple Components Profile

Components: Native Codesets:
normal: ISO 8859-1:1987; Latin Alphabet No. 1
wide: ISO/IEC 10646-1:1993; UTF-16
Key: 00

Contenido de la IR anterior: incluye, entre otras cosas, la dirección de un anfitrión y un número de puerto. El protocolo usado es el IIOP versión 1.0

Figura 123. Ejemplo de una IOR. Una referencia interoperable a un objeto puede enviarse por correo electrónico o almacenarse en una base de datos o en un fichero de texto (véase el “problema del Océano Pacífico” en 6.6)

Los tres componentes conceptuales de una IOR (mostrados en la figura 103a) son

- 1) **El nombre del tipo** (también llamado identificador del depósito o *repository ID*) indica el tipo más especializado del objeto CORBA representado con la IOR. El nombre del tipo se llama también identificador del depósito porque sirve como índice dentro del depósito de implementaciones (*Implementation Repository*; se explicará en las páginas siguientes).
- 2) **La información del extremo.** En este componente se especifica un protocolo y la información necesaria para localizar el objeto CORBA mediante ese protocolo. Si se usa el protocolo IIOP, dicha información consiste en un nombre de anfitrión y un número de puerto TCP. En lo sucesivo, considero que siempre se usa IIOP.
- 3) **La clave del objeto.** Este componente es un conjunto de datos binarios opacos a los clientes y cuyo formato depende del fabricante del ORB. La clave consta de dos partes: el **nombre del adaptador del objeto** y el **nombre del objeto**. La primera identifica un determinado adaptador dentro del proceso servidor que atiende las llamadas para el objeto al que hace referencia la IOR. La segunda identifica a qué objeto concreto dentro del adaptador hace referencia la IOR.

En la figura 117, el rectángulo punteado que rodea a la información del extremo y a la clave del objeto simboliza el hecho de que CORBA permita que una IOR contenga varios pares (información del extremo, clave del objeto). Dicho de otra forma: una misma IOR puede usar distintos protocolos y direcciones para un mismo objeto CORBA.

Dependiendo del carácter de los objetos a los que hacen referencia, las IOR pueden ser de dos tipos: transitorias o persistentes.

Una **IOR transitoria** sólo es útil mientras el servidor (un proceso, al fin y al cabo) que la generó se mantiene en ejecución. Una vez detenido el servidor, la IOR deja de ser válida. Incluso aunque se vuelva a ejecutar el servidor, la IOR ya no se podrá usar (el cliente recibirá una excepción si lo intenta).

En una IOR transitoria, en la información del extremo se almacena la dirección del anfitrión donde se ejecuta el proceso servidor y el número de puerto a través del cual se puede acceder al servicio CORBA. Cuando un proceso servidor crea una IOR transitoria, introduce en ella su dirección y su número de puerto TCP, así como el nombre del adaptador del objeto y el nombre del objeto.

Cuando un cliente emplea una IOR transitoria para hacer una llamada a un objeto CORBA, se conecta a la dirección y al número de puerto indicados en ella y envía, entre otras cosas, su clave de objeto. El proceso servidor extrae de la clave el nombre del adaptador y el del objeto, y los usa para localizar el sirviente asociado al objeto (si el objeto está desactivado, creará un sirviente y se lo asociará, o utilizará uno ya existente). El sirviente se encargará de procesar la petición del cliente.

Si el proceso servidor no está en marcha o no está asociado al nombre del anfitrión y al número de puerto que figuran en la IOR, el código del cliente recibirá una excepción `OBJECT_NOT_EXIST`.

Las **IOR persistentes** permiten identificar objetos aunque el proceso servidor que las generó se haya detenido una o más veces. Estas referencias sobreviven a los “apagados” del servidor. Una IOR persistente apunta a un objeto CORBA que siempre existe conceptualmente. Uso “conceptualmente” porque el objeto puede estar guardado en una base de datos o en un archivo de texto mientras no se necesita.

Como vimos, existe una clara separación entre los tiempos de vida de objetos y sirvientes. Cuando un proceso servidor se para, se destruyen todos los sirvientes que había creado; pero no sus objetos, que son virtuales (siempre que sean persistentes, claro está). Una IOR persistente deja de funcionar solamente cuando se destruye el objeto (virtual) que la tiene asociada, no cuando se destruye la implementación del objeto.

Cada vez que se detiene y se vuelve a arrancar un proceso servidor, puede ejecutarse en otro anfitrión. Incluso en el caso de que el nuevo proceso se ejecute en la misma máquina, lo habitual es que use otro puerto. En consecuencia, en una IOR persistente no basta con grabar un nombre del anfitrión y un número de puerto, tal como se hace con las IOR transitorias. Si se desea un mecanismo de persistencia, debe existir algún modo de que los clientes sepan, tras la parada de un servidor, a qué puerto y a qué máquina dirigir sus peticiones.

La solución adoptada es que, cuando se crea una IOR persistente, el servidor CORBA guarda en ella el nombre del tipo del objeto (o identificador del depósito), el nombre del adaptador del objeto, el nombre del objeto y la dirección del depósito de implementaciones (nombre de la máquina donde se ejecuta y número de puerto por el que escucha).

Un **depósito de implementaciones** (*Implementation Repository*) es una especie de contenedor donde se guardan las implementaciones de todos los objetos registrados en una aplicación CORBA. Gracias a él, un ORB puede conocer el nombre de los servidores en ejecución, los nombres de las máquinas que los albergan y el número de puerto que usa cada uno.

Durante la creación de una IOR persistente, el servidor CORBA establece contacto con el depósito de implementaciones y le envía el nombre del objeto (Obj13, p. ej.), el nombre del adaptador del objeto (Adapt1, p. ej.), así como el nombre de la máquina donde se ejecuta (aidima2.aidima.es, p. ej.) y el puerto por el que escucha peticiones (1025, p. ej.). Por así decirlo, todo servidor hace una presentación en toda regla ante un depósito de implementaciones. El servidor sabe dónde se halla el depósito de implementaciones porque la dirección consta en la configuración de la máquina local.

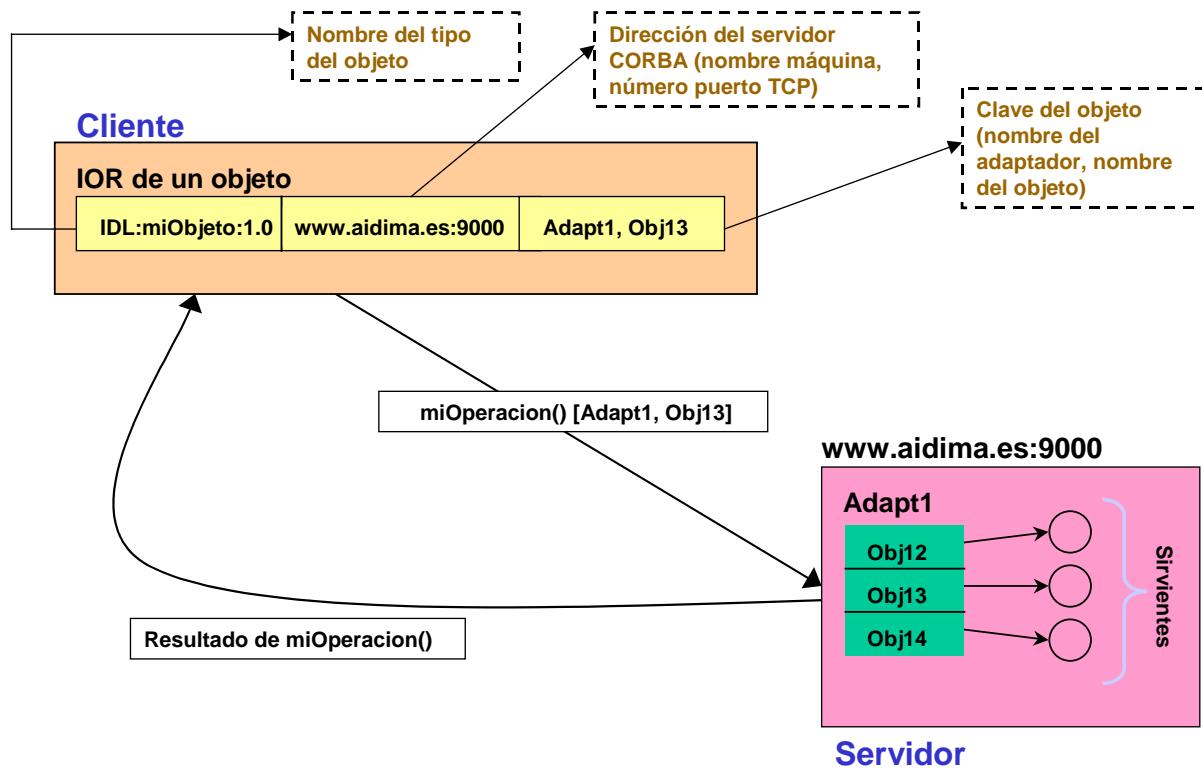
Cuando un cliente llama a algún método mediante una IOR persistente, la llamada va al depósito de implementaciones, no al servidor CORBA que ha generado la IOR. El depósito extrae de la IOR la dirección del servidor y la busca entre su información. Si la encuentra, envía al cliente un mensaje con esta información: el nombre de la máquina actual donde se ejecuta el servidor y el número de puerto. El cliente redirigirá su petición a la máquina y al puerto indicados por el depósito. Puede suceder que el depósito de implementaciones, al intentar comunicarse con el servidor, descubra que éste no está ya en ejecución. En dicho caso, recurrirá a la activación bajo demanda (siempre que el servidor se haya configurado así): enviará al servidor un mensaje de ejecución (*fork/exec*) que lo pondrá en marcha.

Volviendo a las comparaciones antropomorfas, cuando un servidor CORBA que genera objetos persistentes se inicia es como si dijera al depósito de implementaciones: “Buenos días. Acabo de ser creado. Mi nombre es Servidor1; vivo en una máquina apacible y acogedora, que se llama www.aidima.es y pueden

molestarme, perdón, consultarme, por medio del puerto 9000". Cuando el depósito activa a un servidor bajo demanda, éste envía un mensaje a aquél que vendría a decir "Buenas tardes. Su mensaje me ha creado. Vivo en la máquina www.aidima.es y pueden interrumpirme, perdón, consultarme, por medio del puerto 9000. A partir de ahora pueden enviarme peticiones sin necesidad de volver a ejecutarme".

La diferencia entre el uso de una IOR transitoria y una persistente se muestra en las figuras 124 y 125.

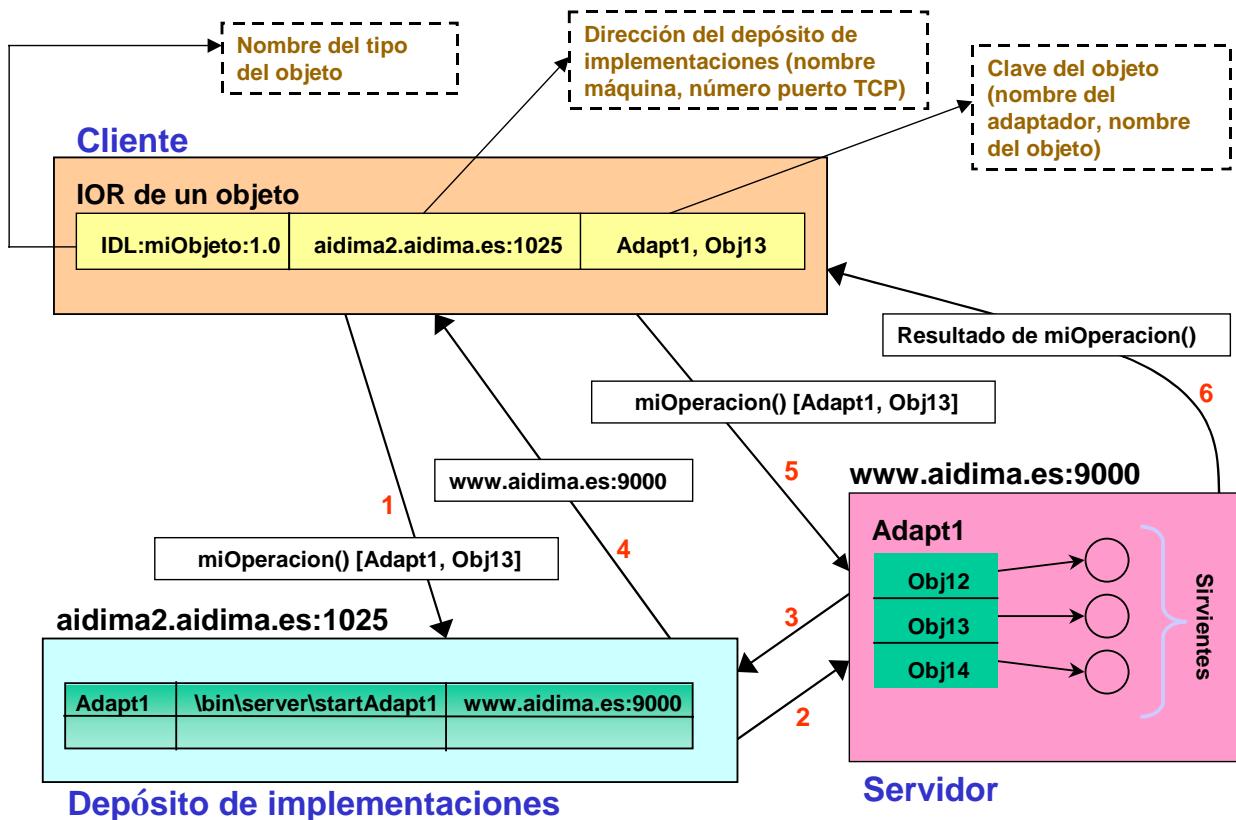
EJEMPLO DE USO DE UNA REFERENCIA TRANSITORIA



Miguel Ángel Abián Julio 2004

Figura 124. Llamada remota mediante una IOR transitoria

EJEMPLO DE USO DE UNA REFERENCIA PERSISTENTE



Miguel Ángel Abián Julio 2004

Figura 125. Llamada remota mediante una IOR persistente

En la figura 125 se considera que el depósito de implementaciones se ejecuta en el puerto 1025 de un anfitrión llamado `aidima2.aidima.es` y que el servidor CORBA se ejecuta en el puerto 9000 de la máquina `www.aidima.es`. La secuencia de pasos de la figura se explica a continuación:

- 1) Mediante una IOR, el cliente llama a la operación `miOperacion()` del objeto `Obj13`, y la llamada va a parar al depósito de implementaciones.
- 2) El depósito extrae de la llamada el nombre del adaptador y lo usa para buscar en su tabla de servidores el servidor CORBA correspondiente. Si no encuentra ningún servidor, devuelve al cliente una excepción `OBJECT_NO_EXIST`. Si encuentra un servidor pero está parado y no ha sido configurado como de activación bajo demanda, devuelve al cliente una excepción `TRANSIENT`. Si encuentra un servidor configurado para la activación bajo demanda y que está parado, envía un mensaje al anfitrión correspondiente para ejecutar el servidor CORBA y, luego, espera una respuesta de éste.
- 3) El servidor CORBA envía un mensaje al depósito de implementaciones en el que le informa de su dirección actual (nombre del anfitrión y número de puerto).

- 4) El depósito devuelve la dirección actual del servidor CORBA al cliente.
- 5) El cliente hace una segunda llamada a `miOperacion()`, dirigida ahora a la dirección proporcionada por el depósito.
- 6) El servidor CORBA extrae de la llamada la clave del objeto y averigua a qué sirviente corresponde. Por último, devuelve los resultados de `miOperacion()` al cliente.

El reenvío de la llamada del cliente al servidor CORBA no es algo que deba ser considerado en el código: la estructura interna del ORB se encarga automáticamente de ellos.

