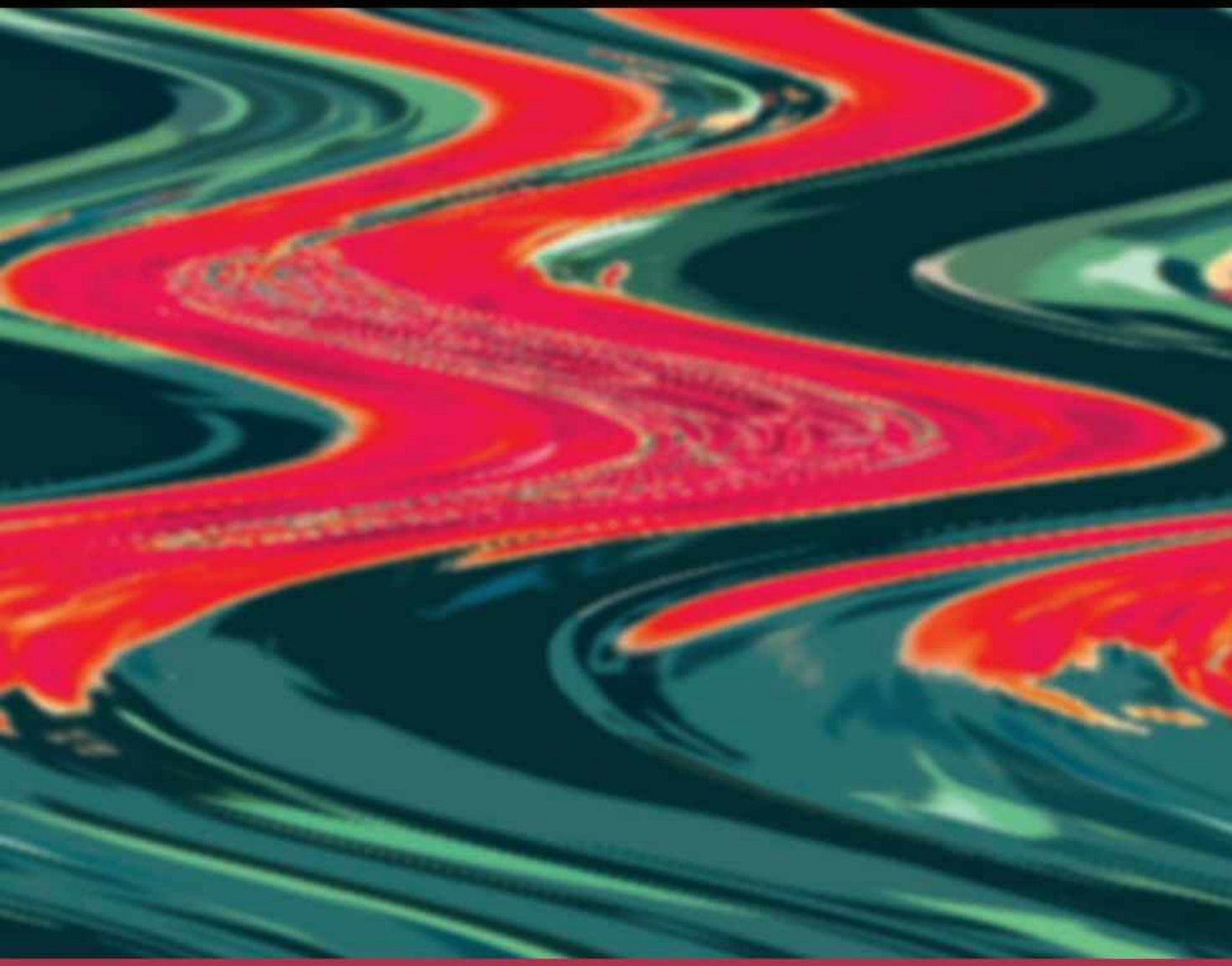
Computación Distribuida

Fundamentos y Aplicaciones



PEARSON
Addison Wesley

M. L. Liu

COMPUTACIÓN DISTRIBUIDA.
FUNDAMENTOS Y APLICACIONES

COMPUTACIÓN DISTRIBUIDA.
FUNDAMENTOS Y APLICACIONES

M. L. Liu
California Polytechnic State University, San Luis Obispo

Traducción:
José María Peña Sánchez
Fernando Pérez Costoya
María de los Santos Pérez Hernández
Víctor Robles Forcada
Facultad de Informática
Universidad Politécnica de Madrid

Madrid • México • Santafé de Bogotá • Buenos Aires • Caracas • Lima • Montevideo
San Juan • San José • Santiago • Sao Paulo • Reading, Massachussets • Harlow, England

Datos de catalogación bibliográfica

LIU, M. L.

COMPUTACIÓN DISTRIBUIDA. FUNDAMENTOS Y APLICACIONES

PEARSON EDUCACIÓN, S.A. Madrid, 2004

ISBN: 84-7829-066-4
Materia: Informática 681.3

Formato 195 x 250 Páginas: 424

COMPUTACIÓN DISTRIBUIDA. FUNDAMENTOS Y APLICACIONES
LIU, M. L.

ISBN: 84-7829-066-4
Depósito Legal: M-

PEARSON ADDISON-WESLEY es un sello editorial autorizado de PEARSON EDUCACIÓN, S.A.

Authorized translation from the English language edition, entitled *DISTRIBUTED COMPUTING: PRINCIPLES AND APPLICATIONS*, 1st Edition by LIU, M. L., published by Pearson Education, Inc, publishing as Addison-Wesley.

ISBN: 0-201-79644-9

Equipo editorial:
Editor: David Fayerman Aragón
Técnico editorial: Ana Isabel García Borro
Equipo de producción:
Director: José Antonio Clares
Técnico: José Antonio Hernán
Diseño de cubierta: Equipo de diseño de Pearson Educación, S.A.
Composición: COMPOMAR, S.L.
Impreso por:
IMPRESO EN ESPAÑA - PRINTED IN SPAIN

Este libro ha sido impreso con papel y tintas ecológicos

Índice de contenido

Prefacio

En el modelo de curriculum de computación del año 2001 (*Computing Curricula 2001 http://www.computer.org/education/cc2001/report/index.html*) desarrollado por *Joint IEEE Computer Society/ACM Task Force*, la computación en red se incluye como un área clave de las ciencias de la computación:

> *Los últimos avances en las redes de computación y telecomunicaciones, particularmente en aquellas basadas en TCP/IP, han incrementado la importancia de las tecnologías de red en las disciplinas de computación. La computación en red cubre una serie de subespecialidades, incluyendo protocolos y conceptos de redes de comunicación de computadores, sistemas multimedia, estándares y tecnologías web, seguridad en redes, computación sin cables y móvil y sistemas distribuidos.*
>
> *El dominio de esta área de trabajo necesita de teoría y de práctica. Las técnicas de aprendizaje que supongan la realización de experimentos y análisis propios, son muy recomendadas, ya que refuerzan la comprensión de los conceptos de los estudiantes y su aplicación a los problemas de la vida real. Las prácticas de laboratorio deberían incluir la recolección y síntesis de datos, modelado empírico, análisis de protocolos a nivel de código fuente, monitorización de paquetes de red, construcción de software y evaluación de modelos de diseño alternativos. Todos estos son conceptos importantes que pueden ser mejor comprendidos en las prácticas de laboratorio.*

El modelo de curriculum de ACM enumera una serie de temas dentro de esta área, especificando un mínimo de 15 horas de temas troncales y temas de libre elección adicionales. La mayor parte de estos temas se cubren en una serie de cursos de computación distribuida que inicié e impartí en *California Polytechnic State University* (*Cal Poly*), San Luis Obispo, desde 1996. Para estos cursos utilicé material de varias publicaciones, así como material propio, incluyendo transparencias, códigos de ejemplo y tareas de laboratorio, problemas e investigación, que proporcionaba como un curso completo a mis estudiantes.

Este libro de texto es una síntesis de los materiales del curso que he acumulado a lo largo de seis años y ha sido diseñado para impartir temas técnicos optativos a estudiantes universitarios.

Motivaciones del libro

Tradicionalmente, los cursos de computación distribuida se ofertan a estudiantes de postgrado. Sin embargo, con el crecimiento de Internet y de las aplicaciones para *intranets*, cada vez más estudiantes universitarios emplean la computación basada en red, ya sea en el trabajo o en casa. La computación distribuida es diferente de (1) comunicaciones y redes y (2) sistemas operativos distribuidos. Opera a un mayor nivel de abstracción que el nivel de red y que los sistemas operativos y se ocupa de los paradigmas de programación, de API y conjuntos de herramientas y de protocolos y de estándares en el contexto de la computación basada en red.

Aunque hay numerosos libros disponibles sobre programación de redes y tecnologías, existe una carencia de libros escritos en estilo de libro de texto que combinen la teoría y la práctica de la computación distribuida.

Este libro tiene las siguientes características diferenciadoras:

- Está diseñado para introducir a los estudiantes universitarios en los fundamentos de la computación distribuida, temas normalmente destinados a estudiantes de postgrado.
- Se centra en las capas más altas de la arquitectura de la computación basada en red, y más específicamente en los paradigmas y abstracciones de dicha computación.
- Incorpora temas conceptuales y prácticos, utilizando programas de ejemplo y ejercicios para ilustrar y reforzar los conceptos presentados.
- Está diseñado como un libro de texto, con un estilo narrativo apropiado para el ámbito académico, con diagramas ilustrativos de los temas, ejercicios al final de los capítulos y listas de referencias para que los estudiantes puedan realizar sus investigaciones.
- Está diseñado para el aprendizaje con la experimentación: se presentan ejemplos de programación sobre los temas presentados y se incorporan actividades de laboratorio y ejercicios al final de cada capítulo.
- El autor proporcionará materiales de enseñanza complementarios, incluyendo transparencias de presentación, un sitio web y un manual para instructores.
- Además del libro impreso y de los artículos, este libro cita referencias fiables accesibles a través de la Web. Por ejemplo, las referencias incluyen enlaces a sitios de archivo donde se tiene acceso a los *Requests for Comments* (RFC) de Internet. El autor opina que los estudiantes universitarios están más motivados para buscar referencias que se encuentran disponibles en la Web. (Nota: aunque he elegido incluir como referencias enlaces web que son fiables y estables, es posible que alguno de ellos se quede obsoleto con el tiempo. Apreciaría cualquier información sobre dichos enlaces inactivos).

Lo que no encontrarás en este libro

- Este libro no es de redes. En sentido general las redes abarcan la computación distribuida ya que forma parte de las redes de computadoras. Sin embargo, en la universidad, los cursos de redes se suelen centrar en los niveles más bajos de la arquitectura de red y tratan temas como la transmisión de la señal, los errores de conexión, los protocolos del nivel de enlace, los protocolos del nivel de transporte y el protocolo del nivel de Internet. En comparación, este libro trata

los niveles más altos de la arquitectura de red, en concreto los niveles de aplicación, presentación y sesión, haciendo hincapié en el punto de vista de los paradigmas y abstracciones de computación y no en la arquitectura del sistema.

- Este libro no es sobre sistemas distribuidos. El enfoque no está en la arquitectura del sistema o en recursos del sistema.
- Este libro no es sobre el desarrollo de aplicaciones web. Aunque Internet es la red más popular, la programación para Internet es una forma específica de computación distribuida. Este libro trata la computación distribuida en general, incluyendo Internet, intranets y redes de área local.
- Este libro no es sobre API o tecnologías. Aunque a lo largo del libro se introducen una serie de API, son presentadas como conjuntos de herramientas para paradigmas particulares; la introducción de estas API tiene la intención de permitir a los estudiantes escribir programas para los ejercicios de laboratorio con el fin de reforzar los conceptos y fundamentos.

Para los profesores

Este libro está diseñado para su uso en un curso técnico optativo en el ámbito universitario. Los doce capítulos del libro pueden ser cubiertos en un cuatrimestre como mínimo o en un semestre de forma más pausada. Los materiales no requieren conocimientos avanzados de redes, de sistemas operativos o de ingeniería del software. Se puede dar un curso basado en este libro para estudiantes de segundo ciclo.

En un campo tan amplio como la computación distribuida es imposible cubrir todos los aspectos en un solo libro. De forma particular, no se intentan cubrir las últimas tecnologías. Este libro tiene la intención de dar a conocer los conceptos fundamentales de la intercomunicación entre procesos.

A lo largo de todo el libro está la idea de abstracción, en el sentido de encapsulación: cómo se aplica esta idea en varios paradigmas de computación distribuida y las diferencias entre herramientas que proporcionan diversos niveles de abstracción. Creo firmemente que estos conceptos e ideas son importantes para todos los estudiantes de ciencias e ingenierías de la computación, independientemente del área de especialización. Con la comprensión de estos conceptos fundamentales, los estudiantes deberían estar bien preparados para explorar nuevas herramientas y tecnologías por sí mismos, como se espera que hagan a lo largo de su carrera profesional.

Los tres primeros capítulos del libro contienen material de introducción que puede ser explicado en la primera o dos primeras semanas del curso académico. Durante este tiempo se introducirá a los estudiantes en multitud de temas que pueden o no ser nuevos para ellos. Los siguientes capítulos son más técnicos y detallados y pueden ser explicados a razón de un capítulo por semana.

Debido a la amplitud del tema de la computación distribuida, es probable que se quiera aportar materiales adicionales a este texto que se consideren importantes. Por ejemplo, se puede añadir una introducción a los algoritmos distribuidos o se puede profundizar más en el área de seguridad. Para tener tiempo para estos añadidos se puede considerar prescindir de algunos capítulos del libro.

No se asume que los lectores de este libro tienen experiencia previa en computación distribuida. En Cal Poly, he utilizado este material para enseñar a estudiantes con diversos perfiles, desde estudiantes sin ninguna experiencia en programación multiproceso, hasta estudiantes que han desarrollado complejo software de red. Sin em-

bargo, los estudiantes con conocimientos avanzados también encontrarán estos temas de interés.

Nota para los lectores

Unos comentarios sobre los tipos de letra en la redacción de este libro:

- Los términos y frases clave están en negrita; por ejemplo: este libro trata la **computación distribuida**.
- Palabras especiales, tales como las utilizadas como identificadores de programas, nombres de protocolo no estándares o nombres de operaciones son expresados en letra itálica para su diferenciación del resto de la frase; por ejemplo: ¿cuál es la salida esperada cuando se ejecuta *thread3*? Compílelo y ejecútelo.
- Las palabras reservadas y los identificadores, tales como los definidos por el lenguaje Java o por los protocolos conocidos, aparecen en itálica; por ejemplo: para dar soporte a los hilos en un programa, Java proporciona la clase denominada *thread*, además de la interfaz denominada *runnable*.

Sobre la introducción de artículos y enlaces web:

A lo largo del texto de este libro, se insertan extractos de artículos previamente publicados en otros medios. Estos artículos han sido escogidos por su relevancia en los temas y por el interés que pueden suponer para los lectores.

Muchas de las referencias listadas al final de cada capítulo son enlaces web. Esta es una elección premeditada, porque el autor cree que el acceso a contenido disponible en la Web incrementará la iniciativa de los estudiantes. Los enlaces web elegidos son aquellos que el autor considera fiables y estables. Sin embargo, la obsolescencia de alguno de estos enlaces es inevitable, un cuyo caso el autor ofrece sus disculpas y agradece su notificación.

Contactar con el autor

La compilación de un libro de texto es una tarea larga y laboriosa. Hasta el punto que pueda intentaré maximizar la precisión de los materiales presentados en este libro. Si se descubre algún error o falta de coherencia o si se tienen sugerencias de mejora, estaré encantada de saberlo. Por favor, mándeme un correo a *mliu@csc.calpoly.edu*.

Materiales complementarios

Los materiales complementarios, incluyendo los programas fuentes de los ejemplos de programación y las transparencias, están disponibles en el URL *www.aw.com*.

Los materiales adicionales para profesores están sólo disponibles contactando con el representante de ventas de Addison Wesley.

Agradecimientos

Agradezco la generosidad de diversos autores y editores que han concedido permiso de reimpresión para la inclusión de trabajos previamente publicados, en el texto de este libro.

Estoy en deuda para siempre con mis directores de programa de doctorado, Dr. Divyakant Agrawal y Dr. Amr El Abbadi, Computer Science Department, University of California en Santa Barbara, que me introdujeron en el campo de la computación distribuida.

Debo dar las gracias a los estudiantes de Cal Poly Jared Smith, Domingo Colon, Vinh Pham, Hafeez Jaffer, Erik Buchholz y Lori Sawdey por el regalo de su tiempo y su esfuerzo en la revisión del libro. También tengo que agradecer a los estudiantes que se matricularon en estos cursos de Cal Poly en los años académicos 2001-2003: Computer Engineering 369, Computer Engineering 469, y Computer Science 569, por su paciencia con los numerosos errores en los borradores del manuscrito de este libro.

Agradezco a mis compañeros del Computer Science Department y del College of Engineering en Cal Poly, que me dieron la oportunidad de dar clases en estos cursos que inspiraron el libro, y sin cuyos ánimos el libro no hubiera sido posible.

Agradezco al personal de Addison Wesley por su inestimable dirección y asistencia que permitió que el libro viera la luz, y a los siguientes revisores por la generosidad de compartir su tiempo y sabiduría:

Anup Talukdar
Motorola Labs, Schaumburg

Dr. Ray Toal
Loyola Marymount University

Mr. David Russo
Computer Science and Engineering
Senior Lecturer, University of Texas en Dallas

Dr. Alvin Lim
Computer Science and Software Engineering Department

Auburn University
Isaac Ghansah, Professor
Computer Science and Computer Engineering
California State University, Sacramento

Bruce Char
Department of Computer Science
Drexel University

Finalmente, agradezco a mi familia y en particular a mi hijo, Marlin, el haberme proporcionado razones para esforzarme.

M. L. Liu
Cal Poly, San Luis Obispo
Enero, 2003

CAPÍTULO 1

Introducción a la computación distribuida

Este libro trata sobre **computación distribuida**. Este capítulo comenzará indicando qué se entiende por computación distribuida en el contexto de este libro. Para ello, se muestra la historia de la computación distribuida y se compara este tipo de computación con otras formas de computación. A continuación, se presentan algunos conceptos básicos en los campos de **sistemas operativos**, **redes** e **ingeniería del software**, conceptos con los que el lector debe familiarizarse para comprender el material presentado en capítulos sucesivos.

1.1. DEFINICIONES

Uno de los orígenes de la confusión existente en el campo de la computación distribuida es la falta de un vocabulario universal, tal vez debido al increíble ritmo al que se desarrollan nuevas ideas en este campo. A continuación se definen algunos de los términos claves utilizados en el contexto de este libro. Durante la lectura del libro es necesario mantener en la mente estas definiciones, teniendo en cuenta que algunos de dichos términos pueden tener diferentes definiciones en otros contextos.

En sus orígenes, la computación se llevaba a cabo en un solo procesador. Un **monoprocesador** o la **computación monolítica** utiliza una única unidad central de proceso o CPU (*Central Processing Unit*) para ejecutar uno o más programas por cada aplicación.

Un **sistema distribuido** es un conjunto de computadores independientes, interconectados a través de una red y que son capaces de colaborar con el fin de realizar una tarea. Los computadores se consideran independientes cuando no comparten memoria ni espacio de ejecución de los programas. Dichos computadores se denominan computadores **ligeramente acoplados**, frente a computadores **fuertemente acoplados**, que pueden compartir datos a través de un espacio de memoria común.

La **computación distribuida** es computación que se lleva a cabo en un sistema distribuido. Este libro explora las formas en que los programas, ejecutando en computadores independientes, colaboran con otros con el fin de llevar a cabo una determinada tarea de computación, tal como los servicios de red o las aplicaciones basadas en la Web.

- Un **servicio de red** es un servicio proporcionado por un tipo de programa especial denominado servidor en una red. La WWW (*World Wide Web*) o simplemente *Web* es un servicio de este tipo, así como el correo electrónico (*email*) y la transferencia de ficheros (FTP: *File Transfer Protocol*). Un programa de servidor es justamente la mitad del denominado modelo cliente-servidor de la computación distribuida. Este modelo se estudiará detalladamente en sucesivos capítulos de este libro.
- Una **aplicación de red** es una aplicación para usuarios finales, que se ejecuta en computadores conectados a través de una red. Existe un gran número de aplicaciones de red, que van desde aplicaciones comerciales, tales como carritos de la compra y subastas electrónicas, a aplicaciones no comerciales tales como salones de *chat* o juegos de red.

La diferencia entre servicios y aplicaciones de red no es siempre nítida y estos términos frecuentemente se intercambian.

1.2. LA HISTORIA DE LA COMPUTACIÓN DISTRIBUIDA

Al comienzo se utilizaban computadores aislados, cada uno de los cuales era capaz de ejecutar programas almacenados. La conexión de computadores aislados de forma que los datos se pudieran intercambiar fue una progresión natural. La conexión rudimentaria de computadores a través de cables fue utilizada ya en los años 60 para la compartición de ficheros. No obstante, esta práctica requiere intervención manual y no puede denominarse aplicación de computación a uno o más programas que ejecutan de forma autónoma con el objetivo de realizar una determinada tarea. Tal aplicación requiere comunicación de datos, donde dos computadores intercambien datos espontáneamente y utilicen determinado software y hardware para la realización de las tareas inherentes de dicha aplicación.

El primer RFC (*Request For Comments*) de Internet, RFC 1, es una propuesta que especifica cómo las máquinas participantes pueden intercambiar información con otras a través del uso de **mensajes**. Mientras pudiera haber intentos individuales de crear aplicaciones de red a pequeña escala (tal vez mediante la conexión de dos o más computadores a través de cable), la primera aplicación de red fue el correo electrónico, también denominado *email*, ya que el primer mensaje fue enviado y registrado en 1972 en una red ARPANET de cuatro nodos. (Un nodo de una red es un computador o máquina que participa en la misma.) Los mecanismos de transferencia de ficheros automatizados, que permiten el intercambio de ficheros de datos entre las máquinas, supusieron otra progresión natural y ya en 1971 hubo una propuesta para dicho tipo de mecanismo (*véase* los RFC 114 y RFC 141). Hasta el día de hoy, el correo electrónico y la transferencia de ficheros siguen siendo dos de los más populares servicios de red. Sin embargo, el más conocido servicio de red es indudablemente la ***World Wide Web*** (**WWW**). La Web fue concebida originalmente a finales de los años 80 por científicos del centro de investigación suizo CERN en Ginebra como una aplicación que permite el acceso a hipertexto sobre una red. Desde entonces, la WWW se ha convertido en una plataforma para aplicaciones y servicios de red, incluyendo el correo electrónico, motores de búsqueda y comercio electrónico (*e-commerce*).

Request for Comments (Petición de comentarios) son especificaciones propuestas por ingenieros de Internet que invitan a realizar comentarios públicos. A través de los años, se han realizado miles de especificaciones, que se han archivado y que están accesibles en un gran número de sitios web, incluyendo los Archivos de Internet RFC/STD/FYI/BCP [faqs.org, 5].

La red ARPANET, creada en 1970, fue la predecesora de Internet.

La WWW ha sido responsable de la gran explosión que ha habido en el uso de Internet. Hasta 1990, ARPANET, el predecesor de Internet, fue utilizado principalmente por científicos, investigadores y académicos como una red de datos. Estimulado por la popularidad de la WWW, la red creció espectacularmente en los años 90, tal y como se muestra en las Figuras 1.1 y 1.2.

Algunos sitios de la Web con información de la historia de la red y que son muy visitados son [vlmp.museophile.com, 1], [zakon.org, 2] y [isoc.org, 38]. Además, [Hafner and Lyon, 4] es un fascinante informe de los primeros pasos de Internet, incluyendo información sobre las personas y las organizaciones que estuvieron implicados.

El término **hipertexto** fue creado por el visionario Ted Nelson, para referirse a los documentos textuales desde los que se puede acceder a documentos adicionales a través de rutas asociadas. El ejemplo más conocido de hipertexto es una página web que contiene enlaces.

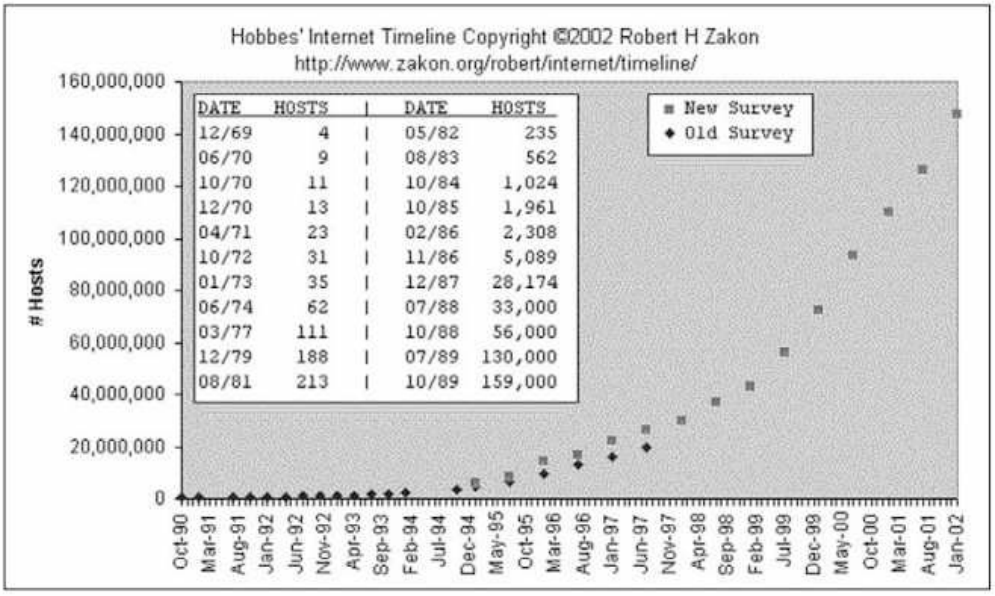
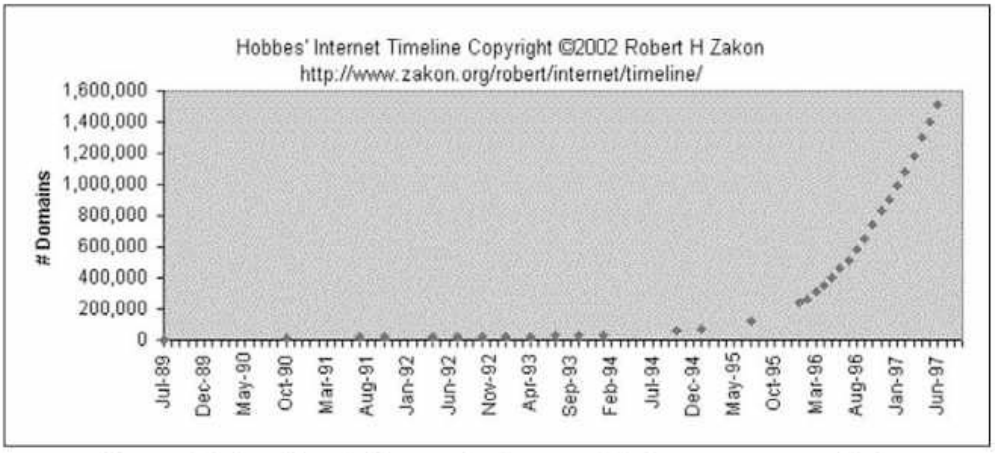


Figura 1.1. El crecimiento de los servidores de Internet [zakon.org, 2] (reimpreso con permiso).



Un dominio Internet es parte del esquema de nombrado de recursos de Internet.

Figura 1.2. Dominios de Internet [zakon.org, 2] (reimpreso con permiso).

Tendencias históricas

por Richard Gabriel y Jim Waldo, Sun MicroSystems.
(Obtenido de http://www.sun.com/jini/overview/ [7].)
Reimpreso con permiso de Jim Waldo.

¿Cómo hemos llegado hasta el punto en que los servicios y dispositivos conectados son las principales fuerzas que condicionan las nuevas formas de computación?

La razón más significativa es nuestro mejor entendimiento de la física, la química, que son las bases físicas de la computación, y el proceso de creación de *chips*. Hoy en día, un computador significativamente potente se puede construir a partir de uno o dos *chips* pequeños y un sistema de computación completo en una pequeña tarjeta.

Existen tres dimensiones que pueden mejorarse: el tamaño, el coste y la potencia computacional. Desde los años 60, el tamaño y el coste de los computadores se han decrementado dramáticamente, mientras que la potencia computacional ha ido superando distintos topes.

Una máquina o *mainframe* de los años 60 estaba formada por una colección de enormes cajas, situadas en una habitación grande, costaba millones de dólares, lo que suponía un freno para incrementar la potencia computacional. Sólo una compañía completa podía afrontar su coste.

La aparición del minicomputador fue posible cuando la funcionalidad del *mainframe* pudo abordarse mediante el uso de pequeñas cajas. El minicomputador tenía la potencia computacional de la generación anterior de *mainframes*, pero un único departamento podía comprarlo. La mayoría de los minicomputadores se conectaron a terminales interactivos, esto constituyó los comienzos de la cultura basada en los computadores, una comunidad.

Cuando un computador con la potencia de un minicomputador se redujo a una única caja que podía situarse en la mesa de un despacho, se obtuvo la estación de trabajo. Un departamento podía comprar aproximadamente una estación de trabajo por cada dos profesionales. Una estación de trabajo tenía suficiente potencia computacional para abordar tareas tales como diseño sofisticado, aplicaciones de ingeniería y científicas, y soporte gráfico para las mismas.

Cuando el computador personal se hizo suficientemente pequeño para situarse en un escritorio y suficiente potente para soportar interfaces gráficas de usuario intuitivas, los individuos podían utilizarlo sin grandes problemas y las compañías empezaron a comprarlos para cada uno de sus empleados.

Eventualmente los procesadores se hicieron suficientemente pequeños y baratos para colocarlos en un coche sustituyendo el sistema de encendido anterior, o en un televisor en lugar de los anteriores dispositivos electrónicos discretos. Hoy en día, los coches pueden tener más de 50 procesadores, y los hogares alrededor de 100.

La potencia computacional tiene otro problema. La tendencia hacia procesadores más pequeños, más rápidos y más baratos implica que menos gente tiene que compartir una CPU, pero también implica que la gente de una organización en cierto modo se aísla. Cuando se comparte una herramienta, se crea una comunidad; debido a que las herramientas se reducen, menos personas la utilizan conjuntamente y, por tanto, la comunidad se dispersa. No obstante, una comunidad no suele darse por vencida. Afortunadamente, la potencia computacional está relacionada con la reducción de los procesadores, y ya que la comunidad utiliza un sistema de computación reducido, existe suficiente potencia para permitir la comunicación entre los sistemas. Así, por ejemplo, las estaciones de trabajo empiezan a dar resultados satisfactorios una vez que se pueden comunicar e intercambiar datos.

El tramo final de la dimensión de la potencia computacional lo constituye el hecho de que ahora los procesadores son suficientemente potentes para soportar un lenguaje de programación de alto nivel, orientado a objetos y que permite el movimiento de objetos entre diferentes procesadores. Dichos procesadores son suficientemente pequeños y baratos como para constituir los dispositivos más sencillos.

Una vez que hay suficiente potencia computacional, la habilidad para conectarse y comunicarse se convierte en el factor determinante. Actualmente para la mayoría de la gente, un computador sólo ejecuta unas pocas aplicaciones y principalmente utilidades relacionadas con la comunicación: correo electrónico, la Web. Recuerda cómo Internet se hizo popular rápidamente, primero con el correo electrónico, y, más recientemente, con el uso de la Web y los navegadores.

1.3. DIFERENTES FORMAS DE COMPUTACIÓN

Para comprender qué significa la computación distribuida en el contexto de este libro, resulta instructivo analizar diferentes formas de computación.

Computación monolítica

En la forma más sencilla de computación, un único computador, tal como un computador personal (PC, *Personal Computer*) se utiliza para la computación. Dicho computador no está conectado a ninguna red y, por tanto, sólo puede utilizar aquellos recursos a los que tiene acceso de manera inmediata. Esta forma de computación se denomina **computación monolítica**. En su forma más básica, un único usuario utiliza el computador a la vez. Las aplicaciones de usuario sólo pueden acceder a aquellos recursos disponibles en el sistema. Un ejemplo de este tipo de computación, que puede denominarse *computación monolítica monousuario*, es el uso de aplicaciones tales como un programa de procesamiento de texto u hojas de cálculo en un PC.

La computación monolítica permite la convivencia de múltiples usuarios. Esta forma de computación (*véase* la Figura 1.3a), donde varios usuarios pueden compartir de forma concurrente los recursos de un único computador a través de una técnica denominada **tiempo compartido**, fue popular en los años 70 y 80. El computador que proporciona el recurso centralizado se denomina *mainframe* para diferenciarlo de pequeños computadores tales como minicomputadores o microcomputadores. Los usuarios, que podrían estar dispersos geográficamente, se pueden conectar al *mainframe* e interaccionar con el mismo durante una sesión a través de dispositivos denominados *terminales*. Algunos *mainframes* ampliamente utilizados incluyen las series IBM 360 y las series Univac 1100. Las aplicaciones que utilizan esta forma de computación son típicamente programas separados y diseñados para llevar a cabo una sola función, tal como programas de nóminas o contabilidad para una empresa o una universidad.

Computación distribuida

En contraste, la **computación distribuida** implica el uso de múltiples computadores conectados a la red, cada uno de los cuales tiene su propio procesador o procesadores y otros recursos (*véase* la Figura 1.3b). Un usuario que utilice una estación de trabajo puede usar los recursos del computador local al que la estación de trabajo está conectada. Adicionalmente, a través de la interacción entre el computador local y los computadores remotos, el usuario también puede acceder a los recursos de estos últimos. La Web es un excelente ejemplo de este tipo de computación. El uso de un na-

vegador para visitar un sitio web, tal como Netscape o Internet Explorer, implica la ejecución de un programa en un sistema local que interactúa con otro programa (conocido como servidor web) que se ejecuta en un sistema remoto, a fin de traer un fichero desde dicho sistema remoto.

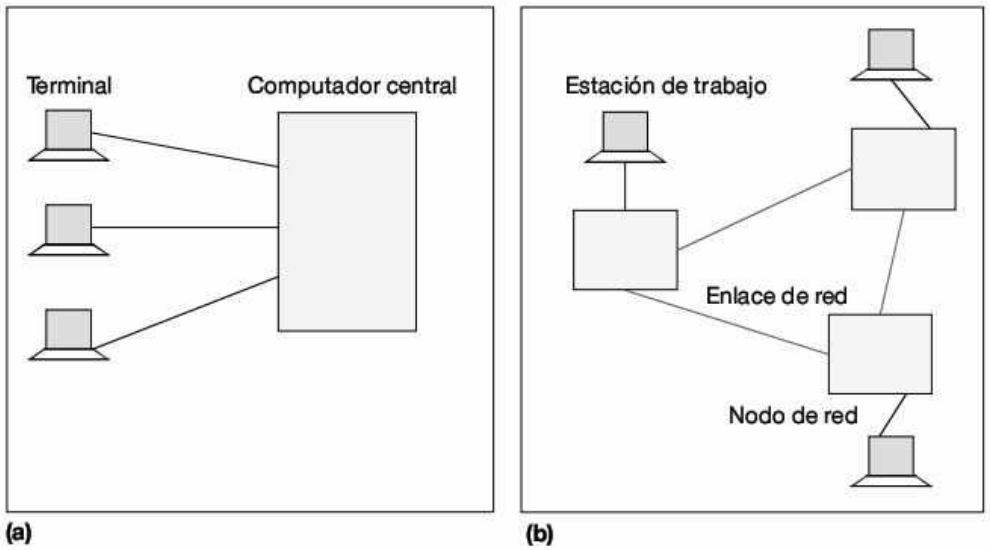


Figura 1.3. Computación centralizada (a) frente a computación distribuida (b).

Computación paralela

Similar a la computación distribuida, aunque diferente, es la denominada **computación paralela** o **procesamiento paralelo**, que utiliza más de un procesador simultáneamente para ejecutar un único programa. «Idealmente, el procesamiento paralelo permite que un programa ejecute más rápido porque hay más motores (más CPU) ejecutándolo. En la práctica, suele ser difícil dividir un programa de forma que CPU separadas ejecuten diferentes porciones del programa sin ninguna interacción» [Koniges, 9]. La computación paralela se suele realizar sobre un único computador que tiene múltiples CPU, aunque, de acuerdo a Koniges, es también posible «llevar a cabo procesamiento paralelo mediante la conexión de varios computadores en una red. Sin embargo, este tipo de procesamiento paralelo requiere software muy sofisticado denominado software de procesamiento distribuido» [Koniges, 9].

Mediante la computación paralela se pueden resolver problemas que de otra manera sería imposible resolver con un único computador. También permite la resolución de problemas de computación intensiva que de otra forma serían insostenibles económicamente. Hoy en día, la computación paralela se utiliza principalmente en computación científica a gran escala, en áreas tales como la biología, la aeronáutica, la predicción atmosférica y el diseño de semiconductores. Aunque se trata de una materia fascinante, la computación paralela se encuentra fuera de los objetivos de este libro.

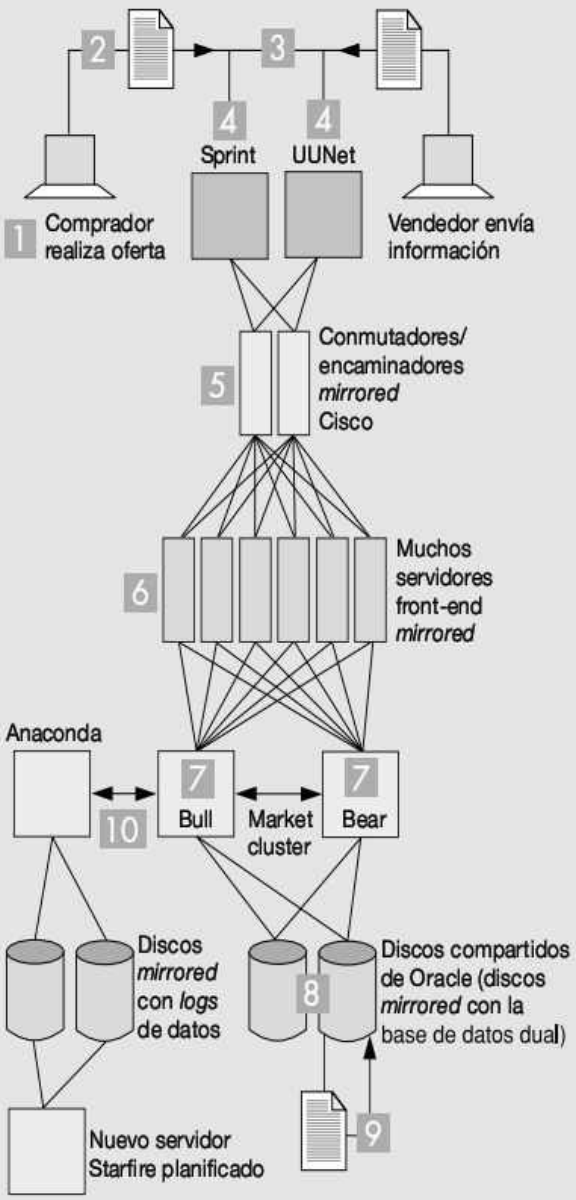
Por dónde viaja

por *Joseph Menn.*
(De *Los Angeles Times*, Los Angeles, Calif., Dec. 2, 1999, Joseph Menn

Los usuarios de eBay raramente piensan sobre el proceso que implica la realización de ofertas, hasta que el sistema falla. Internamente, el sistema de subasta tiene un número de salvaguardas que se basan principalmente en la duplicación o *mirroring* de la tecnología, para el caso en el que una pieza hardware o software falle. La información debe pasar a través de muchas compañías y tipos de equipamiento diferentes para que todo funcione correctamente.

1. La persona que hace la oferta se registra y envía una oferta electrónica desde su PC.
2. La oferta viaja desde el proveedor de servicios de Internet del consumidor, a través de los conmutadores (*switches*) y encaminadores (*routers*), al ISP de los servidores de la compañía.
3. Se envía la oferta a través del *backbone* de Internet.
4. La oferta viaja desde uno de los ISP de eBay, muy probablemente Sprint o UUNET, y a través de tuberías a eBay.
5. La oferta pasa a través de los conmutadores y encaminadores Cisco de eBay.
6. La información llega a uno de los aproximadamente 200 servidores *front-end* Compaq, que ejecutan Windows NT. Los servidores son duplicados, de forma que si uno falla, el otro realiza la tarea.
7. La oferta pasa por uno de los servidores Starfire de Sun Microsystem, llamados Bull y Bear, estando duplicados entre sí.
8. La oferta se añade a las dos bases de datos de almacenamiento de información, que ejecutan Oracle, donde se comprueba la información del vendedor.
9. El flujo de información se envía de vuelta, mandando correos electrónicos al vendedor y a los posibles compradores del producto de la oferta. También se envía una confirmación a la persona que envió la oferta.
10. Desde Bull, se envía la cantidad de la oferta y otra información a otro servidor Starfire, denominado Anaconda, y se guarda en discos de almacenamiento duplicados.

Fuente: Times staff, EBay.



Ebay tiene planificado añadir otro servidor Starfire para los discos de datos finales, sirviendo de *mirror* de Anaconda.

Computación cooperativa

Un propietario de un computador interesado *descargará* de SETI@home una pieza de software libre (por ejemplo, un salvapantallas). A continuación, cuando su computador esté ocioso, el software *descarga* un fichero de datos de un sitio de Internet para analizarlo en dicho computador. Los resultados del análisis se vuelven a enviar al sitio de Internet, donde se combinan con el resto de participantes de SETI@home, en el intento de buscar señales de vida extraterrestre.

Recientemente, el término *computación distribuida* se ha aplicado también a proyectos de computación cooperativa tales como el de la búsqueda de inteligencia extraterrestre (SETI, *Search for Extraterrestrial Intelligence*) [setiathome.ssl.berkeley.edu, 10] y distributed.net [distributed.net, 33]. Estos proyectos dividen la computación a gran escala entre las estaciones de trabajo de las máquinas de Internet, utilizando ciclos de CPU excedentes, como se describe en el margen (*Nota*: La discusión más avanzada sobre este tipo de computación queda fuera del alcance de este libro.)

1.4. VIRTUDES Y LIMITACIONES DE LA COMPUTACIÓN DISTRIBUIDA

Antes de la aparición de la Web, la computación monolítica, tal como la ejecución de aplicaciones de empresa en *mainframes* o la ejecución de un procesador de texto o una hoja de cálculo en un computador personal por parte de un único usuario, era la forma de computación dominante. Se dice que Thomas Watson, el fundador de IBM, dijo esta frase en 1943: «creo que hay un mercado mundial para tal vez cinco computadores». Sin embargo, desde los años 80, la computación distribuida se ha vuelto tan importante, si no más, como la computación monolítica.

Existen diferentes razones para la popularidad de la computación distribuida:

- **Los computadores y el acceso a la red son económicos.** Los computadores personales actuales tienen una potencia superior a los primeros *mainframes*, además de tener mucho menor tamaño y precio. Unido al hecho de que la conexión a Internet está disponible universalmente y es económica, el gran número de computadores que existen interconectados se convierte en una comunidad ideal para la computación distribuida.

- **Compartición de recursos.** La arquitectura de la computación distribuida refleja la arquitectura de computación de las organizaciones modernas. Cada organización mantiene de forma independiente los computadores y recursos locales, mientras permite compartir recursos a través de la red. Mediante la computación distribuida, las organizaciones pueden utilizar sus recursos de forma efectiva. La Web, por ejemplo, consiste en una plataforma muy potente para la compartición de documentos y otros recursos dentro y entre las organizaciones.

- **Escalabilidad.** En la computación monolítica, los recursos disponibles están limitados por la capacidad de un computador. Por el contrario, la computación distribuida proporciona escalabilidad, debido a que permite incrementar el número de recursos compartidos según la demanda. Por ejemplo, se pueden añadir más computadores que proporcionen servicio de correo electrónico si se produce un incremento en la demanda de este servicio.

- **Tolerancia a fallos.** Al contrario que la computación monolítica, la computación distribuida permite que un recurso pueda ser replicado (o reflejado) con el fin de dotar al sistema de tolerancia a fallos, de tal forma que proporcione disponibilidad de dicho recurso en presencia de fallos. Por ejemplo, las copias o *backups* de una base de datos se pueden mantener en diferentes sistemas de la red, de modo que si un sistema falla, se puede acceder al resto de las copias sin interrumpir el servicio. Aunque no es posible construir un sistema distribuido completamente fiable en presencia de fallos [Fischer, Lynch, and Paterson, 30], el desarrollador que diseña e implementa un sistema es el responsable de maximizar la tolerancia a fallos del mismo. La tolerancia a fallos en la computación distribuida es un tema complejo que ha recibido mucha atención en la comunidad investigadora. Los lectores interesados en este tema pueden leer el trabajo de Pankaj Jalote [Jalote, 31].

En cualquier forma de computación, existe siempre un compromiso entre sus ventajas y sus desventajas. Además de las ventajas descritas, la computación distribuida también tiene algunas desventajas. Las más significativas son:

- **Múltiples puntos de fallo.** Hay más puntos de fallo en la computación distribuida. Debido a que múltiples computadores están implicados en la computación distribuida, y todos son dependientes de la red para su comunicación, el fallo de uno o más computadores, o uno o más enlaces de red, puede suponer problemas para un sistema de computación distribuida. Existe una cita popular, atribuida al notable científico Leslie Lamport, que dice que «un sistema distribuido es aquél en el que el fallo de un computador que ni siquiera sabes que existe, puede dejar tu propio computador inutilizable».
- **Aspectos de seguridad.** En un sistema distribuido hay más oportunidades de ataques no autorizados. Mientras que en un sistema centralizado todos los computadores y recursos están normalmente bajo el control de una administración única, en un sistema distribuido la gestión es descentralizada y frecuentemente implica a un gran número de organizaciones independientes. La descentralización hace difícil implementar y ejecutar políticas de seguridad; por tanto, la computación distribuida es vulnerable a fallos de seguridad y accesos no autorizados, que desafortunadamente puede afectar a todos los participantes en el sistema. Este problema está claramente ilustrado por ataques bien conocidos en Internet, tales como los gusanos y los virus [Eichen and Rochlis, 21; Zetter, 22].

  Debido a su importancia, la seguridad en la computación es un tema ampliamente investigado y estudiado, y se han desarrollado varias técnicas para la escritura y realización de aplicaciones seguras. Tales técnicas incluyen cifrado, claves, certificados, firmas digitales, entornos seguros, autenticación y autorización. La seguridad es un amplio tema que queda fuera del alcance de este libro. Se recomienda a los lectores proseguir el estudio de este tema en referencias tales como [Oaks, 32].

Ahora que se han clarificado los objetivos de este libro, se van a analizar algunos conceptos básicos en tres disciplinas de informática relacionadas: los sistemas operativos, las redes y la ingeniería del software. Aunque para este libro no se requiere un conocimiento muy profundo de estas disciplinas, el libro sí se refiere a algunos conceptos y términos asociados con las mismas. El resto de este capítulo introducirá dichos conceptos y términos.

Los ataques web podrían tener muchos orígenes

por Matt Richtel y Sara Robinson (NYT), 11 de febrero de 2000.
Reimpreso con permiso del New York Times.

San Francisco, 10 Feb. Los expertos de seguridad en informática dijeron hoy que las pruebas sugieren que los ataques durante tres días a importantes sitios web podrían ser obra de más de una persona o grupo.

El análisis de que más de un grupo estuviera detrás cuestiona las conclusiones de algunos expertos de seguridad que fueron inicialmente escépticos de que siguiendo el ataque del lunes a Yahoo, múltiples vándalos se hubieran unido para realizar asaltos «*copy cat*» a otros sitios.

Mientras la comunidad de Internet busca de forma vehemente a los responsables, los expertos informáticos afirman que será difícil determinar incluso qué computadores iniciaron los ataques.

CERT, una organización de seguridad de computadores, financiada federalmente, y anteriormente conocida como la *Computer Emergency Response Team*, ha afirmado que nunca había visto este número inusual de informes de ataques. Desde el lunes hasta el miércoles, los servicios de varios sitios web punteros, incluyendo los del portal Yahoo, la compañía de la Bolsa E*Trade Group, la empresa de subastas eBay y el sitio de noticias CNN de Time Warner, fueron interrumpidos y en algunos casos parados por asaltos provocados por docenas de computadores inundando a los servidores con múltiples datos.

Pero los expertos en seguridad han afirmado que los sitios web e Internet en general quedará vulnerable en un futuro cercano, debido a que muchas organizaciones no están tomando medidas oportunas para evitar que vándalos puedan iniciar ataques en sus computadores.

Un oficial del gobierno ha dicho hoy que es necesario endurecer las leyes a fin de combatir tales ataques. «Nosotros no consideramos esto una travesura» dijo el subfiscal general Eric Holder. «Estos hechos son muy serios».

Además hoy se ha revelado que más sitios web importantes, que habían sido advertidos, fueron asaltados el miércoles, último día de los asaltos. Estos incluyen a Excite@Home, un proveedor de acceso a Internet a alta velocidad sobre cable módem, que fue atacado en la tarde del miércoles a pesar de haber tomado precauciones para defender su red.

Al menos otras dos compañías importantes de comercio electrónico fueron atacadas el miércoles, de acuerdo a IFsec, una compañía de seguridad informática de Nueva York, que rechazó dar el nombre de las compañías, afirmando que una de ellas fue un cliente.

«Nosotros estamos viendo más de los que aparecen en los medios de comunicación», afirmó David M. Remnitz, el jefe ejecutivo de IFsec.

Adicionalmente, usuarios del I.R.C. (*Internet Relay Chat*) dijeron que en las dos últimas semanas el foro había sufrido ataques similares a los de las compañías de comercio electrónico.

Mientras tanto, los proveedores de servicio de redes e investigadores continúan analizando las pruebas, incluyendo los paquetes de datos utilizados para sobrecargar y paralizar a los sitios web víctimas.

Expertos de seguridad informática de la Universidad Stanford en Palo Alto, California, dijeron que las primeras pruebas sugieren que los ataques podrían haber sido obra de más de una persona o grupo.

David J. Brumley, asistente oficial de seguridad informática en Stanford, afirmó que el tipo de datos incluido en los paquetes utilizados para atacar a Yahoo el lunes eran diferentes de los datos del asalto el martes a eBay.

«Los ataques fueron completamente diferentes esos dos días» dijo Mr. Brumley. «Las personas que atacaron Yahoo eran diferentes de las que atacaron eBay y CNN.»

Los proveedores de servicio de redes dijeron que los asaltos recientes incluían dos tipos de ataques, sugiriendo que más de un grupo podría estar implicado. Ambos tipos de ataques corresponden a lo que se denomina denegación de servicio, ya que estos ataques evitan que los sitios puedan servir a los clientes.

En el primer tipo, conocido como *SYN flood*, los atacantes accedieron e instalaron software en un gran número de computadores. A continuación, utilizaron estas máquinas para bombardear al sitio víctima con peticiones para comenzar una sesión de comercio electrónico. El gran número de peticiones sobrecargó a los servidores, que no permitieron que los clientes pudieran acceder al sitio.

Para evitar que se pudiera realizar una traza de estas peticiones, los vándalos emplearon una técnica denominada *spoofing*, que consiste en alterar la dirección de origen.

El segundo tipo de ataque, conocido como *smurf attack*, de nuevo implica el uso de otras máquinas, pero además emplea una gran red de computadores externa para «amplificar» el número de datos utilizados en el ataque y, de esta forma, incrementar la efectividad del asalto. Se cree que la red de computadores de Stanford se podría haber utilizado de esta forma en el ataque a Yahoo.

Los expertos de seguridad afirman que es sencillo configurar las redes de forma que no se puedan utilizar en un *smurf attack*, aunque muchos administradores no conocen cómo llevar a cabo esta configuración.

Los expertos de seguridad informática hicieron hincapié en que el gran número de computadores utilizados para iniciar los ataques esta semana hace muy difícil realizar una traza de los mismos.

«En este punto, existe tanto tráfico que es muy difícil realizar una traza», afirmó Joel de la Garza de Kroll-O'Gara Information Security Group, una compañía para minimizar los riesgos.

Además, expertos de seguridad informática han comentado que las compañías cuyos computadores han sido asaltados y utilizados como plataformas para un asalto, normalmente no tienen ninguna idea del problema, incluso cuando el asalto continúa. Los vándalos pueden activar el asalto desde una ubicación remota, y a una compañía o a un individuo cuyo computador se está utilizando; el único impacto que puede provocar es un decremento de la velocidad en la actividad de la red.

Las compañías víctimas y los expertos de seguridad afirmaron hoy que en algunos casos los ataques parecen más complicados de lo que originalmente se pensó, reforzando la dificultad de prevenirlos.

Excite@Home, por ejemplo, ha afirmado que intentó tomar medidas de precaución a la luz de los ataques anteriores, pero que fue incapaz de prevenir el ataque a su sitio web durante al menos media hora.

«Según nuestro conocimiento, un sitio no puede tomar medidas preventivas contra los ataques sin la ayuda de otros», afirmó Kelly Distefano, una portavoz de Excite@Home. Dijo que la compañía hubiera necesitado más cooperación de las compañías que proporcionan servicios de red a Excite.

Peter Neumann, principal científico de SRI International en Menlo Park, California, reiteró que el éxito de los ataques ha mostrado que los sitios de Internet no están tomando la precauciones adecuadas para evitar ser asaltados mediante ataques de este tipo.

«Es hora de que la gente despierte», dijo Mr. Neumann. «La gente está acelerándose en utilizar el comercio electrónico en la Red sin conocer los riesgos, y hay grandes riesgos, como hemos podido ver aquí.»

1.5. CONCEPTOS BÁSICOS DE SISTEMAS OPERATIVOS

La computación distribuida supone la ejecución de programas en múltiples computadores. A continuación se describen algunos de los conceptos implicados en la ejecución de programas en computadores actuales.

Programas y procesos de computación

Un programa software es un artefacto construido por un desarrollador software utilizando alguna forma de lenguaje de programación. Típicamente, el lenguaje es de alto nivel y requiere un compilador o intérprete para traducirlo a lenguaje máquina. Cuando un programa se ejecuta en un computador se representa como un **proceso**. En los computadores modernos, un proceso consiste en un programa que se ejecuta, sus valores actuales, la información de estado y los recursos utilizados por el sistema operativo para gestionar la ejecución del programa. En otras palabras, un proceso es una entidad dinámica que sólo existe cuando un programa se ejecuta.

La Figura 1.4 muestra las transiciones de estados durante la vida de un proceso. Un proceso entra en el estado listo cuando un programa inicia su ejecución, y el sistema operativo lo sitúa en una cola, junto a otros programas que van a ejecutarse. Cuando los recursos del sistema (tales como la CPU) se encuentran disponibles, un proceso es lanzado para ejecutarse, de tal forma que entra en el estado de ejecución. Continúa ejecutándose hasta que debe esperar por la ocurrencia de un evento (tal como la finalización de alguna operación de E/S), momento en el que el proceso entra en el estado bloqueado. Una vez que el evento ocurra, el sistema operativo coloca al proceso en la cola de ejecución, esperando a su turno para ejecutar de nuevo. El proceso repite el ciclo listo-ejecución-bloqueado tantas veces como sea necesario hasta que la ejecución del proceso se complete, momento en el cual el proceso queda terminado.

En este libro se utilizarán programas Java, o fragmentos de ellos, como ejemplos de código. Hay tres tipos de programas Java: **aplicaciones** (Figura 1.5), **applets** (Figura 1.6) y **servlets** (Figura 1.7). Independientemente del tipo de programa, cada programa se escribe como una **clase Java**. Una aplicación Java tiene un método principal (*main*) y se ejecuta como un proceso independiente (*stand-alone*).

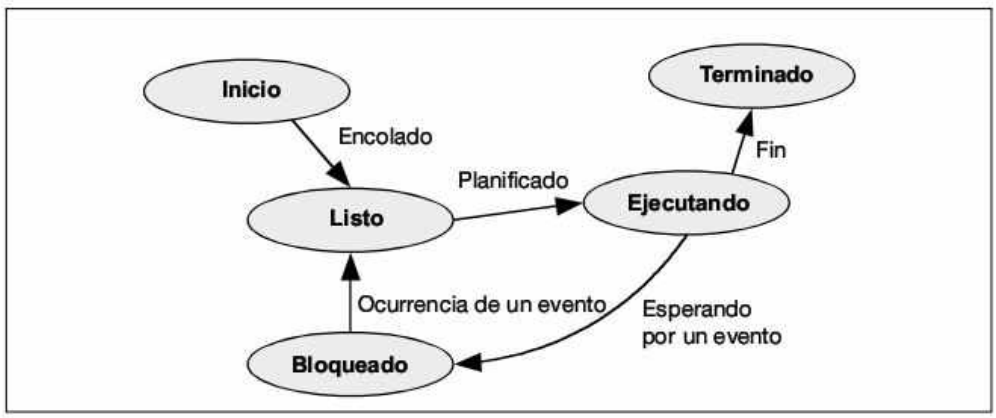
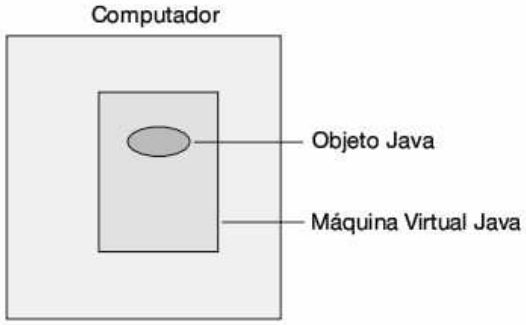


Figura 1.4. Un diagrama simplificado de transición de estados de un proceso.

Por otro lado, un applet no tiene un método *main* y se ejecuta mediante el uso de un navegador o de la herramienta que permite ver applets, *appletviewer*. Un servlet es parecido al applet en el hecho de que no tiene un método *main*. Se ejecuta en el contexto de un servidor web. En este libro se mostrarán ejemplos de los tres tipos de programas y fragmentos de programas, siendo las aplicaciones las más frecuentemente utilizados.

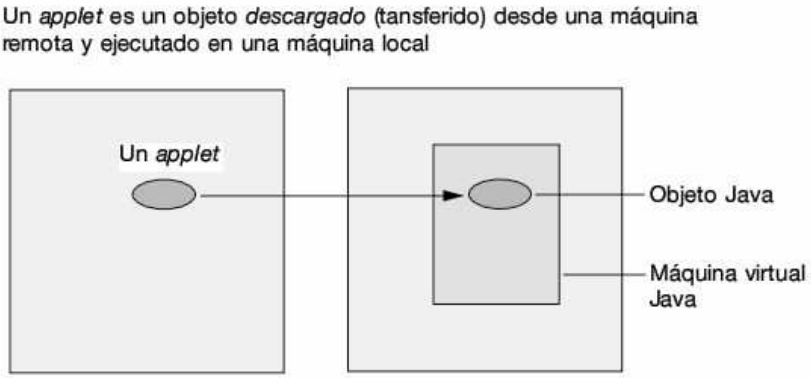
Un programa Java se compila y se convierte a un código denominado ***bytecode***, un código objeto universal. Cuando se ejecuta, la Máquina Virtual Java (JVM, *Java Virtual Machine*) traduce el *bytecode* a código máquina nativo del computador, siguiendo las transiciones de estados que se han estudiado anteriormente.

La ejecución de una aplicación *stand-alone* Java en una máquina local.




```
/*******************************************************
* Un ejemplo de una aplicación Java sencilla.
* M. Liu 1/8/02
*******************************************************/

import java.io.*;

class MiPrograma{
  public static void main(String[ ] args)
    throws IOException{

    BufferedReader teclado = new
      BufferedReader(new InputStreamReader(System.in));
    String nombre;
    System.out.println("¿Cuál es tu nombre?");
    nombre = teclado.readLine( );
    System.out.print("Hola " + nombre);
    System.out.prinln(" – bienvenido a CSC369.\n");
  } // fin main
} // fin clase
```

Figura 1.5. Una aplicación *stand-alone* Java (arriba) y el código correspondiente (abajo).

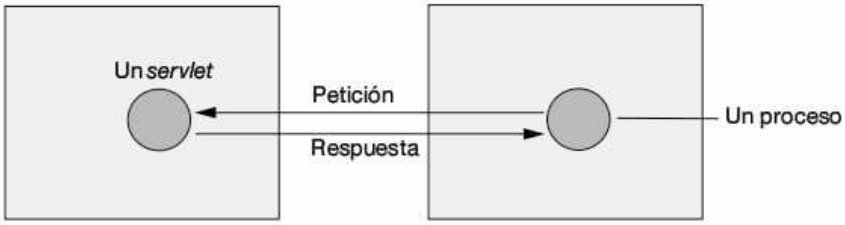



```
/************************************************
* Un ejemplo de un applet sencillo.
* M. Liu 1/8/02
************************************************/

import java.applet.Applet;
import java.awt.*;

public class MiApplet extends Applet{

   public void paint(Graphics g){
      setBackground(Color.blue);

      Font Claude = new Font("Arial", Font.BOLD, 40);
      g.setFont(Claude);
      g.setColor(Color.yellow);
      g.drawString("Hola Mundo", 100, 100);
   } // fin paint

} // fin clase
```

```
<!— Una página web que cuando se abre, ejecuta>
<!— el applet MiApplet>
<!— M. Liu 1/8/02>

<title>Un applet sencillo</title>
<hr>

<applet code="MiApplet.class" width=500 height=500>
</applet>

<hr>
<a href=" HolaMundo.java">El fichero fuente.</a>
```

Figura 1.6. Un *applet* (arriba) y la página web (abajo) que permite activar el *applet*.

Un *servlet* es un objeto que ejecuta en una máquina remota e interactúa con un proceso local mediante un protocolo de petición-respuesta.




```
/*****************************************************
* Un ejemplo de un servlet sencillo.
* M. Liu 1/8/02
*****************************************************/

import java.io.*;
import java.text.*;
import java.util.*;
import javax.servleeeeservlevlet.*;
import javax.servlet.http.*;

public class MiServlet extends HttpServlet {

   public void doGet (HttpServletRequest request,
                      HttpServletResponse response)
     throws ServletException, IOException {

     PrintWriter salida;
     String titulo = "Salida de MiServlet";
     //primero establecer el tipo de contenido y otros
     //campos de la cabecera
     response.setContentType("text/html");
     //a continuación escribir los datos de respuesta
     salida = response.getWriter();
     salida.println("<HTML><HEAD><TITLE>");
     salida.println(titulo);
     salida.println("</TITLE></HEAD><BODY>");
     salida.println("<H1>" + titulo + "</H1>");
     salida.println("<P>Hola Mundo");
     salida.println("</BODY></HTML>");
     salida.close();
   } //fin doGet
} //fin clase
```

Figura 1.7. Un *servlet* (arriba) y la página web (abajo) que permite activar el *servlet*.

Debido a que el *bytecode* es un código intermedio independiente del tipo de máquina y que se traduce al código de máquina específico en tiempo de ejecución, se dice que los programas Java son **independientes de la plataforma**, indicando que el mismo programa se puede ejecutar en cualquier tipo de máquina que tenga una JVM.

Este libro asume que el lector tiene un conocimiento básico del lenguaje de programación Java, hasta el punto de que es capaz de compilar y ejecutar una aplicación ***stand-alone*** o un applet. Un programa *stand-alone* es aquél que se ejecuta por sí solo, sin intercambiar mensajes con otros programas.

Programación concurrente

La computación distribuida supone el uso de la programación concurrente, que consiste en la ejecución simultánea de procesos. Los siguientes párrafos muestran tres clases de programación concurrente.

- **Procesos concurrentes ejecutados en múltiples computadores.** Gran parte del material de este libro incluye ejemplos de procesos separados ejecutando concurrentemente en computadores independientes interconectados a través de una red. Los procesos interactúan con otros procesos mediante el intercambio de datos sobre la red, pero su ejecución es por otra parte completamente independiente. Cuando alguien accede a una página web utilizando un navegador, un proceso del mismo que ejecuta en la máquina local interactúa con un proceso que se ejecuta en la máquina del servidor web.

  La programación concurrente que implica a múltiples máquinas requiere determinado soporte de programación; es decir, el software utilizado para los programas participantes debe contener la lógica necesaria para permitir la interacción entre los procesos. Uno de los temas principales de este libro es cómo se expresa esta lógica en los programas.
- **Procesos concurrentes ejecutados en un único computador.** Los computadores modernos utilizan sistemas operativos multitarea, que permiten la ejecución concurrente de múltiples tareas o procesos. La concurrencia puede ser real o virtual. La verdadera concurrencia multitarea sólo es posible si el computador tiene múltiples CPU, de forma que cada CPU pueda ejecutar un proceso. En un computador con una sola CPU, se utiliza tiempo compartido (*véase* la Figura 1.8) o rodajas de tiempo para permitir que los procesos puedan ejecutarse por turnos, creando la ilusión de que se ejecutan en paralelo.

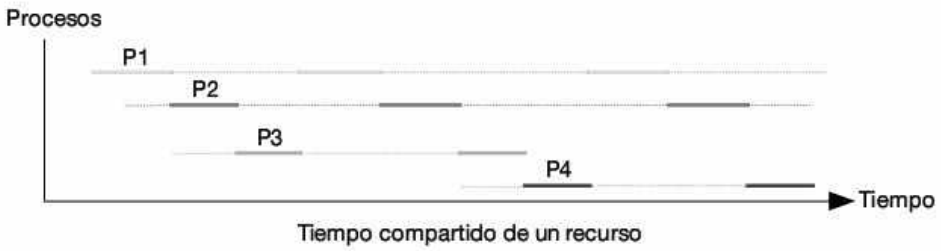


Figura 1.8. Tiempo compartido en un computador.

Ya que la multitarea es una funcionalidad del sistema operativo, no se necesita una programación especial para llevar a cabo este tipo de programación concurrente, es decir, no se necesita una lógica software especial en el programa para iniciar la multitarea.

- **Programación concurrente dentro de un proceso.** Además de la programación concurrente entre diferentes procesos, muchas veces un único programa necesita iniciar diferentes tareas que se ejecuten concurrentemente. Por ejemplo, un programa podrá necesitar realizar otras tareas mientras espera indefinidamente por la entrada de un usuario en una interfaz de ventanas. También podría ser aconsejable que un programa ejecute varias tareas en paralelo, por motivos de rendimiento. La programación concurrente dentro de un proceso se lleva a cabo a través de dos tipos de herramientas proporcionadas por el sistema operativo.

Procesos padres e hijos

En tiempo de ejecución, un proceso puede crear procesos subordinados, o **procesos hijos**. A través de la multitarea real o virtual, el proceso original, denominado **proceso padre**, continúa ejecutando simultáneamente con el proceso hijo (*véase* la Figura 1.9). Un proceso hijo es un proceso completo, que consiste en un programa en ejecución, sus propios valores actuales e información de estado, parte de la cual es heredada del proceso padre. Un proceso padre puede saber cuándo un proceso hijo ha finalizado.

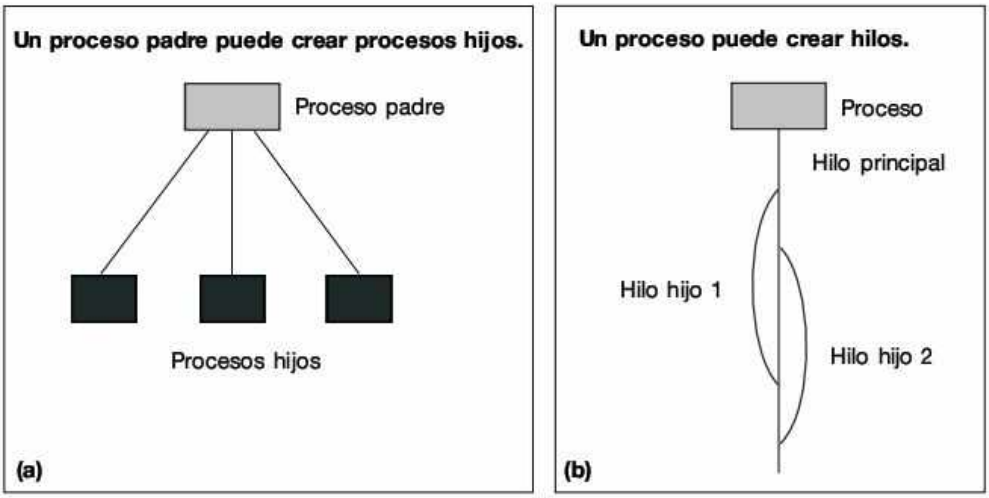


Figura 1.9. Ejecución concurrente dentro de un proceso.

Threads o hilos

En lugar de procesos hijos, un proceso puede crear ***threads*** o **hilos**, también conocidos como **procesos ligeros**. Los hilos poseen un mínimo de información de estado, comportándose por lo demás de la misma forma que los procesos. Debido a que implican menos sobrecarga, es preferible utilizar hilos que utilizar procesos hijos.