El **depósito de implementaciones** (*Implementation Repository*) almacena la información sobre las implementaciones de los objetos registrados. Mediante él, el ORB controla el proceso de localizar y activar objetos remotos. En caso de que el ORB necesite información sobre la implementación de un objeto concreto, puede consultar a este depósito. Como ya vimos, tres son sus funciones:

- Mantener un registro de todos los servidores conocidos
- Registrar todos los servidores que están en ejecución (máquina y número de puerto).
- Arrancar bajo demanda los servidores si están registrados para la activación automática.

Los depósitos de implementaciones son elementos optativos en la arquitectura de CORBA, si bien resultan necesarios cuando se quiere dotar a los objetos de persistencia.

El **depósito de interfaces** (*Interface Repository*) viene a ser un diccionario donde se almacenan las definiciones de todas las interfaces de los objetos registrados (es decir, los métodos de éstas y los argumentos y valores de retorno para cada método). Generalmente, un depósito de este tipo se implementa como un proceso que se ejecuta en un anfitrión y un número de puerto fijados. El anfitrión donde se ejecuta no tiene por qué coincidir con el del servidor CORBA.

No todos los productos CORBA implementan este servicio. Los que lo hacen tienen la ventaja de poder trabajar con objetos que no existían cuando se compilaron las aplicaciones mediante llamadas dinámicas (éstas se verán al final de este subapartado). También puede usarse para comprobar si las llamadas a un objeto son correctas: si el nombre del método o los tipos de los argumentos no coinciden con la información contenida en el depósito de interfaces para la interfaz del objeto, el ORB rechazaría la llamada (arrojará una excepción).

La información de las interfaces está en las descripciones IDL de éstas, pero los depósitos de interfaces proporcionan un modo de acceder a esta información en tiempo de ejecución. Si un cliente tiene una referencia remota a un objeto CORBA, puede consultar al depósito de interfaces sobre los métodos y sus argumentos y valores de retorno.

Al igual que RMI, CORBA usa un **servicio de nombres**. Un servicio de nombres es un servicio que permite que los clientes busquen objetos a partir de nombres lógicos ("miObjeto", p. ej.). El servicio de nombres está disponible en dos versiones:

- a) El servicio original de nombres de CORBA (conocido como *COS Naming Service* en la documentación de Java).
- b) El servicio interoperable de nombres de CORBA (conocido como *CORBA Interoperable Naming Service* o INS en la documentación del OMG). Este servicio es una extensión más actualizada del anterior y permite usar cadenas de tipo URL para los nombres de los objetos (p. ej., `corbaloc:iiop:1.2@mianfitrion:9000/miobjeto`).

Aquí sólo consideraré el primer servicio de nombres. Como la terminología que usa para él la documentación del OMG es bastante abstracta, buscaré en todo momento la similitud con un sistema de archivos.

En la jerarquía de nombres de CORBA, un contexto de nombrado (*Naming Context*) correspondería a un directorio en una jerarquía de archivos, mientras que el nombre del objeto corresponde a un archivo. El nombre completo de un objeto se conoce como un nombre compuesto; en éste, el primer componente da el nombre de un contexto de nombrado; el segundo, el de otro, y así sucesivamente, hasta que el último da el nombre del objeto. El contexto inicial de nombrado, o contexto raíz, es proporcionado por CORBA. Dentro de un contexto de nombrado, un nombre de un objeto es único.

La sintaxis para un nombre compuesto es la siguiente:

(contexto de nombrado₁) (contexto de nombrado₂) ... (nombre del objeto)

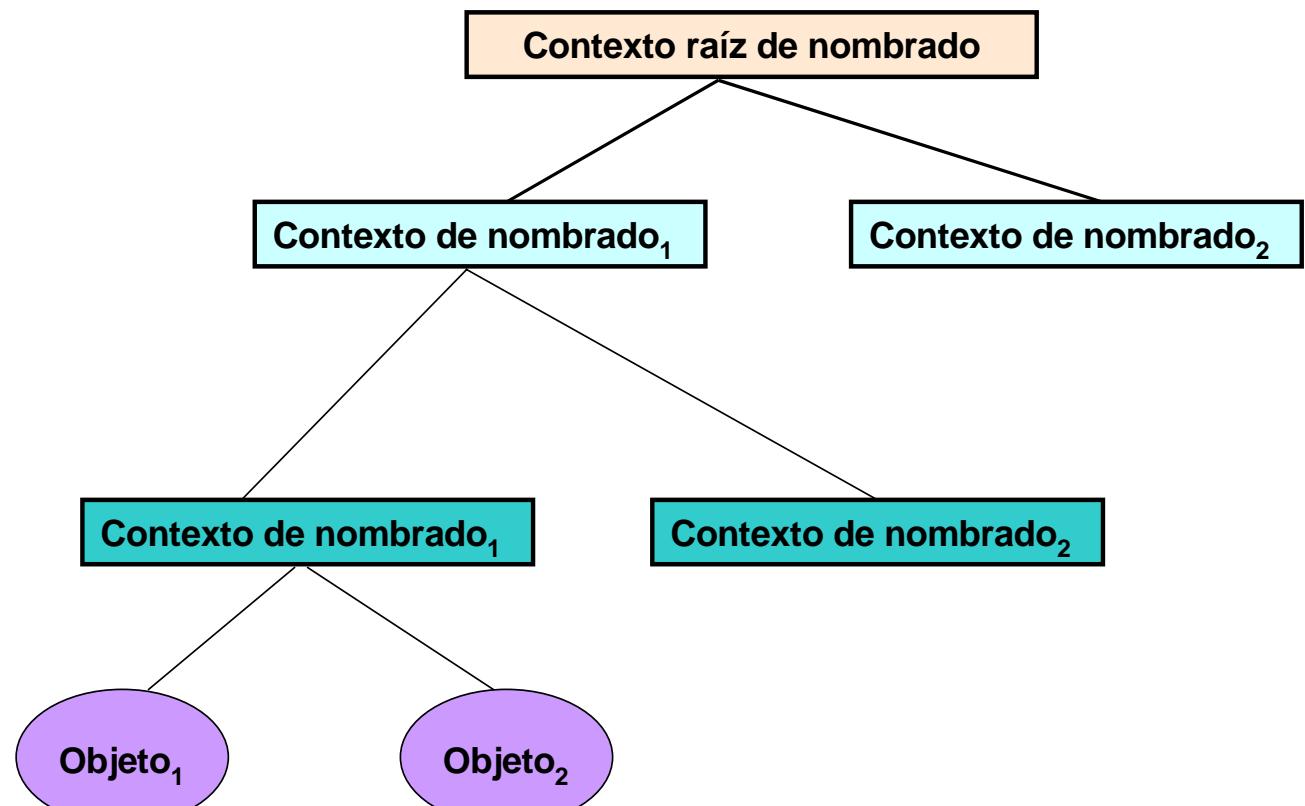
El servicio de nombres proporciona una API para poder crear contextos de nombrado y para asociar nombres a objetos. En Java, la interfaz `org.omg.CosNaming.NamingContext` representa cada rama o contexto de nombrado del árbol jerárquico de nombres. A cada objeto `NamingContext` se le puede preguntar: ¿Está aquí el objeto representado por este nombre?

Si se quiere registrar un objeto `jOvidio` en un contexto llamado `JefesDpto`, el código Java sería así:

```
ORB orb = ORB.init(args, null);
EmplImp jOvidio = new EmplImp(orb);
orb.connect(jOvidio);
// Se obtiene una referencia al contexto raíz de CORBA
org.omg.CORBA.Object refObj =
        orb.resolve_initial_references("NameService");
// Se transforma la referencia a un contexto de nombrado (es una
// conversión de tipos, pues la referencia apunta a un objeto
// CORBA genérico). El sistema de nombres INS usa NamingContextExt.
NamingContext Refnc = NamingContextHelper.narrow(refObj);
NameComponent nc1 = new NameComponent("JefesDpto","");
NameComponent nc2 = new NameComponent("JoseOvidio","");
NameComponent camino[] = {nc1, nc2};
Refnc.rebind(camino, jOvidio);
```

El nombre compuesto del objeto `jOvidio` sería `(JefesDpto) (JoseOvidio)`. El primer componente es el contexto de nombrado y el segundo el nombre del objeto. Mediante el nombre compuesto y el servicio de nombres, los clientes podrían acceder a los métodos remotos de `jOvidio`.

SERVICIO DE NOMBRES DE CORBA



Miguel Ángel Abián Julio 2004

Figura 126. Esquema del servicio de nombres de CORBA

FUNCIONAMIENTO DEL SERVICIO DE NOMBRES (1)

El servicio de nombres se ejecuta sobre el ORB y debe estar activo antes que los servidores y los clientes

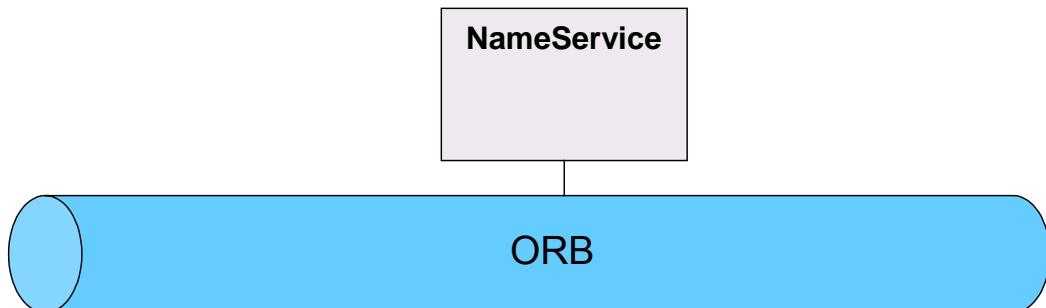


Figura 127. Primer paso para obtener una referencia a un objeto CORBA

FUNCIONAMIENTO DEL SERVICIO DE NOMBRES (2)

Un objeto de la clase Persona es registrado por el servidor en el servicio de nombres de CORBA

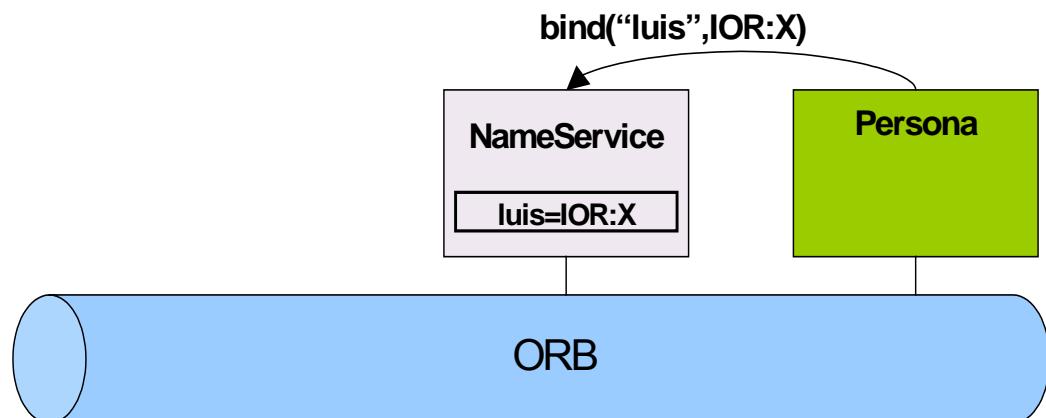


Figura 128. Segundo paso para obtener una referencia a un objeto CORBA

FUNCIONAMIENTO DEL SERVICIO DE NOMBRES (3)

Un objeto cliente usa el método *resolve_initial_reference* para localizar el servicio de nombres

`IOR:NS=resolve_initial_references("NameService")`

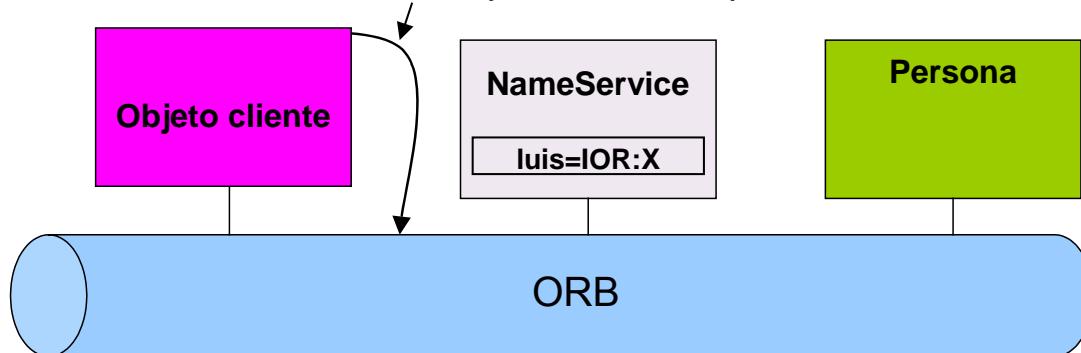


Figura 129. Tercer paso para obtener una referencia a un objeto CORBA

FUNCIONAMIENTO DEL SERVICIO DE NOMBRES (4)

Un objeto cliente usa el método *resolve* para obtener una referencia al objeto llamado "luis". A este proceso se le llama resolución del nombre "luis". Con la referencia obtenida, el cliente puede llamar a los métodos remotos del objeto CORBA "luis"

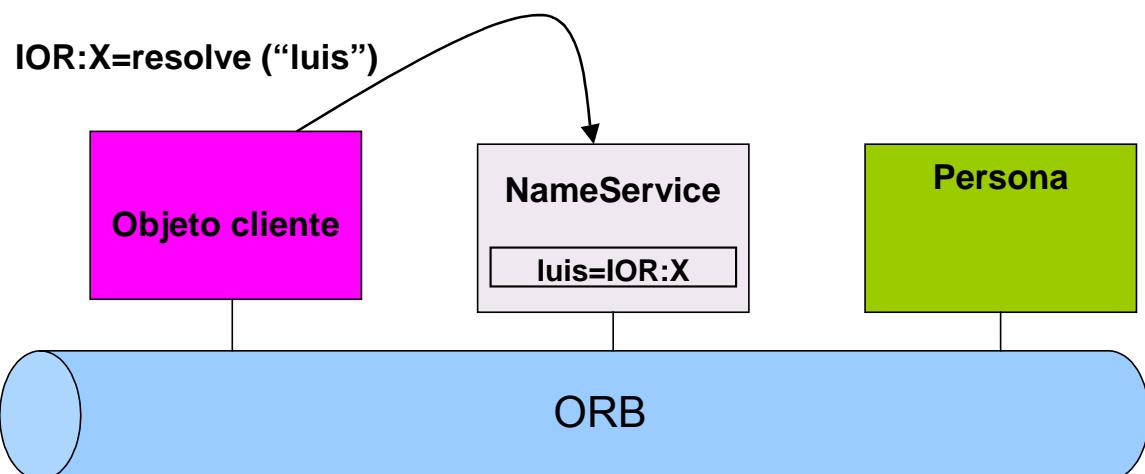


Figura 130. Cuarto paso para obtener una referencia a un objeto CORBA

¿Cómo se realizan las llamadas remotas en la arquitectura de CORBA? La respuesta no es sencilla. En las páginas anteriores, vimos cómo las llamadas alcanzan a los sirvientes, sea mediante referencias transitorias o persistentes; pero faltan ver muchos subprocessos que se ocultan tras las llamadas (creación de objetos, envío de peticiones IIOP, funcionamiento de los adaptadores y esqueletos, etc). Para empezar, deben distinguirse dos casos: el de las llamadas estáticas y el de las dinámicas.