La creación y coordinación de hilos requiere el soporte de la programación. El software correspondiente al programa debe escribirse con la lógica necesaria para la creación de hilos y la coordinación, o sincronización de la ejecución de la familia de hilos creados por el hilo padre.

La ejecución concurrente de los hilos puede ocasionar una **condición de carrera**. Una condición de carrera ocurre cuando una serie de mandatos de un programa se ejecutan de una forma arbitraria e intercalada, pudiendo llevarse a cabo una ejecución no determinista. La Figura 1.10 ilustra esta situación. Siendo *contador* una variable compartida entre dos hilos concurrentes. Al ejecutar la secuencia 1, en la cual las instrucciones de los dos procesos se ejecutan secuencialmente, el valor del contador se incrementa en dos. Por otro lado, si se ejecuta la secuencia 2, en la cual el conjunto de instrucciones se intercala, el contador sólo se incrementa en uno.

Las condiciones de carrera se pueden evitar si se utiliza **exclusión mutua** dentro de un segmento de código, de forma que se asegure que los mandatos de dicho segmento sólo puedan ejecutarse por parte de un único hilo en un determinado momento. A dicho segmento de código se le denomina **región crítica**. En nuestro ejemplo, la región crítica comprende el código en el cual se accede y se incrementa la variable *contador*.

La programación que utiliza hilos se denomina **programación *multi-threaded* o programación multihilo**. Un programa *multi-threaded* escrito para evitar condiciones de carrera se dice que es ***thread-safe***. El desarrollo de programas *thread-safe* complejos requiere unos avanzados conocimientos de programación. Afortunadamente, en este libro apenas se utilizan hilos de forma explícita, ya que muchas de las herra-

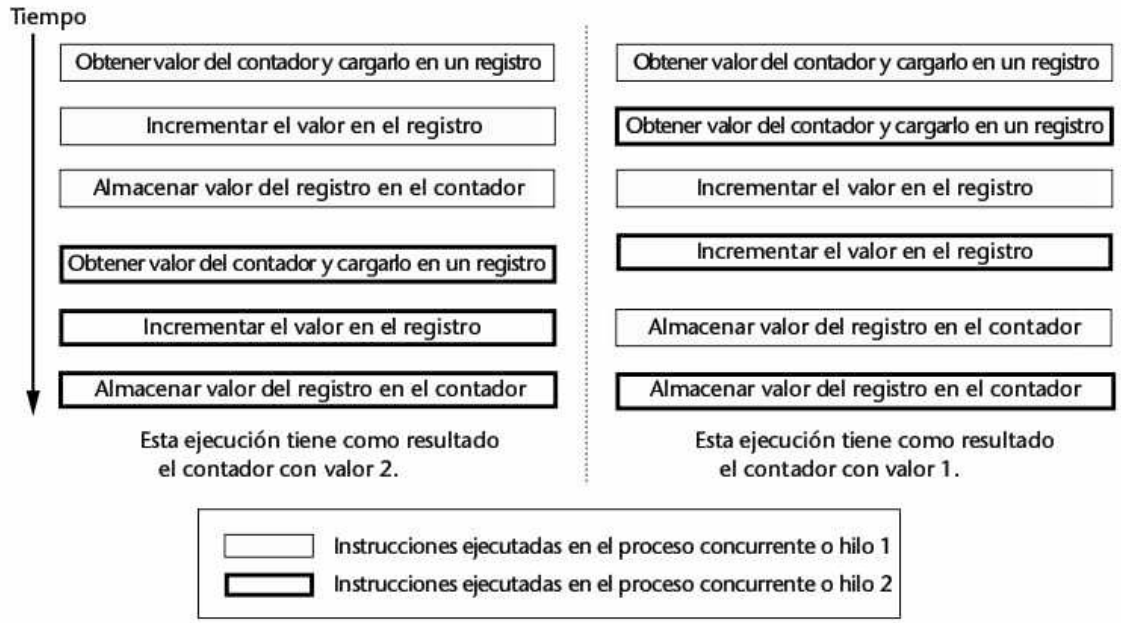


Figura 1.10. Condición de carrera procedente de procesos o hilos concurrentes no sincronizados.

mientas que dan soporte a aplicaciones de red frecuentemente utilizan programación multihilo internamente.

Hilos Java

La máquina virtual Java permite que una aplicación tenga múltiples hilos ejecutando concurrentemente. Cuando una máquina virtual Java se inicia, hay normalmente un único hilo (aunque en algunos sistemas un programa se puede iniciar con más de un hilo), que típicamente invoca al método denominado *main* de alguna clase, tal como la clase de una aplicación escrita por el programador. Se pueden crear otros hilos adicionales a partir de un hilo activo, y cada hilo se ejecutará independientemente y en paralelo con el resto hasta que termine.

Para permitir el uso de hilos en un programa, Java proporciona una clase denominada *thread* y una interfaz denominada *runnable*.

En un programa Java hay dos formas de crear un nuevo hilo de ejecución:

1. Declarar una clase como subclase de la clase *thread*. Esta subclase debe sobreescribir el método *run* de la clase *thread*. Cuando se crea e inicia una instancia de la subclase, el código del método *run* se ejecuta concurrentemente con el hilo principal.
2. Declarar una clase que implemente la interfaz *runnable*. Esta clase implementa el método *run* de dicha interfaz. Cuando se crea e inicia una instancia de la subclase, el código del método *run* se ejecuta concurrentemente con el hilo principal.

La Figura 1.11 muestra el uso de la primera forma de crear un nuevo hilo de ejecución, mientras que la Figura 1.12 muestra el uso de la segunda forma.

```
public class EjecHilos
{
   public static void main (String[] args)
   {
      HiloEjemplo p1 = new HiloEjemplo(1);
      p1.start();

      HiloEjemplo p2 = new HiloEjemplo(2);
      p2.start();

      HiloEjemplo p3 = new HiloEjemplo(3);
      p3.start();
   }
} //fin clase EjecHilos
```

```
public class HiloEjemplo extends Thread
{
   int miIdent;

   HiloEjemplo(int ident) {
      this.miIdent = ident;
   }

   public void run() {
      int i;
      for (i = 1; i < 11; i++)
         System.out.println
            ("Hilo"+ miIdent + ": " + i);
   }
} //fin clase HiloEjemplo
```

Figura 1.11. Aplicación sencilla que crea tres hilos utilizando una subclase de la clase *thread*.

```
public class EjecHilos2
{
   public static void main (String[] args)
   {
      Thread p1 = new Thread(new
HiloEjemplo2(1));
      p1.start();

      Thread p2 = new Thread(new
HiloEjemplo2(2));
      p2.start();

      Thread p3 = new Thread(new
HiloEjemplo2(3));
      p3.start();
   }
} //fin clase EjecHilos2
```

```
class HiloEjemplo2 implements Runnable
{
   int miIdent;

   HiloEjemplo2(int ident) {
      this.miIdent = ident;
   }

   public void run() {
      int i;
      for (i = 1; i < 11; i++)
         System.out.println
("Hilo"+miIdent + ": "   + i);
   }
} //fin clase HiloEjemplo2
```

Figura 1.12. Aplicación sencilla que crea tres hilos utilizando una implementación de la interfaz *runnable*.

En Java, la manera más sencilla de evitar las condiciones de carrera es a través de la utilización de los **métodos estáticos sincronizados**. Un método estático que contenga en su cabecera la palabra reservada *synchronized* puede ejecutarse por un único hilo simultáneamente. Por tanto, se garantiza que el código de un método estático sincronizado sea mutuamente exclusivo. En el ejemplo mostrado en la Figura 1.10, el código correspondiente al incremento de la variable *contador* debe encapsularse dentro de un método estático sincronizado, tal que el contador sólo se incremente por parte de un único hilo simultáneamente. Un ejemplo de código Java que ilustra el uso de hilos y métodos estáticos sincronizados puede encontrarse en el Ejercicio 2(d.) al final de este capítulo.

En subsecuentes capítulos, se utilizarán los términos **proceso** e **hilo** frecuentemente. Si el lector no se encuentra familiarizado con los *threads*, al final de este capítulo hay un conjunto de ejercicios que le permitirán practicar a través del uso de hilos Java.

1.6. CONCEPTOS BÁSICOS DE REDES

Después de analizar algunos conceptos básicos de los sistemas operativos que son relevantes para la computación distribuida, se procederá a realizar lo mismo con conceptos básicos de redes.

Protocolos

En el contexto de las comunicaciones, un protocolo es un conjunto de reglas que los participantes deben seguir. En una reunión, los seres humanos siguen instintivamente

un protocolo no establecido explícitamente, basado en la visión, el lenguaje del cuerpo y los gestos. Este protocolo establece que sólo una persona debe hablar en un momento determinado mientras el resto escucha. En una conversación telefónica, una parte inicia la llamada, y a continuación, después de que la llamada es aceptada, las dos partes se turnan en la comunicación, utilizando pausas y preguntas para indicar cuando la otra parte tiene la palabra.

En la comunicación entre computadores, los protocolos deben estar definidos formalmente e implementados de una manera precisa. Para cada protocolo, debe existir un conjunto de reglas que especifiquen las siguientes cuestiones:

- ¿Cómo se codifican los datos intercambiados?
- ¿Cómo los eventos (envío, recepción) se sincronizan (ordenan) de modo que los participantes puedan enviar y recibir información de una forma coordinada?

El concepto de protocolo se concretará más cuando se estudien diferentes protocolos en el resto del libro.

Debe destacarse que un protocolo es un conjunto de reglas. La especificación de un protocolo no indica cómo deben implementarse dichas reglas. Por ejemplo, el protocolo HTTP (*Hypertext Transfer Protocol*) especifica las reglas que deben seguir un proceso del navegador web y un proceso del servidor web. Cualquier programa en el servidor web que cumpla las reglas establecidas satisfará el protocolo, sin importar el lenguaje de programación o sintaxis utilizada. Por tanto, es importante entender que un protocolo (tal como HTTP) es distinto de su implementación (tal como indica la gran variedad de navegadores web, incluyendo los navegadores Netscape e Internet Explorer).

La **sintaxis** de un lenguaje de programación es el conjunto de reglas de lenguaje, incluyendo la ortografía y la gramática del mismo.

Análogamente, las reglas de un deporte, tal como el baloncesto, son especificadas por alguna autoridad, como por ejemplo la NBA (*National Basketball Association*), pero es responsabilidad de cada equipo y cada jugador llevar a cabo dicho juego, siempre siguiendo dichas reglas.

Arquitectura de red

En los libros sobre redes de datos, normalmente se presentan las funciones de una red utilizando una arquitectura de red (*véase* la Figura 1.13). Dicha arquitectura de red clásica se denomina arquitectura OSI (*Open System Interconnect*) y divide las funciones complejas de una red en siete capas. Todas o parte de estas funciones deben estar presentes en un computador que participa en la comunicación de datos y, por tanto, también en la computación distribuida. El lector interesado en cuestiones más específicas del modelo OSI deberá buscar más información en libros de redes tales como [Tanenbaum, 35]. Para los objetivos de este libro, se presenta una arquitectura apropiada para Internet.

OSI significa *Open System Interconnect*, y es el nombre dado al modelo de arquitectura de red realizado por una organización denominada *International Organization for Standardization* (ISO).

La arquitectura de red de Internet está representada en la Figura 1.14, y está formada por cuatro capas: física, Internet, transporte y aplicación. La **capa física** proporciona las funciones de transmisión de señales, representando los flujos de datos entre los computadores. La **capa Internet** permite dirigir y entregar un paquete de datos a un computador remoto. La **capa de transporte** proporciona las funciones necesarias para la entrega de paquetes de datos a un proceso específico que se ejecuta en el computador remoto. Finalmente, la **capa de aplicación** permite que los mensajes se puedan intercambiar entre los programas, a fin de posibilitar el funcionamiento de una aplicación, tal como la Web.

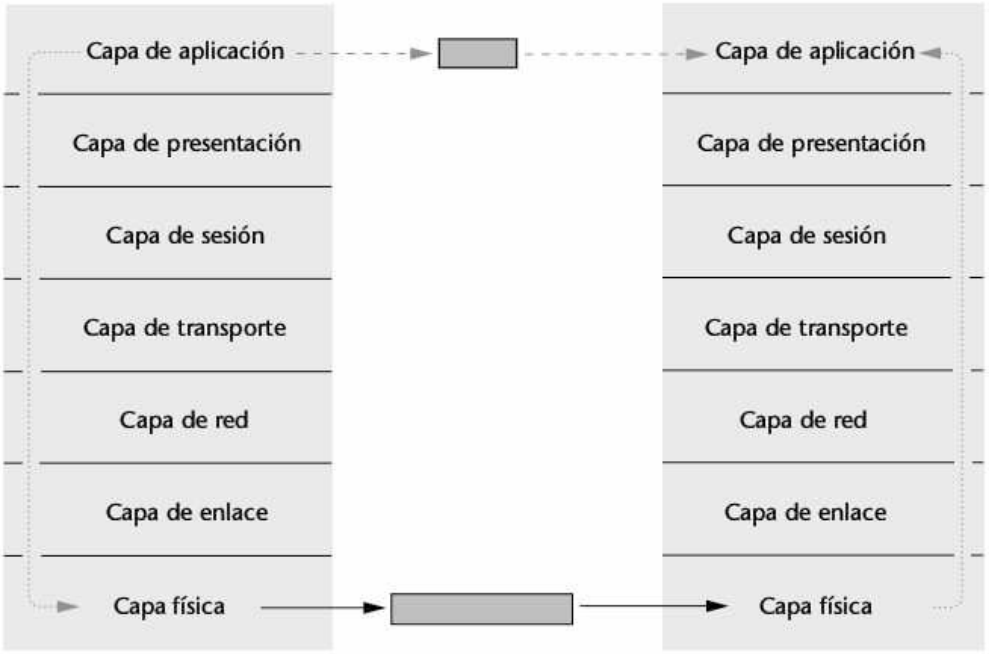


Figura 1.13. La arquitectura de red de siete capas OSI.

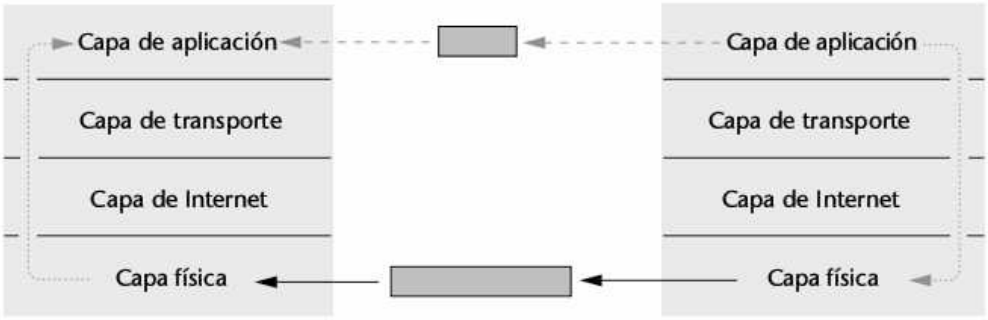


Figura 1.14. La arquitectura de red de Internet.

La división de las capas es conceptual: la implementación de estas funciones no necesita dividirse claramente en el hardware y software que implementa la arquitectura. La división conceptual de la arquitectura en capas sirve al menos para dos propósitos útiles. Primero, permite que los protocolos se puedan especificar de una forma sistemática; es decir, utilizando una arquitectura de red, estos protocolos se pueden especificar capa a capa, resolviendo la problemática asociada a cada capa. Segundo, la arquitectura de capas permite que los detalles de las funciones de la red se puedan abstraer u ocultar. Cuando se escribe una aplicación, es útil no preocuparse por los detalles de la comunicación de los datos y concentrarse en el protocolo de la aplicación. Una arquitectura de capas hace posible que los programas se escriban como si los datos se intercambiaran directamente (*véase* las líneas disjuntas de las Figuras 1.13 y 1.14). Realmente, el mensaje enviado desde una aplicación debe procesarse por todas las capas de la arquitectura de la red (*véanse* las líneas punteadas). Eventualmente,

la señal de datos que representa el mensaje se transmite a través del enlace físico que une los computadores (*véanse* las líneas continuas). Una vez que las señales llegan al computador receptor, éste las procesa en el orden inverso, hasta que finalmente los datos son reconstruidos en el mensaje original y éste es entregado al proceso apropiado.

Protocolos de la arquitectura de red

Ahora se van a describir algunos protocolos específicos de la arquitectura de Internet. El protocolo para la capa de Internet se denomina, como su nombre indica, Protocolo de Internet (IP, *Internet Protocol*). Este protocolo utiliza un esquema de nombrado particular, que se estudiará a continuación, para identificar los computadores en una red y para encaminar los datos. En la capa de transporte, hay dos protocolos ampliamente utilizados: el Protocolo de Control de Transmisión (TCP, *Transmission Control Protocol*), que proporciona una comunicación **orientada a conexión** y el Protocolo de Datagrama de Usuario (UDP, *User Datagram Protocol*), que ofrece una comunicación **sin conexión**. (En la próxima sección se analizarán más detenidamente estos conceptos y posteriormente en el Capítulo 2). Finalmente, en la capa de aplicación, existen diferentes protocolos especificados para aplicaciones de red, tales como el Protocolo de Transferencia de ficheros (FTP, *File Transfer Protocol*), el Protocolo de Correo Electrónico Sencillo (SNMP, *Simple Network Mail Protocol*) y el Protocolo de Transmisión de Hipertexto (HTTP, *Hypertext Transmission Protocol*). El conocido protocolo *Transmission Control Protocol/Internet Protocol* (TCP/IP) es un conjunto de protocolos que incluye a los protocolos de Internet y transporte de esta arquitectura; estos protocolos se utilizan universalmente para la comunicación a través de Internet. Por tanto, una aplicación de Internet debe ejecutarse en un computador que implemente esta parte de la arquitectura de Internet, coloquialmente denominada **pila de protocolos TCP/IP**.

Los lectores interesados en protocolos de capas inferiores pueden consultar libros de texto tales como [Stallings, 12; Tanenbaum, 13]. Este libro está orientado al estudio de protocolos de la capa de aplicación. El libro comienza analizando algunos de los protocolos de aplicación más populares, como los que se mencionan en el párrafo anterior. A continuación, se estudiará cómo tales aplicaciones se utilizan en la computación distribuida.

Comunicación orientada a conexión frente a comunicación sin conexión

Aunque la comunicación orientada a conexión y la comunicación sin conexión son temas relacionados con las redes de datos, en esta sección se discutirán los aspectos que distinguen ambos tipos de comunicación.

En una comunicación orientada a conexión, una conexión, que puede ser **física** (es decir, tangible, proporcionada mediante dispositivos hardware tales como cables, módem y receptores) o **lógica** (es decir, *abstracta* o *virtual*, utilizando software que emula una conexión), se establece entre dos partes, el emisor y el receptor. Por ejemplo, alguien (el emisor) puede marcar un número para realizar una llamada telefónica a un amigo (el receptor). Una vez establecida la conexión, los datos (voz en el caso de una llamada telefónica) pueden enviarse continuamente a través de la conexión hasta que la sesión finaliza. En el caso de la llamada telefónica, la sesión termina cuando el emisor cuelga el teléfono al final de la conversación, punto en el cual la conexión se

corta. Hay que destacar que en este modo de comunicación no es necesario especificar explícitamente la dirección del destinatario para cada paquete de datos individual durante el tiempo que la conexión se utilice.

Como su nombre indica, la comunicación sin conexión implica que no existe conexión. En su lugar, los datos se envían mediante el uso de paquetes y cada emisor debe indicar de forma explícita en cada paquete la dirección del receptor. Un ejemplo de comunicación sin conexión es la correspondencia entre dos amigos a través de mensajes de correo electrónico o cartas. Cada correo electrónico o carta, que contiene un mensaje, deben contener la dirección del destinatario. El intercambio continúa hasta que la correspondencia o sesión finaliza.

Una red de datos transmite datos; una red de voz transmite voz. Las redes modernas transmiten tanto datos como voz.

En una red de datos es más sencillo proporcionar una comunicación sin conexión, ya que ésta no necesita mantener conexiones separadas. Sin embargo, la falta de conexión puede implicar la pérdida de paquetes de datos en la entrega o la entrega fuera de orden de los mismos. Por ejemplo, cuando se envían múltiples y sucesivos correos electrónicos o cartas a una única persona, cada uno de los cuales contiene parte de un mensaje, es posible que el destinatario reciba los correos electrónicos o cartas desordenados, ya que cada correo o carta se entregan de forma independiente.

Por otro lado, la comunicación orientada a conexión puede asegurar la entrega segura y ordenada de los paquetes de datos a través de una conexión establecida, pero con el coste adicional de la sobrecarga que implica este proceso. Este es otro ejemplo de compromiso entre ambas soluciones.

La Figura 1.15 muestra gráficamente la diferencia entre estas dos formas de comunicación. En el Ejercicio 3 que se encuentra al final de este capítulo, se realiza un análisis sencillo de las diferencias existentes entre ambas formas de comunicación a través de un estudio guiado.

En cualquier capa de una arquitectura de red, la comunicación se puede lograr a través de utilidades orientadas a conexión o sin conexión. En la capa de transporte de la pila TCP/IP, el protocolo UDP es un protocolo sin conexión, mientras

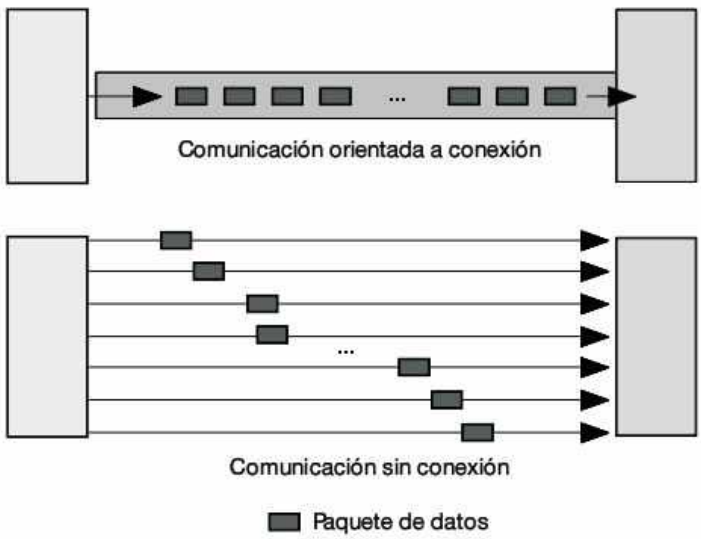


Figura 1.15. Comunicación orientada a conexión frente a comunicación sin conexión.

que el protocolo TCP es un protocolo orientado a conexión. Se dice que una aplicación o protocolo que utiliza UDP para transmitir datos es una aplicación o protocolo sin conexión a nivel de transporte, mientras que uno que utiliza TCP es orientado a conexión en dicha capa o nivel. Es necesario destacar que es posible que una aplicación sea orientada a conexión en una capa y sin conexión en otra. Por ejemplo, una aplicación que utiliza el protocolo HTTP, un protocolo orientado a conexión en la capa de aplicación, podría utilizar UDP en la capa de transporte para transmitir y recibir los datos.

La Tabla 1.1 compara los dos modos de comunicación.

Tabla 1.1. Comparación de la comunicación entre procesos orientada a conexión y sin conexión.

	Orientado a conexión	**Sin conexión**
Direccionamiento	Se especifica en el momento de la conexión; posteriormente no es necesario volver a especificarlo con cada operación (envío o recepción).	Se especifica en cada operación.
Sobrecarga de la conexión	Existe sobrecarga de establecimiento de la conexión.	No se aplica.
Sobrecarga del direccionamiento	No existe sobrecarga de direccionamiento en cada operación.	Existe sobrecarga en cada operación.
Orden de llegada de los datos	La abstracción de la conexión permite al mecanismo de comunicación entre procesos mantener el orden de llegada de los paquetes de datos.	La falta de conexión hace difícil a la aplicación que hace uso del mecanismo de comunicación entre procesos mantener el orden de llegada.
Protocolos	Este modo de comunicación es apropiado para protocolos que requieren el intercambio de grandes conjuntos de datos y/o un gran número de intercambios.	Este modo de comunicación es apropiado para protocolos que intercambian un pequeño conjunto de datos y realizan un número pequeño de intercambios.

Recursos de red

Este libro utilizará frecuentemente el término **recursos de red**. Por recursos de red se entiende aquellos recursos que están disponibles para los participantes de una comunidad de computación distribuida. Por ejemplo, en Internet los recursos de red incluyen hardware tal como los computadores (incluyendo **servidores de Internet** y **encaminadores**) o equipamiento (impresoras, máquinas de fax, cámaras, etc.), y software, tal como procesos, buzones de correo electrónico, ficheros o documentos web. Una clase importante de recursos de red son los **servicios de red**, tales como la Web o el servicio de transferencia de ficheros, que son implementados por procesos que ejecutan en computadores.

Aunque la idea parece simple, uno de los retos claves en la computación distribuida es la identificación única de los recursos disponibles en la red. La próxima sección describe cómo se lleva a cabo la identificación de recursos en Internet.

Un **nodo de Internet** es un computador que implementa la arquitectura de protocolos de Internet y, por tanto, es capaz de participar en comunicaciones de Internet.

Un **encaminador** o ***router*** es un computador especializado en encaminar los datos entre las redes. En Internet, un encaminador implementa la funcionalidad de la capa de Internet.

Identificación de nodos y direcciones del protocolo de Internet

Físicamente, Internet es una gigantesca malla de enlaces de red y computadores o nodos. Conceptualmente (*véase* la Figura 1.16), las principales arterias de Internet son un conjunto de enlaces de red de alto ancho de banda que constituyen el esqueleto central o «*backbone*» de la red. Conectadas a este *backbone* existen redes individuales, cada una de las cuales tiene un identificador único. Los computadores con soporte TCP/IP, denominados **nodos** o **máquinas** de Internet, están unidos a **redes** individuales. A través de este sistema de «autopistas de la información», los datos pueden transmitirse desde una máquina M_1 en una red R_1 a otra máquina M_2 en una red R_2. Para realizar la transferencia de datos desde un programa es necesario identificar de forma única al proceso receptor, de forma similar al proceso de envío de una carta por parte del servicio postal.

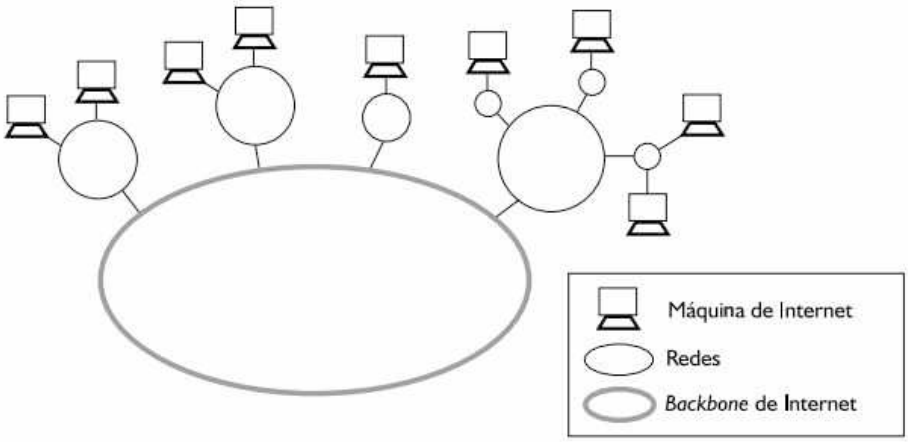


Figura 1.16. La topología de Internet.

Como se mencionó anteriormente, un proceso es una representación en tiempo real de un programa cuando se ejecuta en un computador. Por otro lado, también se ha descrito que en Internet un computador o una máquina se encuentra enlazado a una red. Por tanto, para identificar un proceso es necesario indicar la red, el computador enlazado a dicha red y a continuación el proceso que se ejecuta en el computador.

En la arquitectura de Internet, la identificación de la máquina es parte del protocolo Internet (IP), el cual, como ya se describió, es el protocolo de la capa Internet de la pila de protocolos TCP/IP. A continuación se analiza el esquema de identificación de nodos especificado en la versión 4 de IP, conocida como IPv4. Aunque el esquema se ha modificado en la versión 6 (IPv6) para permitir el uso de más direcciones Internet, los principios del esquema son los mismos para las dos versiones, habiéndose escogido IPv4 por su mayor simplicidad. En el contexto de este libro, las diferencias entre las dos versiones no son significativas.

En IPv4, cada máquina de Internet se identifica por una única cadena de 32 bits. Dada una longitud de 32 bits, el número total de direcciones posibles es 2^{32}, o lo que es lo mismo, el espacio de direcciones de IPv4 permite 2^{32} (4.294.967.296 o sobre cuatro mil millones) direcciones.

Cada dirección IP debe identificar tanto la red en la cual la máquina reside como la máquina en sí. El esquema de direccionamiento IPv4 funciona de la siguiente forma:

El espacio de direcciones se divide en cinco clases, que van desde la clase A hasta la clase E. Como queda representado en la Figura 1.17, cada clase tiene un único prefijo. La clase A comienza con el bit 0, la clase B comienza con la secuencia de bits 10, la clase C con la secuencia 110 y así sucesivamente. El resto de los bits de cada dirección se utilizan para identificar la red y la máquina correspondiente. Por tanto, una dirección de clase A tiene 31 bits para identificar el par red-máquina, una dirección de clase B tiene 30 bits y así sucesivamente. Esto significa que hay un total de 2^{31} (aproximadamente dos mil millones) direcciones de clase A disponibles, mientras que hay un máximo de 2^{32} (aproximadamente mil millones) direcciones de clase B disponibles. El número máximo de direcciones de clase C, D o E se puede calcular de forma similar. Es importante destacar que dentro de cada clase un pequeño número de direcciones (tales como la dirección con todos los bits a 0 o todos los bits a 1) queda reservado para propósitos especiales.

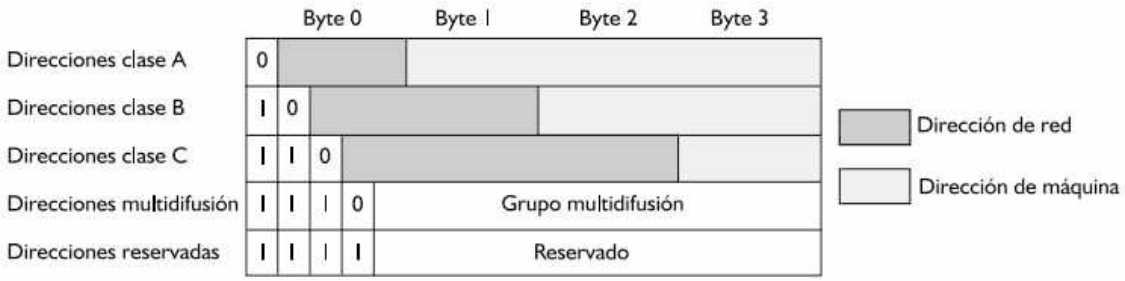
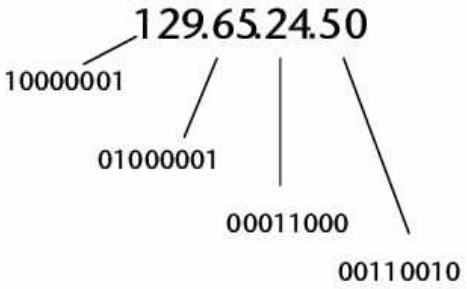


Figura 1.17. El esquema de direcciones IPv4.

Es importante saber por qué es necesario tener diferentes clases de direcciones. Esto está relacionado con el número de computadores que cada red individual puede soportar. Considérese una dirección de clase A (*véase* la la Figura 1.17): los 7 bits que siguen al prefijo 0 se utilizan para la identificación de la red y el resto (32-8 = 24 bits) identifican la máquina dentro de la red. Por tanto, cada red de clase A puede albergar hasta 2^{24} (aproximadamente 16 millones) computadores, aunque no puede haber más de 2^{7} o 128 redes de este tipo. Del mismo modo, se puede analizar que cada una de las 2^{14} (16.384) redes de clase B pueden tener hasta 2^{16} (65.536) máquinas. Igualmente, existen muchas más redes de clase C que de clase B, aunque cada red de clase C puede contener muchos menos computadores.

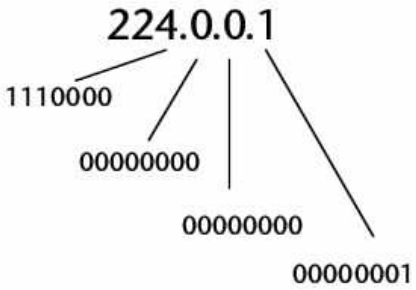
Como se ha mencionado anteriormente, en raras ocasiones tendremos ocasión de identificar máquinas IP utilizando la dirección de 32 bits. En las ocasiones que utilicemos la dirección numérica, lo más probable es que usemos la denominada notación decimal con puntos en su lugar. La notación decimal con puntos de una dirección IP utiliza un valor decimal para cada uno de los 4 bytes de la dirección IP.

Como un ejemplo, supongamos que la notación decimal con puntos de una dirección Internet particular es 129.65.24.50. La expansión binaria de 32 bits de dicha notación es la siguiente:



Ya que la secuencia de bits comienza por 10, la dirección es una dirección de clase B. Dentro de esta clase, la porción correspondiente a la identificación de la red corresponde a los restantes bits en los primeros 2 bytes, es decir, 00000101000001, y la porción correspondiente a la identificación de la máquina corresponde a los últimos 2 bytes, 0001100000110010. Por conveniencia, el prefijo binario que identifica la clase se suele incluir como parte de la porción de red de la dirección, de forma que se dice que esta dirección tiene como identificación de red 129.65 y como identificación del computador en la red 24.50.

Veamos otro ejemplo. Dada la dirección 224.0.0.1, se puede expandir de la siguiente forma:



El prefijo binario 1110 indica que esta dirección es de clase D, también denominada dirección de multidifusión (*multicast*). Los paquetes de datos enviados a esta dirección deben ser entregados al grupo de multidifusión 00000000000000000000000000000001.

Las direcciones IP las asigna una autoridad conocida como **IANA** (***Internet Assigned Numbers Authority***) [community-ml.org, 25] a organizaciones tales como universidades o proveedores de servicio en Internet (ISP, *Internet Service Provider*). (*Nota*: La asignación de esta autoridad es dinámica. *Véase* http://www.wia.org/pub/iana.html, donde se describe la historia de la evolución de esta autoridad.) Dentro de cada red, la asignación de la identificación de la máquina se hace de forma interna a la organización. Normalmente, una organización hace uso de esta porción de la dirección para subdividir su red en una jerarquía de subredes, con un único número de nodo asignado a cada computador unido a una subred. Por ejemplo, el administrador de la clase B 129.65 podría designar el segundo byte (es decir, los 8 bits de la parte izquierda de la porción del nodo) como identificador de una subred. Bajo este esquema de subredes, la dirección IP 129.65.32.3 identifica a una máquina con identificador 3 en una subred de identificador 32 de esta red.

Desde la década de los 90, la demanda de direcciones IP se ha disparado hasta el punto de que el espacio de direcciones se ha agotado. El esquema de direccionamiento estático que se acaba de describir se ha aumentado con numerosos cambios en respuesta a la demanda creciente de direcciones, incluyendo el esquema de **direccionamiento dinámico** que se ha hecho popular con los proveedores de servicio de Internet o ISP, tales como American Online (AOL). Utilizando direccionamiento dinámico, un ISP o grandes organizaciones pueden extender el espacio de direcciones para una red IP dada mediante la unión de direcciones. Por ejemplo, una dirección estática de red de clase B puede tener hasta 2^{16} o 65.536 máquinas estáticas. Uniendo las aproximadamente 65.000 direcciones y asignando cada una de ellas a una sesión activa bajo demanda, es posible dar soporte a millones de computadores IP, asumiendo que no hay más de 65.000 sesiones activas en un determinado momento. Por esta razón, cuando se accede a Internet a través de un ISP, la dirección IP del computador puede variar de una sesión a otra.

La mayoría de los usuarios tienen problemas para memorizar una cadena de 32 bits, incluso con la ayuda de la notación decimal con puntos. Por tanto, es preferible utilizar un nombre simbólico para identificar un computador. Esta es la razón por la que la comunidad de Internet adoptó el **sistema de nombrado de dominio** (**DNS**, *Domain Name System*). El acrónimo DNS también se refiere al servicio de nombrado de dominio (*Domain Name Service*), que consiste en el servicio que proporciona un sistema de nombrado de dominio. Cada vez que se utiliza el correo electrónico o se visualiza una página web, se identifica la máquina de Internet utilizando un nombre de dominio basado en el protocolo DNS.

Cada nombre de dominio contiene al menos dos componentes separados por puntos. En una dirección como acme.com, el último componente, com en este caso, se denomina **dominio de primer nivel**. A la izquierda del punto en el nombre, acme en este caso, se encuentra lo que se denomina **dominio de segundo nivel**. Es posible también que existan subdominios, tales como marketing.acme.com. Los nombres de dominios no son sensibles a las mayúsculas y minúsculas, por lo que se pueden utilizar indistintamente.

Actualmente, los dominios de primer nivel están clasificados como se muestra en la Tabla 1.2 [Brain, 15].

Tabla 1.2. Nombres de dominio de alto nivel.

.com	Para entidades comerciales, que cualquiera, desde donde sea, puede registrar.
.net	Originalmente se designó para organizaciones directamente relacionadas con las operaciones de Internet. Actualmente, este dominio también se está utilizando para negocios, cuando el nombre .com deseado ya está registrado por otra organización. Hoy en día cualquiera puede registrar un nombre en el dominio .net.
.org	Para organizaciones misceláneas, incluyendo aquellas organizaciones sin ánimo de lucro.
.edu	Para instituciones de educación superior.
.gov	Para entidades del gobierno federal de los EEUU.
.mil	Para el ejército de EEUU.
Códigos de países	Para países individuales basados en la organización de estándares internacionales; por ejemplo, .ca para Canadá, y .jp para Japón. *Véase* [Connolly, 18] para ver una lista de los códigos existentes de países.

Los dominios de segundo nivel combinados con los dominios de primer nivel (por ejemplo, calpoly.edu) normalmente, aunque no siempre, corresponden a la porción de red de una dirección IP, mientras que el resto del nombre del dominio (por ejemplo, www.csc) se utiliza para identificar la subred, si existe, y el nombre de la máquina. *Véase* la Figura 1.18 donde se describe gráficamente esta característica.

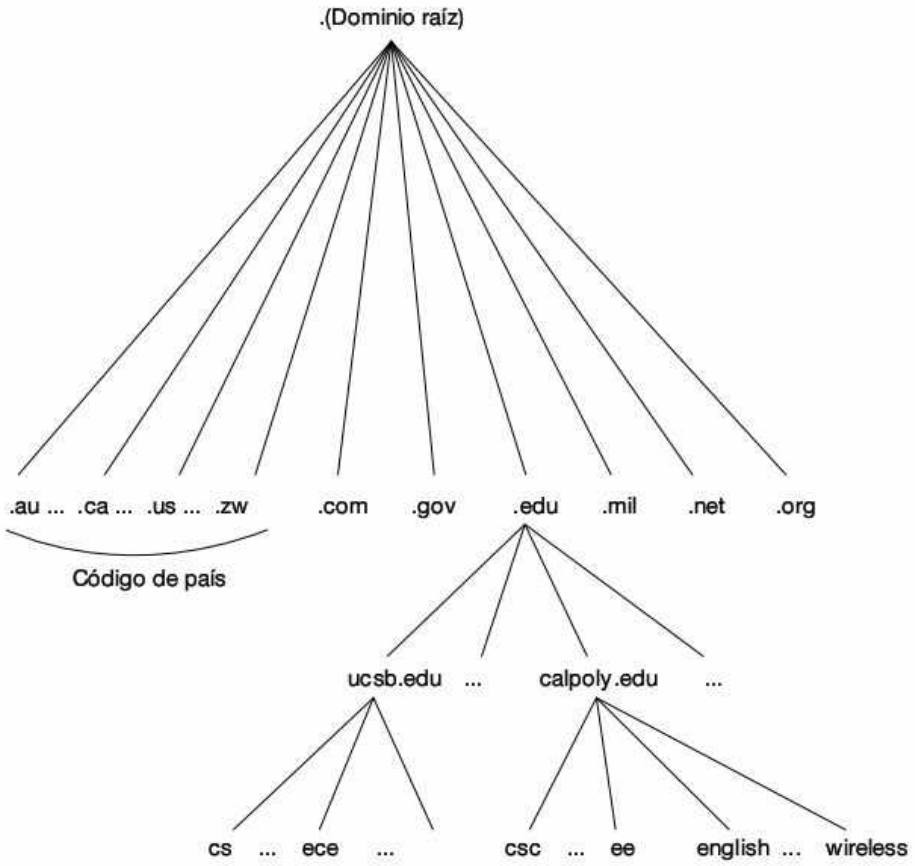


Figura 1.18. Jerarquía de nombres de dominio.

Cada nombre de dominio corresponde a una dirección IP, aunque esta asociación no tiene por qué ser permanente. Por ejemplo, el nombre de dominio ebay.com actualmente corresponde a la dirección IP 216.32.120.133. La resolución de un nombre de domino para obtener la correspondiente dirección IP y viceversa se lleva a cabo a través de un servicio conocido como **resolución de nombres DNS**. El Ejercicio 4 muestra una forma de probar este servicio.

Finalmente, el nombre de dominio *localhost* se puede utilizar para identificar el computador en el cual se ejecuta el proceso. Este nombre se corresponde siempre a la dirección IP 127.0.0.1 y permite identificar al computador local.

Una vez que se identifica una máquina utilizando una dirección IP o un nombre de dominio, se pueden identificar también recursos individuales dentro de la máqui-

na. A continuación se mostrarán tres ejemplos de esquemas de identificación de un proceso, un receptor de correo electrónico y documentos web, respectivamente.

Identificación de procesos a través de puertos

Especificar el nombre de dominio o su dirección IP de forma correcta permite localizar una máquina o computador en Internet. Pero en las aplicaciones de red, los datos deben entregarse a un proceso específico que se ejecuta en un computador. Por tanto, se necesita un esquema de nombrado que permita identificar de forma única un proceso. Hay muchos esquemas que permiten realizar esto. Por ejemplo, una posibilidad es utilizar un único **identificador de proceso (PID**, *Process Identifier*), que le asigna el sistema operativo al proceso (*véase* el Ejercicio 4). En Internet, el protocolo utilizado para identificar un proceso usa una entidad lógica conocida como **puerto de protocolo**, o simplemente **puerto**. Es importante recordar que la capa de transporte de la arquitectura de Internet es la encargada de distribuir los datos a los procesos, siendo los dos protocolos más conocidos de esta capa TCP y UDP. Cada uno de estos protocolos utiliza un conjunto separado de puertos en cada máquina para llevar a cabo esta tarea. Un proceso que desee intercambiar datos con otro proceso utilizando TCP o UDP debe tener asignado uno de estos puertos. Una aplicación que desee enviar datos a un proceso actualmente asociado al puerto p en la máquina M, debe dirigir los datos a (M, p). En IPv4, existen 2^{16} puertos (desde el puerto 0 hasta el puerto 65.535) en cada máquina, bajo los protocolos TCP o UDP. Por ejemplo, cuando se accede a un sitio web, normalmente se hace uso del servicio de un proceso que ejecuta en la máquina que se especificó (por ejemplo, www.calpoly.edu) utilizando el protocolo TCP y el puerto 80.

En los protocolos TCP y UDP, los números entre el 0 y el 1023 (2^{10}) están reservados para servicios conocidos. Estos puertos se denominan **puertos bien conocidos (*well-known*)** y los asigna la autoridad IANA [isoc.org, 38]. En algunos sistemas sólo los procesos del sistema o los programas que ejecutan usuarios privilegiados pueden utilizan estos números. A cada servicio de red popular, tal como telnet, FTP, HTTP o SMTP, se le asigna uno de estos números de puerto (23, 21, 80, 25, respectivamente), que permite localizar cada uno de estos servicios. En los ejemplos de programación de este libro se especificarán en muchas ocasiones estos números de puerto.

Direcciones de correo electrónico

Una dirección de correo electrónico utiliza el formato nombreUsuario@nombreDominio. Por ejemplo, mliu@csc.calpoly.edu identifica al autor de este libro. Cuando se envía un correo electrónico identificando esta dirección como destinatario, un programa de correo electrónico en el computador correspondiente al nombre de dominio especificado entrega el correo al buzón del usuario especificado en este sistema; en este caso, el autor de este libro.

URL

Los usuarios de los navegadores web están familiarizados con los **URL (*Uniform Resource Locators*)**. Cuando se introduce una cadena, tal como http://www.csc.calpoly.edu en el navegador para visitar un determinado sitio web, se está utilizando un URL.

Un URL es un esquema de nombrado que se encuentra debajo de un esquema más general denominado **URI (*Uniform Resource Identifiers*)**. Los URI son cadenas cortas que identifican recursos en la Web, incluyendo documentos, imágenes, ficheros, servicios y buzones de correo. El esquema URI permite identificar de una forma uniforme estos recursos, bajo una variedad de esquemas de nombrado utilizados en protocolos de aplicación específicos, tales como HTTP, FTP y correo electrónico.

URL es un término informal asociado con populares esquemas URI para protocolos tales como HTTP, FTP o correo electrónico.

URN (*Uniform Resource Name*) es un esquema especificado por el RFC2141 y otros documentos relacionados, que permite el uso de identificadores de recursos persistentes e independientes de su localización. Un URN proporciona nombres persistentes dentro de un espacio de nombres, permitiendo de esta forma que un objeto permanente tenga varias copias en varios sitios conocidos; si uno de los sitios no está disponible, el objeto podría encontrarse en cualquiera de los otros sitios. Existen varias propuestas para la implantación de los URN, pero ninguna de ellas ha sido ampliamente adoptada aún [aboutdomains.com, 16].

Aunque sea un término informal, el URL es con diferencia el mejor conocido de todos estos términos. Un URL proporciona una forma no persistente (es decir, no necesariamente permanente) de identificar un objeto de forma única dentro de un espacio de nombres. Un espacio de nombres, en el contexto de un sistema de nombrado, se refiere al conjunto de nombres que el sistema proporciona. En su forma más general, el formato de un URL es:

<protocolo>//<usuario>:<clave>@<id-máquina>:<puerto>/<ruta>

donde

- **<protocolo>** es el nombre no sensible a mayúsculas o minúsculas del protocolo de la capa de aplicación utilizado para acceder al recurso; por ejemplo, HTTP en el caso de acceder a un navegador web;
- **<usuario>:<clave>** es la autorización de acceso, en el caso de que sea requerida por el protocolo;
- **<id-máquina>** es el nombre de dominio o dirección IP decimal con puntos de la máquina que proporciona el servicio a través del protocolo; por ejemplo, www.calpoly.edu;
- **<puerto>** es el puerto para el protocolo de la capa de transporte del proceso que proporciona el servicio en la máquina remota; por ejemplo, el puerto 80 (por defecto) para los servidores HTTP o Web;
- **<ruta>** especifica la ruta dentro del sistema de ficheros de la máquina remota donde se encuentra el recurso; por ejemplo, mliu/csc102/index.html.

Cuando se introduce un URL en un navegador, es posible no especificar el protocolo (en cuyo caso se asume que se utiliza el protocolo HTTP), el par usuario:clave (no utilizado en HTTP), el número de puerto (80 por defecto), y la ruta (se supone que se selecciona la raíz de la jerarquía de directorios de documentos). Por ejemplo, el URL www.csc.calpoly.edu introducido en el navegador Netscape permite acceder a la página inicial de la Universidad Politécnica de California de San Luis Obispo, que se encuentra en la máquina con nombre de dominio www.csc.calpoly.edu, y que utiliza el puerto 80.

A veces se puede utilizar una forma acortada de un URL, denominada **URL relativo**. Durante una sesión, cuando se está accediendo a un documento (por ejemplo, http://www.csc.calpoly.edu/index.html), se puede utilizar un URL relativo para nom-

brar otro fichero en el mismo directorio, trayéndose dicho fichero desde el mismo servidor web. Por ejemplo, si hay otro fichero en el mismo directorio llamado courses.html, entonces el URL courses.html se puede utilizar en lugar del URL completo http://www.csc.calpoly.edu/courses.html.

Servicio de nombres extensible

El servicio de nombres extensible (XNS, *Extensible Name Service*) es un servicio de nombres de Internet gestionado por la Organización XNS Public Trust Organization (XNSORG), una organización independiente. El servicio permite un esquema de nombrado con una dirección única y universal para llevar a cabo «todos los tipos de comunicaciones: teléfono, fax, páginas web, mensajería instantánea, e incluso correo ordinario... Como cualquier servicio de nombrado, XNS opera a más alto nivel que DNS. DNS está diseñado para traducir un nombre a una dirección de una máquina Internet. XNS está diseñado para resolver una dirección universal en cualquier otro tipo de direcciones de cualquier tipo de red de comunicaciones. Se podría decir que XNS es a DNS como DNS es a un número de teléfono (y, de hecho, XNS utiliza DNS para resolver la dirección Internet de una agencia XNS)» [omg.org, 27]. Un XNS es una cadena de caracteres. Hay tres tipos de nombres XNS: nombres personales, nombres de negocio y nombres generales, cada uno de los cuales empieza por un único carácter (=, @, y +, respectivamente) y cada uno de los cuales puede contener hasta 64 caracteres Unicode.

Unicode es un estándar para representar caracteres. De acuerdo con la página inicial de Unicode, «Unicode proporciona una [representación numérica] única para cada carácter, sin importar cuál sea la plataforma, el programa o el lenguaje» [Unicode.org, 29].

Resolución de nombres

Siempre que se utiliza un nombre simbólico para identificar un recurso, el nombre debe traducirse a la correspondiente dirección física para localizar dicho recurso. Como se ha mencionado anteriormente, un nombre de dominio tiene el siguiente formato:

nombreComputador.nombreDivisión.nombreCompañía.com

que para una máquina Internet debe ser traducido a la dirección numérica, por ejemplo 129.65.123.7, del computador correspondiente. Al proceso de traducción se le conoce como **resolución de nombres**, o simplemente **búsqueda de nombres**.

Para realizar la resolución de nombres, se debe utilizar una base de datos (también llamada directorio o registro) que contenga las asociaciones entre nombres simbólicos y nombres físicos. Si el espacio de nombres de un esquema de nombrado tiene un tamaño limitado, entonces es posible realizar la resolución de nombres manualmente. En el caso de DNS o XNS, un proceso manual no tiene sentido; en su lugar se utiliza un servicio de red para permitir la resolución de nombres dinámicamente.

En el caso de DNS, los **servidores DNS** se encargan de realizar el servicio de búsqueda de nombres. Una autoridad central mantiene la base de datos de nombres y permite que la base de datos se distribuya a través de Internet a los servidores DNS. Cuando se especifica un nombre de dominio, tanto si se introduce en un navegador como si se codifica en un programa que se va a ejecutar, dicho nombre se envía al servidor DNS más cercano para su resolución. Si el servidor más cercano no tiene la asociación, envía la petición a otro servidor DNS. La petición se propaga hasta que el nombre es resuelto, respuesta que es enviada de nuevo al proceso que originó la petición.

En posteriores capítulos de este libro se tendrá la oportunidad de trabajar con esquemas de nombrado y sus utilidades asociadas.

1.7. CONCEPTOS BÁSICOS DE INGENIERÍA DEL SOFTWARE

La ingeniería del software es la disciplina de informática que aborda el proceso de desarrollo de las aplicaciones. Aunque este libro proporciona el conocimiento técnico necesario para construir aplicaciones de red, no cubre el proceso de desarrollo de tales aplicaciones. De la misma forma, algunos de los conceptos básicos de la ingeniería del software son relevantes para este libro. A continuación se introducen estos conceptos.

Programación procedimental frente a programación orientada a objetos

A la hora de construir aplicaciones, hay dos clases de lenguajes de programación: lenguaje procedimental y lenguaje orientado a objetos. (Aunque existen otros tipos de lenguajes, tales como el lenguaje funcional, estos lenguajes no son normalmente utilizados en aplicaciones de red.)

Los lenguajes procedimentales (siendo el lenguaje C el prototipo de este tipo de lenguajes) utilizan procedimientos para reducir la complejidad de las tareas de la aplicación. Por ejemplo, una aplicación puede implementarse utilizando un **procedimiento** (también llamado función, aunque en algunos contextos el término *procedimiento* se utiliza para funciones que no devuelven nada) que lleve a cabo la entrada, otro procedimiento para realizar la computación, y un tercer procedimiento para generar la salida.

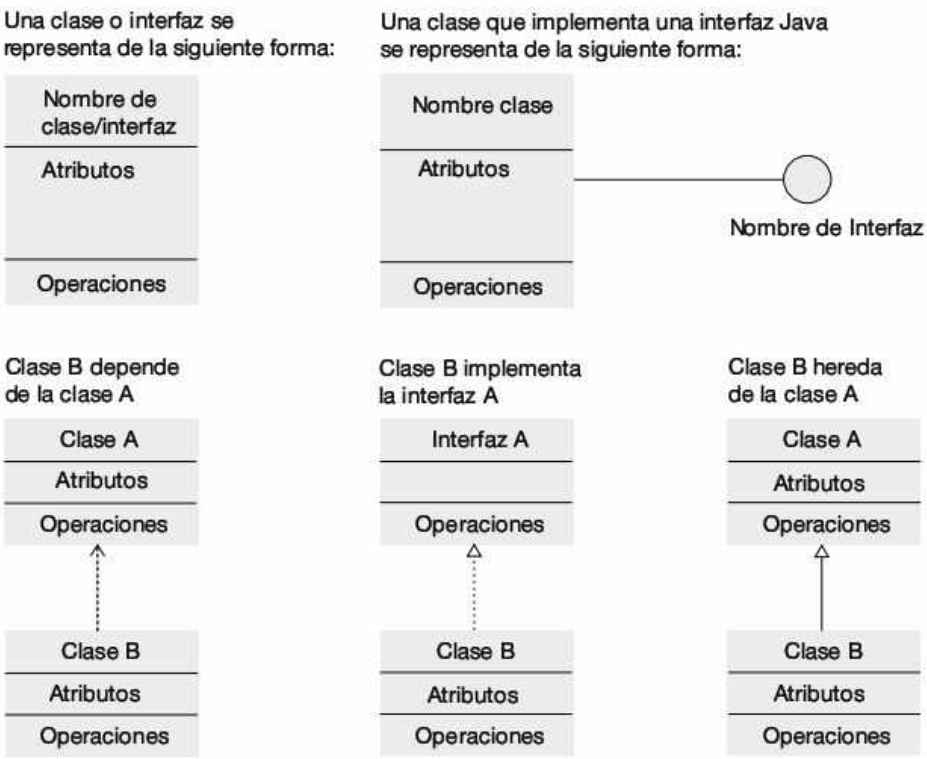
Los lenguajes orientados a objetos, como por ejemplo Java, que es el lenguaje elegido para este libro, utiliza objetos para encapsular los detalles. Cada objeto simula un objeto de la vida real, almacenando tanto los datos de estado como los diferentes comportamientos del mismo. Los datos de estado se representan como variables de instancia (en Java) o variables miembro (en C++). Los comportamientos se representan mediante los métodos.

UML

Un paso importante en la ingeniería del software es la producción de **artefactos**, o documentos, para realizar el diseño conceptual de la aplicación que se está desarrollando. Por legibilidad, estos documentos deben escribirse utilizando un conjunto de notaciones y lenguajes universales. El **lenguaje de modelado unificado (UML, *Unified Modeling Language*)**, desarrollado por la organización *Object Management Group* [omg.org, 27] es una utilidad de este tipo. UML proporciona un conjunto común de lenguaje y notaciones «para especificar, visualizar, construir y documentar los artefactos de los sistemas software» [omg.org, 27].

OMG-UML proporciona un conjunto rico de herramientas para todas las facetas de la ingeniería del software. Estas herramientas son explicadas en cursos de ingeniería del software. En este libro se utilizará ocasionalmente una de estas notaciones: los diagramas de clase de UML (y sólo un subconjunto de ellos), para docu-

mentar las relaciones de algunas de las clases Java que aparecen en la presentación. La Figura 1.19 presenta el subconjunto de diagramas de clase que se utilizará en este libro.



Nota: El estilo de las líneas y la forma de las flechas son significativas.

Figura 1.19. Un subconjunto de los diagramas de clases UML.

La arquitectura de aplicaciones distribuidas

La idea de utilizar una arquitectura multicapa para organizar las funciones de una red de datos se puede aplicar a las aplicaciones distribuidas. La Figura 1.20 presenta un ejemplo de dicha arquitectura.

Figura 1.20. Arquitectura de las aplicaciones distribuidas.

Utilizando esta arquitectura, las funciones de una aplicación distribuida se pueden clasificar en tres capas:

- La **capa de presentación** proporciona la interfaz de usuario. Por ejemplo, si la aplicación es un carrito de la compra, esta capa genera el conjunto de páginas web de la tienda correspondiente que se ven desde un navegador.
- La **capa lógica de aplicación** proporciona la computación necesaria para la aplicación. Esta capa también se llama **capa lógica de negocio** para las aplicaciones empresariales. En una aplicación de carrito de la compra, esta capa se encarga de tareas tales como la verificación de la tarjeta de crédito, calcular las cantidades correspondientes a las peticiones, calcular los impuestos de venta o el coste de la entrega.
- La **capa de servicio** proporciona los servicios necesarios para soportar las funciones de las otras dos capas. Los servicios pueden incluir utilidades de acceso a los datos (tales como un sistema de gestión de base de datos), servicios de directorio para búsquedas de nombres (tal como DNS) y comunicación entre procesos (que permita que se puedan intercambiar datos entre los procesos).

Este libro se va a centrar en la capa de servicio. Las otras dos corresponden a temas de ingeniería del software.

Conjuntos de herramientas, marcos de desarrollo y componentes

Los conjuntos de herramientas (más conocidos por su palabra inglesa *toolkits*), entornos de desarrollo (*frameworks*) y componentes son términos asociados con la ingeniería del software para sistemas empresariales (es decir, aplicaciones comerciales a gran escala).

En el contexto del desarrollo de software, un ***toolkit*** o ***framework*** es una colección de clases, herramientas y ejemplos de programación. Por ejemplo, el *toolkit* JDK (*Java Development Toolkit*) es una colección de herramientas para desarrollar programas Java, mientras que el *framework* .NET de Microsoft está orientado a la construcción de aplicaciones web. Se supone que el lector sabe desarrollar programas Java mediante JDK; otros *toolkits* para computación distribuida (por ejemplo, el *Java Socket Toolkit*) se cubrirán en otros capítulos de este libro.

El desarrollo de software basado en componentes es una técnica para la construcción de sistemas software empresariales. Utilizando esta técnica, el software se desarrolla y evoluciona mediante la unión de componentes ya probados y reutilizables. Esta técnica se basa en la reutilización del software y tiene como principal ventaja que reduce significativamente los costes y errores de desarrollo [Pour, 37]. Los entornos *Enterprise Java Bean* (EJB) y *Component Object Model* (COM) (Microsoft) son plataformas que dan soporte a aplicaciones basadas en componentes. Aunque estas plataformas son importantes para la computación distribuida empresarial, su estudio queda fuera del alcance de este libro.

RESUMEN

En este capítulo introductorio se han discutido los siguientes temas:

- Qué se entiende por computación distribuida y cómo se relaciona y diferencia de otros temas como los sistemas distribuidos y la computación paralela.

- Conceptos básicos de sistemas operativos que son importantes para el estudio realizado en este libro. Estos conceptos incluyen a los procesos y los hilos.
- Conceptos básicos de comunicación de datos relevantes para este libro. Dichos temas incluyen
  - Arquitecturas de red: el modelo OSI y el modelo de Internet
  - Comunicación orientada a conexión frente a comunicación sin conexión
  - Esquemas de nombrado para recursos de la red, incluyendo
    - — Sistema de Nombrado de Dominio (DNS)
    - — Sistema de Nombres Extensible (XNS)
    - — Números de puertos de protocolo
    - — Identificador de Recursos Uniforme (URI) y Localizador de Recursos Uniforme (URL)
    - — Dirección de correo electrónico
- Conceptos básicos de ingeniería del software que son importantes para el estudio realizado en este libro. Tales conceptos incluyen
  - Programación procedimental comparada con la programación orientada a objetos
  - Diagramas de clases que utilizan la notación del Lenguaje de Modelado Unificado (UML)
  - La arquitectura de tres capas de las aplicaciones distribuidas, que consiste en (i) la capa de presentación, (ii) la capa de aplicación o de lógica de negocio, y (iii) la capa de servicios
  - Los términos *toolkit*, *framework* y componente en el contexto de la ingeniería del software

EJERCICIOS

1. Computación distribuida
   a. Considérese la computación distribuida tal y como se ha definido en este capítulo. Para cada una de las siguientes actividades, determine y explique si es un ejemplo de computación distribuida:
      i. Utilizar Excel en un computador personal aislado
      ii. Realizar navegación web
      iii. La mensajería instantánea
      iv. Compilar y probar un programa escrito en Cobol en una máquina departamental sin conexión de red
      v. Utilizar el correo electrónico en el computador de un departamento para enviarlo a uno mismo.
      vi. Utilizar Napster.com para *descargar* música
   b. En este ejercicio se utilizará un modelo matemático simplificado para analizar los fallos de un sistema distribuido. Explique las respuestas.

Supóngase que cada computador tiene una probabilidad p de fallar cada cierto tiempo, $p<1$.

i. Si n computadores se interconectan y se necesita que todos los computadores estén disponibles para mantener un determinado servicio, dado que se utiliza un sistema distribuido que incluye a estos computadores,

   a. ¿Cuál es la probabilidad p de que el servicio no esté disponible en un determinado momento, asumiendo que ningún otro componente del sistema distribuido falle? Exprese p como un función matemática de n y p.

   b. Basándose en la respuesta del apartado a, ¿cuál es la probabilidad p cuando el sistema no es distribuido, es decir, para el caso n=1?

   c. Basándose en la respuesta del apartado a, utilice p=0.2 y n=3 para calcular la probabilidad p. ¿Cómo es esta probabilidad comparada con la probabilidad de fallo en el caso de utilizar computación monolítica, es decir, en un único computador?

ii. Ahora supóngase que el servicio proporcionado sólo requiere uno de los tres computadores, con los otros dos sirviendo como copias de respaldo o *backups* (es decir, cada uno de los tres computadores, es capaz por sí mismo de proporcionar el servicio). ¿Cuál es la probabilidad de que el servicio no esté disponible en un determinado momento, asumiendo que ningún otro componente del sistema distribuido falle? ¿Cómo es la probabilidad de fallo de este sistema comparada con la probabilidad de fallo del mismo sistema utilizando computación monolítica, es decir, un único computador?

c. Investigue el gusano de Internet [Eichin and Rochlis, 21] o un ataque de virus tal como el del virus I-Love-you [Zetter, 22] y resuma en qué consisten y cómo se originaron. ¿Por qué estos hechos son significativos en la computación distribuida? Piense en algunas medidas para evitar estos problemas.

d. Investigue en la «computación distribuida» (o, de forma más precisa, computación colaborativa) de los proyectos seti@home [setiathome.ssl.berkeley.edu, 10] y genome@home [genomeathome.stanford.edu, 23]. Escoja uno de ellos. Escriba un informe para (i) explicar el objetivo del proyecto, (ii) explicar cómo se lleva a cabo la computación en el sistema distribuido, y (iii) explicar qué hay que hacer para participar en el proyecto.

e. Investigue sobre los comienzos de Internet (*véanse* las referencias [vlmp.museophile.com, 1], [zakon.org, 2] [silkroad.com, 3], o [Hafner and Lyon, 4], por ejemplo) y escriba un informe corto sobre una de las organizaciones clave y una de las figuras prominentes en la historia de Internet.

2. Programación concurrente

   a. Busque la especificación del API de Java [java.sun.com, 20].

      Escoja el enlace de la interfaz *runnable* y a continuación la clase *thread*. Lea detenidamente cada una de ellas, leyendo las especificaciones de los métodos de cada una.

i. De acuerdo a las especificaciones, ¿cuál de las dos, la interfaz *runnable* o la clase *thread*, es preferible si sólo se pretende implementar el método *run*? ¿Por qué?

ii. ¿Qué hace el método *sleep* de la clase *thread*? Escriba la sentencia(s) Java que aparece(n) en el código para que un hilo suspenda la ejecución durante 5 segundos.

iii. ¿Qué hace el método *activeCount* de la clase *thread*? ¿Qué debería devolver el método en un programa donde se crean tres hilos?

iv. Se dice que el método *stop* de la clase *thread* es *deprecated*. ¿Qué significa que un método sea *deprecated*?

v. ¿Cuántos métodos hay en la interfaz *runnable*? Nómbrelos.

vi. ¿Cómo se utiliza la interfaz *runnable* para crear un hilo? Explíquelo.

Napster.com es un servicio de música digital. AudioGalaxy y KaZaA ofrecen servicios similares.

b. Compile y ejecute los ficheros *class* Java que se muestran en la Figura 1.11 y que se encuentran en la carpeta de ejemplos del programa. ¿Cuál es el resultado? Capture la salida de la ejecución y escriba un párrafo explicando la salida, prestando especial atención al orden de las líneas de la salida.

c. Compile y ejecute los ficheros *class* Java que se muestran en la Figura 1.12. ¿Cuál es el resultado? Capture la salida de la ejecución y escriba un párrafo explicando la salida, prestando especial atención al orden de las líneas de la salida. Además, compare la salida con la salida del apartado b (la segunda parte).

d. Considérense las siguientes clases Java:

i. ¿Cuál es la salida esperada cuando se ejecuta *EjecHilo3*? Compílelo y ejecútelo.

ii. Comente la palabra reservada *synchronized* en la cabecera del método *update*. Compílelo y ejecute de nuevo *EjecHilo3*. ¿Cuál es la salida? Explíquelo.

```
public class EjecHilo3
{
   public static void main (String[ ] args)
   {
      int numHilosOrig = Thread.activeCount();
      for (int i=0; i<10; i++) {
         Thread p = new Thread (new Thread3Ejemplo());
         p.start();
      }
      System.out.println("numero hilos ="+
Thread.activeCount());
      while (Thread.activeCount() > numHiloOrig) {
         // bucle hasta que todos los hilos hayan finalizado.
      }
System.out.println("Finalmente, numero ="+ Hilo3Ejemplo.numero);
   }
} // Fin clase EjecHilo3
```

```
class Hilo3Ejemplo implements Runnable {
     static int numero=0;

     Thread3Ejemplo() {
        super();
     }

     public void run() {
        update();
     }

     static public synchronized void update () {
        int miNumero = numero;

        int segundo = (int) (Math.random( ) * 500.0);
        try {
           Thread.sleep(segundo);
        }
        catch (InterruptedException e) {
        }
        miNumero++;
        numero = miNumero;
        System.out.println("numero="+numero+
        "; numero hilo =" + Thread.activeCount());
     }
  } // Final clase Hilo3Ejemplo
```

3. Comunicación orientada a conexión frente a comunicación sin conexión

   En este ejercicio se utilizará un modelo matemático simplificado para comparar la comunicación orientada a conexión y la comunicación sin conexión. Explique las respuestas.

   En una determinada red se proporcionan ambas formas de comunicación:

   - Utilizando comunicación orientada a conexión, establecer una conexión supone 50 segundos, después de los cuales un paquete de hasta 10 caracteres se envía en 1 segundo sobre la conexión, en cualquier dirección.
   - Utilizando comunicación sin conexión, se puede enviar un paquete de hasta 10 caracteres en 1,2 segundos (el envío de cada paquete lleva un tiempo ligeramente superior al del modelo orientado a conexión, debido a que cada paquete debe encontrar el camino al receptor).

   Supóngase que los procesos A y B intercambian mensajes en esta red. A inicia la comunicación y envía un mensaje de 100 caracteres, que se reparten en 10 paquetes. En respuesta, B envía un mensaje de 50 caracteres, repartidos en 5 paquetes.

   Asumiendo que no hay ningún retardo diferente del correspondiente al establecimiento de la conexión (en el caso orientado a conexión) y la transmisión de los paquetes:

a. ¿Cuánto tiempo dura la sesión entre A y B, utilizando comunicación orientada a conexión? Explíquelo.

b. ¿Cuánto tiempo dura la sesión entre A y B, utilizando comunicación sin conexión? Explíquelo.

c. ¿Cuántos datos (en número de caracteres) deben ser intercambiados entre A y B para que la comunicación orientada a conexión lleve a una sesión más corta que la comunicación sin conexión? Explíquelo.