Las Llamadas estáticas ocurren cuando los adaptadores y los esqueletos existen antes de la compilación del cliente. Su funcionamiento es muy parecido al de las llamadas remotas con RMI, salvo por el detalle de que hay un adaptador de objetos. Los pasos que se siguen se exponen aquí:

- a) El cliente obtiene una referencia a un objeto CORBA (una referencia interoperable). Apenas obtenida, el ORB crea un adaptador. Los adaptadores o *stubs* son generados automáticamente a partir del archivo IDL de la interfaz del objeto CORBA.
- b) El cliente llama a uno de los métodos de la interfaz del objeto (definida en el archivo IDL asociado).
- c) El adaptador comunica la llamada al ORB y le da el nombre del objeto, el método al cual se llama y los argumentos necesarios (si los hubiere). Esta información se codifica dentro de una petición CORBA (las peticiones también son objetos).
- d) El ORB envía un mensaje IIOP para localizar el objeto mediante el servicio de nombres de CORBA. El mensaje IIOP no es más que un flujo de bytes codificado según unas reglas (dadas por la CDR) y en el cual se han “empaquetado” los argumentos de la llamada, el nombre del método y el identificador del objeto llamado.
- e) La aplicación o el proceso del servidor, al recibir el mensaje IIOP, crea (si no existía ya) un objeto sirviente correspondiente al objeto CORBA, lo registra (mediante el OA) e informa al ORB de que el objeto es accesible.
- f) El ORB pasa la petición al OA.
- g) El OA determina cuál es el objeto llamado, así como el método al que se llama. A continuación, pasa la petición al esqueleto asociado a la interfaz del objeto CORBA.
- h) Del flujo de bytes de la petición, el esqueleto extrae (“desempaquetta”) el nombre del método, los argumentos de la llamada y el identificador del objeto llamado. Acto seguido, pasa la llamada, con sus argumentos, a un objeto sirviente.
- i) El objeto sirviente procesa la llamada y devuelve el resultado (si procede) al esqueleto.
- j) El esqueleto recibe la respuesta del objeto sirviente y la envía al ORB mediante un mensaje IIOP.
- k) En el cliente, el ORB extrae los datos del mensaje de respuesta y se los envía al adaptador.
- l) El adaptador lee los datos enviados por el ORB y se los devuelve al cliente.

Teóricamente, se podría integrar el OA dentro del ORB, y así el esquema de las llamadas remotas en CORBA quedaría muy parecido al de RMI. Sin embargo, en la práctica es más recomendable mantener la separación entre el OA y el ORB, pues así se evita tener que recompilar el código que implementa al ORB cuando se añade o se modifica algún método de la interfaz.

Las Llamadas dinámicas ocurren cuando se llama a un objeto cuya interfaz era desconocida cuando se compiló el cliente o cuya interfaz ha cambiado tras la compilación de éste. Si la interfaz del objeto llamado era desconocida cuando se compiló el cliente, éste no dispone de adaptadores que le digan cuál es la interfaz del objeto al que llama. Si se ha cambiado la interfaz de un objeto tras la compilación del cliente (añadiendo o quitando un método, pongamos por caso), el cliente no tiene un adaptador actualizado que simule los métodos del objeto llamado. En ambos casos, las llamadas estáticas resultarían inservibles.

Para poder trabajar con situaciones como las descritas, CORBA permite las llamadas dinámicas. Para poder hacerlas, el lado del cliente en la arquitectura CORBA dispone de la API DII (*Dynamic Invocation Interface*: interfaz de llamadas dinámicas) para poder llamar a métodos que no figuran en los adaptadores. Usando esta API, los clientes pueden descubrir nuevos métodos en tiempo de ejecución.

La DII tiene su contrapartida en el lado del servidor: la API DSi (*Dynamic Skeleton Interface*: interfaz de esqueletos dinámicos) permite que un ORB entregue peticiones a un sirviente que desconocía en tiempo de compilación el tipo del objeto al que implementa. Los servidores pueden usar la DII para crear en tiempo de ejecución objetos desconocidos durante la compilación.

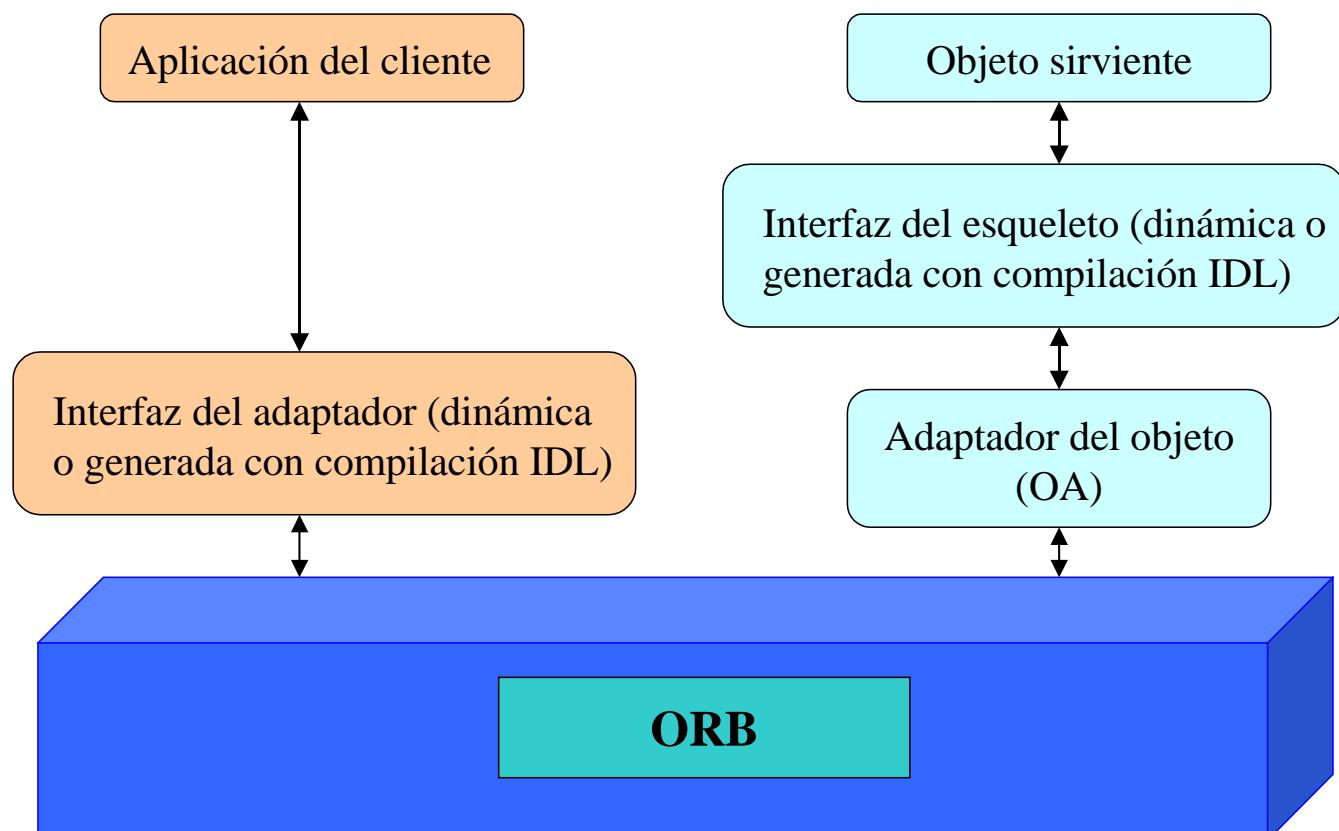
En una llamada dinámica, los pasos son los dados para las llamadas estáticas (la diferencia estriba en que, en vez de llamar al adaptador o al esqueleto, se llama a la *Dynamic Skeleton Interface* (DSI) o a la DSi del objeto CORBA).

En general, el uso de estos las API DII y DSi no es necesario para el programador medio de CORBA. Los únicos usos que me vienen ahora a la mente para el uso explícito en el código de estos mecanismos de enlace dinámico sería

- la creación de herramientas de desarrollo que permitieran crear nuevos objetos y modificar el código sin necesidad de parar y recompilar la aplicación CORBA en marcha (algo parecido a lo que permite hacer Eclipse con los objetos Java);
- la construcción de “traductores” entre distintas arquitecturas distribuidas de objetos (por ejemplo, entre CORBA y RMI, o entre CORBA y DCOM).

Utilicemos llamadas dinámicas o estáticas, el resultado final se describe en la siguiente figura.

ESQUEMA DE LAS LLAMADAS REMOTAS EN CORBA



Miguel Ángel Abián Julio 2004

Figura 131. Esquema de las llamadas remotas en CORBA

6.5. CORBA y RMI: puntos de intersección, puntos de fuga

Cuando apareció la RMI de Java, muchos programadores pensaron que era una herramienta creada para hacer la competencia a CORBA. Sin embargo, no ha sido así: Sun se ha caracterizado por cuidar mucho las implementaciones de las herramientas Java basadas en CORBA y por una política de mejora continua. La versión empresarial de Java (J2EE) usa RMI-IIOP e incorpora muchas características de CORBA, lo cual hace que se integre muy bien con aplicaciones CORBA.

Hoy día, Java es una estupenda elección para el desarrollo con CORBA. Esta afirmación no es fruto de la lectura de la propaganda de Sun, sino de muchas experiencias con herramientas para C++ y Java. Las herramientas actuales para usar CORBA con Java no tienen nada que envidiar a las existentes para C++ (y no olvidemos que C++ ya estaba completamente maduro, e incluso un poco podrido, cuando Java irrumpió en el mercado).

Desde la versión 1.2 del JDK, Sun ha incluido herramientas para trabajar con CORBA desde Java. Con ellas, se puede usar Java para generar componentes CORBA tal y como se puede usar C++ o COBOL, pero con todas las ventajas de aquel lenguaje. En la versión 1.4 del JDK se incluyen estas herramientas:

- Un compilador de IDL a Java, llamado `idlj`.
- Un ORB que se ejecuta como un demonio (`ordb`, de *Object Request Broker Daemon*), donde se suministra servicio de arranque, un servicio de nombres transitorios, un servicio de nombres persistentes y un gestor del servidor.
- Una herramienta `servertool`, que permite registrar, desregar, comenzar y apagar un servidor.
- Una herramienta `tnameserv`, que es un servicio de nombres transitorios, incluido por compatibilidad con las versiones anteriores del JDK.

Todas estas herramientas forman parte de lo que se conoce como la **tecnología IDL de Java o el IDL de Java (Java™ IDL)**; la documentación de Sun está en <http://java.sun.com/products/jdk/idl/index.jsp>, basada en la versión 2.3.1. de CORBA. Actualmente, Sun no proporciona ningún depósito de interfaces. Así pues, los usuarios del IDL de Java no pueden usar la API DII.

Desde luego, existen ORB con depósitos de interfaces, amén de con otros muchos servicios adicionales. Algunos son de pago y otros son gratuitos. Un producto excepcionalmente bueno es **VisiBroker ORB**, de Borland (es de pago, pero se pueden descargar versiones de prueba que funcionan al 100% durante un tiempo). **Orbacus** (<http://www.ooc.com>) también es un producto comercial bastante recomendable: se basa en la versión 2.5 de CORBA, permite trabajar con Java y C++, incorpora depósito de interfaces, se distribuye en forma de código fuente y está disponible para Windows, HP-UX, Linux, IRIS, AIX, Solaris y Tru64. En la página web del producto se puede descargar una versión de evaluación que no caduca (mientras escribo esto, la última versión es la 4.1.3). Si consideramos productos libres, **JacORB** (distribuido bajo licencia GNU) es un buen producto para el programador medio de CORBA (va por la versión 2.2). Proporciona DII y DSII, depósito de interfaces y de implementación, además de permitir el empaquetado de datos IIOP en mensajes HTTP (técnica conocida como *HTTP tunneling* o pasarela HTTP).

Mientras escribía este tutorial, he podido contar unos cincuenta ORB, casi todos compatibles con Java: ORBit (<http://www.labs.redhat.com/orbit>), omniORB2

(<http://www.uk.research.att.com/omniORB/omniORB.html>), COOL ORB, ILU, Arachme, TAO, Electra, COPE, Jorba, CORBAPlus, Jyhu, Fnorb, ROBIN, DOME, Bionic búfalo, Orbix, OrbixWeb... Aunque seguro que existen más, este hecho da cuenta del interés que despierta la unión entre Java y CORBA.

Las similitudes entre RMI y CORBA se resumen en una: ambas son tecnologías para construir aplicaciones distribuidas basadas en objetos.

Las diferencias entre CORBA y RMI son numerosas, pero hay una sustancial: RMI se centra en Java, mientras que CORBA no privilegia a ningún lenguaje (ni plataforma). A primera vista, esto puede parecer un claro argumento en contra de RMI; pero no hay que ajusticlarla o descartarla antes de examinar minuciosamente las pruebas de la defensa (hasta las tecnologías merecen un juicio ecuánime).

La primera prueba a favor de RMI es que, al ser una solución 100% Java, puede ejecutarse en cualquier plataforma donde exista una máquina virtual Java. Cualquier parte de una aplicación distribuida que use RMI puede cambiar de ubicación y colocarse en cualquier anfitrión con una MVJ.

La segunda prueba consiste en un ataque directo a CORBA: RMI permite hacer cosas vedadas para la primera tecnología. En las llamadas remotas, RMI permite enviar objetos por valor de una forma natural (las últimas versiones de CORBA también lo permiten; pero de una forma poco " limpia " y con una especificación donde abundan los huecos y los retales). En cambio, los objetos CORBA nacen y mueren atados a las máquinas donde fueron creados: no se puede enviar objetos de un anfitrión a otro (en lugar de ellos se envían sus adaptadores o *stubs*). Por otro lado, la serialización de objetos, imposible en CORBA, permite descargar código de forma dinámica. En consecuencia, la distribución de las aplicaciones RMI es mucho más fácil que en CORBA (el talón de Aquiles de CORBA siempre ha sido la distribución en el lado del cliente). Si se opta por usar *applets* como clientes CORBA, la instalación y configuración de las aplicaciones en el lado del cliente se vuelve trivial.

En la tercera prueba, cualquier juez apreciaría premeditación. Mucho antes del día de autos, Sun e IBM trabajaron para desarrollar un RMI basado en el protocolo IIOP (lo que se conoce como RMI/IIOP o "RMI sobre IIOP"). Ya en la versión 1.3 del J2SE, Sun incorporó la capacidad de que RMI trabajara con el protocolo IIOP (usado por CORBA), además de con JRMP. Si se usa RMI-IIOP en lugar de JRMP, las aplicaciones RMI pueden acceder a objetos CORBA, y viceversa (luego veremos qué condiciones deben satisfacerse para lograr la interoperabilidad entre RMI y CORBA). Con RMI/IIOP desaparece, pues, la limitación de que las aplicaciones RMI sólo servían cuando todas las partes estaban escritas en Java.

Finalmente, la cuarta prueba para lograr la absolución del acusado se basa en el empirismo. RMI es mucho más fácil de usar que CORBA: su lenguaje de definición de interfaces coincide con el usado para escribir las aplicaciones (Java), no necesita ningún ORB ni ningún adaptador de objetos, cuenta con su recolector distribuido de basura (en el subapartado siguiente veremos por qué CORBA carece de uno). Irreparablemente, más posibilidades implica más complicación.

Concluido el juicio, debe matizarse que, si la arquitectura de CORBA es tan compleja, es porque permite trabajar con muchos lenguajes y proporciona características de seguridad y escalabilidad más completas que las de RMI. En otras palabras: CORBA es más pesado que RMI, pero más robusto. No expondré otra vez los puntos débiles de RMI, pues se detallaron en 5.6.

Como ya he adelantado, IIOP permite que las aplicaciones RMI pueden acceder a objetos CORBA, y viceversa. Se vuelve falsa, pues, la afirmación de que las aplicaciones RMI sólo sirven cuando todas las partes están escritas en Java. Si se generan adaptadores y esqueletos (o sólo adaptadores) que usen IIOP, cualquier objeto CORBA podrá interaccionar con ellos. Podemos tener un servidor RMI que atienda peticiones de clientes escritos en COBOL, Smalltalk o C++, y viceversa. **La única condición que debemos exigir es que las interfaces de todos los componentes de la aplicación se hayan definido como interfaces remotas de Java.** Dicha exigencia resulta imprescindible, porque –por ejemplo– un servidor escrito en C++ a partir de una interfaz IDL podría ser incapaz de entender lo que quiere decirle un cliente Java escrito a partir de una interfaz remota Java. ¿Por qué? Pues porque el IDL es más rico en “contenido” que Java: el cliente Java sería incapaz de entender una respuesta del servidor C++ en la que se usaran argumentos `inout` (C++ sí permite estos argumentos).

Esta condición sólo constituye un ligero problema cuando se trabaja con aplicaciones ya escritas y que no pueden modificarse: entonces hay que usar la tecnología IDL de Java. Supongamos que tenemos un servidor CORBA escrito en C++ y que interesa construir un cliente Java que envíe peticiones al servidor. Para conseguirlo, deberemos obtener el archivo IDL de la interfaz del servidor y compilarlo mediante `idlj` (el compilador IDL de Java). La compilación generará los archivos que nos servirán como base para la aplicación (adaptadores, esqueletos, etc.)

Si trabajamos con aplicaciones nuevas, aún no implementadas ni distribuidas, podemos usar RMI (con IIOP) para trabajar con objetos CORBA. En el ejemplo anterior, lo primero que haríamos sería escribir una interfaz remota RMI para el servidor. Después, con la aplicación `rmic`, generaríamos un archivo IDL para la interfaz RMI (usando `rmic -idl archivointerfazRMI`). En la parte del servidor, un compilador IDL para C++ se encargaría de generar los adaptadores y esqueletos a partir del archivo IDL generado antes. En la parte del cliente Java, aquéllos se generarían mediante `rmic -iiop archivointerfazRMI`. Una vez implementado el código necesario en ambas partes, el servidor C++ será un servidor CORBA “puro” y el cliente Java será un cliente RMI “puro”. Ambos podrán interoperar sin ningún problema.

La regla para usar la tecnología IDL de Java o RMI es muy sencilla: para aplicaciones CORBA ya escritas y en funcionamiento, conviene usar la primera; para las aplicaciones CORBA aún no materializadas, conviene usar RMI (con IIOP) para todas las partes escritas en Java. Desde luego, si todos los elementos de una aplicación distribuida van a escribirse en Java, lo natural es usar RMI (recomendaría usar RMI con IIOP para este caso, pues nunca se sabe si algún día se necesitará interoperabilidad con objetos CORBA).

Aunque por motivos bien distintos, CORBA y RMI tienen una desventaja común: su escasa eficacia para las transferencias masivas de datos. En el caso de CORBA, por la complejidad de la arquitectura y por la conversión de los datos al formato CDR; en el caso de RMI, por la serialización de objetos: una llamada remota puede implicar serializar y deserializar cientos o miles de objetos.

CORBA ha contado con el problema añadido de luchar al principio con un enemigo muy poderoso: los cortafuegos. Como los servidores CORBA escuchan en puertos aleatorios, suelen provocar conflictos con las políticas de seguridad de los cortafuegos, que sólo permiten que unos pocos puertos (como el de HTTP o FTP) puedan ser accedidos desde el exterior. Este problema hizo que cundiera la

desconfianza de las empresas hacia esta tecnología. Ahora existen ORB que evitan los conflictos introduciendo los mensajes IIOP en mensajes HTTP (*HTTP tunneling* o pasarela HTTP); pero el problema es que no existían cuando las primeras implementaciones de CORBA vieron la luz. RMI siempre ha permitido usar esta treta para evitar problemas con los cortafuegos (este comportamiento por defecto puede desactivarse con `java.rmi.server.disableHttp=true`).

6.6. El problema del océano Pacífico: náufragos en los mares de CORBA

CORBA no dispone de un recolector distribuido de basura por motivos conceptuales, no técnicos. Al permitir CORBA que las IOR se guarden en correos electrónicos, en bases de datos o en ficheros de texto, el significado de una recolección automática de los objetos persistentes que no se usan es incompatible con el de una IOR persistente. En el artículo *Binding, Migration, and Scalability in CORBA [Communications of the ACM, Vol. 41, No 10, Octubre 1998]*, Michi Henning definió lo que se conoce desde entonces como “el problema del océano Pacífico”.

Consideré el siguiente escenario: se encuentra perdido en una isla desierta en el océano Pacífico, aislado y con un servidor CORBA como único enlace con el resto del mundo. Puede, por tanto, contestar las peticiones CORBA que reciba, pero no puede enviar mensajes CORBA. Harto de pasar sus días en una isla desierta y de alimentarse a base de cocos y agua de lluvia, y maldiciéndose por no llevar consigo un ejemplar de *Robinson Crusoe*, decide crear en el servidor CORBA un objeto persistente que avise de su situación (llamémoslo un objeto *SOS*). Una vez creado el objeto *SOS*, obtiene su IOR en forma de cadena de texto (hay un ejemplo en la figura 123), la escribe en un trozo de la corteza de algún árbol, introduce ésta en una botella, le pone un tapón y la arroja al mar.

La botella flota por el océano durante meses y finalmente acaba en una playa de Australia, donde es abierta por una persona con conocimientos de CORBA. Al leer la cadena de texto, reconoce al instante que es una referencia interoperable CORBA y la utiliza para acceder al objeto *SOS*, por medio del cual averigua que está usted en una isla desierta y que necesita ayuda. La historia acaba bien: él acude en su rescate y le libera de su exilio involuntario.

Este ejemplo nos muestra una cuestión importante: como CORBA permite que las referencias persistentes se propaguen de maneras incontrolables (por correo electrónico, por correo ordinario, dentro de una bolleta perdida en el océano, etc.), no se puede saber si una IOR es todavía de interés para algún cliente. En este ejemplo, la IOR del objeto *SOS* continúa siendo de interés cuando flota –dentro de una botella– por el océano Pacífico, y la persona que encuentra la botella tiene derecho a suponer que su llamada CORBA mediante esa IOR llegará al objeto *SOS*.

Por todo ello, el significado de una IOR en CORBA hace imposible que se pueda ejecutar un recolector de basura sin correr el riesgo de dejar a algún cliente con una IOR inválida, esto es, con un enlace a ninguna parte.

6.7. Java y CORBA: un par de ejemplos

Advertencia: Tanto en las explicaciones que siguen como en el código de los ejemplos usaré las herramientas para CORBA que vienen en el **JDK versión 1.4 y posteriores**. Suponen tantas ventajas para los programadores de CORBA que creo que vale la pena saltarse la compatibilidad con las versiones anteriores. Si en la documentación que maneja encuentra compiladores como `idltojava` es que está usando alguna versión del JDK anterior a la 1.3 (no considero aquí la versión 1.3 puesto que no era compatible con el POA).

En este subapartado expondré dos ejemplos del uso de CORBA con Java. El primero consiste en una aplicación que devuelve al cliente los mensajes enviados por éste, junto con alguna información del servidor (hace de eco, por así decirlo). El segundo, en una aplicación de *chat* muy sencilla.

Para ambos ejemplos considero que todas los archivos están en el fichero `bin` del JDK (en caso contrario, se ha de configurar el `CLASSPATH`) y que todos los procesos (incluyendo el servicio de nombres) se ejecutan en la máquina local. Como esa situación sólo es aplicable en el período de pruebas y depuración, daré luego instrucciones detalladas para distribuir las aplicaciones y hacerlas funcionar en un verdadero sistema distribuido.

Nota: La versión del JDK usada para la capturas de pantallas es la 1.4.2, que viene por defecto en el JBuilder X; pero también he comprobado que todo el código funcionaba en la versión 1.2 y en el JDK 1.5.0 Beta 1. Todo el código que usa CORBA se ha probado con los sistemas operativos Windows XP Profesional (en un PC doméstico), Red Hat 9.0 (en un PC doméstico) y Solaris 8.0 (en una estación de trabajo SUN BLADE 150).

Para el ejemplo del servicio de eco, los pasos para construir la aplicación y ejecutarla se detallan a continuación.

Paso a) Se escribe la interfaz IDL.

Ejemplo 19a: Mensajeria.idl

```
module Mensajeria {  
    interface Eco {  
        string enviarTexto(in string texto);  
    };  
};
```

Paso b) Se compila el fichero Mensajeria.idl:

```
idlj -fall Mensajeria.idl
```

La opción –fall indica que se generen todos los archivos, tanto los del servidor como los del cliente. La compilación genera un directorio Mensajeria, dentro del cual se encuentran los siguientes archivos:

- `_EcoStub.java`. Es la clase adaptadora para el cliente. Implementa la interfaz `Eco` y actúa como clase base para los clientes. Esta clase se encarga de codificar el argumento de `enviarTexto()` al formato que usa IIOP y de enviarlo a través de la red, así como de decodificar la respuesta del método (un `String`).
- `Eco.java`. Es la interfaz Java equivalente al fichero IDL. Hereda de la clase `org.omg.CORBA.Object` e implementa la interfaz `EcoOperations`.
- `EcoHelper.java`. Las clases `Helper` permiten leer objetos `Eco` de flujos CORBA de entrada, así como escribirlos en flujos CORBA de salida. Además, ayudan a convertir referencias genéricas a objetos CORBA en objetos concretos mediante el método `narrow()`.
- `EcoHolder.java`. Las clases `Holder` permiten que Java pueda usar objetos de tipo `Eco` en argumentos de tipo `out` o `inout` de las interfaces IDL (luego veremos un ejemplo). Si trabajásemos con C++, el compilador IDL no generaría clases `Holder`, pues este lenguaje permite el paso por referencia.
- `EcoOperations.java`. Se encarga de
- `EcoPOA.java`. Esta clase abstracta es la clase esqueleto del servidor y actúa como la clase base para los sirvientes. Implementa la interfaz `EcoOperations`. Esta interfaz contiene el método `enviarTexto()`.

Advertencia: Si se editan los archivos `.java` generados en la compilación IDL, no deben modificarse. En general, nunca se necesita usarlos.

Java y las redes. Introducción a las redes, a java.io, java.zip, java.net, java.nio, a RMI y a CORBA

La clase `Eco` no alberga mucho misterio. Es una interfaz Java normal y corriente:

```
package Mensajeria;

/*
 * Mensajeria/Eco.java .
 * Generated by the IDL-to-Java compiler (portable), version "3.1"
 * from Mensajeria.idl
 * Junes 26 de julio de 2004 23H52' CEST
 */

public interface Eco extends EcoOperations, org.omg.CORBA.Object,
org.omg.CORBA.portable.IDLEntity
{
} // interface Eco
```

La clase `EcoOperations` tampoco nos saca de pobres:

```
package Mensajeria;

/*
 * Mensajeria/EcoOperations.java .
 * Generated by the IDL-to-Java compiler (portable), version "3.1"
 * from Mensajeria.idl
 * Junes 26 de julio de 2004 23H52' CEST
 */

public interface EcoOperations
{
    String enviarTexto (String texto);
} // interface EcoOperations
```

La clase `EcoHelper` ya nos da pistas sobre lo que hace la compilación IDL. Esta clase sirve para crear flujos donde puedan ir mensajes IIOP, así como para introducir datos en los flujos de salida o extraerlos de los flujos de entrada. El programador no necesita usarla directamente, pero el IDL de Java sí.

```
package Mensajeria;

/*
 * Mensajeria/EcoHelper.java .
 * Generated by the IDL-to-Java compiler (portable), version "3.1"
 * from Mensajeria.idl
 * Junes 26 de julio de 2004 23H52' CEST
 */

abstract public class EcoHelper
{
    private static String _id = "IDL:Mensajeria/Eco:1.0";

    public static void insert (org.omg.CORBA.Any a, Mensajeria.Eco that)
    {
        org.omg.CORBA.portable.OutputStream out = a.create_output_stream ();
        a.type (type ());
```

```
        write (out, that);
        a.read_value (out.create_input_stream (), type ());
    }

    public static Mensajeria.Eco extract (org.omg.CORBA.Any a)
    {
        return read (a.create_input_stream ());
    }

    private static org.omg.CORBA.TypeCode __typeCode = null;
    synchronized public static org.omg.CORBA.TypeCode type ()
    {
        if (__typeCode == null)
        {
            __typeCode = org.omg.CORBA.ORB.init ().create_interface_tc (Mensajeria.EcoHelper.id (),
"Eco");
        }
        return __typeCode;
    }

    public static String id ()
    {
        return _id;
    }

    public static Mensajeria.Eco read (org.omg.CORBA.portable.InputStream istream)
    {
        return narrow (istream.read_Object (_EcoStub.class));
    }

    public static void write (org.omg.CORBA.portable.OutputStream ostream, Mensajeria.Eco
value)
    {
        ostream.write_Object ((org.omg.CORBA.Object) value);
    }

    public static Mensajeria.Eco narrow (org.omg.CORBA.Object obj)
    {
        if (obj == null)
            return null;
        else if (obj instanceof Mensajeria.Eco)
            return (Mensajeria.Eco)obj;
        else if (!obj._is_a (id ()))
            throw new org.omg.CORBA.BAD_PARAM ();
        else
        {
            org.omg.CORBA.portable.Delegate delegate =
((org.omg.CORBA.portable.ObjectImpl)obj)._get_delegate ();
            Mensajeria._EcoStub stub = new Mensajeria._EcoStub ();
            stub._set_delegate(delegate);
            return stub;
        }
    }

}
```

Devuelve el TypeCode
(tipo de dato CORBA) de
un objeto Eco

Lee un objeto Eco de un
flujo CORBA de entrada

Escribe un objeto
Eco en un flujo
CORBA de entrada

Convierte referencias
genéricas a objetos
CORBA en referencias a
objetos concretos

El método `public static void insert (org.omg.CORBA.Any a, Mensajeria.Eco that)` de la clase `EcoHelper` permite insertar valores en un objeto `Any`. En este caso,

```
EcoHelper.insert(any, eco)
```

introduciría en un objeto `Any` un objeto de tipo `Eco`.

Si se deseara extraer el objeto de tipo `Eco` del objeto `Any` se usaría el método `public static Mensajeria.Eco extract (org.omg.CORBA.Any a)`:

```
EcoHelper.extract (any)
```

La clase `_EcoStub.java` también es interesante (su comportamiento es similar al de los adaptadores de RMI):

```
package Mensajeria;
```

```
/*
 * Mensajeria/_EcoStub.java .
 * Generated by the IDL-to-Java compiler (portable), version "3.1"
 * from Mensajeria.idl
 * Junes 26 de julio de 2004 23H52' CEST
 */

public class _EcoStub extends org.omg.CORBA.portable.ObjectImpl implements Mensajeria.Eco
{
    public String enviarTexto (String texto)
    {
        org.omg.CORBA.portable.InputStream $in = null;
        try {
            org.omg.CORBA.portable.OutputStream $out = _request ("enviarTexto", true);
            $out.write_string (texto);
            $in = _invoke ($out);
            String $result = $in.read_string ();
            return $result;
        } catch (org.omg.CORBA.portable.ApplicationException $ex) {
            $in = $ex.getInputStream ();
            String _id = $ex.getId ();
            throw new org.omg.CORBA.MARSHAL (_id);
        } catch (org.omg.CORBA.portable.RemarshalException $rm) {
            return enviarTexto (texto);
        } finally {
            _releaseReply ($in);
        }
    } // enviarTexto

    // Type-specific CORBA::Object operations
    private static String[] __ids = {
```

```
"IDL:Mensajeria/Eco:1.0";  
  
public String[] _ids ()  
{  
    return (String[])__ids.clone ();  
}  
  
private void readObject (java.io.ObjectInputStream s) throws java.io.IOException  
{  
    String str = s.readUTF ();  
    String[] args = null;  
    java.util.Properties props = null;  
    org.omg.CORBA.Object obj = org.omg.CORBA.ORB.init (args, props).string_to_object (str);  
    org.omg.CORBA.portable.Delegate delegate = ((org.omg.CORBA.portable.ObjectImpl)  
obj)._get_delegate ();  
    _set_delegate (delegate);  
}  
  
private void writeObject (java.io.ObjectOutputStream s) throws java.io.IOException  
{  
    String[] args = null;  
    java.util.Properties props = null;  
    String str = org.omg.CORBA.ORB.init (args, props).object_to_string (this);  
    s.writeUTF (str);  
}  
} // class _EcoStub
```

La existencia de las clases `Holder` es imprescindible para que Java pueda manejar argumentos IDL del tipo `out` o `inout`, incluidos en el IDL del OMG. La sintaxis de Java no admite argumentos de esos tipos (Java pasa por valor los tipos primitivos y los objetos) y se hace necesario simularlos mediante clases `Holder`. Una instancia de estas clases puede ser pasada como argumento a un método Java, y el método puede modificar el atributo `value` de la instancia.