4. Nombrado

a. ¿Cuál es el tamaño del espacio de direcciones (es decir, el número total de direcciones posibles) en cada una de las cinco clases de direcciones IPv4? Muestre los cálculos.

b. Descubra la IP de la dirección de red asignada a la organización del lector. ¿Qué clase de red es (A hasta E)?

c. Descubra el nombre de dominio del servidor web de la organización del lector. ¿Cuál es su dirección IP?

d. El programa de red *nslookup* se puede utilizar para obtener el servicio de búsqueda de nombres de DNS. Se puede invocar de al menos tres formas:

- En un sistema UNIX, ejecutando *nslookup* desde el intérprete de mandatos.
- En un sistemas Windows, ejecutando *nslookup* desde la ventana de intérprete de mandatos.
- Accediendo a la página http://cc-www.uia.ac.be/ds/nslookup.html.

Utilice este servicio para completar la siguiente tabla:

Dirección IP	**Nombre de dominio**
127.0.0.1	
	ifi.uio.no
	ie.technion.ac.il
204.198.135.62	
224.0.1.24	
	cse.cuhk.edu.hk
129.65.2.119	
	www.mit.edu

e. Completar la siguiente tabla:

Dirección IP	**Nombre dominio**	**Clase de dirección (A-E)**	**Ident. red (en notación decimal con puntos)**	**Ident. nodo (en notación decimal con puntos)**
18.181.0.31				
129.65.2.119				
204.198.135.62				
224.0.1.24				

f. Utilice los códigos de país correspondientes a dominios de alto nivel listados por la autoridad IANA [iana.org, 19] para encontrar los códigos de países de nombres de dominios correspondiente a los siguientes países:

   Armenia, Brasil, Canadá, Cuba, Alemania, España, Francia, Guatemala, India, Méjico, Qatar, Singapur, Suiza, El Salvador, Turquía.

   Identifique las naciones correspondientes a los siguientes códigos:

   td, tv, zw, nz, ph, pk, eg, bt, ao.

g. Considérese esta URI: http://www.algunsitio.org:8081/foo/index.html.

   i. ¿Cuál es el protocolo especificado?

   ii. ¿Cuál es el nombre del servidor?

   iii. ¿Cuál es el número de puerto del proceso que proporciona el servicio?

   iv. ¿Dónde se encuentra ubicado el documento?

h. Busque los números de puertos bien conocidos a través de la página web http://www.iana.org/assignments/port-numbers.

   i. ¿Cuál es el número de puerto asignado a cada uno de estos servicios: (i) FTP, (ii) telnet, (iii) SMTP, y (iv) ***World Wide Web*** HTTP? ¿Estos servicios están disponibles utilizando TCP, UDP, o ambos?

   ii. ¿Qué servicios están asociados a los puertos 13 y 17 respectivamente?

   iii. En un sistema UNIX, o desde una ventana de intérprete de mandatos de un sistema Windows, una forma de acceder a un servicio de red es a través del siguiente mandato:

      telnet<espacio><nombre dominio o dirección IP de un sistema conocido><espacio><número de puerto asignado al servicio>

      Por ejemplo, el mandato telnet foo.com 13 permite acceder al servicio del proceso que ejecuta en el puerto 13 del nodo de Internet foo.com.

      Utilice este método para acceder a los servicios ofrecidos por el puerto 13 de alguna máquina conocida. Describa el resultado.

i. En lugar de utilizar el esquema de Internet que usa el número de puerto del protocolo como parte de la dirección para la entrega de datos a un proceso en una determinada máquina, considérese un esquema alternativo donde el proceso se localiza utilizando un identificador de proceso único (PID), de forma que el sistema operativo UNIX se encarga de asignar dicho identificador a cada proceso activo. Obsérvese que el PID se asigna de forma dinámica a cada proceso cuando se crea, de forma que no es posible conocer a priori el identificador del proceso. Además, el rango de valores para los PID varía de un sistema a otro. ¿Cuáles son los problemas, si existen, de este esquema de direcciones?

j. Un esquema de nombrado se dice que proporciona **transparencia de localización** [community-ml.org, 25] si permite acceder a los objetos sin especificar de forma explícita su ubicación física. Por ejemplo, el sistema de numeración de teléfonos de EEUU proporciona transparencia de localización, ya que quien llama no necesita conocer donde se encuentra la persona que recibe la llamada. Por otro lado, el sistema de direcciones del servicio postal de EEUU no permite transparencia de localización, ya que es necesario conocer la dirección física del receptor (excluyendo los números de apartado de correos).

Considérese los siguientes esquemas de nombrado. Para cada uno de ellos, determine si proporcionan transparencia de localización. Justifique la respuesta.

i. El sistema de nombres de dominio (DNS)

ii. El localizador de recursos uniforme (URL)

iii. El localizador de recursos de nombre (URN)

iv. El servicio de nombres extensible (XNS)

5. Diagrama de clases UML

   a. Utilizando la notación mostrada en la Figura 1.19, dibújese el diagrama de clases de las clases de la Figura 1.11.

   b. Utilizando la notación mostrada en la Figura 1.19, dibújese el diagrama de clases de las clases de la Figura 1.12.

REFERENCIAS

1. El museo virtual de la computación, http://www.museophile.com/computing.html
2. Descripción temporal de Internet de Hobbes – la definitiva historia de ARPAnet & Internet, http://www.zakon.org/robert/internet/timeline/
3. El grupo *Silk Road, Ltd*, una breve historia de las redes, http://www.silkroad.com/net-history.html
4. Katie Hafner and Matthew Lyon. *Where Wizards Stay Up Late: The Origins of the Internet*. New York, NY: Simon & Schuster, 1996.
5. Archivos de Internet RFC/STD/FYI/BCP, http://www.faqs.org/rfcs/
6. Todd Campbell, «*The first email message*», PRETEXT Magazine, 1998. http://www.pretext.com/mar98/features/story2.htm
7. http://www.sun.com/jini/overview/, Septiembre 2000.
8. webopedia, http://webopedia.internet.com
9. Alice E. Koniges. *Industrial Strength Parallel Computing*. San Francisco, CA: Morgan Kaufman Publishers, 2001.
10. SETI@home, *the Search for Extraterrestrial Intelligence*, http://setiathome.ssl.berkeley.edu/
11. Recursos Java, http://www.csc.calpoly.edu/~mliu/javaResources.html
12. William Stallings. *Data and Computer Communications*. Upper Saddle River, NJ: Prentice Hall, 1999.
13. Andrew S. Tanenbaum. *Computer Networks*. Upper Saddle River, NJ: Prentice Hall, 1996.
14. Chuck Semeria, 3Com Corporation, *Understanding IP Addressing: Everything You Ever Wanted To Know*, 1996, http://www.3com.com/other/pdfs/infra/corpinfo/en_US/501302.pdf
15. Marshall Brain, Cuestión del día, http://www.howstuffworks.com/question549.htm, *HowStuffWorks*
16. Acerca de dominios, *about domains*, http://www.aboutdomains.com/News/basics.htm
17. El Centro Nacional de Aplicaciones de Supercomputación (NCSA). Una guía para principiantes sobre URL, http://archive.ncsa.uiuc.edu/SDG/Experimental/demoweb/url-primer.html
18. Dan Connolly. Nombrado y direcciones: URI, URL, ..., http://www.w3.org/Addressing/
19. *Internet Assigned Number Authority*. Códigos de países de los dominios de alto nivel, http://www.iana.org/cctld/cctld.htm
20. Especificación del API de la plataforma Java 2 v1.4, http://java.sun.com/j2se/1.4/dos/api/index.html

21. Mark W. Eichin and Jon A. Rochlis, Massachusetts Institute of Technology. ***With Microscopes and Tweezers: An Analysis of the Internet Virus of 1988***, http://www.mit.edu/people/eichin/virus/main.html
22. Kim Zetter. PCWorld.com, «***When Love Came to Town: A Virus Investigation***», 13 Noviembre, 2000.
23. ***The*** genome@home ***project homepage***, http://genomeathome.stanford.edu/
24. ***Internet Assigned Numbers Authority***, http://www.iana.org/
25. Modelo de referencia para ODP (***Open Distributed Processing***) (RM-ODP), ISO/IEC IS 10746|ITU-T X.900, http://community-ml.org/RM-ODP/
26. XNS – ***frequently asked questions***, http://www.xns.org/, XNSORG.
27. ¿Qué es OMG UML? http://www.omg.org/gettingstarted/what_is_uml.htm
28. Guía de notación de UML – Versión 1.1, http://www.informatik.fh-luebeck.de/~st/UML/UML1.1/
29. Página inicial de Unicode, http://www.unicode.org/
30. Michael Fischer, Nancy Lynch, and Michael S. Paterson. «***Impossibility of distributed consensus with one faulty process***». ***Proceedings of the 2nd ACM Symposium on Principles of Database Systems***, páginas 1-7, 1983.
31. Pankaj Palote. ***Fault Tolerance in Distributed Systems***. Upper Saddle River, NJ: Prentice Hall, 1994.
32. Scott Oaks. ***Java Security***. Sebastopol, CA: O'Reilly Press, 2001.
33. ***distributed.net: Node Zero***, http://www.distributed.net/
34. ***Internet Security Alliance***, http://www.isalliance.org/
35. Andrew Tanenbaum. ***Computer Networks***. Uppers Saddle River, NJ: Prentice Hall, 1996.
36. http://www.iana.org/assignments/port-numbers
37. Gilda Pour. «***Web-Based Architecture for Component-Based Application Generators***». ***The International Multiconference in Computer Science***, Las Vegas, Nevada, 2002.
38. ***Internet Society*** (ISOC), Todo sobre Internet: Historia de Internet, http://www.isoc.org/internet/history/

CAPÍTULO 2

IPC - Comunicación entre procesos

Un paradigma es un modelo abstracto de cómo se realiza una determinada tarea.

La espina dorsal de los sistemas distribuidos son los mecanismos de **comunicación entre procesos** (*interprocess communication* o IPC): la posibilidad de que procesos separados e independientes (como podrá recordar el lector, los procesos son representaciones en tiempo de ejecución de programas) se comuniquen entre sí para colaborar en una tarea. En este capítulo, se van a ver los fundamentos, características, paradigmas e implementaciones de los mecanismos de comunicación entre procesos.

La Figura 2.1 ilustra un mecanismo básico de comunicación entre procesos: dos procesos independientes, con la posibilidad ejecutarse en máquinas separadas, intercambian datos sobre una red de comunicaciones. En este caso, el proceso 1 actúa como **emisor**, que transmite datos al proceso 2, el **receptor**.

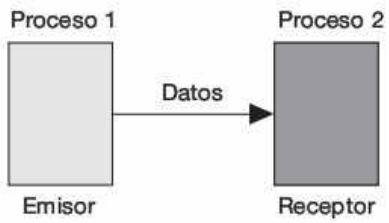


Figura 2.1. Comunicación entre procesos.

En sistemas distribuidos, dos o más procesos establecen una comunicación entre ellos por medio de un protocolo (un conjunto de reglas que deben ser observadas por los participantes en la

comunicación de los datos) acordado por los procesos. Un proceso puede ser emisor en determinados puntos durante el protocolo y receptor en otros. Cuando la comunicación es desde un proceso a únicamente otro proceso, el modelo de comunicación entre procesos se dice que es **unidifusión** o ***unicast***. Cuando la comunicación es desde un proceso con un grupo de procesos, el mecanismo de comunicación se denomina **multidifusión** o ***multicast***, que será tratado en el Capítulo 6. La Figura 2.2 ilustra el concepto de estos dos tipos de IPC.

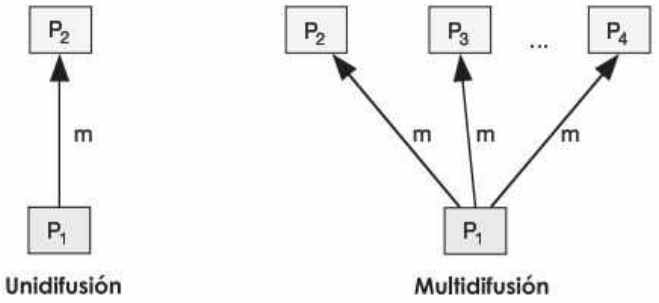


Figura 2.2. Unicast y multicast.

Los sistemas operativos actuales, como UNIX y Windows proporcionan funcionalidades para la comunicación entre procesos. Llamaremos a estas funcionalidades mecanismos de comunicación entre procesos a nivel de sistema operativo, para distinguirlos de los mecanismos de comunicación de alto nivel. Los mecanismos de comunicación a nivel de sistema operativo incluyen colas de mensajes, semáforos, y regiones de memoria compartida. (Si el lector no ha realizado un curso de sistemas operativos, no se preocupe si no le resultan familiares estos términos; no van a ser casos de estudio en este libro). Es posible desarrollar software de red usando directamente estas funcionalidades a nivel de sistema operativo. Ejemplos de estos programas son los manejadores (*drivers*) de red y los programas de evaluación de prestaciones.

En ingeniería del software, se denomina **abstracción** a un mecanismo para ocultar las complejidades internas de una tarea. Por ejemplo los lenguajes de alto nivel, como Java, proporcionan una abstracción que permite al programador tener que comprender los detalles al nivel del sistema operativo.

Se pueden desarrollar también aplicaciones distribuidas muy rudimentarias aunque normalmente no se hace, debido a que la complejidad típica de estas aplicaciones requiere el uso de algún tipo de **abstracción** para separar al programador de los detalles a nivel de sistema operativo. El lector puede encontrar más información sobre mecanismos de comunicación a nivel de sistema operativo en libros sobre dichos sistemas operativos.

2.1. UN ARQUETIPO DE INTERFAZ DE PROGRAMACIÓN PARA COMUNICACIÓN ENTRE PROCESOS

A continuación se va a presentar una interfaz de programación que proporciona el mínimo nivel de abstracción para facilitar la comunicación entre procesos. Para ello se necesitan cuatro operaciones primitivas básicas. Los detalles acerca de estas operaciones (tales como los argumentos y los valores devueltos) se verán cuando se comenten herramientas y funcionalidades específicas a lo largo de los siguientes capítulos. Estas operaciones son:

- **Enviar**. Esta operación se invoca por el proceso emisor con el propósito de transmitir datos al proceso receptor. La operación debe permitir al proceso emisor identificar al proceso receptor y especificar los datos a transmitir
- **Recibir**. Esta operación es invocada por el proceso receptor con el objetivo de aceptar datos de un proceso emisor. La operación debe permitir al proceso receptor identificar al proceso emisor así como especificar el área de memoria que permitirá almacenar el mensaje, que posteriormente será accedida por el receptor.
- **Conectar**. Para mecanismos de comunicación orientados a conexión deben existir operaciones que permitan establecer una conexión lógica entre el proceso que lo invoca y otro proceso determinado: un proceso invoca una operación de **solicitar-conexión** (denominada **conectar** en adelante) mientras que el otro proceso solicita la operación de **aceptar-conexión**.
- **Desconectar**. Para mecanismos de comunicación orientados a conexión, esta operación permite que una conexión lógica, previamente establecida, sea liberada en ambos extremos de la comunicación.

Un proceso involucrado en un mecanismo de comunicación invoca estas operaciones en un orden determinado. La invocación de cada operación implica la ocurrencia de un **evento**. Por ejemplo, una operación de enviar solicitada por el proceso emisor desemboca en el evento de transmisión de datos hacia el proceso receptor, mientras tanto la invocación de la operación recibir por parte del proceso receptor implica que dichos datos sean entregados al proceso. Es necesario indicar que los procesos participantes invocan operaciones de forma independiente, ya que no hay forma de conocer el estado del otro proceso.

Todos los paradigmas de sistemas distribuidos que se van a ver en este libro proporcionan, implícita o explícitamente, operaciones de comunicación entre procesos. El siguiente capítulo (Capítulo 3) presentarán una jerarquía de los paradigmas de sistemas distribuidos. En los siguientes capítulos, se verán ejemplos de cómo se utilizan estos paradigmas en protocolos, herramientas y funcionalidades.

Los protocolos de servicio de red pueden ser implementados usando operaciones de comunicación entre procesos primitivas. Por ejemplo, en el protocolo HTTP básico (*HyperText Transfer Protocol*, protocolo de transferencia de hipertexto, usado de forma extensiva en el entorno web, que será estudiado en próximos capítulos) un proceso, el navegador web, invoca una operación *conectar* para establecer una conexión lógica con otro proceso, el servidor web, seguido de una operación *enviar* hacia el servidor web, para transmitir los datos que representan una solicitud. El servidor web como respuesta invoca una operación enviar para transmitir los datos solicitados por el navegador web. Al finalizar la comunicación, cada proceso invoca su operación *desconectar* para finalizar con la conexión. La Figura 2.3 ilustra la sequencia de opera-

ciones. Se verán protocolos de servicio de red como HTTP más adelante en este y otros capítulos.

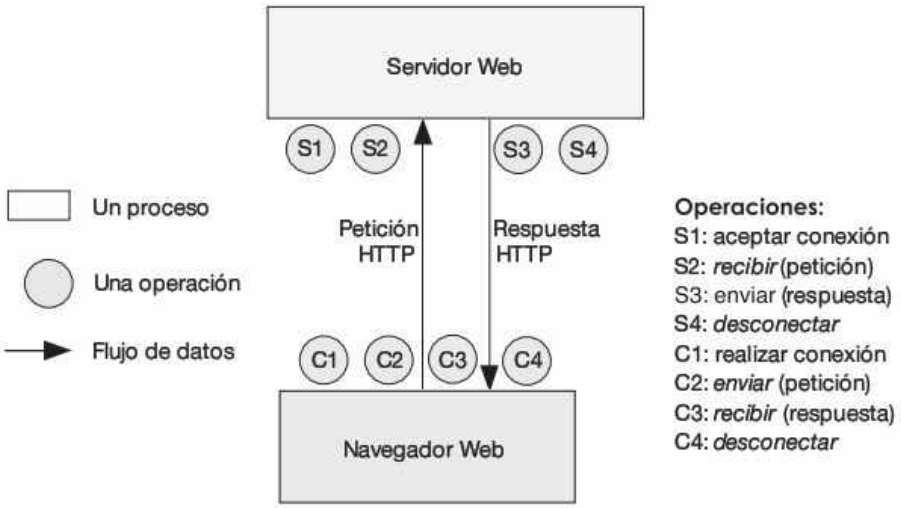


Figura 2.3. Comunicación entre procesos en HTTP básico.

En el resto de este capítulo se van a tratar determinados aspectos clave de estas operaciones de comunicación entre procesos.

2.2. SINCRONIZACIÓN DE EVENTOS

Una de las mayores dificultades cuando se trabaja con mecanismos de comunicación entre procesos es que cada proceso involucrado ejecuta de forma independiente sin que ninguno de ellos sepa qué ocurre en el proceso en el otro extremo.

Si tenemos en cuenta el caso del protocolo básico HTTP, tal y como lo hemos descrito antes, como se puede observar, los dos extremos involucrados en el protocolo invocan operaciones de comunicación en un orden determinado. Por ejemplo, el proceso del navegador web no debe invocar ninguna operación *enviar* antes de haber completado la operación *conectar*. También es importante que el servidor web no empiece a transmitir datos antes de que el proceso del navegador web esté preparado. Aún más, el proceso navegador necesita conocer cuándo los datos solicitados se han transmitido, de tal manera que el subsiguiente procesamiento de los mismos tenga lugar, incluyendo el dar formato y mostrar los contenidos al usuario.

La forma más sencilla que tiene un mecanismo de comunicación de procesos para proporcionar sincronización de eventos es por medio de peticiones **bloqueantes**, que es la supresión de la ejecución del proceso hasta que la operación invocada haya finalizado.

Para ilustrar el uso de las llamadas bloqueantes en la sincronización de eventos, se va a considerar de nuevo el caso del protocolo HTTP básico. Un proceso de na-

vegador web invoca una operación bloqueante para *conectar*, que bloquea su posterior ejecución hasta que la conexión haya sido aceptada por el lado servidor. Posteriormente, el proceso navegador invoca una operación *recibir* bloqueante, la cual suspende la ejecución del proceso hasta que la operación se haya completado (con éxito o no). La opción bloqueante o no bloqueante se realiza a nivel del sistema operativo y se inicia por medio de las funcionalidades proporcionadas por el mecanismo de comunicación entre procesos, no por el programador. Los programas de los dos procesos se muestran en la Figura 2.4.

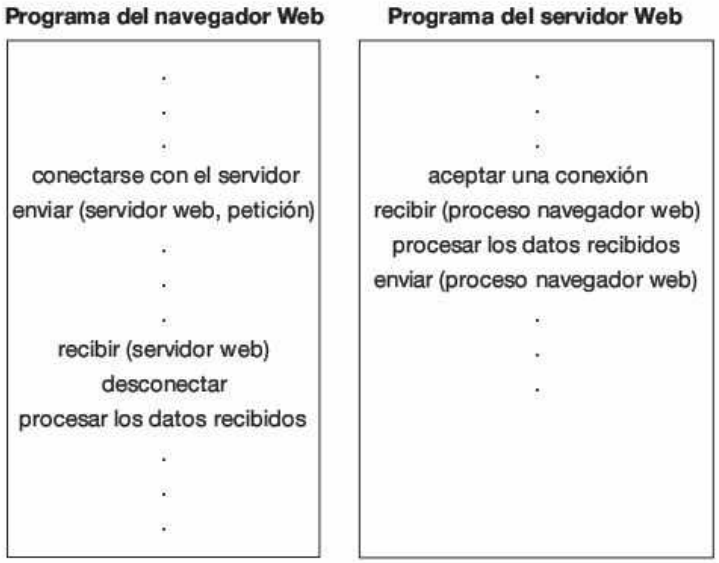


Figura 2.4. Flujo de ejecución de dos programas que intervienen en una comunicación entre procesos.

Durante su ejecución, el proceso se suspende después de que se invoque cada llamada bloqueante. Cada vez que se ejecuta una invocación a una operación bloqueante, la condición de bloqueo se inicia por las funcionalidades de comunicación entre procesos en conjunto con el sistema operativo sobre el que se apoya. La condición de bloqueo termina posteriormente cuando la operación ha sido completada, en dicho instante se dice que el proceso se encuentra *desbloqueado*. Un proceso desbloqueado transita al estado de listo y en su momento continuará la ejecución. En el caso de que la operación no pueda ser completada, un proceso bloqueado sufrirá un **bloqueo indefinido**, durante el cual el proceso permanecerá en el estado de bloqueado indefinidamente, a menos que se tomen las medidas de intervención apropiadas.

Las operaciones bloqueantes a menudo se llaman **operaciones síncronas**. Como alternativa, las operaciones de comunicación entre procesos también pueden ser **asíncronas** o **no bloqueantes**. Una operación asíncrona invocada por un proceso no causará bloqueo, y por consiguiente el proceso es libre de continuar con su ejecución una vez que se invoca al mecanismo de comunicación para realizar la operación asíncrona. Se informará posteriormente al proceso si la operación se ha completado y si lo ha sido con éxito o no.

Un proceso puede invocar una operación no bloqueante cuando el proceso puede continuar sin esperar a que se complete el evento iniciado por la operación. Por ejemplo, la operación *recibir* invocada por el navegador web debe esperar a la respuesta del servidor para poder continuar con el procesamiento. Por el contrario, la operación *enviar* invocada por el servidor web puede ser no bloqueante, porque el servidor web no necesita esperar a que la operación se haya completado antes de proceder con la siguiente operación (la operación *desconectar*), de forma que puede continuar sirviendo a otros procesos navegadores.

Es responsabilidad del programador reconocer la necesidad de sincronización y de cuándo una llamada es bloqueante. Por ejemplo, si se invoca una operación *recibir* no bloqueante en el código del navegador web, debido a un programador descuidado, se asume que los datos se reciben inmediatamente después de que se invoque la operación, el procesamiento posterior puede mostrar datos que no sean válidos o, peor aún, generar errores.

Debido a que el uso de operaciones *enviar* y *recibir* es fundamental para los sistemas distribuidos vamos a ver diferentes escenarios en los cuales se usan diferentes combinaciones de estos modos.

Enviar síncrono y *recibir* síncrono

La Figura 2.5 es un diagrama, que a lo largo de este libro vamos a denominar **diagrama de eventos**, que ilustra la sincronización de los eventos para una sesión de un protocolo implementada por medio de una operación *enviar* y una *recibir* síncronas. En este escenario, la invocación de la operación *recibir* causa la suspensión del proceso que la invoca (proceso 2) hasta que se reciben los datos y se completa la operación. De la misma forma, la operación enviar invocada causa que el proceso emisor (proceso 1) se suspenda. Cuando se reciben los datos enviados al proceso 2, el

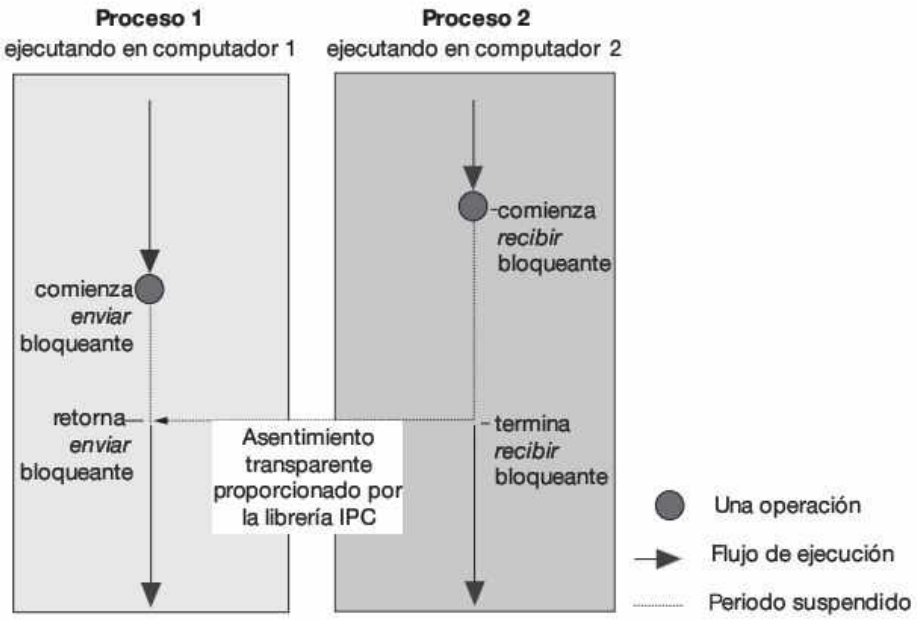


Figura 2.5. *Enviar* y *recibir* síncronos.

mecanismo de comunicación entre procesos en el ordenador 2 envía un asentimiento al mecanismo de comunicación análogo del ordenador 1, y el proceso 1 puede posteriormente desbloquearse. Es necesario remarcar que el asentimiento se gestiona por medio de los mecanismos de comunicación de ambos computadores y que es transparente para el proceso.

El uso de un enviar síncrono y un recibir síncrono es aconsejable cuando la lógica de la aplicación de ambos procesos necesita que los datos enviados se reciban antes de continuar con el procesamiento.

Dependiendo de la implementación de las funcionalidades de comunicación entre procesos, la operación recibir síncrona puede no completarse hasta que la cantidad de datos esperada por el receptor haya llegado. Por ejemplo, si el proceso 2 invoca una operación de recibir para 300 bytes de datos, y la operación enviar sólo transmite 200 bytes, es posible que el bloqueo del proceso 2 continúe incluso después de haberse entregado los primeros 200 bytes; en este caso, el proceso 2 no se desbloqueará hasta que el proceso 1 transmita posteriormente los 100 bytes de datos restantes.

Enviar asíncrono y *recibir* síncrono

En la Figura 2.6 se ilustra un diagrama de eventos para una sesión de protocolo implementada usando una operación de *enviar* asíncrona y una operación de *recibir* síncrona. Como antes, la operación *recibir* bloquea al proceso que la invoca hasta que lleguen datos para completar la operación. Sin embargo, la operación *enviar* invocada no va a causar que el proceso emisor se suspenda. En este caso, como el proceso emisor no se bloquea no resulta necesario un asentimiento por parte del mecanismo de comunicación en el ordenador del proceso 2. Este uso de una operación de *enviar* asíncrona y una de *recibir* síncrona es apropiado cuando la lógica de la aplicación emisora no depende de la recepción de los datos por parte del otro extremo. Sin em-

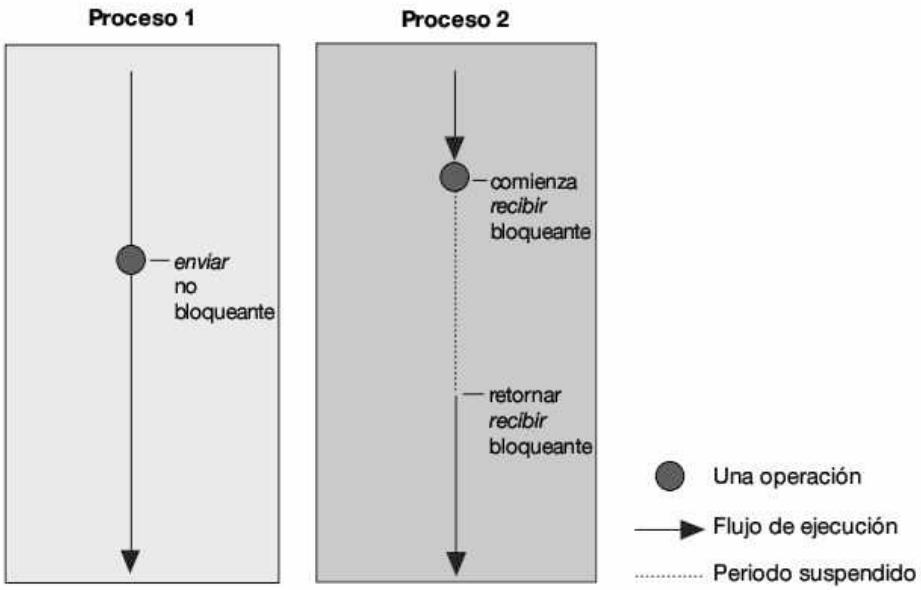


Figura 2.6. *Enviar* asíncrono y *recibir* síncrono.

bargo, dependiendo de la implementación del mecanismo de comunicación, no se garantiza que los datos enviados, realmente sean entregados al receptor. Por ejemplo, si la operación *enviar* es ejecutada antes de que la correspondiente de *recibir* sea invocada en el otro extremo, es posible que los datos no se entreguen al proceso receptor, a menos que el mecanismo de comunicación proporcione espacio para almacenar datos enviados de forma prematura.

Enviar síncrono y *recibir* asíncrono

La Figura 2.7 ilustra los diferentes escenarios de un protocolo de sesión que emplee operaciones de *enviar* síncrono y de *recibir* asíncrono.

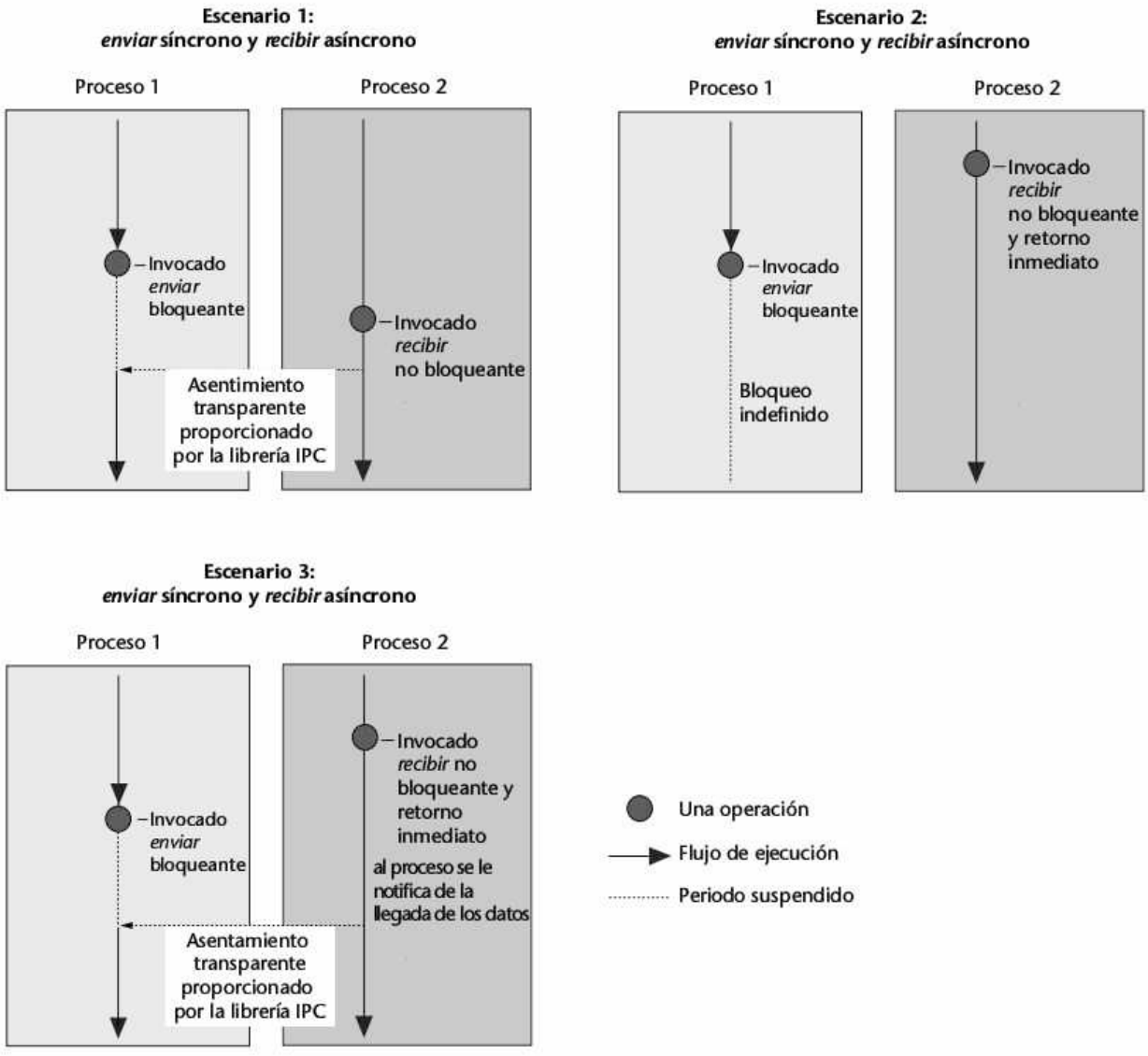


Figura 2.7. *Enviar* síncrono y *recibir* asíncrono.