Un ejemplo clarificará la situación. Supongamos una interfaz IDL de este estilo:

```
// Código IDL  
Interface Calculadora {  
  
    double sumar (in double a, inout double b);  
  
};
```

Una implementación en Java del estilo

```
// Código Java  
double sumar (double a, double b) {  
    b = a + b;  
    return (b);  
}
```

implicaría la pérdida del significado de la operación (el argumento `b` no retendrá el valor de la suma cuando termine la llamada). Para mantener la semántica de la interfaz IDL original, se usaría una clase `Holder`:

```
// Código Java
Calculadora calc = new Calculadora();
double a = 2.0;
// Java proporciona clases Holder para todos los tipos
// primitivos (double, int, etc.)
DoubleHolder b = new DoubleHolder(3.0);
calc.sumar(a, b);
// Se imprimirá 5.0
System.out.println(b.value);
```

En este ejemplo, la variable `b` conserva el valor que tomó en el cuerpo del método y se puede recuperar con `value`. Java proporciona clases `Holder` para todos los tipos primitivos (`double`, `int`, etc.). En C++ no se necesitan clases `Holder`, pues el lenguaje admite directamente el paso por referencia de objetos.

Paso c) El siguiente paso consiste en escribir y compilar un archivo que contenga dos clases: la clase sirviente y la servidora (también se puede escribir cada clase en un archivo). La primera es `EcoImpl`, la cual implementa la interfaz IDL `Eco` y es una subclase de `EcoImpl`. Cada objeto CORBA `Eco` será implementado por una instancia de `EcoImpl`. La segunda clase es `ServidorEco`, que

- crea e inicia un objeto ORB;
- obtiene una referencia al POA raíz y activa el gestor POA (`POAManager`);
- crea un sirviente y lo asocia al ORB;
- extrae una referencia asociada al sirviente y la transforma en un objeto del tipo apropiado;
- obtiene el contexto inicial de nombrado y registra el sirviente en el servicio de nombres, con el nombre “miEco”;
- espera las llamadas de los clientes al nuevo objeto.

Ejemplo 19b: ServidorEco.java

```
import Mensajeria.*; // Dentro del paquete FechaHora están los stubs.
import org.omg.CosNaming.*;// Paquete que implementa el servicio de nombres CORBA.
import org.omg.CosNaming.NamingContextPackage.*; // Paquete que contiene excepciones
// para el servicio de nombres CORBA.
import org.omg.CORBA.*; // Paquete con clases necesarias para todas las aplicaciones CORBA.
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;
```

```
import java.util.*;  
  
// IMPORTANTE: SÓLO FUNCIONARÁ CON J2SE 1.4 O SUPERIOR  
  
// Clase que implementa la interfaz Eco.  
class Ecolmp extends EcoPOA {  
  
    private ORB orb;  
  
    public void setORB(ORB orb) {  
        this.orb = orb;  
    }  
  
    public String enviarTexto (String texto) {  
        return (">Ha enviado al servidor CORBA el texto: ** " + texto + " ** en la fecha " +  
               new Date().toString() + " (hora del servidor)");  
    }  
  
}  
  
public class ServidorEco {  
  
    public static void main (String args[]) {  
        try {  
            // Se crea y se inicia el ORB.  
            ORB orb = ORB.init (args, null);  
  
            // Se obtiene una referencia a la raíz de POA y se lanza el gesto del POA  
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));  
            rootpoa.the_POAManager().activate();  
  
            // Se crea un objeto Ecolmp y se registra en el ORB.  
            Ecolmp ei = new Ecolmp();  
            ei.setORB(orb);  
  
            // Se obtiene una referencia CORBA de ei.  
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(ei);  
            Eco eco = EcoHelper.narrow(ref);  
  
            // Se registra el objeto en el servicio de nombres de CORBA y se le asigna un nombre.  
            // Al estar registrado, podrá recibir peticiones de los clientes.  
            org.omg.CORBA.Object refObj = orb.resolve_initial_references("NameService");  
            NamingContextExt refnc = NamingContextExtHelper.narrow(refObj);  
            NameComponent camino[] = refnc.to_name("miEco");  
            refnc.rebind(camino, eco);  
  
            System.out.println("Servicio CORBA de eco activo.");  
  
            //El servidor CORBA queda a la espera de peticiones de los clientes.  
            orb.run();  
        }  
        catch (Exception e) {  
            System.out.println("Error fatal.");  
            e.printStackTrace ();  
        }  
    }  
}
```

}

Paso d) El siguiente paso consiste en escribir y compilar el cliente para el servicio CORBA de eco.

Ejemplo 19c: ClienteEco.java

```
import Mensajeria.*; // Dentro del paquete FechaHora están los stubs.
import org.omg.CORBA.*; // Paquete con clases necesarias para todas las aplicaciones CORBA.
import org.omg.CosNaming.*; // Paquete que implementa el servicio de nombres CORBA.
import org.omg.CosNaming.NamingContextPackage.*;

import java.io.*; // Paquete de E/S.

// IMPORTANTE: SÓLO FUNCIONARÁ CON J2SE 1.4 O SUPERIOR

public class ClienteEco {

    public static void main(String args[]) {
        try {
            // Se crea y se inicializa el ORB.
            ORB orb = ORB.init(args, null);

            // Se busca el objeto miEco en el servicio de nombres de CORBA.
            org.omg.CORBA.Object refObj = orb.resolve_initial_references("NameService");
            NamingContextExt refnc = NamingContextExtHelper.narrow(refObj);
            Eco ecolimpl = EcoHelper.narrow(refnc.resolve_str("miEco"));

            // Se lee la entrada y se envía al servidor.
            BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
            System.out.println(">Comience a escribir, por favor.");
            while (true) {
                String linea = br.readLine();
                System.out.println(ecolimpl.enviarTexto(linea));
            }
        } catch (Exception e) {
            System.out.println("No se pudo contactar con el servicio de eco.");
            e.printStackTrace();
        }
    }
}
```

Paso e) Tras compilar los archivos ServidorEco.java y ClienteEco.java (lo que compilará también todos los archivos dentro del directorio Mensajeria), el siguiente paso consiste en ejecutar el servicio de nombres transitorios del IDL de Java.

En Windows, se arranca este servicio mediante

```
start tameserv
```

En UNIX se usa tnameserv &.

Si se desea arrancar este servicio para que escuche por un puerto TCP que no sea el estándar (900), hay que usar

```
start tnameserv -ORBInitialPort numpuerto (Windows)
```

o

```
tnameserv numpuerto& (UNIX)
```

Paso e) Se ejecuta la clase servidora (ServidorEco).

Paso f) Se ejecuta la clase cliente (ClienteEco). Si todos los archivos están en una misma máquina y el servicio de nombres se ejecuta en ella, bastará usar

```
java ClienteEco
```

Si se ha usado un puerto TCP diferente del estándar (900) para arrancar tnameserv, deberá usarse

```
java ClienteEco -ORBInitialPort numpuerto
```

En mi sistema, los pasos anteriores se representan en las siguientes capturas de pantalla.

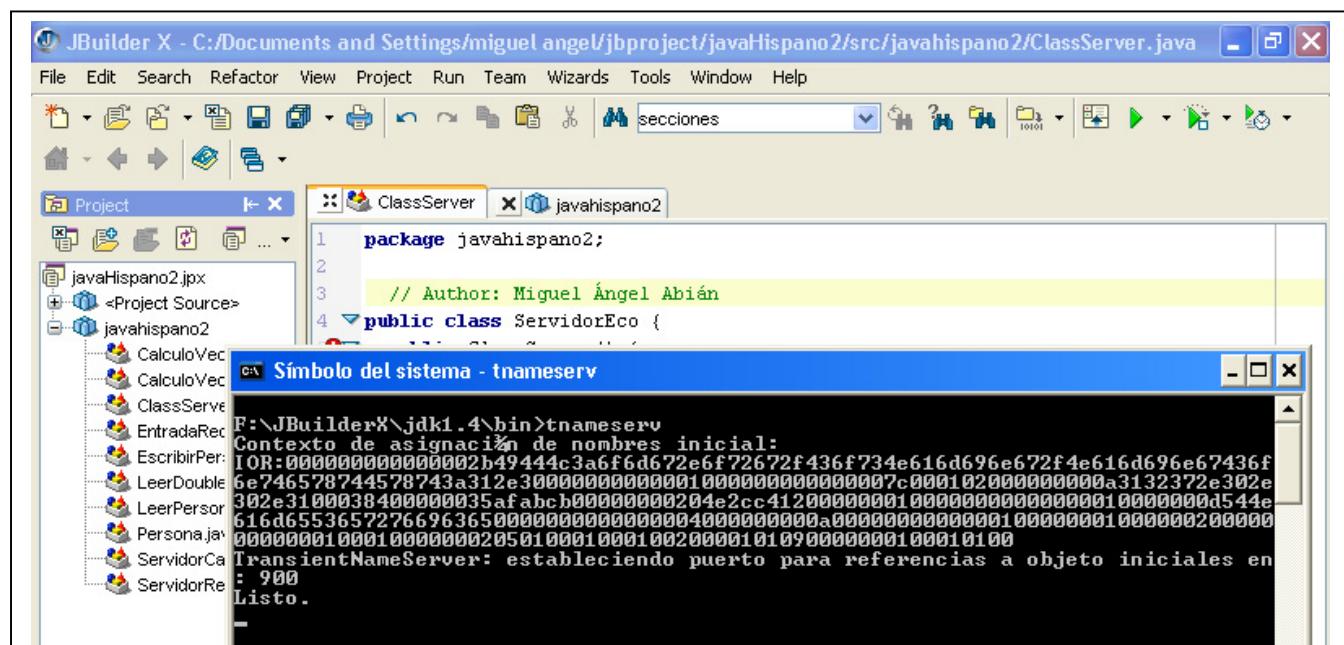


Figura 132. Captura de pantalla del lanzamiento del servicio de nombres

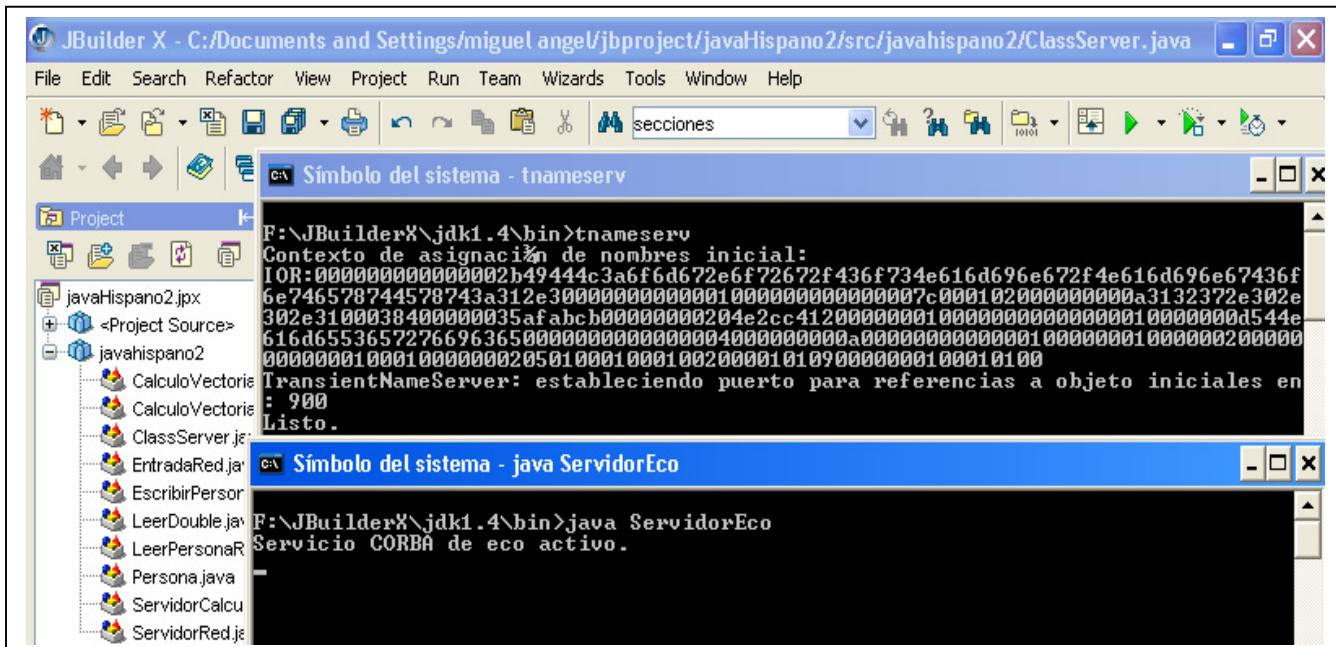
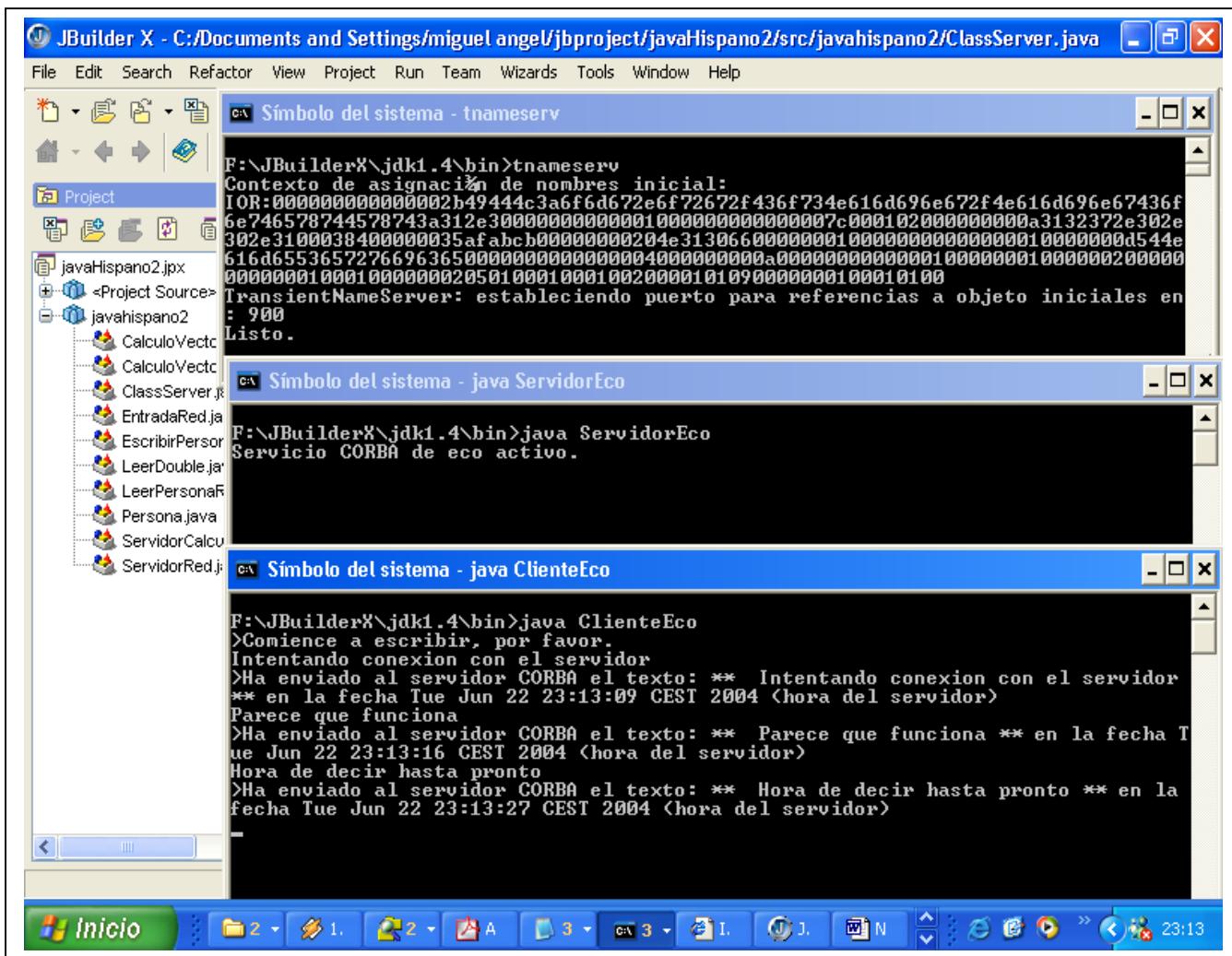
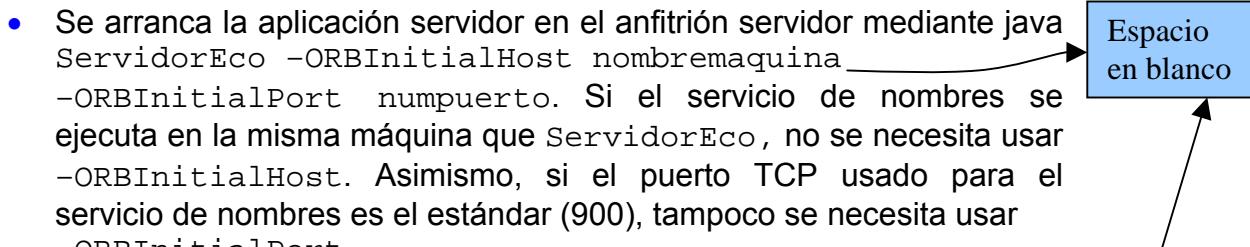


Figura 133. Captura de pantalla de la ejecución de ServidorEco



Para distribuir y ejecutar de modo general la aplicación de eco, el proceso consiste en

- Se compila el archivo Mensajeria.idl en el anfitrión servidor con idlj -fserver Mensajeria.idl. Se generará un directorio Mensajeria con los archivos que necesita el servidor: Eco.java, EcoPOA.java y EcoOperations.java.
- Se compila el archivo Mensajeria.idl en el anfitrión cliente con idlj -fclient Mensajeria.idl. Se generará un directorio Mensajeria con los archivos que necesita el cliente: _EcoStub.java, Eco.java, EcoHelper.java, EcoHolder.java y EcoOperations.java.
- Como el código del servidor usa la clase EcoHelper, habrá que dejar dentro del directorio Mensajeria del anfitrión servidor los archivos EcoHelper.java y _EcoStub.java (la clase Helper hace uso de la Stub).
- Se compilán en el anfitrión servidor todos los archivos dentro del directorio Mensajeria.
- Se compilán en el anfitrión cliente todos los archivos dentro de Mensajeria.
- Se compila el archivo ServidorEco.java en el anfitrión servidor.
- Se compila el archivo ClienteEco.java en el anfitrión cliente.
- Se ejecuta el servicio de nombres en un anfitrión (puede ser distinto de los dos anteriores).
- Se arranca la aplicación servidor en el anfitrión servidor mediante java ServidorEco -ORBInitialHost nombremaquina -ORBInitialPort numpuerto. Si el servicio de nombres se ejecuta en la misma máquina que ServidorEco, no se necesita usar -ORBInitialHost. Asimismo, si el puerto TCP usado para el servicio de nombres es el estándar (900), tampoco se necesita usar -ORBInitialPort.

- Se arranca la aplicación cliente en el anfitrión cliente mediante java ClienteEco -ORBInitialHost nombremaquina -ORBInitialPort numpuerto. Si el servicio de nombres se ejecuta en la misma máquina que ClienteEco, no se necesita usar -ORBInitialHost. Asimismo, si el puerto TCP usado para el servicio de nombres es el estándar (900), tampoco se necesita usar -ORBInitialPort.

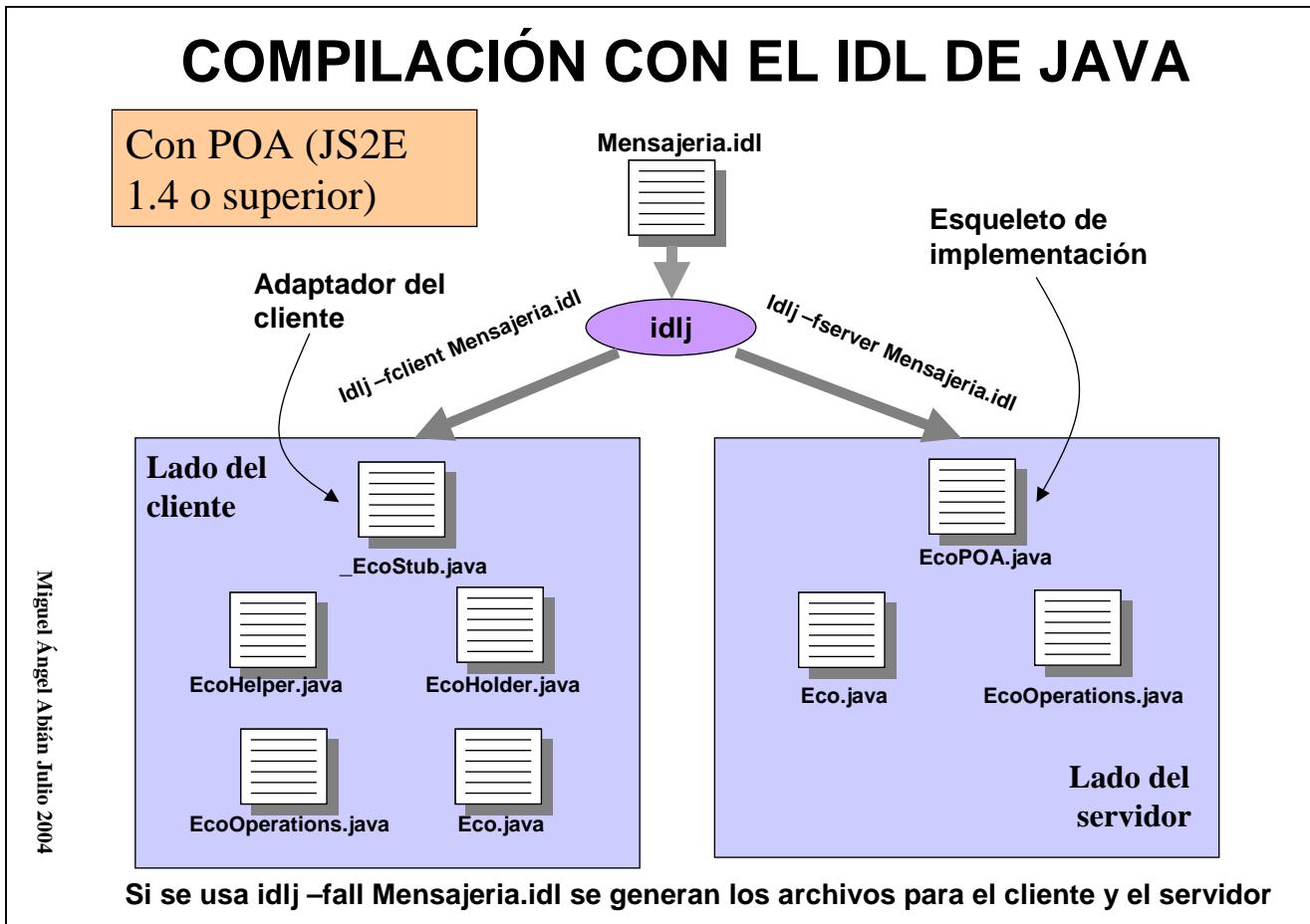


Figura 135. Compilación IDL de la aplicación CORBA del ejemplo 19. La figura tiene validez general para las aplicaciones CORBA escritas en Java

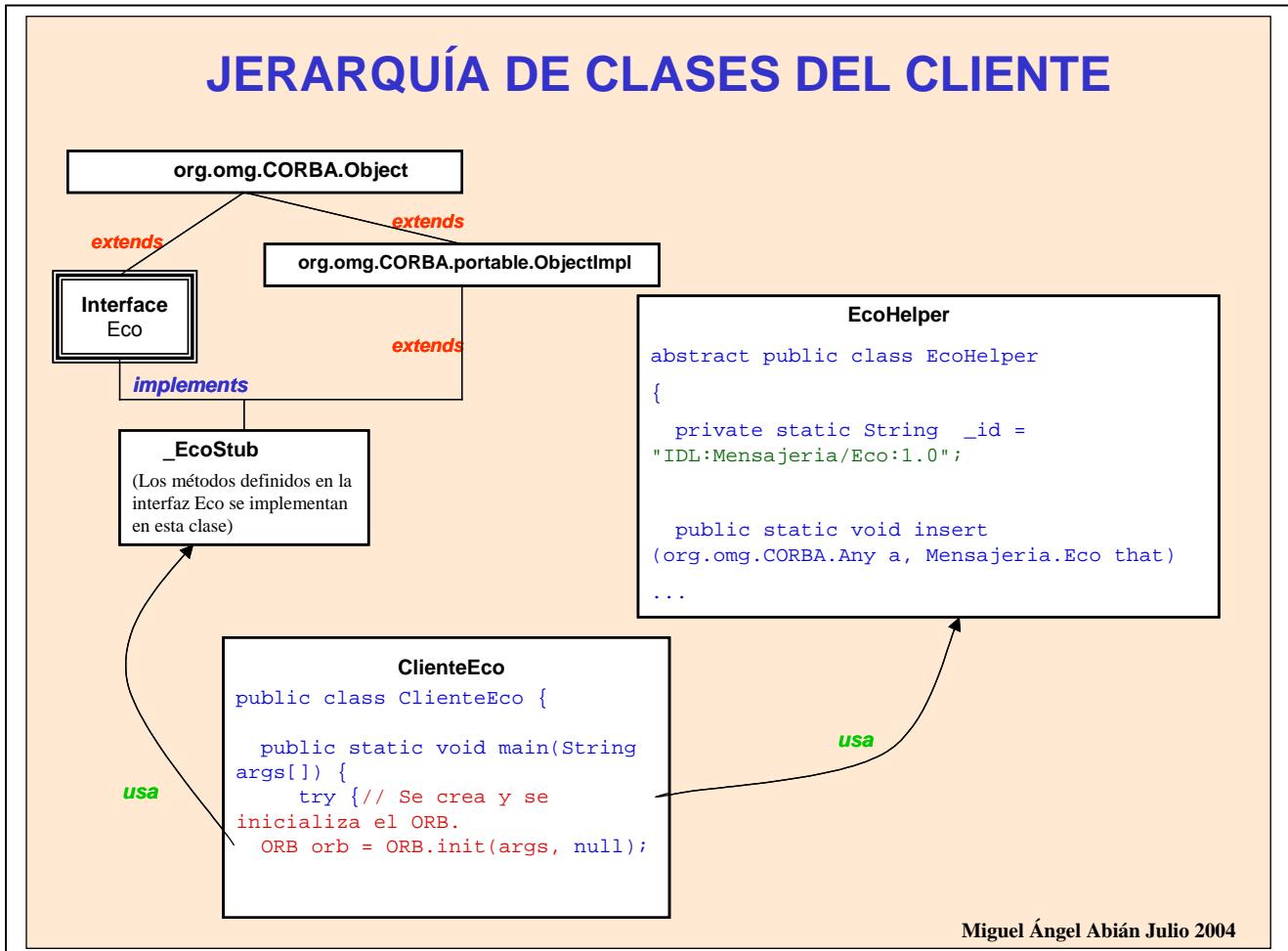


Figura 136. Jerarquía de clases en la parte del cliente del ejemplo 19.

Si el lector no tiene ningún inconveniente en tener algún archivo de más en el cliente y en el servidor, los pasos anteriores se pueden simplificar en éstos:

- Se compila el archivo Mensajeria.idl con idlj -fall Mensajeria.idl. Se generará un directorio Mensajeria con los archivos Eco.java, EcoPOA.java y EcoOperations.java, _EcoStub.java, EcoHelper.java y EcoHolder.java.
- Se compilan los archivos anteriores.
- Los archivos .class generados en el paso anterior se copian en el anfitrión del cliente y del servidor, dentro de sendos directorios llamados Mensajeria.
- Se compila el archivo ServidorEco.java en el anfitrión servidor.
- Se compila el archivo ClienteEco.java en el anfitrión cliente.
- Se ejecuta el servicio de nombres en un anfitrión (puede ser distinto de los dos anteriores).
- Se arranca la aplicación servidor en el anfitrión servidor mediante java ServidorEco -ORBInitialHost nombremaquina

Espacio en blanco

Espacio en blanco

- Se arranca la aplicación cliente en el anfitrión cliente mediante `java ClienteEco -ORBInitialHost nombremaquina -ORBInitialPort numpuerto.`
- Espacio en blanco

Como segundo ejemplo del uso de CORBA con Java incluyo aquí el ejemplo de una aplicación de *chat*. Es una versión muy simplificada de una aplicación que construí hace unos años para un club de deportes de nieve. En esta versión uso POA y el **modelo de delegación** (también conocido en la documentación de Sun como *Tie Model*: modelo de ligadura).

En el ejemplo anterior, hemos usado implícitamente el **modelo de herencia**, en el cual se implementa la interfaz IDL mediante una clase sirviente (`EcoImp`) que también extiende al esqueleto generado por el compilador IDL para Java. Este modelo tiene el inconveniente de que, como Java no admite herencia múltiple, no se puede heredar del esqueleto y de una clase sirviente. El modelo de delegación permite heredar de una clase sirviente, posibilidad inexistente en el otro modelo.

Al trabajar con el modelo de delegación y con POA, hay que ejecutar dos veces al compilador `idlj` (al usar `-fall`, considero que quiero generar los archivos para el servidor y el cliente):

```
idlj -fall interfazIDL.idl  
idlj -fallTie interfazIDL.idl
```

La segunda ejecución de `idlj` generará un archivo de tipo `interfazIDLPOATie.java`.