Una operación de *recibir* asíncrono causa que el proceso que la invoca no se bloquee, y el comportamiento dependerá mucho de cómo se encuentre implementado el mecanismo de comunicación. La operación recibir, en todos los casos, retornará inmediatamente, existiendo tres posibles escenarios de qué ocurriría a continuación:

- *Escenario 1.* Los datos solicitados por la operación del receptor ya han llegado en el momento que la operación *recibir* se invoca. En este caso los datos se entregan al proceso 2 inmediatamente, y el proceso 1 se desbloqueará por medio de un asentamiento transmitido por el mecanismo de comunicación del ordenador 2.
- *Escenario 2.* Los datos solicitados por la operación recibir no han llegado todavía; el proceso receptor no recoge ningún dato. Es responsabilidad del proceso receptor cerciorarse de que los datos se han recibido, si es necesario, repetir el proceso hasta que los datos hayan llegado. (*Nótese* que es común que el programa utilice un bucle para invocar a la operación *recibir* repetidas veces hasta que el dato esperado llegue. Este técnica de repetir intentos se denomina **muestreo**, en inglés *polling*.) El proceso 1 se queda bloqueado de forma indefinida hasta que el proceso 2 vuelve a invocar a *recibir* y el asentimiento finalmente le llega por parte del mecanismo de comunicación del ordenador 2.
- *Escenario 3.* Los datos solicitados por la operación *recibir* no han llegado aún. El mecanismo de comunicación del ordenador 2 notificará a dicho proceso cuando los datos solicitados hayan llegado, en dicho instante el proceso 2 puede pasar a procesarlos. Este escenario requiere que el proceso 2 proporcione un manejador de evento que puede invocarse por parte del mecanismo de comunicación para notificar al proceso que los datos esperados han llegado.

Enviar asíncrono y *recibir* asíncrono

Sin ningún bloqueo en cualquiera de los dos lados, la única forma en que los datos puede entregarse al receptor es que el mecanismo de comunicación pueda almacenar los datos recibidos. El proceso receptor puede ser notificado de la llegada de los datos (*véase* la Figura 2.8). Otra alternativa es, que el proceso receptor haga muestreo en busca de la llegada de datos, para su posterior procesamiento.

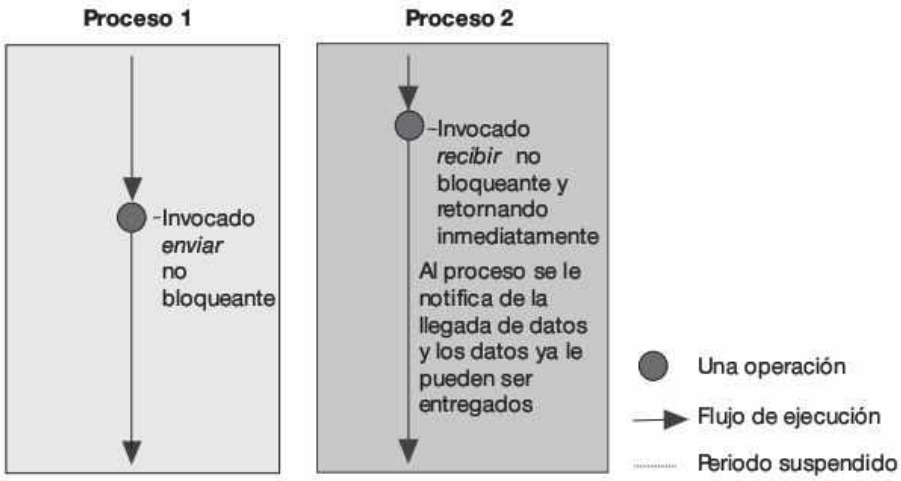


Figura 2.8. *Enviar* asíncrono y *recibir* asíncrono.

2.3. TEMPORIZADORES E HILOS DE EJECUCIÓN

Aunque el bloqueo proporciona la sincronización necesaria para los mecanismos de comunicación, es por lo general inaceptable que un proceso se quede suspendido de forma indefinida. Existen dos medidas para solucionar este problema. La primera es el uso de **temporizadores** (*timeouts*), que se pueden utilizar para fijar el tiempo máximo de bloqueo. Los temporizadores los proporciona el propio mecanismo de comunicación y pueden ser fijados desde el programa por medio de una operación. En segundo lugar, un proceso puede lanzar otro **proceso hijo** o un **hilo de ejecución** (*thread*) independiente para invocar la operación bloqueante, permitiendo de esta manera al hilo de ejecución principal o al proceso padre del programa seguir ejecutando otras tareas de procesamiento mientras el hilo de ejecución o proceso hijo se suspende. La Figura 2.9 ilustra este uso de los hilos de ejecución.

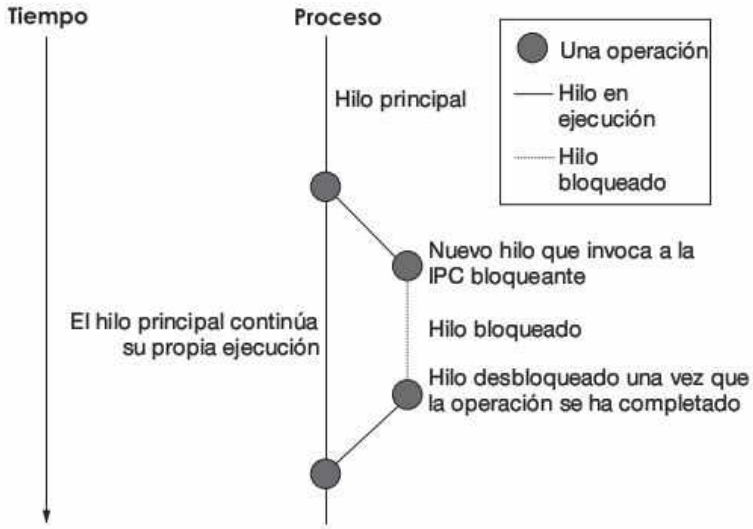


Figura 2.9. Utilización de un hilo de ejecución para una operación bloqueante.

Los temporizadores son importantes si la ejecución de operaciones síncronas puede potencialmente dar como resultado un **bloqueo indefinido**. Por ejemplo, una operación bloqueante de *conectar* puede causar que el proceso que la solicita se quede suspendido de forma indefinida si la conexión no está establecida y no se puede establecer debido a una caída en la red de interconexión entre ambos ordenadores. En esta situación, es típicamente inaceptable para el proceso solicitante quedarse «colgado» de forma indefinida. Los bloqueos indefinidos pueden resolverse usando temporizadores. Por ejemplo, se puede fijar un temporizador de 30 segundos para la operación de *conectar*. Si la petición no se completa en aproximadamente 30 segundos, el mecanismo de comunicación la abortará, en dicho instante el proceso que la solicitó se desbloqueará, pudiendo así continuar con su procesamiento.

2.4. INTERBLOQUEOS Y TEMPORIZADORES

Otra causa para sufrir un bloqueo indefinido son los **interbloqueos** (*deadlocks*). En comunicación entre procesos, un interbloqueo puede causarse por una operación in-

vocada de forma no apropiada, quizás por culpa de una mala interpretación del protocolo o por errores de programación. La Figura 2.10 muestra este caso. El proceso 1 ha invocado una operación de *recibir* para recoger datos del proceso 2. A la vez, el proceso 2 ha invocado otro *recibir* bloqueante cuando debería ser una operación de *enviar* lo apropiado. Como resultado, ambos procesos se encuentran bloqueados a esperas de datos del otro, cosa que nunca puede ocurrir (debido a que cada proceso se encuentra bloqueado). En resumen, cada proceso por su parte se encontrará suspendido indefinidamente hasta que salte un temporizador o hasta que el sistema operativo aborte el proceso.

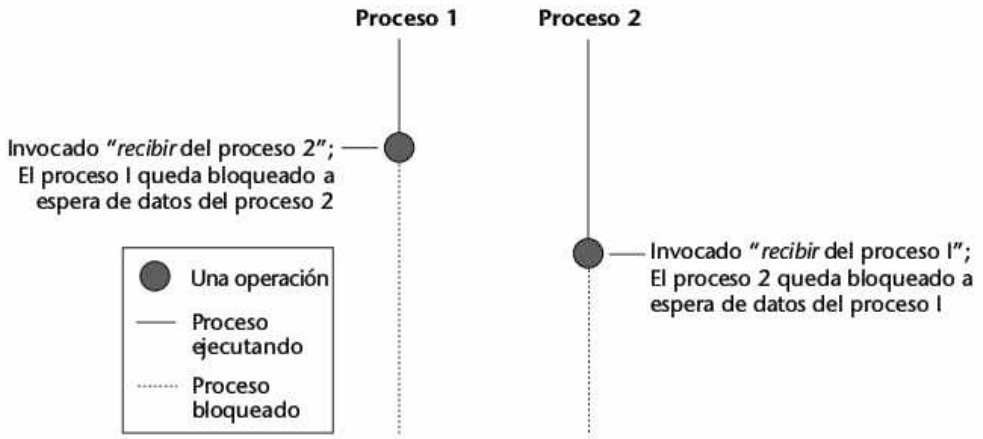


Figura 2.10. Un interbloqueo causado por operaciones bloqueantes.

2.5. REPRESENTACIÓN DE DATOS

En el nivel físico de una arquitectura de red (que es el nivel más bajo, en oposición al nivel de aplicación que es el más alto), los datos se transmiten como señales **analógicas**, las cuales representan un **flujo binario**. En el nivel de aplicación, se necesita una representación más compleja de los datos transmitidos con el objeto de dar soporte a la representación de tipos de datos y estructuras proporcionadas por los lenguajes de programación, tales como cadenas de caracteres, enteros, valores en coma flotante, vectores, registros y objetos.

Analógico es lo opuesto a digital: hace referencia a algo mecánico, en oposición a algo que representa datos. El procesamiento de señales analógicas es un área de trabajo dentro de redes de computadores.

Si consideramos el caso simple de dos procesos, proceso 1 en el ordenador A y proceso 2 en el ordenador B, que participan en un protocolo que implica el intercambio de un valor entero durante su ejecución. El proceso 1 calcula el valor e invoca una operación *enviar* para mandar el valor al proceso 2, el cual invoca una operación *recibir* para recoger dicho valor, para que el proceso 2 realice las correspondientes operaciones con dicho valor, todo de acuerdo al protocolo establecido.

Si nos centramos en el valor entero que el proceso 1 tiene que enviar. El valor entero se encuentra representado en el formato de representación del ordenador A, que es una máquina de 32-bits que utiliza representación *big-endian* para tipos de datos de varios bytes. (Los términos *big-endian* y *little-endian* indican cuál es el byte más significativo en representaciones de valores de varios bytes de tamaño. En arquitec-

Un **flujo binario** es un flujo de bits (0 y 1), tal como 00101...1010111.

turas de tipo *big-endian*, el byte más a la izquierda [el de menor dirección] es el más significativo. En arquitecturas, *little-endian* es el byte más a la derecha el más significativo).

El computador B, por su parte, es una máquina de 16-bits con representación de datos *little-endian*. Supóngase que el dato de 32 bits es transmitido directamente desde el espacio de almacenamiento del proceso 1 al espacio reservado por el proceso 2. Entonces (1) 16 bits de los 32 enviados van a ser truncados ya que el tamaño de un entero en el computador B es de sólo 16 bits y (2) el orden de los bytes de la representación de los enteros debe ser intercambiado de forma que se interprete correctamente el mismo valor por parte del proceso receptor.

Como se ha visto en el ejemplo, cuando computadores heterogéneos participan en una comunicación entre procesos, no basta con transmitir los valores de los datos o las estructuras usando flujos de bits en crudo a menos que los procesos participantes tomen las medidas oportunas para empaquetar e interpretar los datos de forma apropiada. Para nuestro ejemplo hay tres esquemas para hacer esto:

Se denominan **computadores heterogéneos** a aquellos computadores que disponen de diferente hardware y por tanto de diferentes representaciones de datos.

1. Antes de invocar a la operación *enviar*, el proceso 1 convierte el valor entero a 16-bits en formato *little-endian* para el proceso 2.
2. El proceso 1 envía los datos en 32-bits y representación *big-endian*. Tras recibir los datos, el proceso 2 convierte el valor a su formato, 16-bits y *little-endian*.
3. Un tercer esquema es que los procesos cuando intercambien datos lo hagan en una **representación externa**: los datos se van a enviar transformados a esa representación y los datos recibidos se van a interpretar con esa representación y se van a traducir a la representación nativa.

Tomando otro ejemplo, supongamos que el proceso 1, ejecutándose en el ordenador A, desea enviar un simple carácter *a* al proceso 2, ejecutándose en el ordenador B. El programa para el proceso 1 usa ASCII como representación de caracteres, mientras que el programa del proceso 2 usa Unicode. El esquema 1 obligará al proceso 1 a convertir el carácter *a* a Unicode antes de enviarlo. El esquema 2 requerirá que el proceso 2 reciba el dato, y lo convierta de ASCII a la representación Unicode correspondiente. El esquema 3 exigirá que ambos extremos se pongan de acuerdo en una representación externa, digamos ASN.1 (*Abstract Syntax Notation Number 1*), de forma que el proceso 1 convierta el carácter *a* a ASN.1 antes de mandarlo, y el proceso 2 convierta el dato recibido de ASN.1 a la representación Unicode. (La representación ASN.1 se explicará en la Sección 2.6.)

ASCII son las siglas de *American Standard Code for Information Interchange* (Código estándar americano para intercambio de información), y es un esquema de codificación usado para caracteres del alfabeto inglés usando un rango de valores de 0 a 127.

Unicode es un esquema de codificación complejo que permite traducir caracteres, no sólo del alfabeto inglés, a valores numéricos entre 0 y 65535. Para una definición más precisa, véase http://www.unicode.org.

Si se considera otro caso más cuando se transmite una estructura de datos, como por ejemplo una lista de valores, además de lo necesario de representación externa de los valores de datos, existe en este caso la necesidad de «aplanar» o serializar la estructura de datos en el extremo del emisor y deshacer el cambio en el otro extremo, para así reconstruir los datos.

El término de **empaquetamiento de datos** (*data marshaling*) se usa en el contexto de los mecanismos de comunicación entre procesos para referirse a las transformaciones necesarias para transmitir valores de datos o estructuras. El empaquetamiento de datos se necesita para todos los mecanismos de comunicación e incluye los pasos necesarios para acondicionar los datos para ser transmitidos: (1) serialización de las estructuras de datos, y (2) conversión de los valores de datos a las representaciones externas. La Figura 2.11 muestra el concepto de empaquetamiento de datos.

Para aplicaciones de red escritas en lenguajes orientados a objetos como Java, unas estructuras de datos que requieren especial atención en lo referente al empaquetamiento

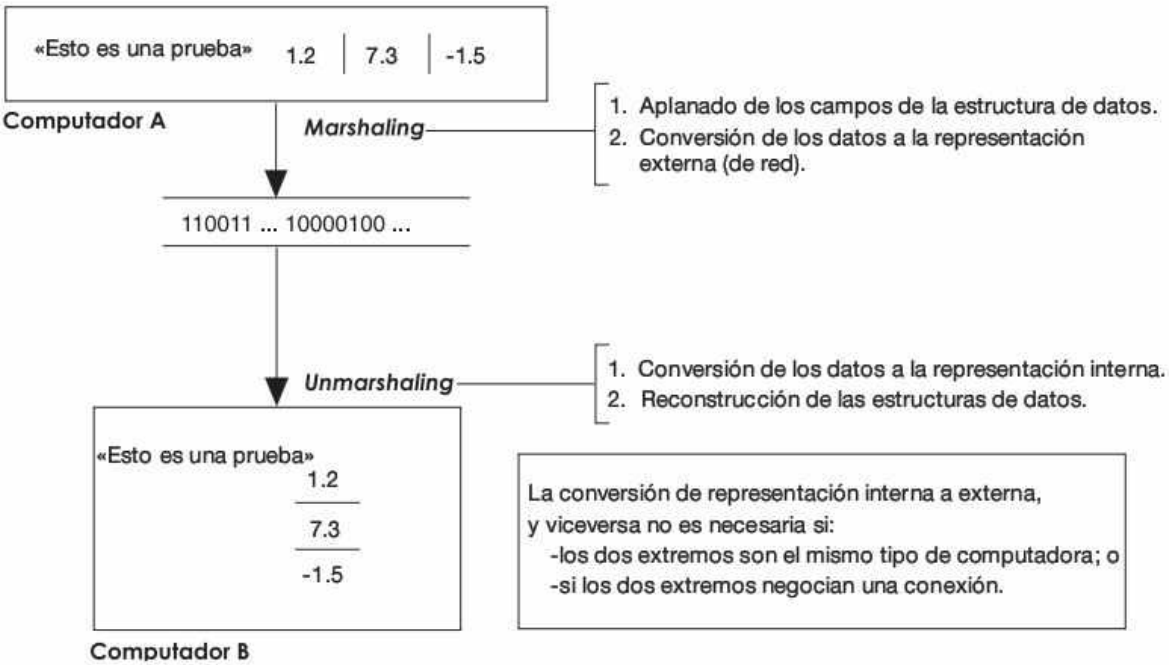


Figura 2.11. Empaquetamiento de datos.

de datos son los propios objetos. A diferencia de estructuras de datos estáticas como vectores o registros, un objeto encapsula tantos datos (representando el estado del objeto), como métodos (representando el comportamiento del objeto). Si se va a transmitir un objeto usando un mecanismo de comunicación entre procesos es necesario que el proceso de empaquetamiento (de nuevo aplanado y codificación) cubra tanto a los datos como a la representación de métodos, incluyendo el estado de ejecución, de forma que el objeto una vez desempaquetado por el proceso receptor pueda funcionar como un objeto ejecutando en el espacio de dicho proceso. Debido a la complejidad existente, el empaquetado de objetos implica un mayor reto que el del resto de estructuras, se le ha dado un nombre específico: **serialización de objetos** [java.sun.com, 11]. En Java, «La serialización de objetos soporta la codificación de objetos, y de los objetos alcanzables desde ellos, en un flujo de bytes, así como el soporte complementario para su reconstrucción en un objeto ... desde el flujo de bytes» [Harold,12].

2.6. CODIFICACIÓN DE DATOS

Aunque determinados programas especializados puedan escribirse para realizar la comunicación entre procesos usando un esquema de representación determinado de mutuo acuerdo, las aplicaciones distribuidas de propósito general necesitan un esquema universal, e independiente de plataforma, para codificar el intercambio de datos. Por esto se han definido estándares de red para la codificación de datos.

Como se muestra en la Figura 2.12, hay estándares para la codificación de datos disponibles a diferentes niveles de abstracción.

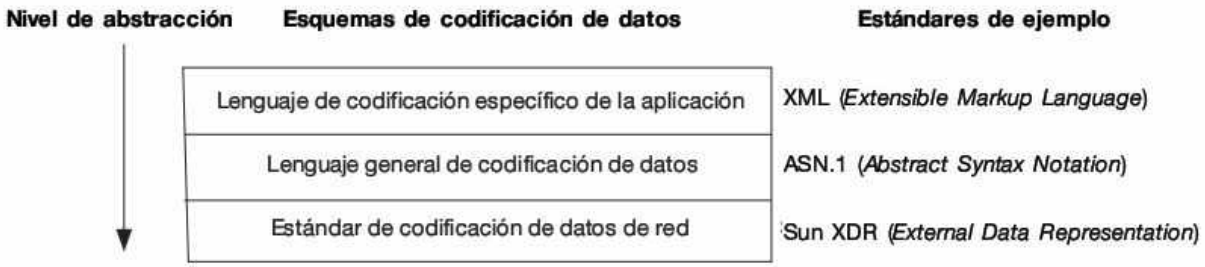


Figura 2.12. Estándares de representación de datos de red.

Al nivel más simple, un esquema de codificación como **XDR** (*External Data Representation*) [ietf.org, 1] permite que un conjunto de tipos de datos determinado y unas estructuras específicas se puedan codificar para usarse con mecanismos de comunicación entre procesos. El empaquetamiento y desempaquetamiento de datos se realizan automáticamente por las funcionalidades de comunicación entre procesos en los dos extremos, de forma transparente al programador.

A un nivel de abstracción más alto (esto es, ocultando más detalles; se entrará en este concepto con más detalle en el próximo capítulo), existen estándares como **ASN.1** (*Abstract Syntax Notation Number 1*) [oss.com, 2]. ASN.1 es un estándar de OSI (*Open Systems Interconnection*) que especifica la sintaxis de transferencia para datos sobre una red. El estándar cubre un amplio rango de estructuras (como conjuntos y secuencias) y tipos de datos (enteros, booleanos y caracteres) y soporta el concepto de etiquetado de datos. Cada elemento de datos transmitido se codifica usando una sintaxis que indica el tipo, la longitud, el valor y opcionalmente una etiqueta que identifica una forma específica de interpretar esta sintaxis.

A un nivel aún más alto de abstracción ha emergido **XML** (*Extensible Markup Language*) [w3.org, 9] como lenguaje de descripción de datos entre aplicaciones que comparten información, principalmente aplicaciones en Internet, usando una sintaxis similar a HTML (*HyperText Markup Language*), que es el lenguaje usado para componer páginas web. XML va un poco más allá que ASN.1 en el sentido que permite al usuario usar etiquetas configurables (tales como <*mensaje*>, <*de*> o <*para*> en el ejemplo de la Figura 2.13) para especificar una unidad de contenido de datos. XML se utiliza para facilitar intercambio de datos entre sistemas heterogéneos, para separar el contenido de una página web (escrito en XML) de la sintaxis para mostrarlo (escrita en HTML), y para permitir que los datos se compartan entre aplicaciones. Desde su aparición en 1998, XML ha ganado una atención considerable y en la actualidad se utiliza en un amplio rango de aplicaciones.

```
<mensaje>
   <a>MaryJ@BigU.edu</a>
   <de>JohnL@OpenU.edu</de>
   <sobre>Comunicación entre procesos</sobre>
   <texto> La espina dorsal de los sistemas distribuidos son los ...
</texto>
</mensaje>
```

Figura 2.13. Un fichero XML de ejemplo.

2.7. PROTOCOLOS BASADOS EN TEXTO

El empaquetamiento de datos es, en el caso más simple, cuando se intercambian cadenas de caracteres o texto codificado en una representación de tipo ASCII. El intercambio de datos en texto tiene la ventaja adicional de que puede ser fácilmente analizado por un programa y mostrado a un usuario humano. Por eso es relativamente habitual en varios protocolos intercambiar peticiones y respuestas en forma de cadenas de caracteres. Estos protocolos se denominan **basados en texto**. Muchos protocolos habituales son basados en texto, incluyendo FTP (*File Transfer Protocol*, protocolo de transferencia de ficheros), HTTP y SMTP (*Simple Mail Transfer Protocol*, protocolo simple de transferencia de correo). El lector tendrá la oportunidad de investigar y experimentar con estos protocolos en ejercicios al final de este capítulo, y se estudiarán varios de estos protocolos en detalle en posteriores capítulos.

2.8. PROTOCOLOS DE SOLICITUD-RESPUESTA

Un tipo importante de protocolos son los **protocolos de solicitud-respuesta**. En estos protocolos un lado invoca una petición y espera una respuesta del otro extremo. Posteriormente, puede ser enviada otra solicitud, esperándose de nuevo respuesta. El protocolo se desarrolla basándose en interacciones de este tipo hasta que la tarea solicitada se ha cumplido. FTP, HTTP y SMTP son protocolos habituales del tipo solicitud-respuesta.

2.9. DIAGRAMA DE EVENTOS Y DIAGRAMA DE SECUENCIA

Un diagrama de eventos, ya presentado en la sección 2.2, es un diagrama que se puede utilizar para documentar la secuencia detallada de eventos y bloqueos durante la ejecución de un protocolo. La Figura 2.14 es un diagrama de eventos para un protocolo solicitud-respuesta en el que participan dos procesos concurrentes, A y B. La ejecución de cada proceso respecto del tiempo se representa usando una línea vertical, que avanza hacia abajo. Un intervalo de línea continua durante la línea de ejecución representa un periodo de tiempo en el cual el proceso estaba activo. Un intervalo de línea discontinua representa cuándo el proceso está bloqueado. En el ejemplo, ambos procesos están inicialmente activos. El proceso B invoca un *recibir* bloqueante con antelación a la solicitud 1 del proceso A. El proceso A mientras tanto lanza la solicitud 1 que se estaba esperando, usando para ello un *enviar* no bloqueante y acto seguido un *recibir* bloqueante a la espera de la respuesta de B. La llegada de la solicitud 1 reactiva al proceso B que inicia el procesamiento de la solicitud antes de invocar una operación de *enviar* con respuesta 1 para el proceso A. El proceso B invoca luego un *recibir* bloqueante para la solicitud 2. La llegada de la respuesta 1 desbloquea al proceso A, que reanuda su ejecución para trabajar con la respuesta y preparar la petición 2, que desbloquea al proceso B. Una secuencia similar de eventos se sigue.

Cabe resaltar que cada turno de solicitud-respuesta enlaza una pareja de operaciones *enviar* y *recibir* para intercambiar dos mensajes. El protocolo se puede extender hasta cualquier número de turnos de intercambios con este patrón.

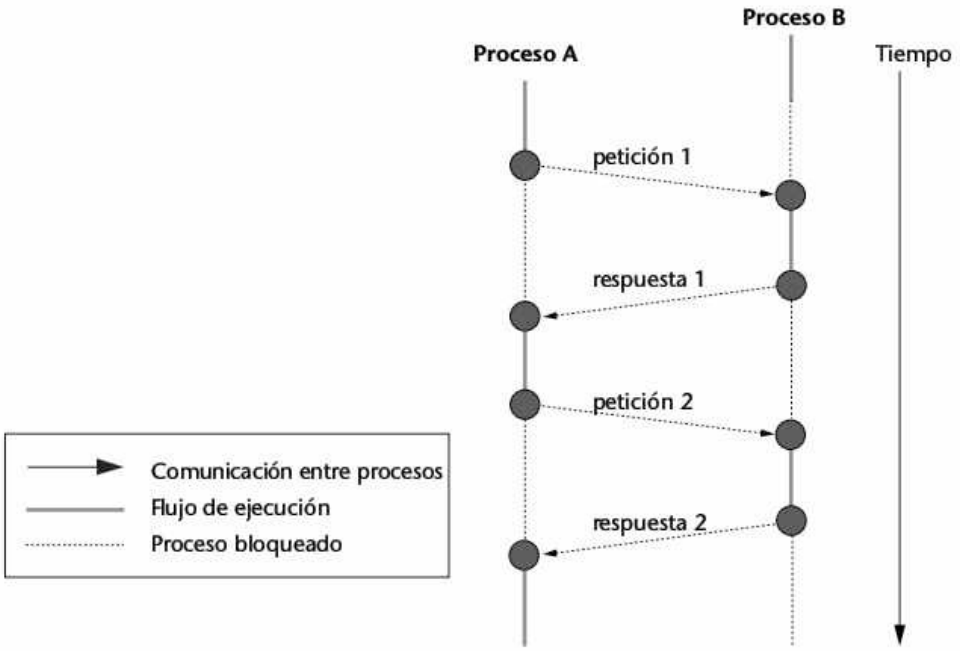


Figura 2.14. Un diagrama de eventos.

También es necesario indicar que es esencial que los programas que implementan el protocolo deben estar escritos para invocar las operaciones de *enviar* y *recibir* en el orden preestablecido, de otra forma uno o los dos procesos participantes puede quedarse esperando por una solicitud o una respuesta que nunca llega y los procesos pueden quedarse bloqueados de forma indefinida.

La Figura 2.15 presenta, por medio de un diagrama de eventos un protocolo HTTP. En su forma más básica, HTTP es un protocolo basado en texto del tipo petición-respuesta que utiliza sólo un turno de intercambio de mensajes. Un servidor web es un proceso que escucha constantemente peticiones realizadas desde procesos navegadores. Un navegador web establece una conexión con el servidor, tras ello invoca una

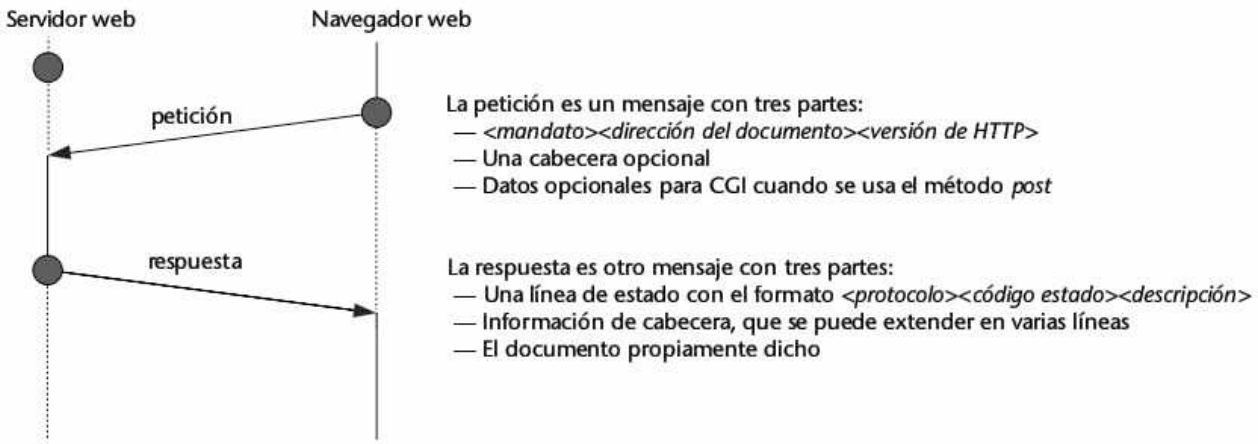


Figura 2.15. Diagrama de eventos de una sesión HTTP.

petición en el formato dictado por el protocolo. El servidor procesa la petición y responde con una línea de estado, una cabecera de información y el documento solicitado por el navegador. Tras recibir la respuesta, el navegador analiza el documento y lo presenta en pantalla. (Se estudiará el modelo cliente-servidor y el protocolo HTTP con más detalle en el Capítulo 5.)

Un diagrama de eventos es una herramienta muy útil para ilustrar la sincronización entre eventos. Pero es, sin embargo, demasiado detallado para documentar protocolos que sean complejos. Una forma simplificada de este diagrama, denominada **diagrama de secuencia** y parte de la notación UML, se usa más habitualmente para denotar la comunicación entre procesos.

En un diagrama de secuencia, el flujo de ejecución de cada participante del protocolo se representa por una línea discontinua y no se diferencia entre estados de bloqueo y ejecución. Cada mensaje intercambiado entre dos elementos se representa con una línea dirigida que va entre las dos líneas discontinuas de emisor y receptor sobre la que se añade una etiqueta descriptiva del mensaje, tal y como se ilustra en la Figura 2.16.

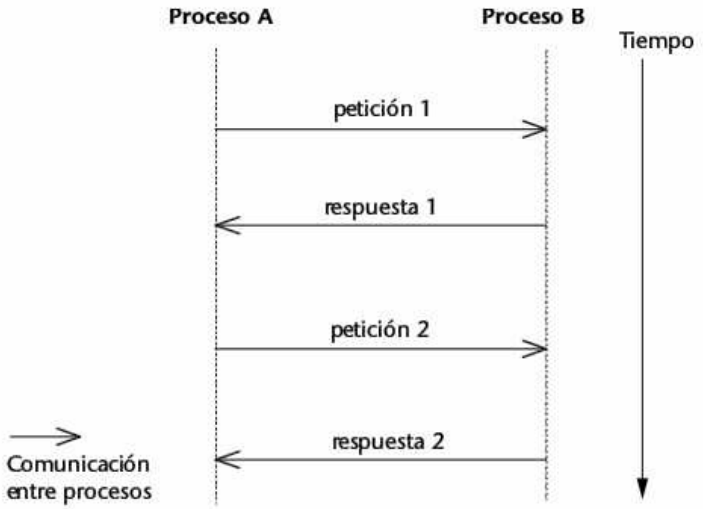


Figura 2.16. Un diagrama de secuencia.

El diagrama de secuencia de una sesión HTTP básica se muestra en la Figura 2.17.

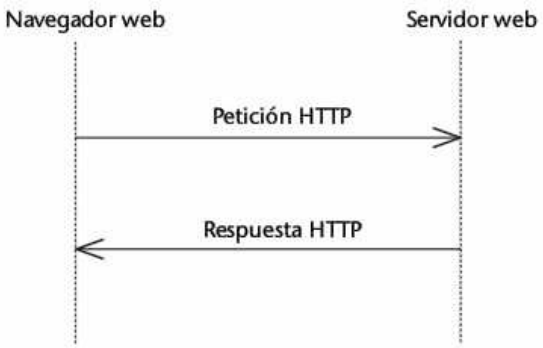


Figura 2.17. El diagrama de secuencia para una sesión HTTP.

La Figura 2.18 muestra el texto de los mensajes intercambiados durante una sesión HTTP de ejemplo. Por medio de un cliente **telnet** (*telnet* es un protocolo utilizado normalmente para una sesión de terminal sobre una máquina remota), se puede conectar a un servidor web e introducir el texto de una petición HTTP a mano. (La utilización de *telnet* para comunicarse con un proceso de la forma aquí indicada permite hacer pruebas con procesos vía IPC sin necesidad de escribir el programa, pero es necesario avisar de que ésta *no es* la forma normal de interactuar con estos procesos. En los siguientes capítulos se aprenderá a programar estas interacciones). En este caso, el servidor web se ejecuta sobre el puerto 80 de la máquina www.csc.calpoly.edu. La petición GET /~mliu/ HTTP/1.0 se teclea y envía. La respuesta del servidor web se remite a continuación. En el Capítulo 9 se estudiará el significado de las peticiones y respuestas cuando se exponga el protocolo HTTP en detalle.

```
Script started on Tue Oct 10 21:49:28 2000
9:49pm telnet www.csc.calpoly.edu 80
Trying 129.65.241.20...
Connected to tiedye2-srv.csc.calpoly.edu.
Escape character is '^]'.
GET    /~mliu/    HTTP/1.0 <-------------------- Petición HTTP

HTTP/1.1 200 OK
Date: Wed, 11 Oct 2000 04:51:18 GMT <---------- Línea de estado HTTP
Server: Apache/1.3.9 (Unix) ApacheJServ/1.0 <-- Cabecera de respuesta HTTP
Last-Modified: Tue, 10 Oct 2000 16:51:54 GMT
ETag: "1dd1e-e27-39e3492a"
Accept-Ranges: bytes
Content-Length: 3623
Connection: close
Content-Type: text/html

<HTML>
<HEAD>
<TITLE> Mei-Ling L. Liu's  Home Page
</TITLE>
</HEAD>
<BODY bgcolor=#ffffff>
...
```

Contenido del documento

Figura 2.18. El diálogo durante una sesión HTTP.