Si todos los archivos están en el fichero *bin* del JDK y todos los procesos se ejecutan en la máquina local, los pasos para ejecutar la aplicación de *chat* son similares a los de la aplicación de *eco*. La única diferencia reside en que, en el paso b), `serviciochat.idl` debe compilarse así:

```
idlj -fall serviciochat.idl  
idlj -fallTie serviciochat.idl
```

JERARQUÍA DE CLASES EN EL LADO DEL SERVIDOR

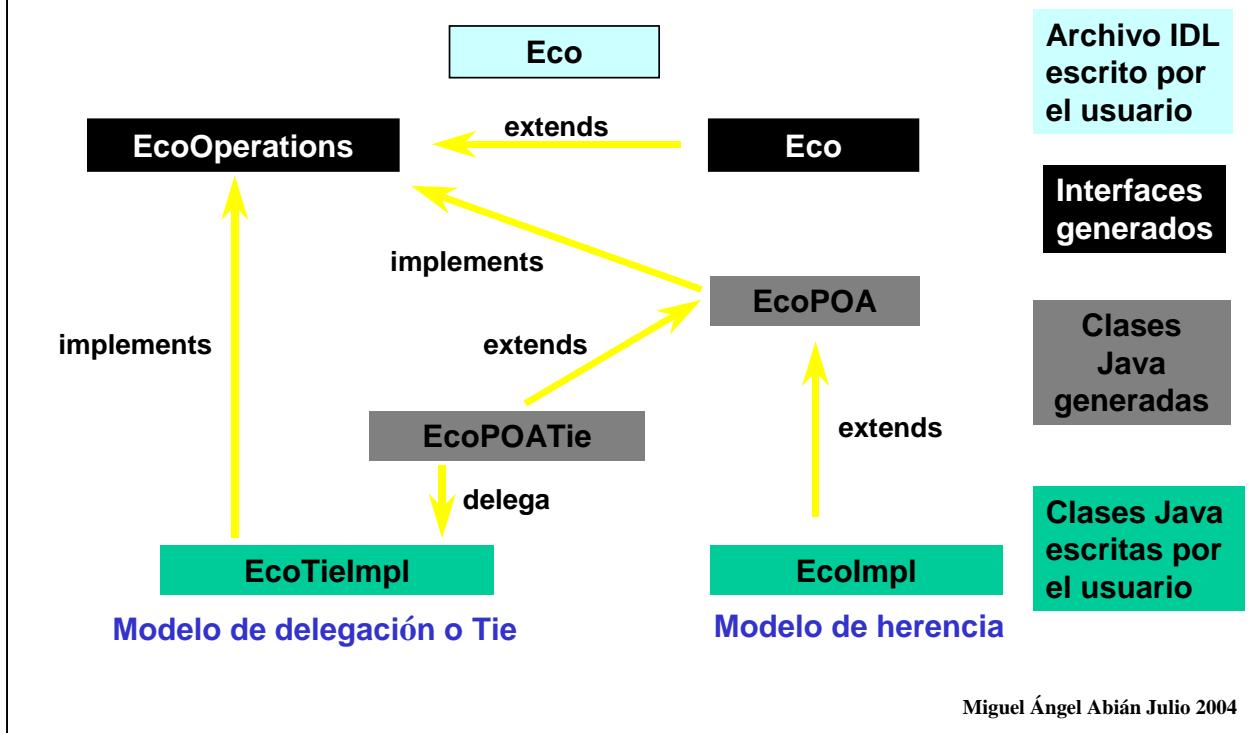


Figura 137. Jerarquía parcial de clases en el lado del servidor del ejemplo 19, con ambos modelos: el de herencia y el de delegación.

He escogido este ejemplo de un *chat* porque me permite ilustrar con código el funcionamiento de las **retrollamadas** (callbacks). Una retrollamada es una llamada del servidor a algún método implementado en un cliente remoto (quizás ubicado en otro anfitrión o en otro proceso dentro de la misma máquina). Como ya se dijo en 2.3, “cliente” y “servidor” son papeles en una comunicación, y pueden variar: por ejemplo, un objeto que se comporta como cliente para otro puede ser servidor para un tercero.

Con frecuencia, las retrollamadas se usan cuando los clientes necesitan acceder de forma periódica a alguna información de servidor o averiguar si se ha lanzado algún evento en éste. Imaginemos, por ejemplo, una aplicación informática encargada de controlar el aterrizaje y despegue de los aviones en un aeropuerto. Sería ineficaz que el módulo de control de la aplicación tuviera que enviar señales a cada avión, cada cierto período de tiempo, para saber dónde está cada uno. Si muchos aviones estuvieran inactivos, se perdería mucho tiempo de CPU comprobando posiciones inalteradas. Es mucho más lógico que, cada cierto tiempo, los aviones informen al módulo de control de los cambios en su posición (si ha lugar). Así se evita que los aviones inactivos consuman tiempo y recursos de la aplicación.

En una retrollamada, el objeto que actuaba como servidor pasa a actuar como cliente. Debe, por tanto, conseguir una referencia al objeto cliente y usarla para hacer sus peticiones. En CORBA hay dos maneras para obtener las referencias de los objetos: usar un servicio de nombres que el servidor pueda consultar o escribir el código del cliente de forma que se llame a un método del servidor pasándole como

argumento el propio cliente. En la segunda opción, no hay que olvidar que CORBA no pasa realmente objetos: pasa referencias a los objetos. En el ejemplo de la aplicación de *chat*, la referencia al objeto cliente se pasa en este fragmento de código:

```
try {
    servidor.registrarUsuario(tie._this());
}
catch (Exception e) {
    System.out.println("Error al intentar registrar el usuario");
    e.printStackTrace();
}
```

Envio al servidor de una referencia a sí mismo

La llamada del servidor al cliente se produce en

```
public synchronized void enviarATodos(ClienteChat usuario,
                                      String texto) {
    // Nombre por defecto de los usuarios no registrados: "Anónimo"
    String nombreUsuario = "";
    try {
        nombreUsuario = usuario.getIdentificador();
        if ( nombreUsuario.equals("") )
            nombreUsuario = "Anonimo";
        Iterator it = usuarios.iterator();
        ClienteChat tmp =null;
        while (it.hasNext()) {
            tmp = (ClienteChat) it.next();
            try {
                // No envía el mensaje a aquel usuario que lo ha enviado.
                if ( !(tmp.equals(usuario)) )
                    tmp.enviar(nombreUsuario + ":" + texto);
            }
            catch (Exception e1) {
                // Si hay problemas al enviar el texto a un
                // usuario se le borra de la lista de
                // usuarios activos.
                it.remove();
            } // fin del try-catch interno
        }
    }
    catch (Exception e2) {
        System.out.println("Error desconocido.");
        e2.printStackTrace();
    }
}
```

Retrollamada

Ejemplo 20a: serviciochat.idl

```
module serviciochat {  
  
    interface ClienteChat;  
  
    interface ServidorChat {  
  
        // El método registrarUsuario registra a un usuario en el servicio de chat.  
        void registrarUsuario(in ClienteChat ch);  
  
        // El método enviarATodos envía un texto a todos los usuarios en activo del chat.  
        void enviarATodos(in ClienteChat ch, in string texto);  
    };  
  
    interface ClienteChat {  
  
        // El método getIdentificador devuelve el identificador del cliente.  
        string getIdentificador();  
  
        // El método enviar envía un texto a un cliente (no a todos).  
        void enviar(in string texto);  
    };  
};
```

Ejemplo 20b: ServidorChatImpl.java

```
import serviciochat.*;  
import org.omg.CORBA.*;  
import org.omg.CosNaming.*;  
import org.omg.PortableServer.*;  
import org.omg.PortableServer.POA;  
  
import java.util.*;  
  
// IMPORTANTE: SÓLO FUNCIONARÁ CON J2SE 1.4 O SUPERIOR  
  
/*  
 *  
 * ServidorChatImpl es una implementación Java de serviciochat.idl.  
 * Debe ejecutarse tras activar tnameserv.  
 */  
  
public class ServidorChatImpl extends ServidorChatPOA {  
  
    private ORB orb;  
    private List usuarios = new ArrayList(); // Lista con los usuarios.  
  
    /*  
     * Crea un objeto ORB.  
     */
```

```
public void setORB(ORB orb) {
    this.orb = orb;
}

/*
 * Registra un usuario en el chat. Es necesario que esté sincronizado para
 * evitar situaciones como que dos usuarios se registren a la vez.
 */
public synchronized void registrarUsuario(ClienteChat usuario) {
    System.out.println("Bienvenido: " + usuario);
    usuarios.add(usuario);
}

/*
 * Envía el texto del argumento a todos los clientes en activo.
 *
 */
public synchronized void enviarATodos(ClienteChat usuario, String texto) {
    // Nombre por defecto de los usuarios no registrados: "Anónimo"
    String nombreUsuario = "";
    try {
        nombreUsuario = usuario.getIdentificador();
        if ( nombreUsuario.equals("") )
            nombreUsuario = "Anonimo";
        Iterator it = usuarios.iterator();
        ClienteChat tmp =null;
        while (it.hasNext()) {
            tmp = (ClienteChat) it.next();
            try { // No envía el mensaje a aquel usuario que lo ha enviado.
                if ( !(tmp.equals(usuario)) )
                    tmp.enviar(nombreUsuario + ": " + texto);
            }
            catch (Exception e1) {
                // Si hay problemas al enviar el texto a un usuario se le
                // borra de la lista de usuarios activos.
                it.remove();
            } // fin del try-catch interno.
        }
    }
    catch (Exception e2) {
        System.out.println("Error desconocido.");
        e2.printStackTrace();
    }
}

// Se crea y arranca el servidor, registrándolo en el servicio de nombres de CORBA.
// El argumento del main es la URL del servidor CORBA. Si no se especifica ninguna,
// se asume que es localhost (el equipo local).
public static void main(String args[]) {
    try {
        ORB orb = ORB.init(args, null);
        ServidorChatImpl sci = new ServidorChatImpl();
        ServidorChat ref = sci._this(orb);
        sci.setORB(orb);
        POA POAraiz = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
        POAManager poaMgr = POAraiz.the_POAManager();
```

```
        NamingContextExt nc =
            NamingContextExtHelper.narrow(orb.resolve_initial_references("NameService"));
        NameComponent [] nombreComponente = new NameComponent[1];
        nombreComponente[0] = new NameComponent("miChat", "");
        nc.bind(nombreComponente, ref);
        poaMgr.activate();
        System.out.println("Se ha arrancado el servidor CORBA de chat.");
        orb.run();
    }
    catch (Exception e) {
        System.out.println("Error al intentar arrancar el servidor.");
        e.printStackTrace();
    }
}
}
```

Ejemplo 20c: ClienteGraficoChat.java

```
import serviciochat.*;

import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.PortableServer.POA;
import org.omg.PortableServer.*;

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

// IMPORTANTE: SÓLO FUNCIONARÁ CON J2SE 1.4 O SUPERIOR

/*
 *
 * Cliente del chat con interfaz gráfica.
 * Debe ejecutarse tras ejecutar ServidorChatImpl.
 * ClienteChatOperations es generada automáticamente por idlj -fall.
 */

public class ClienteGraficoChat extends JFrame implements ClienteChatOperations {

    private ClienteChatPOATie tie = null;
    private static ORB orb = null;
    private ClienteChat cliente = null;
    private ServidorChat servidor = null;
    private String nombreUsuario = "Anonimo";

    // Parte gráfica.
    private JPanel panel1 = new JPanel();
    private JPanel panel2= new JPanel();
    private JTextArea areaTexto = new JTextArea("", 25, 35);
    private JTextField campoTexto = new JTextField(25);
```

```
private JButton enviarTexto = new JButton("Enviar");
private JButton registrar = new JButton("Registrarse");

// Constructor
public ClienteGraficoChat() {
    dibujar();
    try {
        tie = new ClienteChatPOATie(this);
        cliente = tie._this(orb);
        POA POAraiz = POAHelper.narrow(orb.resolve_initial_references("RootPOA"));
        POAManager poaMgr = POAraiz.the_POAManager();
        poaMgr.activate();
        NamingContextExt nc =
            NamingContextExtHelper.narrow(orb.resolve_initial_references("NameService"));
        servidor= ServidorChatHelper.narrow(nc.resolve(nc.to_name("miChat")));
    }
    catch(Exception e) {
        e.printStackTrace();
    }

    // Código de manejo de eventos de la interfaz gráfica. Se usan clases internas.
    registrar.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evento) {
            setIdificador(campoTexto.getText());
            campoTexto.setText("");
        }
    });

    enviarTexto.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evento) {
            try {
                servidor.enviarATodos(cliente, campoTexto.getText());
                campoTexto.setText("");
            }
            catch(Exception e) {
                System.out.println("Problema fatal al enviar texto a los usuarios.");
                e.printStackTrace();
            }
        }
    });
}

// Salida del chat.
addWindowListener( new WindowAdapter() {
    public void windowClosing(WindowEvent event) {
        System.exit(0);
    }
});

try {
    servidor.registrarUsuario(tie._this());
}
catch (Exception e) {
    System.out.println("Error al intentar registrar el usuario.");
    e.printStackTrace();
}
}
```

```
public synchronized void setIdentificador(String nombreUsuario) {
    this.nombreUsuario = nombreUsuario;
}

public String getIdentificador() {
    return nombreUsuario;
}

public void enviar(String texto) {
    areaTexto.append(texto + "\n");
}

public void dibujar() {
    setTitle(" Un chat con CORBA. Miguel Ángel Abián Julio 2004 ");
    Container contenedor = getContentPane();
    contenedor.setLayout(new BorderLayout());
    contenedor.add("South", panel1);
    contenedor.add("North", panel2);
    contenedor.add("Center", areaTexto);
    panel1.add(registrar);
    panel1.add(campoTexto);
    panel1.add(enviarTexto);
    areaTexto.setEditable(false);
    this.pack();
    show();
}

// Punto de entrada en el cliente gráfico del chat.
public static void main(String args[]) {
    orb = ORB.init(args, null);
    ClienteGraficoChat cliente = new ClienteGraficoChat();
}

}
```

La compilación IDL de la aplicación del chat se muestra en la figura siguiente.

COMPILACIÓN CON EL IDL DE JAVA

Con POA (JS2E 1.4 o superior) y el modelo de delegación

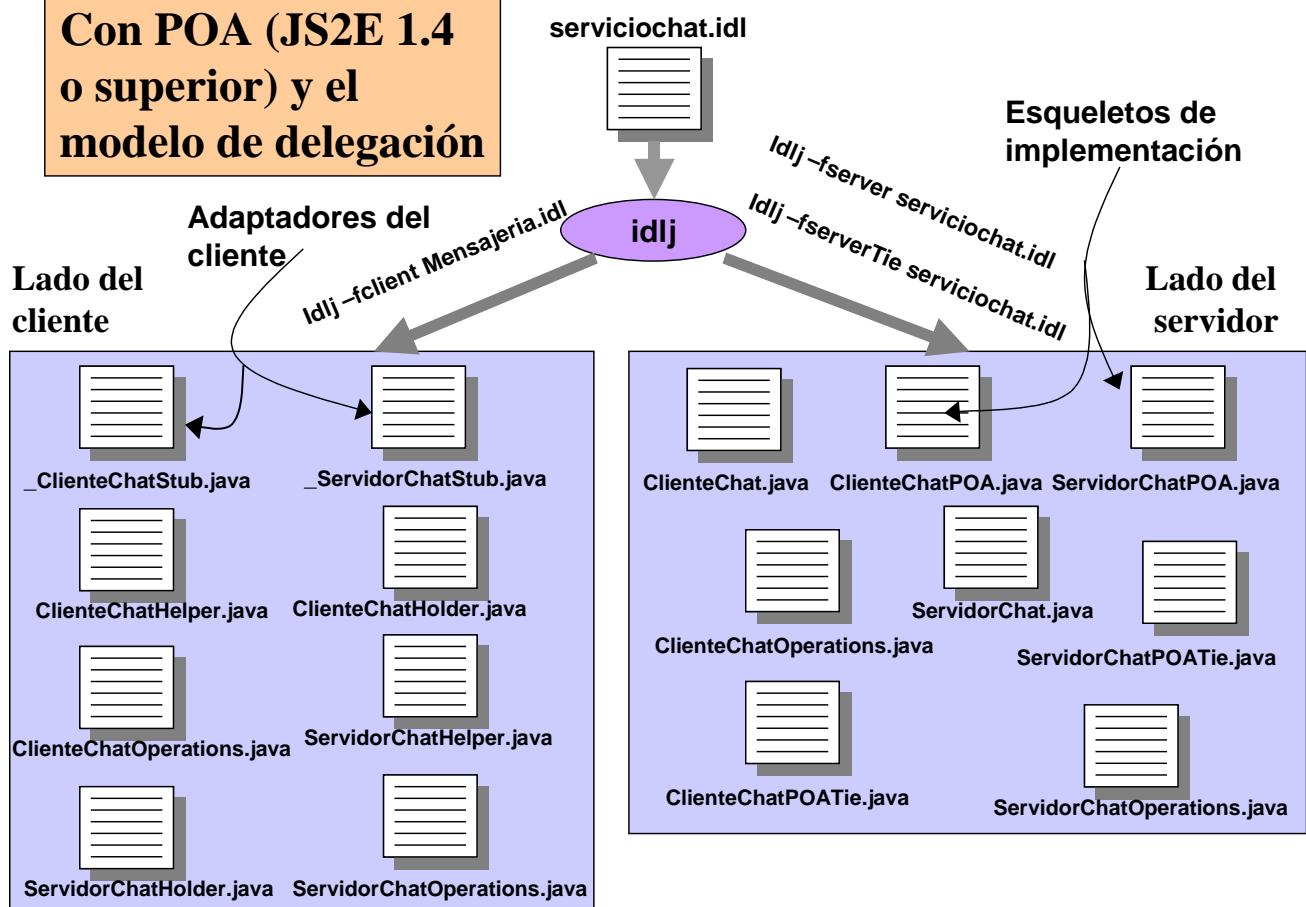


Figura 138. Compilación IDL del ejemplo 20. Distribución de la aplicación CORBA del ejemplo 20. Tiene validez general para las aplicaciones CORBA escritas en Java que usen POA y el modelo de delegación. Si se desea generar todos los archivos de una sola vez hay que ejecutar idlj –fall serviciochat.idl y, luego, idlj –fallTie serviciochat.idl

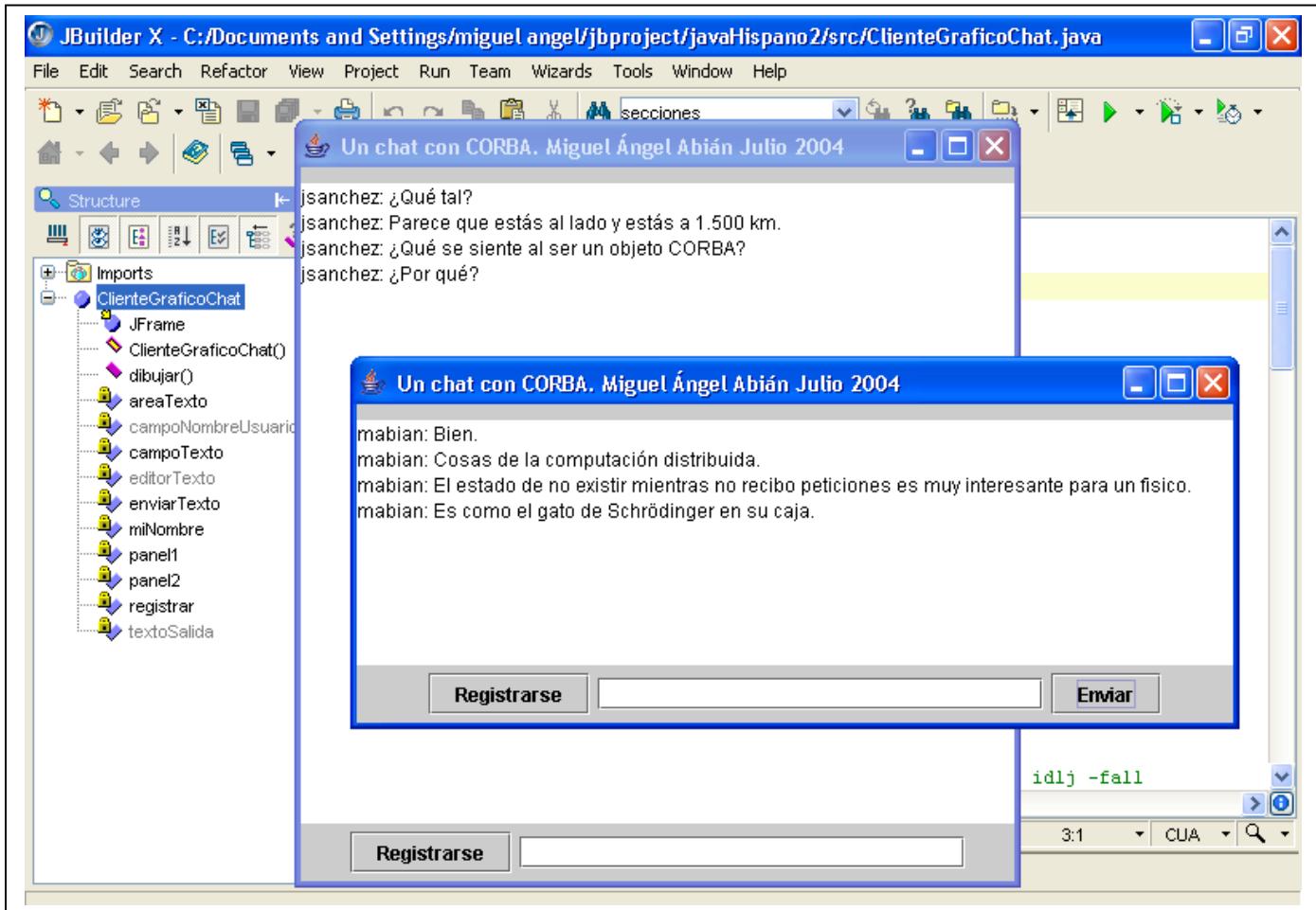


Figura 139. El chat hecho con Java y CORBA

En este caso, como hay retrollamadas (*callbacks*), los papeles de servidor y cliente cambian durante la ejecución de la aplicación de *chat*. Puede hacerse la compilación que muestra la figura 138 y luego mover los archivos que faltan para que el cliente se comporte como servidor, y viceversa; pero es mucho más cómodo compilar con las opciones `-fall` y `-fallTie`. En este caso, el proceso de distribución seguiría los siguientes pasos:

- Se compila el archivo `serviciochat.idl` con `idlj -fall serviciochat.idl` y, luego, con `idlj -fallTie serviciochat.idl`. Dentro del directorio `serviciochat` aparecerán los archivos `_ClienteChatStub.java`, `_ServidorChatStub.java`, `ClienteChatHolder.java`, `ClienteChatHelper.java`, `ServidorChatHolder.java`, `ServidorChatHelper.java`, `ClienteChat.java`, `ClienteChatPOA.java`, `ClienteChatOperations.java`, `ClienteChatPOATie.java`, `ServidorChat.java`, `ServidorChatPOA.java`, `ServidorChatOperations.java` y `ServidorChatPOATie.java`.
- Se compilan todos los archivos anteriores.

- Los archivos .class generados en el paso anterior se copian en el anfitrión del cliente y del servidor, dentro de sendos directorios llamados Mensajería.
- Se compila el archivo ServidorChatImpl.java en el anfitrión servidor.
- Se compila el archivo ClienteGraficoChat.java en el anfitrión cliente.
- Se ejecuta el servicio de nombres en un anfitrión (puede ser distinto a los dos anteriores).
- Se arranca la aplicación servidor en el anfitrión servidor mediante `java ServidorChatImpl -ORBInitialHost nombremaquina -ORBInitialPort numpuerto.`
Si el servicio de nombres se ejecuta en la misma máquina que ServidorChatImpl, no se necesita usar -ORBInitialHost. Asimismo, si el puerto TCP usado para el servicio de nombres es el estándar (900), tampoco se necesita usar -ORBInitialPort.Espacio en blanco
- Se arranca la aplicación cliente en el anfitrión cliente mediante `java ClienteGraficoChat -ORBInitialHost nombremaquina -ORBInitialPort numpuerto.`
Si el servicio de nombres se ejecuta en la misma máquina que ClienteGraficoChat, no se necesita usar -ORBInitialHost. Asimismo, si el puerto TCP usado para el servicio de nombres es el estándar (900), tampoco se necesita usar -ORBInitialPort.Espacio en blanco

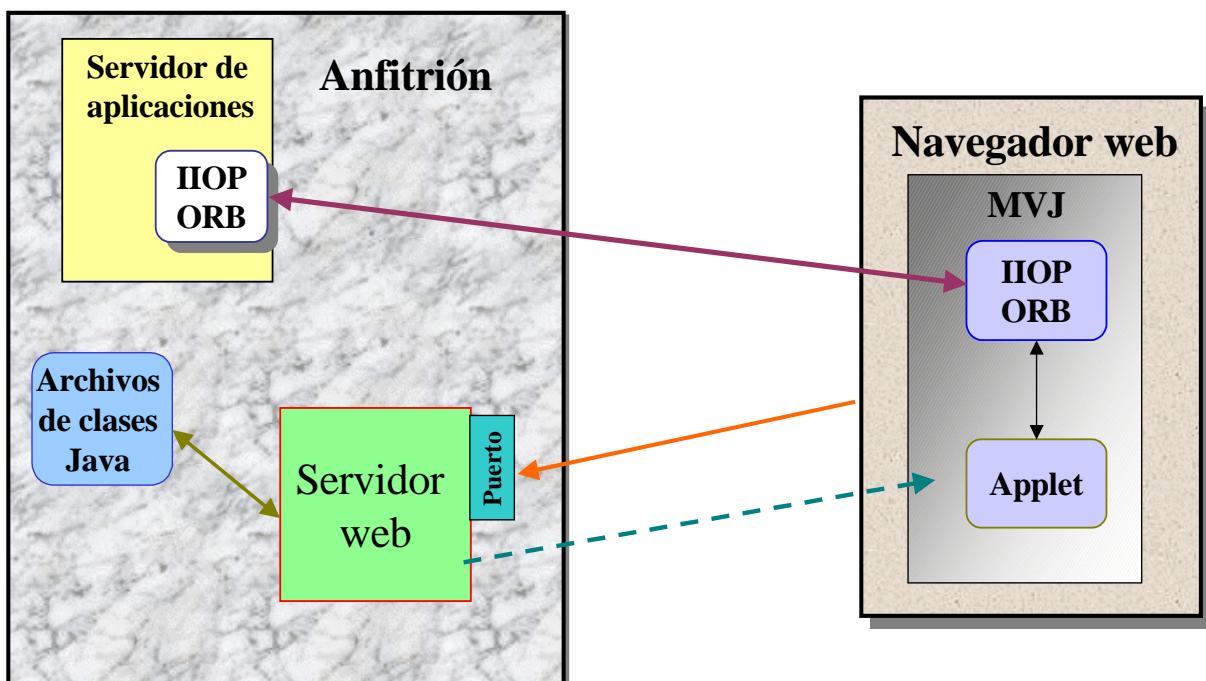
Tanto en este ejemplo como en el anterior, los pasos para distribuir las aplicaciones pueden simplificarse si se almacenan algunos archivos en un servidor HTTP o FTP y se descargan dinámicamente, tal como vimos en 5.5. Esta descarga dinámica sólo es posible en Java: si trabajamos con CORBA mediante lenguajes como Ada o C++, no hay manera de cargar dinámicamente los adaptadores o esqueletos.

La distribución más sencilla para los clientes consiste en que los clientes CORBA sean *applets*. Si se opta por esta solución, hay que colocar los archivos que necesiten los clientes en un servidor web, dentro de una página HTML con una etiqueta APPLET.

Los pasos que se seguirían para ejecutar un *applet* CORBA serían éstos:

- El usuario utiliza un navegador web compatible con Java para acceder a una página web con el *applet* CORBA.
- Se arranca el *applet*.
- El *applet* descarga mediante HTTP los bytecodes de todas las clases que necesita (correspondientes al cliente CORBA) y los ejecuta.
- Las peticiones de los usuarios son conducidas por el *applet* hasta el servidor CORBA.

DISTRIBUCIÓN DE UNA APLICACIÓN CORBA CON CLIENTES DE TIPO APPLET



Nota: el servidor web o HTTP podría estar en un segundo anfitrión

Miguel Ángel Abián Julio 2004

Figura 140. Representación de una aplicación CORBA con Java que usa clientes de tipo applet

En la figura de la página siguiente se muestra el código que hay que escribir en la página HTML con el *applet* y en el cliente CORBA si se opta por distribuir una aplicación CORBA con clientes de tipo *applet*.

En una aplicación de este tipo, lo habitual es usar *applets* firmados o configurar en el lado del cliente un nivel de seguridad para cada anfitrión del cual se puedan descargar archivos.

CÓDIGO PARA USAR APPLETS EN EL LADO DEL CLIENTE

Código para el applet en la página web

```
<APPLET CODE="ClienteCORBA.class" ARCHIVE="Clases.jar"
          WIDTH=400 HEIGHT=300>
  <PARAM NAME="referencia"
         VALUE="IOR:.....">
</APPLET>
```

Código para el cliente CORBA

```
ORB orb = ORB.init(this, new java.util.Properties());
String referencia = getParameter("referencia");
omg.org.CORBA.Object objeto = orb.string_to_object(referencia);
X x = XHelper.narrow(objeto);
```

Miguel Ángel Abián Julio 2004

Figura 141. Código para usar applets en el lado del cliente como medio de distribución de las aplicaciones CORBA escritas en Java

FIN DE LA PRIMERA PARTE

Copyright (c) 2004, Miguel Ángel Abián. Este documento puede ser distribuido sólo bajo los términos y condiciones de la licencia de Documentación de javaHispano v1.0 o posterior (la última versión se encuentra en <http://www.javahispano.org/licencias/>).

Nota biográfica del Autor: Miguel Ángel Abián es licenciado en Ciencias Físicas por la U. de Valencia y obtuvo la suficiencia investigadora dentro del Dpto. Física Aplicada de la U.V con una tesina acerca de relatividad general y electromagnetismo. Además, ha realizado diversos cursos de postgrado sobre bases de datos, lenguajes de programación Web, sistemas Unix, comercio electrónico, firma electrónica, UML y Java. Ha colaborado en diversos programas de investigación TIC relacionados con el estudio de fibras ópticas y cristales fotónicos, ha obtenido becas de investigación del IMPIVA y de la Universidad Politécnica de Valencia y ha publicado diversos artículos en el *IEEE Transactions on Microwave Theory and Techniques* relacionados con el análisis de guías de onda inhomogéneas y guías de onda elípticas.

En el ámbito laboral ha trabajado como gestor de carteras y asesor fiscal para una agencia de bolsa y actualmente trabaja en el Laboratorio del Mueble Acabado de AIDIMA (Instituto Tecnológico del Mueble y Afines), ubicado en Paterna (Valencia), en tareas de normalización y certificación, traducción e interpretación y asesoramiento técnico. En dicho centro se están desarrollando proyectos europeos de comercio electrónico B2B para la industria del mueble basados en Java y XML (más información en www.aidima.es). Ha impartido formación en calidad, normalización y programación para ELKEDE (Grecia), CETEBA (Brasil) y CETIBA (Túnez), entre otros.

Últimamente, aparte de asesorar técnica y financieramente a diversas empresas de la Comunidad Valenciana, es investigador en los proyectos **INTEROP** y **ATHENA** del Sexto Programa Marco de la Comisión Europea, que pretenden marcar las pautas para tecnologías de la información en la próxima década y asegurar el liderazgo de Europa en las tecnologías de la sociedad del conocimiento. Ambos proyectos tienen como fin la interoperabilidad del software (estudian tecnologías como J2EE, .Net y CORBA, servicios web, tecnologías orientadas a aspectos, ontologías, etc.), y en ellos participan empresas como IBM U.K., COMPUTAS, SIEMENS, FIAT, TXT, GRAISOFT, SAP, EADS, además de numerosas universidades europeas y centros de investigación en ingeniería del software.

Sus intereses actuales son el diseño asistido por ordenador de guías de ondas y cristales fotónicos, la evolución de la programación orientada a objetos, Java, UEML, el intercambio electrónico de datos, el surrealismo y París, siempre París.