2.10. COMUNICACIÓN ENTRE PROCESOS ORIENTADA Y NO ORIENTADA A CONEXIÓN

En el Capítulo 1 se introdujo la distinción entre comunicaciones orientadas y no orientadas a conexión. Vamos a aplicar esta distinción a los mecanismos de IPC.

Por medio de un mecanismo de IPC orientado a conexión, dos procesos establecen una conexión (la cual, como recordatorio, decíamos que podía ser lógica, es decir, implementada por software en lugar de verdaderamente física), y posteriormente insertan datos en o extraen datos desde dicha conexión. Una vez que la conexión está establecida, no es necesaria la identificación de emisor y receptor.

Para el caso de una IPC no orientada a conexión, los datos son intercambiados por medio de paquetes independientes cada uno de los cuales necesita explícitamente la dirección del receptor.

Cuando se estudie el API de *sockets* en el Capítulo 4, se verá cómo los mecanismos de IPC orientados y no orientados a conexión se proporcionan al nivel de aplicación.

2.11. EVOLUCIÓN DE LOS PARADIGMAS DE COMUNICACIÓN ENTRE PROCESOS

Una vez expuesto el concepto de comunicaciones entre procesos (IPC), veremos diferentes modelos, o paradigmas, por medio de los cuales la comunicación entre procesos se proporciona al programador que quiere utilizar una IPC en su programa. Al comienzo de este capítulo se vio que los esquemas de codificación de datos se dan a diferentes niveles de abstracción. Lo mismo se puede decir de los paradigmas de comunicación entre procesos, tal y como muestra la Figura 2.19.

En el nivel menos abstracto, la comunicación entre procesos implica la transmisión de ristras binarias sobre una conexión, utilizando una transferencia de datos de bajo nivel, serie o paralelo. Este paradigma de comunicación entre procesos puede ser válido para el desarrollo del software de un *driver* de red, por ejemplo. Una IPC de esta forma cae dentro del dominio de programación de red o de sistema operativo y no lo va a cubrir este libro.

Transferencia de datos serie se refiere a la transmisión de datos bit a bit. Lo opuesto a **transferencia de datos serie** es **transferencia de datos en paralelo**, en la cual varios bits se transmiten concurrentemente.

El siguiente nivel es un paradigma bien conocido, denominado interfaz de **programación de aplicaciones de *sockets*** (el **API de *sockets***). Por medio del paradigma de *sockets*, dos procesos intercambian datos por medio de una estructura lógica denominada ***socket*** habiendo uno de cada tipo en cada extremo. Los datos a transmitir se escriben sobre el *socket*. En el otro extremo, un receptor lee o extrae datos del *socket*. Se estudiará el API de *sockets* del lenguaje Java en el próximo capítulo.

Socket es un término tomado de los primeros días de las comunicaciones telefónicas, cuando un operador tenía que establecer manualmente una conexión entre dos partes insertando los dos extremos de un cable en sendos *sockets*[1].

Los paradigmas de **llamadas a procedimientos remotos** o de **invocación de métodos remotos** proporcionan una mayor abstracción, permitiendo al proceso realizar llamadas a procedimientos o la invocación de métodos de un proceso remoto, con la transmisión de datos como argumentos o valores de resultado. Se estudiará una implementación de este paradigma, la invocación de métodos remotos en Java, en el Capítulo 8.

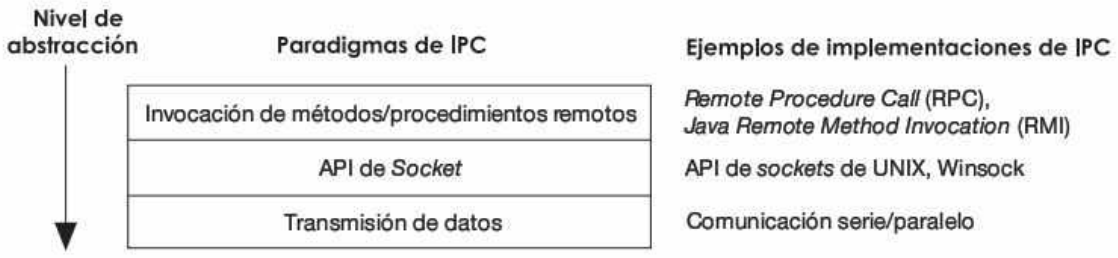


Figura 2.19. Paradigmas de comunicación entre procesos.

[1] (*N. del T.: Socket* en inglés significa enchufe.)

RESUMEN

La comunicación entre procesos (IPC) conforma el eje central de la computación distribuida. En este capítulo se han visto los principios de los mecanismos de IPC, incluyendo lo siguiente:

- Comunicación entre procesos (IPC). La posibilidad de que procesos independientes, y separados se comuniquen entre ellos para colaborar en una tarea. Cuando una comunicación se realiza únicamente de un proceso a otro, el mecanismo IPC se denomina unidifusión. Cuando la comunicación se realiza entre un proceso y un grupo de procesos, el mecanismo IPC es multidifusión.
- Una API básica de funcionalidades para IPC debe proporcionar:
  - — Primitivas de operación: enviar, recibir, conectarse, y desconectarse.
  - — La sincronización de eventos permite que procesos relacionados se ejecuten independientemente, sin conocimiento de lo que ocurre en el otro extremo. La forma más sencilla para que un mecanismo de comunicación permita sincronización es por medio de bloqueos. Las operaciones que son bloqueantes se denominan a menudo operaciones síncronas, mientras que las que no son bloqueantes se llaman también operaciones asíncronas. Los interbloqueos pueden aparecer debido al uso de operaciones bloqueantes. La utilización de hilos de ejecución (*threads*) o procesos permiten la realización de otras tareas a un proceso que aún espera que una operación bloqueante se complete.
  - — El empaquetamiento de datos (*data marshaling*) necesario para preparar los datos para su transmisión por red, está compuesto por los siguientes pasos: (i) serialización de las estructuras de datos, y (ii) conversión de los valores de datos a una representación externa o de red.
- Existen diferentes esquemas de representación de datos de red a diferentes niveles de abstracción. Algunos de los más conocidos son Sun XDR(*External Data Representation*), ASN.1 (*Abstract Syntax Notation Number 1*) and XML (*Extensible Markup Language*).
- El empaquetamiento de datos es más sencillo cuando los datos a transmitir son una secuencia de caracteres o texto representado por medio de una codificación de tipo ASCII. Los protocolos que utilizan texto se denominan protocolos basados en texto.
- Los protocolos del tipo solicitud-respuesta son protocolos que envían iterativamente mensajes de solicitud y de respuesta hasta que se completen las tareas.
- Un diagrama de eventos se utiliza para documentar la secuencia detallada de eventos y bloqueos en un protocolo. Un segmento continuo a lo largo de la línea de ejecución representa el periodo de tiempo durante el cual el proceso está activo. Una línea discontinua representa que el proceso está bloqueado.
- Un diagrama de secuencia es parte de la notación UML y se utiliza para documentar iteraciones entre procesos que son complejas. En un diagrama de secuencia, el flujo de ejecución de cada participante del protocolo se representa por una línea discontinua y no se diferencia entre estados de bloqueo y ejecución.
- Las funcionalidades de comunicación entre procesos pueden ser orientadas o no orientadas a conexión:

— Por medio de los mecanismos orientados a conexión, dos procesos establecen una conexión lógica, para posteriormente intercambiar datos insertándolos y extrayéndolos de la conexión. Una vez que la conexión se ha establecido no es necesario identificar a emisor y receptor.

— En el caso de mecanismos no orientados a la conexión, los datos se intercambian por medio de paquetes independientes, cada uno de los cuales necesita identificar al receptor.

- Las funcionalidades de tipo IPC pueden clasificarse de acuerdo a sus niveles de abstracción, yendo desde transferencia de datos serie/paralelo, al nivel más bajo, pasando por el API de *sockets* al siguiente nivel, hasta llamadas a procedimientos o métodos remotos, al nivel más alto.

EJERCICIOS

1. Teniendo en cuenta el caso de la comunicación entre humanos:
   a. Clasifique cada uno de los siguientes escenarios en términos de **unidifusión** o **multidifusión**:
      i. Un estudiante hablando con un amigo por medio de un teléfono inalámbrico.
      ii. Un ejecutivo hablando vía conferencia telefónica con empresarios en varias ciudades.
      iii. Un profesor dando clase en un aula.
      iv. Un niño jugando con otro usando un *walkie-talkie*.
      v. Un presidente dirigiéndose a la nación en televisión.
   b. ¿Cómo se realiza la sincronización de eventos y la representación de datos durante una sesión de una conversación cara a cara, tal y como se hace cuando se habla con alguien sentado a su lado?
   c. ¿Cómo se realiza la sincronización de eventos y la representación de datos durante una sesión de conversación a distancia, tal y como se hace cuando se habla con alguien por teléfono?
   d. ¿Cómo se realiza la sincronización de eventos y la representación de datos durante una reunión entre dos representantes de dos naciones que hablan idiomas diferentes?
2. El proceso A manda un mensaje sencillo al proceso B usando una llamada IPC no orientada a conexión. Para hacerlo, A invoca una operación *enviar* (especificando un mensaje como argumento) en algún instante durante su ejecución, y B invoca una operación *recibir*. Suponga que la operación *enviar* es asíncrona (no bloqueante) y la operación de *recibir* es síncrona (bloqueante). Dibuje un diagrama de eventos (no un diagrama de secuencia) para cada uno de los siguientes escenarios.
   a. El proceso A invoca la operación *enviar* antes de que el proceso B invoque la operación *recibir*.
   b. El proceso B invoca la operación *recibir* antes de que el proceso A invoque la operación *enviar*.
3. Repita la última pregunta. Esta vez ambas operaciones (*enviar* y *recibir*) son bloqueantes.

4. Considere el API de comunicación entre procesos siguiente:

   Esta API envía y recibe los mensajes de buzones. Un proceso puede comunicarse con otro proceso usando un buzón compartido por esos dos procesos. Por ejemplo, si el proceso A desea comunicarse con los procesos B y C, debe compartir el buzón 1 con el proceso B, y otro buzón, el buzón 2, con C. Los mensajes entre A y B se depositan y recogen del buzón 1, mientras que los mensajes entre A y C se depositan y recogen del buzón 2. Obsérvese la Figura 2.20.

   Las operaciones *enviar* y *recibir* se definen como:

   — enviar (n, mensaje): Envía un mensaje al buzón n, bloqueante (es decir, que el emisor quedará suspendido hasta que llegue una respuesta al buzón compartido).

   — recibir (n, mensaje): Examina el buzón n con antelación a la recepción de un mensaje; es también una operación bloqueante, que significa que el proceso receptor quedará suspendido hasta que llegue un mensaje al citado buzón.

   Un proceso que se encuentre bloqueado a la espera de un mensaje en un determinado buzón no podrá recibir ningún mensaje por otro buzón.

   a. Suponga que un proceso P espera recibir dos mensajes, uno por el buzón 1 y otro por el buzón 2. No se conoce a priori ni el instante ni el orden en el que van a llegar los mensajes. ¿Qué secuencia de *enviar* y *recibir*, si existe, se puede ejecutar para asegurarse que el proceso P no se bloquea permanentemente?

   b. ¿Qué secuencia de *enviar* y *recibir*, si existe, debe ejecutar el proceso P si quiere esperar por un solo mensaje bien por el buzón 1 o por el buzón 2 (o incluso) por ambos? De nuevo, no se conoce por adelantado qué mensaje llegará primero. Asimismo, la secuencia propuesta no debe causar un bloqueo indefinido.

   (*Nota*: Su respuesta debe utilizar únicamente las operaciones dadas en el capítulo; no utilice hilos (*threading*) u otras funcionalidades del sistema operativo.)

5. ¿Se puede dar un interbloqueo durante la comunicación entre procesos (utilizando *enviar/recibir*).

   a. en un sistema de comunicaciones que proporciona una operación de *enviar* bloqueante y de *recibir* bloqueante?

   b. en un sistema de comunicaciones que proporciona una operación de *enviar* no bloqueante y de *recibir* bloqueante?

   Justifique su respuesta. Si la respuesta es afirmativa, proporcione un ejemplo. Si la respuesta es negativa, dé una breve argumentación.

6. Considere el desarrollo de una API de multidifusión sobre una API existente de tipo unidifusión. El API de unidifusión proporciona operaciones *enviar* y *recibir* entre dos procesos. El API de multidifusión debe proporcionar operaciones para (1) enviar a un grupo de procesos, y (2) recibir de un proceso de multidifusión. Describa cómo proporcionaría las operaciones de multidifusión usando las operaciones de unidifusión únicamente. (*Nota*: Su respuesta debe utilizar únicamente las operaciones dadas en el capítulo; no utilice hilos (*threading*) u otras funcionalidades del sistema operativo).

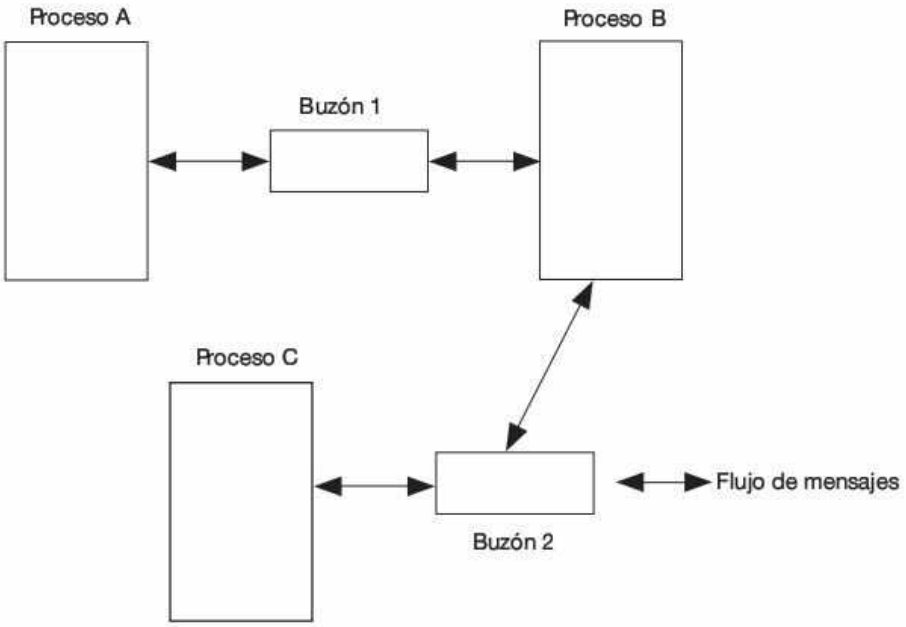


Figura 2.20. Una interfaz de programación de aplicaciones IPC para el Ejercicio 4.

7. En un sistema distribuido, tres procesos *P1*, *P2*, y *P3* participan en una comunicación entre procesos. Suponga que sucede la siguiente secuencia de eventos:

   En el instante 1, *P3* invoca un *recibir* para *P2*.

   En el instante 2, *P1 envía* el mensaje *m1* hacia *P2*.

   En el instante 3, *P2* invoca un *recibir* para *P1*.

   En el instante 4, *P2 recibe* el mensaje *m1*.

   En el instante 5, *P2 envía* el mensaje *m1* hacia *P3*.

   En el instante 6, *P3 recibe* el mensaje *m1*. *P1* invoca un *recibir* para *P2*.

   En el instante 7, *P2* invoca un *recibir* para *P3*.

   En el instante 8, *P3 envía* el mensaje *m2* hacia *P2*.

   En el instante 9, *P2 recibe* el mensaje *m2*.

   En el instante 10, *P2 envía* el mensaje *m2* hacia *P1*.

   En el instante 11, *P1 recibe* el mensaje *m2*.

   a. Para cada uno de los siguientes escenarios, dibuje un diagrama de eventos para mostrar la secuencia de eventos y los bloqueos y desbloqueos de cada proceso:

      i. en un sistema de comunicaciones que proporciona una operación de *enviar* bloqueante y de *recibir* bloqueante,

      ii. en un sistema de comunicaciones que proporciona una operación de *enviar* no bloqueante y de *recibir* bloqueante.

   b. Dibuje un diagrama de secuencia para documentar la comunicación entre los procesos *P1*, *P2*, y *P3*.

8. Este es un ejercicio sobre empaquetamiento de datos (*data marshaling*).
   a. En el contexto de IPC:
      i. ¿Qué es el empaquetamiento de datos? Hay dos pasos dentro del empaquetamiento de datos, nómbrelos y describa cada uno. ¿Por qué es necesario el empaquetamiento de datos?
      ii. ¿Qué es la serialización de objetos?
      iii. ¿Cómo se aplican los dos pasos del empaquetamiento de datos a (i) un vector de enteros, y (ii) un objeto? Describa en términos generales qué es necesario realizar con los datos.
   b. El proceso A envía al proceso B un único dato, una fecha. El proceso A usa el formato americano de fecha: *<mes>/<día>/<año>* (por ejemplo, 01/31/2001). El proceso B usa el formato europeo: *<día>/<mes >/<año>* (por ejemplo, 31/01/2001).
      i. Suponga que no se ha acordado una representación externa de datos.
         a. ¿Cómo A enviaría la fecha a B para que A no necesite hacer ninguna conversión?
         b. ¿Cómo A enviaría la fecha a B para que B no necesite hacer ninguna conversión?
      ii. Suponga que la misma fecha se comunica al proceso C, que utiliza un formato de fecha del tipo: *<año>-<mes>-<día>* (por ejemplo, 2001-31-01). ¿Cómo puede A enviar la fecha a B y C a la vez de forma que A no tenga que realizar conversión alguna?
      iii. Describa una representación externa de datos para la fecha de forma que cualquier proceso emisor convierta la fecha del formato de representación local al formato de representación externo antes de enviarla, y que cualquier proceso receptor convierta la fecha recibida del formato de representación externo al suyo propio.

   Puede resultarle de interés la consulta de la referencia [saqqara.demon.co.uk, 10].

9. Utilice *telnet* para interactuar con un servidor *Daytime* [RFC 867, 4] sobre una máquina a la que tenga acceso. Los servidores *Daytime* escuchan el puerto 13. Desde una ventana de consola sobre el símbolo de sistema teclee:

   `telnet <espacio> <nombre o dirección IP de la máquina><espacio> 13`

   Ejemplo: `telnet maquina.universidad.edu 13`

   Recopile una traza de la sesión y describa sus observaciones.

10. Dibuje un diagrama de secuencia para el protocolo *Daytime*.
11. ¿Es posible para un cliente *Daytime* que se quede bloqueado de forma indefinida? Explíquelo.
12. Utilice *telnet* para interactuar con un servidor *echo* [RFC 862, 6] sobre una máquina a la que tenga acceso. Los servidores *echo* escuchan el puerto 7.
    a. Dibuje un diagrama de eventos para el protocolo *echo*.
    b. ¿Es posible para un cliente *Daytime* que se quede bloqueado de forma indefinida? Explíquelo.
13. Considere el protocolo FTP (*File Transfer Protocol*, Protocolo de Transferencia de Ficheros) [RFC 959, 5].

Nótese que este protocolo utiliza dos conexiones: una para transmitir las peticiones y respuestas y otro para transmitir los ficheros para ser enviados/recibidos.

a. Utilice *telnet* para conectarse a un servidor FTP al que tenga acceso. Luego solicite un mandato para listar el contenido del directorio raíz del servidor.

b. Utilice un diagrama de eventos para describir las interacciones entre los procesos participantes.

c. ¿Cuál es el formato de cada petición?

d. ¿Cuál es el formato de cada respuesta?

e. Considere el mandato MODE del protocolo: permite especificar el tipo de fichero (texto o binario) a transmitir. ¿Cuál es la representación de datos para cada uno de los diferentes modos de fichero?

14. Considere el protocolo SMTP (*Simple Mail Transfer Protocol*, Protocolo Simple de Transferencia de Correo) [RFC 821, 3]. Un extracto de la RFC de este protocolo proporciona la siguiente sesión de ejemplo:

```
    R: 220 USC-ISI.ARPA Simple Mail Transfer Service Ready
    S: HELO LBL-UNIX.ARPA
    R: 250 USC-ISI.ARPA
    S: MAIL FROM:<mo@LBL-UNIX.ARPA>
    R: 250 OK
    S: RCPT TO:<Jones@USC-ISI.ARPA>
    R: OK
    S: DATA
    R: 354 Start mail input; end with <CRLF>.<CRLF>
    S: Bla bla bla...
    S: ...etc. etc. etc.
    S: .
R: 250 OK
S: QUIT
R: 221 USC-ISI.ARPA Service closing transmission channel
```

a. Utilice un diagrama de secuencia para describir las interacciones entre los procesos participantes.

b. ¿Cuál es el formato de cada petición?

c. ¿Cuál es el formato de cada respuesta?

d. Utilice *telnet* para conectarse a un sistema donde usted tenga una cuenta SMTP, y envíese a sí mismo un correo. Conéctese luego al sistema y verifique que el correo ha llegado.

Recopile una traza de la sesión y describa sus observaciones.

REFERENCIAS

(*Nota:* Todos los RFC (*Requests for Comments*) se pueden consultar en línea en la página: IETF RFC Page, http://www.ietf.org/rfc.html.)

1. RFC 1014, External Data Representation.
2. «ASN.1 Overview», *http://www.oss.com/asn1/overview.html.*
3. RFC 821, SMTP.
4. RFC 867, Protocolo Daytime.

5. RFC 959, Protocolo FTP.
6. RFC 862, Protocolo Echo.
7. John Shapley Gray. *Interprocess Communications in UNIX*. Upper Saddle Prentice Hall, 1997.
8. RFC 742, Finger protocol.
9. Extensible Markup Language (XML), *http://www.w3.org/XML/*
10. International Date Format Campaign, *http://www.saqqara.demon.co.uk/*
11. Java Object Serialization, *http://java.sun.com/j2se/1.3/docs/guide/serialization/*
12. Elliotte Rusty Harold. *Java I/O*, Sebastopol, CA: O'Reilly Press, 1999.

CAPÍTULO 3

Paradigmas de computación distribuida

La computación distribuida es una de las más vibrantes áreas en el campo de la informática. Hay una continua evolución hacia nuevas tecnologías que den soporte a las aplicaciones distribuidas, trayendo consigo nuevos modelos conceptuales y terminologías.

Aparentemente nueva jerga, otro acrónimo, o una nueva tecnología rompedora aparece cada día. Para un observador inexperto o para un estudiante que comienza, ordenar esta terminología y sus tecnologías es una ardua tarea.

Este capítulo presenta una clasificación de varios paradigmas para aplicaciones distribuidas, así como una introducción a algunas de las herramientas ya existentes y los protocolos basados en estos paradigmas. En los siguientes capítulos se explorarán algunos de estos paradigmas, herramientas y protocolos en detalle.

3.1. PARADIGMAS Y ABSTRACCIÓN

Los términos **paradigmas** y **abstracción** se han usado ya en previos capítulos, pero aquí se examinarán más de cerca.

Abstracción

Con diferencia el concepto más fundamental dentro de la informática, la abstracción es la idea de **encapsulación**, o de **ocultamiento de detalles**. Citando a David J. Barnes [Barnes, 1]:

> *«Habitualmente usamos la abstracción cuando no es necesario conocer los detalles exactos de cómo algo funciona o se representa, porque podemos utilizarlo en su forma simplificada. A menudo entrar dentro del detalle tiende a os-*

curecer lo que estamos intentando entender en lugar de iluminarlo... la abstracción juega un papel muy importante en la programación porque lo que a menudo queremos modelar, en software, es una versión simplificada de las cosas que existen en el mundo real... sin necesidad de construir cosas reales».

En ingeniería del software, la abstracción se materializa en proporcionar herramientas o funcionalidades que permitan el desarrollo de software sin que el programador tenga que conocer las complejidades subyacentes. No se trata de sobrestimar cuando se dice que las herramientas de abstracción son el empuje detrás del desarrollo moderno de software, y que ellas existen detrás de cada aspecto del desarrollo de una aplicación. Por ejemplo, se utilizan compiladores para abstraerse del detalle de los lenguajes máquina, y los programadores de Java utilizan el paquete AWT (*Abstract Window Toolkit*) para desarrollar rápidamente interfaces gráficas.

En el área de aplicaciones distribuidas, ha habido una explosión de herramientas y funcionalidades basadas en una amplia variedad de paradigmas que ofrecen diferentes grados de abstracción.

Paradigmas

El diccionario Webster define la palabra paradigma como «un patrón, ejemplo, o modelo». En el estudio de cualquier materia de gran complejidad, es útil identificar los patrones o modelos básicos y clasificar los detalles de acuerdo con estos modelos. Este capítulo busca presentar una clasificación de los paradigmas sobre aplicaciones distribuidas. Los paradigmas se van a presentar ordenados basándose en su nivel de abstracción, como se muestra en la Figura 3.1. En el nivel más bajo de abstracción se encuentra el **paso de mensajes**, que encapsula el menos nivel de detalle. Los **espacios de objetos** ocupan el extremo opuesto del espectro, ya que se trata del paradigma más abstracto.

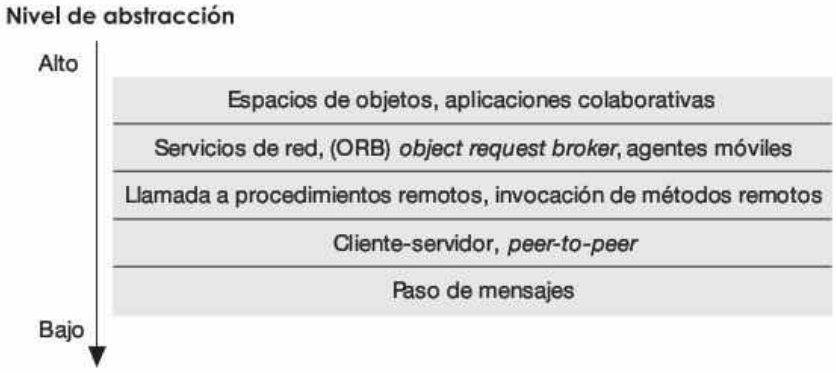


Figura 3.1. Los paradigmas de computación distribuida y su nivel de abstracción.

3.2. UNA APLICACIÓN DE EJEMPLO

A través del resto del capítulo, se usará una misma aplicación para ilustrar cómo se aplica cada uno de los paradigmas.

La aplicación de ejemplo se trata de un sistema de subastas *on-line*. (*Nótese* que las implementaciones descritas en este capítulo intencionadamente ignoran detalles

[como la interfaz con el usuario y el almacenamiento de datos] de la aplicación real. Las implementaciones de ejemplo se han pensado para que sirvan como hilo común de este capítulo. Por medio del uso de estas implementaciones, se podrá comparar y contrastar las diferencias y efectos de las abstracciones proporcionadas por los diferentes paradigmas.) Se simplificará el sistema hasta el punto de tratar sólo un objeto a subastar en cada sesión. Durante cada sesión de subasta, un objeto está abierto a pujas emitidas por los participantes en la subasta. Al final de la sesión, el subastador anuncia el resultado. En la descripción de las diferentes implementaciones, se va a centrar la atención en los aspectos de computación distribuida de la capa de servicio (esto es, dentro de la arquitectura de tres niveles para aplicaciones distribuidas, en el nivel de servicio subyacente que da soporte al resto de niveles superiores) de la arquitectura de la aplicación.

3.3. PARADIGMAS PARA APLICACIONES DISTRIBUIDAS

Paso de mensajes

La aproximación más básica a la comunicación entre procesos es el **paso de mensajes**. En este paradigma, los datos que representan mensajes se intercambian entre dos procesos, un **emisor** y un **receptor**.

El paso de mensajes es el paradigma fundamental para aplicaciones distribuidas. Un proceso envía un mensaje que representa una petición. El mensaje se entrega a un receptor, que procesa la petición y envía un mensaje como respuesta. En secuencia, la réplica puede disparar posteriores peticiones, que lleven a sucesivas respuestas, y así en adelante. La Figura 3.2 ilustra el paradigma de paso de mensajes.

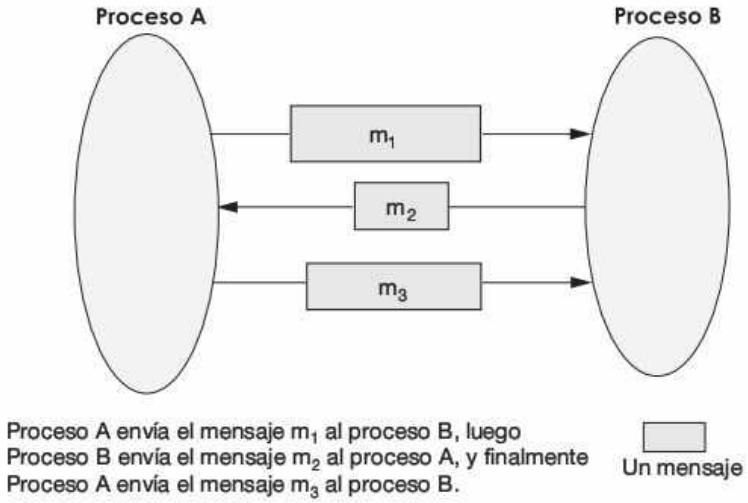


Figura 3.2. El paradigma de paso de mensajes.

Las operaciones básicas necesarias para dar soporte al paradigma de paso de mensajes son *enviar* y *recibir*. Para comunicaciones orientadas a conexión, también se necesitan las operaciones *conectar* y *desconectar*. (Los detalles sobre las operaciones,

tales como argumentos o valores de retorno, se darán junto a las respectivas herramientas y funcionalidades en los capítulos posteriores.) Por medio del grado de abstracción proporcionado por este modelo, los procesos interconectados realizan la entrada y la salida hacia el otro proceso, de una forma similar a la utilizada en la entrada/salida (E/S) de ficheros. Como en el caso de la E/S de ficheros, las operaciones sirven para encapsular el detalle de la comunicación a nivel del sistema operativo, de forma que el programador puede hacer uso de operaciones para enviar y recibir mensajes sin preocuparse por el detalle subyacente.

La interfaz de programación de aplicaciones de ***sockets*** (que se estudiará en el Capítulo 4) se basa en este paradigma. Usando un *socket*, una construcción lógica, dos procesos pueden intercambiarse datos de la siguiente forma: un emisor escribe o inserta un mensaje en el *socket*; en el otro extremo, un receptor lee o extrae un mensaje del *socket*.

La implementación de nuestro sistema de subastas usando paso de mensajes y el paradigma cliente-servidor se describirá en la próxima sección.

Paradigma cliente-servidor

Quizás el más conocido de los paradigmas para aplicaciones de red, el **modelo cliente-servidor** [Comer y Stevens, 2] asigna roles diferentes a los dos procesos que colaboran. Un proceso, el servidor interpreta el papel de proveedor de servicio, esperando de forma pasiva la llegada de peticiones. El otro, el cliente, invoca determinadas peticiones al servidor y aguarda sus respuestas. La Figura 3.3 ilustra este paradigma.

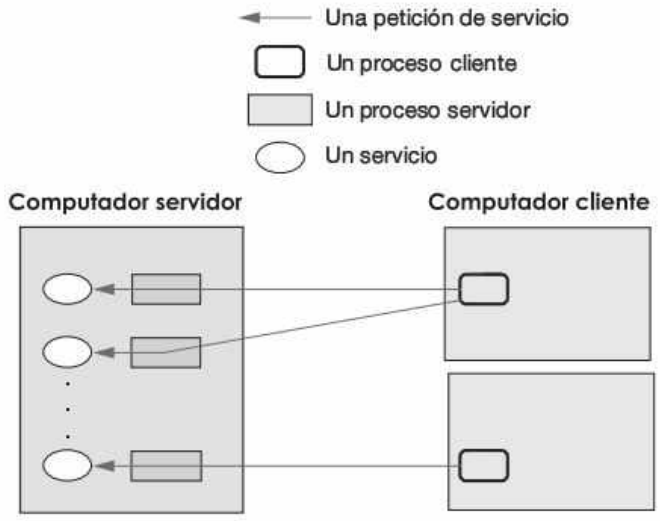


Figura 3.3. El paradigma cliente-servidor.

De una concepción simple, el modelo cliente-servidor proporciona una abstracción eficiente para facilitar servicios de red. Las operaciones necesarias incluyen aquellas que permiten a un proceso servidor escuchar y aceptar peticiones, y a un cliente solicitar dichas peticiones y aceptar las respuestas. Al asignar dos papeles diferentes a

ambos extremos la sincronización de eventos se simplifica: el proceso servidor espera peticiones, y el cliente en su turno espera las respuestas.

Muchos servicios de Internet dan soporte a aplicaciones cliente-servidor. Estos servicios se conocen por el protocolo que implementa la aplicación. Conocidos servicios de Internet son HTTP, FTP, DNS, *finger*, *gopher* y otros, algunos de los cuales ya se han presentado en el Capítulo 2.

Los dos modelos básicos que hemos visto hasta el momento, *cliente-servidor* y *paso de mensajes*, son suficiente como base para la implementación de nuestro sistema de subastas. Cada participante, así como el programa de subastas, asumen a la vez el papel de cliente y servidor, de la siguiente forma:

Para el control de la sesión:

- Como servidor, un participante espera escuchar el anuncio del subastador (1) cuando comience la sesión, (2) en el momento en el que se produzca un cambio en la puja máxima, y (3) cuando la sesión termine.
- Como servidor, el subastador envía una petición que anuncia los tres tipos de eventos antes comentados.

Para la aceptación de pujas:

- Como servidor, un participante envía una nueva puja a un servidor.
- Como cliente, un subastador acepta nuevas pujas y actualiza la puja máxima.

El paradigma cliente-servidor es inherente a muchas aplicaciones distribuidas. El API de *sockets* orientados a conexión proporciona operaciones diseñadas específicamente para servidores y clientes, asimismo, el API de **llamada a procedimientos remotos** (*Remote Procedure Call*) y el API de **Java de invocación de métodos remotos** (*Remote Method Invocation*) (algo más sobre éste último se verá a lo largo de este capítulo) también se refiere a los procesos participantes como clientes y servidores.

Paradigma de igual a igual *peer-to-peer*

En el paradigma cliente-servidor, los procesos participantes juegan diferentes roles: los procesos cliente solicitan peticiones mientras que los procesos servidores escuchan de forma pasiva para servir dichas peticiones y proporcionar los servicios solicitados en respuesta. En particular, el paradigma no da soporte para que el proceso servidor inicie la comunicación.

En el **paradigma *peer-to-peer*** (Figura 3.4), los procesos participantes interpretan los mismos papeles, con idénticas capacidades y responsabilidades (de ahí el término *peer*, en inglés *par*). Cada participante puede solicitar una petición a cualquier otro participante y recibir una respuesta. Un ejemplo bien conocido de un servicio de transferencia de ficheros *peer-to-peer* es *Napster.com*; servicios similares permiten la compartición de ficheros (primordialmente ficheros multimedia) entre ordenadores a través de Internet.

Mientras el paradigma cliente-servidor es un modelo ideal para servicios centralizados de red (donde el proceso servidor proporciona el servicio y los procesos cliente acceden a dicho servicio mediante el citado servidor), el paradigma *peer-to-peer* resulta más apropiado para aplicaciones como mensajería instantánea, transferencia de ficheros, vídeo-conferencia, y trabajo colaborativo. También es posible que una aplicación se base en ambos modelos, cliente-servidor y *peer-to-peer*, Napster.com utiliza un servidor como directorio además de la comunicación *peer-to-peer*.

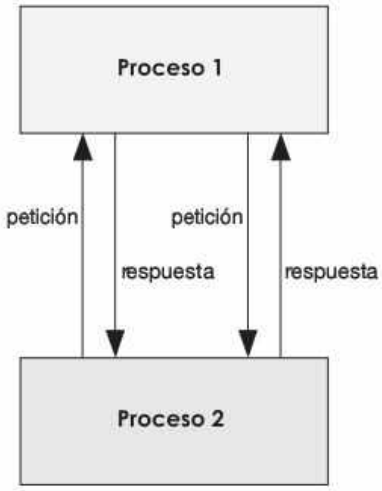


Figura 3.4. El Paradigma *peer-to-peer.*

El paradigma *peer-to-peer* se puede implementar por medio de bibliotecas que proporcionen herramientas de paso de mensajes. Para aplicaciones complejas, como mensajería instantánea o compartición de recursos, se está trabajando en protocolos de alto nivel y en herramientas para su desarrollo. Ejemplos de estos protocolos y sus correspondientes herramientas son el proyecto **JXTA** [jxta.org, 29] y **Jabber** [jabber.org, 30], un protocolo abierto basado en XML para mensajería y presencia instantánea.

En el contexto de la mensajería instantánea, la **presencia** representa el estado de un participante, tal como si está presente (*on-line*) o desconectado, así como otros indicadores de estatus.

La implementación de nuestro sistema de subastas puede simplificarse sustancialmente con la disponibilidad de una herramienta que dé soporte al paradigma *peer-to-peer*. Un participante puede conectarse al subastador directamente para registrarse en la subasta. El subastador posteriormente contacta con cada uno de los participantes para iniciar la sesión de subasta, durante la cual cada uno de los participantes individualmente pueden obtener el estado y realizar pujas. Al concluir la subasta, el subastador notifica al ganador, y el resto de participantes pueden conocer el resultado contactando con el propio subastador.

Paradigma de sistema de mensajes

Middleware hace referencia al software que actúa como intermediario entre procesos independientes. Los sistemas de mensajes son unos de los tipos de *middleware*, los ORB o *broker* de peticiones son otros.

El paradigma de **sistema de mensajes** o ***Middleware* Orientado a Mensajes** (*Message-Oriented Middleware*-MOM), (véase la Figura 3.5) es una elaboración del paradigma básico de paso de mensajes.

El uso de un intermediario es una técnica común en computación distribuida.

En este paradigma, un sistema de mensajes sirve de intermediario entre procesos separados e independientes. El sistema de mensajes actúa como un conmutador para mensajes, a través del cual los procesos intercambian mensajes asíncronamente, de una forma desacoplada. (Asíncronamente, como recordará del Capítulo 2, indica una comunicación sin bloqueo.) Un emisor deposita un mensaje en el sistema de mensajes, el cual redirige el mismo a la cola de mensajes asociada a dicho receptor. Una vez que se ha enviado, el emisor queda liberado para que realice cualquier otra tarea.

Hay dos subtipos de modelos de sistema de mensajes: el modelo de mensajes punto a punto y el modelo de mensajes publicación/suscripción.

El futuro de *peer-to-peer*

por Matthew Fordhal, colaborador de Associated Press.
(http://www.hollandsentinel.com/stories/021801/bus_Napster.shtml, publicado en web el domingo 18 de febrero de 2001.)

Groove Networks, recopilado por AP wire reports.

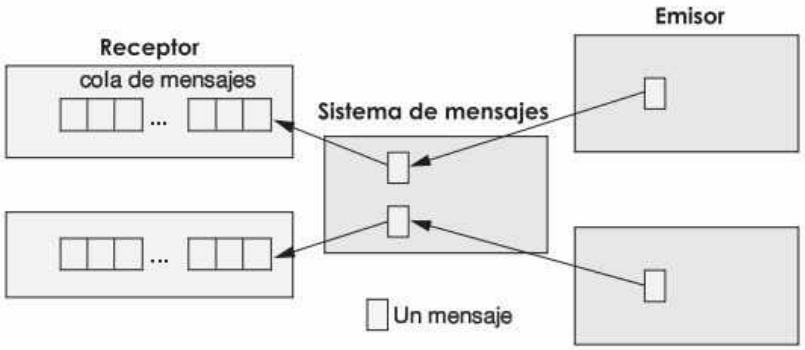


Figura 3.5. El paradigma de sistema de mensajes.

Modelo de mensajes punto a punto

En este modelo, un sistema de mensajes redirige un mensaje desde el emisor hasta la cola de mensajes del receptor. A diferencia del modelo básico de paso de mensajes, el *middleware* proporciona un depósito de los mensajes que permite que el envío y la recepción estén desacoplados. Por medio del *middleware*, un emisor deposita el mensaje en la cola de mensajes del receptor. Un proceso receptor extrae los mensajes de su cola de mensajes y procesa cada mensaje de forma correspondiente.

Comparado con el modelo básico de paso de mensajes, el paradigma de mensajes punto a punto proporciona una abstracción adicional para operaciones asíncronas. Para conseguir el mismo efecto con el paso de mensajes básico, un programador debe hacer uso de hilos (*threads*) o de procesos hijos.

La implementación de nuestro sistema de subastas usando el modelo de mensajes punto a punto es la misma que con el modelo básico de paso de mensajes. La única diferencia es que los mensajes se canalizan por medio del *middleware*, y el envío y la recepción están desacoplados.

Modelo de mensajes publicación/suscripción

En este modelo, cada mensaje se asocia con un determinado tema o evento. Las aplicaciones interesadas en el suceso de un específico evento se pueden suscribir a los mensajes de dicho evento. Cuando el evento que se aguarda ocurre, el proceso publica un mensaje anunciando el evento o asunto. El *middleware* del sistema de mensajes distribuye el mensaje a todos los suscriptores.

El modelo de mensajes publicación/suscripción ofrece una potente abstracción para multidifusión o comunicación en grupo. La operación *publicar* permite al proceso difundir a un grupo de procesos, y la operación *suscribir* permite a un proceso escuchar dicha difusión de mensajes.

Utilizando el modelo de mensajes publicación/suscripción, la implementación de nuestro sistema de subastas se realizaría de la siguiente forma:

- Cada participante se suscribe a los mensajes del evento *comienzo-subasta*.
- El subastador anuncia el comienzo de la sesión de subasta enviando un mensaje de evento *comienzo-subasta*.
- Tras recibir el evento de *comienzo-subasta*, cada participante se suscribe a los mensajes del evento *fin-subasta*.

- El subastador se suscribe a los mensajes del evento *nueva-puja*.
- Un participante que desee realizar una nueva puja lanzará el evento *nueva-puja*, que será reenviado al subastador.

Al final de la sesión, el subastador lanzará en mensaje *fin-subasta* para informar a todos los participantes del resultado. Si se desea, se pueden añadir otros mensajes adicionales que permitan a los participantes conocer el estado de la subasta.

El paradigma MOM ha tenido una larga historia en las aplicaciones distribuidas. Los *Message Queue Services* (MQS) se llevan utilizando desde los 80. El *MQ*Series* de IBM [ibm.com, 6] es un ejemplo de dicho servicio. Otros soportes existentes para este paradigma son *Microsoft's Message Queue* (MSMQ) [Dickman,5; lotus.com, 21] y el *Java's Message Service* (JMS) [Wetherill, 7].

Modelo de llamadas a procedimientos remotos

El modelo de paso de mensajes funciona bien para protocolos básicos de red y para aplicaciones de red básicas. Pero, cuando las aplicaciones crecen en complejidad, resulta necesario un nivel de abstracción mayor para la programación distribuida. En particular, resultaría deseable tener un paradigma que permitiera que el software distribuido se programase de una manera similar a las aplicaciones convencionales que se ejecutan sobre un único procesador. El modelo de llamada a procedimientos remotos (RPC, *Remote Procedure Call*) proporciona dicha abstracción. Utilizando este modelo, la comunicación entre dos procesos se realiza utilizando un concepto similar al de una llamada a un procedimiento local, que resulta familiar a los programadores de aplicaciones.

En lenguajes procedimentales como C, una llamada a procedimiento es una invocación a un procedimiento (función que no devuelve nada) o a una función (una verdadera función, que devuelve un valor).

En un programa que sólo implica a un único proceso, una llamada a un procedimiento del tipo *func(arg1, arg2)* implica un salto en el flujo de ejecución al código de dicho procedimiento. (*Véase* la Figura 3.6). Los argumentos se mandan al procedimiento como parámetros de su ejecución.

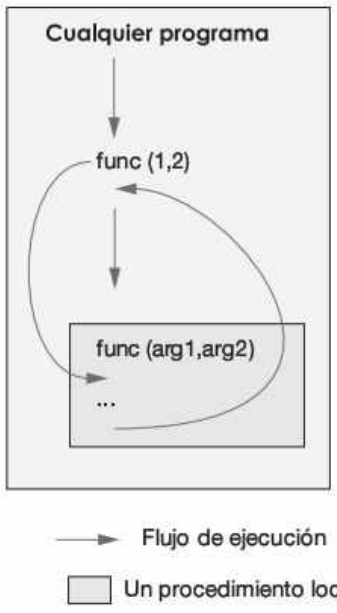


Figura 3.6. Llamada a un procedimiento local.

Una llamada a un procedimiento remoto implica dos procesos independientes, que pueden residir en máquinas diferentes. Un proceso, *A* que quiere realizar una petición a otro proceso, *B* invoca a un procedimiento de *B*, pasando junto a la llamada una lista de valores de argumentos. Como en el caso de la llamada a procedimientos locales, una llamada a un procedimiento remoto dispara una acción predefinida en un procedimiento proporcionado por *B*. Al finalizar el procedimientos, el proceso *B* devuelve un valor al proceso *A*.

La Figura 3.7 muestra el paradigma RPC. Una llamada a un procedimiento se hace desde un proceso al otro, con los datos pasados como argumentos. Después de recibir una llamada, como resultado se ejecutan las acciones codificadas en el procedimiento.

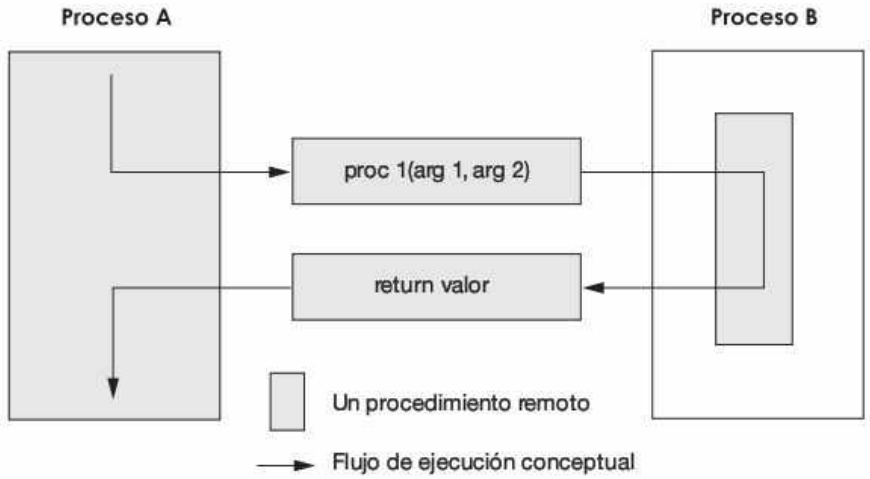


Figura 3.7. El paradigma de llamada a procedimientos remotos.

RPC permite a los programadores construir aplicaciones de red usando una construcción de programación similar a una llamada a procedimiento local, proporcionado un nivel de abstracción conveniente tanto para la comunicación entre procesos como para la sincronización de eventos.

Desde su aparición a principios de los 80, el modelo de llamadas a procedimientos remotos ha sido ampliamente utilizado en las aplicaciones de red. Existen dos API relevantes para RPC: Una es la *Open Network Computing Remote Procedure Call* (ONC RPC), evolucionado a partir del API de RPC desarrollada por Sun Microsystems en el comienzo de los 80. Detalles sobre esta API se pueden encontrar en [ietf.org, 8]. La otra API más conocida es la RPC del *Distributed Computing Environment* (DCE) de *Open Group* [opennc.org, 9]. Además, existe el *Simple Object Access Protocol* (SOAP), que se estudiará en el Capítulo 11, y sus implementaciones que dan soporte a las llamadas a procedimientos remotos basadas en web.

Para utilizar RPC para implementar nuestro sistema de subastas procederemos de la siguiente forma:

- El programa de subastas proporciona un procedimiento remoto para que cada participante se registre y otro procedimiento para que éstos hagan pujas.
- Cada programa participante proporciona los siguientes procedimientos remotos: (1) el que permite al subastador llamar al participante para anunciar el comienzo de una sesión, (2) el que permite al subastador informar al participante de la puja más alta, y (3) el que permite que el subastador anuncie el final de la sesión.

Paradigmas de objetos distribuidos

La idea de aplicar la orientación a objetos a las aplicaciones distribuidas es una extensión del desarrollo software orientado a objetos. Las aplicaciones acceden a objetos distribuidos sobre una red. Los objetos proporcionan métodos, a través de cuya invocación una aplicación obtiene acceso a los servicios. Varios paradigmas, descritos en los siguiente párrafos, se basan en la idea de objetos distribuidos.

Invocación de métodos remotos

La invocación de métodos remotos (*Remote Method Invocation*-RMI) (Figura 3.8) es el equivalente en orientación a objetos de las llamadas a procedimientos remotos. En este modelo, un proceso invoca métodos de un objeto, el cual reside en un ordenador remoto.

Como en RPC, los argumentos se pueden pasar con la invocación, y se puede devolver un valor cuando el método ha concluido.

La implementación de nuestro sistema de subastas es esencialmente la misma que con RPC, la excepción es que los métodos de los objetos reemplazan los procedimientos:

- El programa de subastas proporciona un método remoto para que cada participante se registre y otro método para que éstos hagan pujas.
- Cada programa participante proporciona los siguientes métodos remotos: (1) el que permite al subastador llamar al participante para anunciar el comienzo de una sesión, (2) el que permite al subastador informar al participante de la puja más alta, y (3) el que permite que el subastador anuncie el final de la sesión.

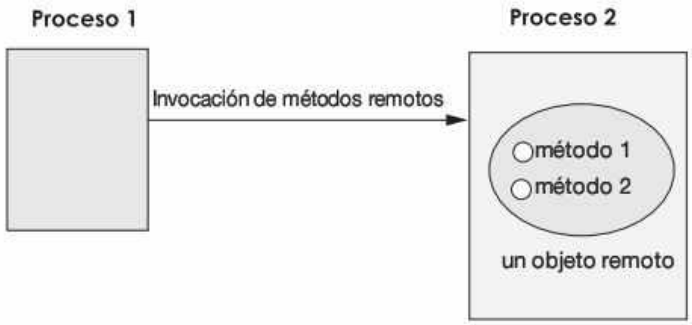


Figura 3.8. El paradigma de invocación de métodos remotos.

Paradigma basado en *Object Request Broker*

En el paradigma basado en *Object Request Broker*[2] (Figura 3.9), un proceso solicita una petición a un **ORB** (*Object Request Broker*), el cual redirige la petición al obje-

[2] N. del T. La traducción correcta de *Object Request Broker* sería «agente o intermediario de peticiones de objetos». Se ha optado por usar el término inglés debido, por un lado, a su uso extendido (incluso en la literatura en español) y por otro porque la traducción está sujeta a confusión (especialmente con terminología similar dentro de la tecnología de agentes).

Figura 3.9. El paradigma basado en *Object Request Broker.*

to apropiado que proporciona dicho servicio. El paradigma se parece bastante al modelo de invocación de métodos remotos en su soporte para acceso remoto a objetos. La diferencia es que el ORB en este paradigma funciona como *middleware*, permitiendo a una aplicación, como solicitante de un objeto, acceder potencialmente a varios objetos remotos (o locales). El ORB puede funcionar también como mediador para objetos heterogéneos, permitiendo la interacción entre objetos implementados usando diferentes API y/o ejecutando sobre diferentes plataformas.

La implementación del sistema de subastas usando un ORB es similar a la que usa RMI. Con la excepción de que cada objeto (subastador y participantes) deben registrarse en el ORB y son, en realidad, invocados por este mismo. Cada participante solicita peticiones al objeto subastador para registrarse para la sesión y hacer pujas. A través del ORB, el subastador invoca los métodos de cada participante para anunciar el comienzo de la sesión, actualizar el estado de las pujas, y anunciar el final de la sesión.

Este paradigma es la base de la arquitectura **CORBA (*Common Object Request Broker Architecture*)** de OMG (*Object Management Group*) que está desarrollada en el Capítulo 10. Herramientas basadas en esta arquitectura son Visibroker de Inspire, Java IDL (*Java Interface Definition Language*), Orbix de IONA, y TAO de Object Computing, Inc.

Las **tecnologías basadas en componentes**, como Microsoft COM, Microsoft DCOM, Java Beans, y Enterprise Java Beans, también se basan en los paradigmas de objetos distribuidos, ya que los componentes son esencialmente objetos empaquetados y especializados que se han diseñado para interactuar con otros por medio de interfaces estándar. Además, los **servidores de aplicaciones**, populares para aplicaciones empresariales, son herramientas de tipo *middleware* que proporcionan acceso a objetos o componentes.

Espacio de objetos

Quizás el más abstracto de los paradigmas orientados a objetos, el paradigma de Espacio de Objetos asume la existencia de entidades lógicas conocidas como **espacios de objetos**. Los participantes en una aplicación convergen en un espacio de objetos común. Un suministrador coloca objetos como entidades dentro de un espacio de objetos, y los solicitantes que se subscriben al espacio pueden acceder a dichas entidades. La Figura 3.10 ilustra este paradigma.

Además del grado de abstracción proporcionado por los otros paradigmas, el paradigma del espacio de objetos proporciona un espacio virtual o sala de reunión entre suministradores y solicitantes de recursos de red, como objetos. Esta abstracción oculta los detalles implicados en la búsqueda de recursos u objetos que son necesarios en paradigmas como la invocación de métodos remotos, ORB, o servicios de red. Además, la exclusión mutua es inherente al paradigma, debido a que un objeto en el espacio puede sólo ser usado por un participante a la vez.

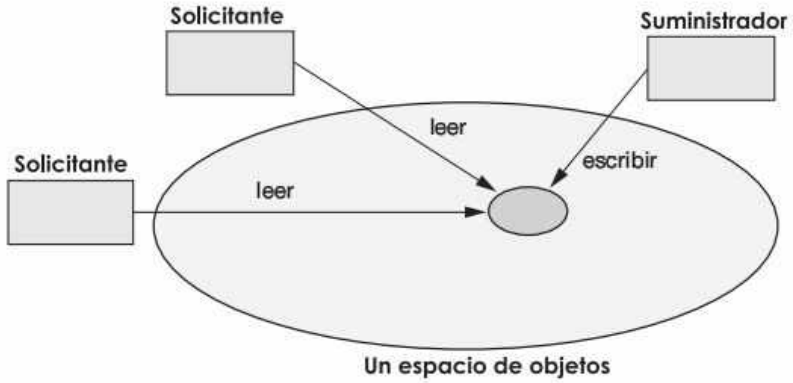


Figura 3.10. El paradigma de Espacio de Objetos.

Para el sistema de subastas, todos los participantes así como el suministrador de servicios se suscribirían al mismo espacio de objetos. Cada participante depositaría un objeto en el espacio de objetos para registrarse en la sesión y para ser notificado del comienzo de la misma. Al comienzo de la sesión, el subastador deposita otro objeto en el espacio de objetos. El objeto contiene información sobre el elemento a subastar así como el histórico de las pujas. Un participante que desee realizar una puja tomará el objeto del espacio y, si así lo quiere, añadirá una nueva puja en el objeto antes de devolverlo al espacio de objetos. Al final de la sesión el subastador retirará el objeto del espacio y contactará con el participante con la mayor puja.

Un conjunto de herramientas basado en este paradigma es JavaSpaces [java.sun.com, 15].

Paradigma de agentes móviles

Un **agente móvil** es un programa u objeto transportable. En este paradigma, un agente se lanza desde un determinado ordenador. El agente entonces viaja de forma autónoma de un ordenador a otro de acuerdo con un itinerario que posee. En cada parada, el agente accede a los recursos o servicios necesarios y realiza las tareas correspondientes para completar su misión. El paradigma se ilustra en la Figura 3.11.

El paradigma ofrece la abstracción de **programa u objeto transportable**. En lugar de intercambio de mensajes, los datos son transportados por el programa/objeto mientras el propio objeto se transfiere entre los participantes.

El paradigma de agentes móviles proporciona una novedosa forma de implementar nuestro sistema de subastas. En comienzo, cada participante lanza un agente móvil hacia el subastador. El agente transporta la identidad, incluyendo la dirección de red, del participante al que representa. Una vez que la sesión ha comenzado el subastador lanza un agente móvil que transporta el itinerario de los participantes, así como la mayor puja. El agente móvil circula entre los participantes y el subastador hasta que finalice la sesión, en cuyo instante el subastador lanza el agente para dar un recorrido más entre todos los participantes para comunicar el resultado.

Paquetes comerciales que den soporte al paradigma de agentes móviles son el sistema concordia [meitca.com, 16], y el sistema Aglet [trl.ibm.co.jp, 17]. Hay también muchos sistemas experimentales basados en este paradigma, incluyendo D'agent [agent.cs.dartmouth.edu, 13] y el proyecto Tacoma [tacoma.cs.uit.no, 14].

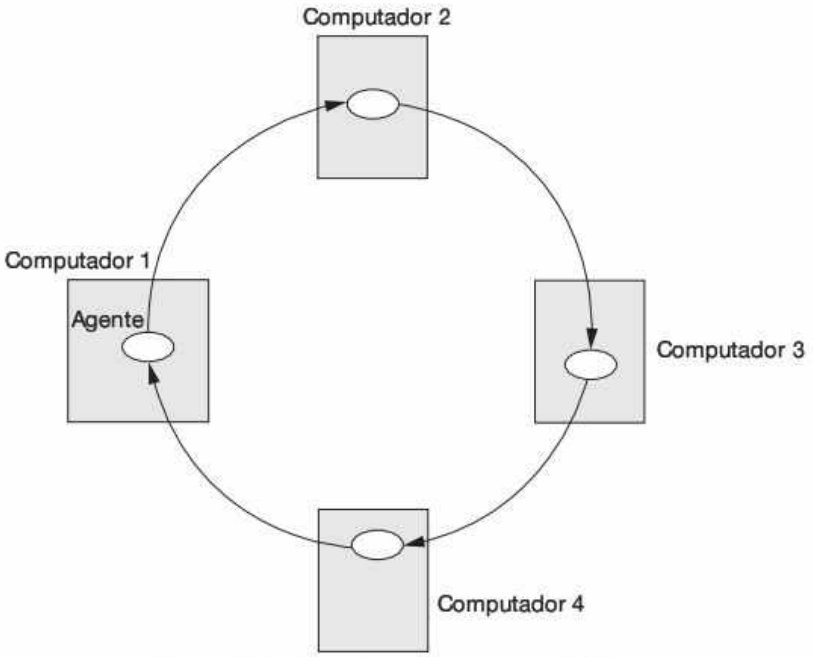


Figura 3.11. Paradigma de agentes móviles.

Paradigma de servicios de red

En el paradigma mostrado en la Figura 3.12, los proveedores de servicios se registran en los servidores de directorio de una red. Un proceso que desee un servicio particular contacta con el servidor de directorio en tiempo de ejecución, y, si el servicio está disponible, al proceso se le dará una referencia a dicho servicio. Usando esta referencia, el proceso interactuará con el servicio.

El paradigma es esencialmente una extensión del paradigma de invocación de métodos remotos. La diferencia es que los objetos de servicio se registran en un directorio global, permitiéndoles ser localizados y accedidos por solicitantes de servicios dentro de una red federada. Idealmente, los servicios se pueden registrar y localizar usando un identificador único global, en cuyo caso el paradigma ofrece un nivel de abstracción extra: **transparencia de localización**. La trasparencia de localización permite a los desarrolladores de software acceder a un objeto o servicio sin tener que ser consciente de la localización del objeto o servicio.

Un **método de *callback*** es un método proporcionado por quien solicita un servicio de forma que el proveedor del servicio pueda iniciar una llamada a un solicitante particular. Los métodos de *callback* se utilizan también en el paradigma de objetos distribuidos. El Capítulo 8 cubre el tratamiento de *callbacks* en Java RMI.

La implementación de nuestro sistema de subastas es la misma que bajo el paradigma RMI excepto que el subastador se registra en el servicio de directorio, permitiendo que los participantes lo localicen y, una vez que la sesión ha comenzado hacer las pujas. Los participantes proporcionan un método *callback* que permite que el subastador anuncie el comienzo y el final de la sesión y que actualice el estado de la misma.

La tecnología Jini de Java [jini.org, 4] se basa en este paradigma. El protocolo SOAP (*Simple Object Access Protocol*), que se estudiará en el Capítulo 11, aplica este paradigma a los servicios accesibles en la web.

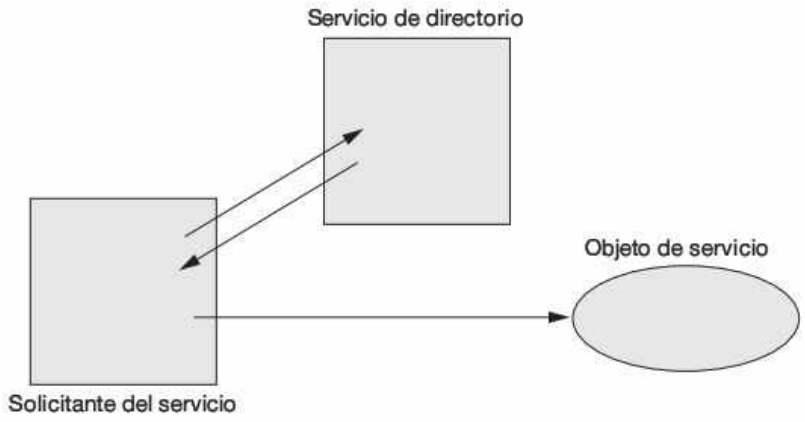


Figura 3.12. El paradigma de servicios de red.

Paradigma de aplicaciones colaborativas (*groupware*)

En este modelo para trabajo cooperativo basado en ordenador, los procesos participan en grupo en una sesión colaborativa. Cada proceso participante puede hacer contribuciones a todos o parte del grupo. Los procesos pueden hacer eso, usando multidifusión para enviar los datos o usar pizarras o tablones virtuales, los cuales permiten a cada participante leer y escribir datos sobre una visualización compartida. Los dos paradigmas de *groupware* se muestran en la Figura 3.13.

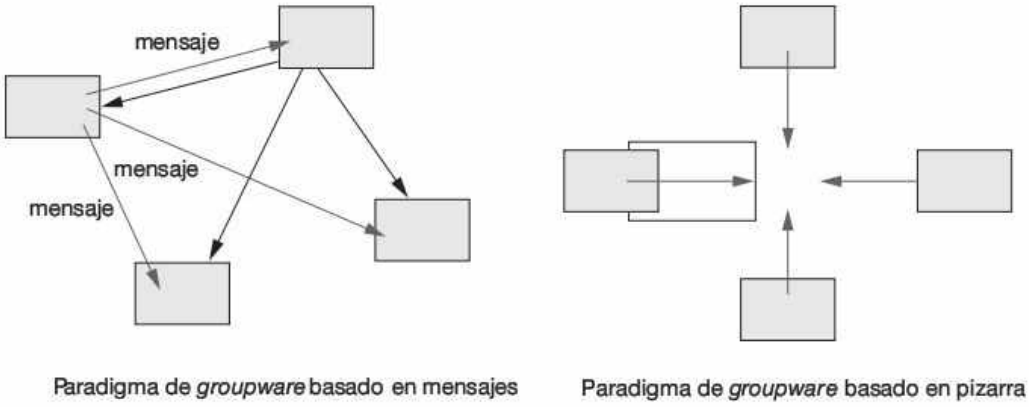


Figura 3.13. El paradigma de aplicaciones colaborativas.

Para implementar el sistema de subastas usando el paradigma de *groupware* basado en mensajes, el subastador iniciará un grupo, al que se incorporarán los participantes interesados. Al comienzo de la sesión, el subastador difundirá un mensaje de multidifusión anunciando el comienzo. Durante la sesión, cada puja es un mensaje de multidifusión a todos los participantes de forma que cada uno puede independientemente acceder al estado de la subasta. Finalmente, el subastador termina la sesión por medio de un mensaje de multidifusión anunciando el resultado.

No es complicado ver cómo el paradigma basado en pizarra se puede utilizar en nuestro sistema de subastas. El subastador y los participantes comparten una pizarra

virtual. El subastador comienza el proceso de puja escribiendo el anuncio sobre la pizarra. Posteriormente, cada participante puede realizar una puja escribiéndola en la pizarra. Finalmente, el subastador termina la sesión escribiendo el anuncio final. El paradigma colaborativo es la base de un gran número de programas *groupware* como Lotus QuickPlace [lotus.com, 21]. Interfaces de programación de aplicaciones para la comparición vía intercambio de mensajes incluye el API de Java de multidifusión y el *Java Shared Data Toolkit* (JSDT) [java.sun.com, 18]. El paradigma basado en pizarra es la base de un número de aplicaciones como SMART Board [smarttech.com, 19], NetMeeting [Microsoft.com, 20], y Groove [groove.com, 27]. Se ha propuesto al protocolo NSTP (*Notification Service Transfer Protocol*) como «una infraestructura para construir aplicaciones *groupware* síncronas. Se basa en la idea de un servidor de notificación coordinado, que es independiente de cualquier aplicación *groupware* específica. Se ha intentado que NSTP sea de alguna manera el equivalente síncrono de HTTP (*Hypertext Transfer Protocol*)» [Day, Patterson, y Mitchell, 28].

3.4. COMPARATIVA

Como se ha mostrado en nuestra discusión, una misma aplicación se puede implementar usando cualquiera de los paradigmas. Dado el gran número de paradigmas y herramientas disponibles, ¿cómo un desarrollador de software puede decidir la más apropiada para una tarea dada?

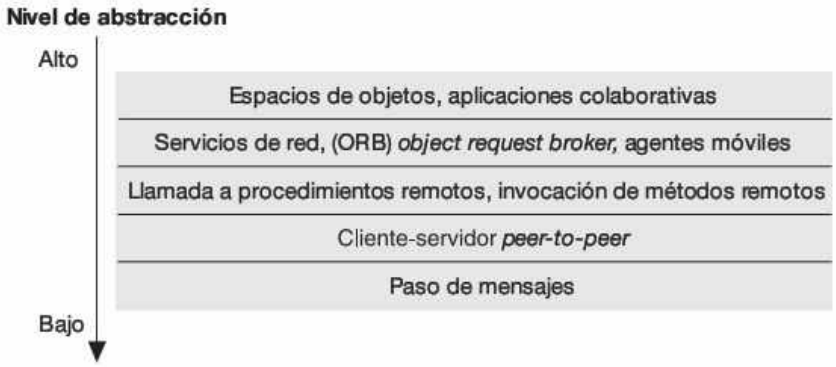


Figura 3.14. Los paradigmas y los niveles de abstracción.

Para responder a esta pregunta, uno debe tener cuidado con lo que ofrece cada una de las diferentes alternativas. Es suficiente decir que cada paradigma o herramienta tiene alguna ventaja sobre los otros. Cada ventaja, sin embargo, puede ser anulada por una desventaja. En los siguiente párrafos se verán algunas de las cuestiones que se deben considerar.

Nivel de abstracción frente a sobrecarga

La Figura 3.14 muestra los paradigmas que se han visto y sus niveles de abstracción correspondientes. En el nivel más bajo se encuentran los paradigmas más básicos: *paso de mensajes*, y *cliente-servidor*. En el nivel más alto están los *espacios de objetos* o la *computación colaborativa*, que proporciona el nivel más alto de abstracción. El

desarrollo de una aplicación altamente compleja se puede ver muy ayudado por una herramienta que ofrece un gran nivel de abstracción. Pero la abstracción tiene un precio: sobrecarga.

Considérese el paradigma de *invocación de métodos remotos*, por ejemplo. Como se podrá ver en el Capítulo 8, para poder proporcionar la abstracción de un método remoto, es necesaria la presencia de unos módulos, conocidos como resguardo (*stub*) y esqueleto (*skeleton*), en tiempo de ejecución para manejar detalles de la comunicación entre procesos. El soporte en tiempo de ejecución y la existencia de módulos adicionales requieren recursos extra del sistema así como tiempo de ejecución. Así, en el caso de que todo lo demás fuese igual, una aplicación escrita usando RMI requerirá más recursos del sistema y tardará más en ejecutar que otra igual que se comunicase por medio del API de *sockets*. Por esta razón, el API de *sockets* puede ser la más apropiada para una aplicación que requiere un tiempo de respuesta rápido y unos recursos mínimos de sistema. Por otro lado, RMI y otras herramientas que proporcionan mayor nivel de abstracción permiten que una aplicación se desarrolle más rápidamente, y por tanto es más apropiada si el tiempo de respuesta y el consumo de recursos no es una pega.

Escalabilidad

La complejidad de una aplicación distribuida se incrementa significativamente, posiblemente de forma exponencial, cuando el número de participantes (procesos u objetos) se incrementa. Considérese la aplicación de ejemplo, el sistema de subastas. Tal y como se ha descrito, el subastador debe manejar las direcciones de los participantes de forma que pueda anunciar el comienzo y final de la sesión. Además, el subastador tiene que contactar repetidas veces con los participantes, de forma individual, para notificarles la puja más alta. Utilizando paso de mensajes, el programador debe codificar la gestión de direcciones y la forma de contactar individualmente con cada uno de ellos. La complejidad crece cuando crece el número de participantes.

Con un conjunto de herramientas de un paradigma de alto nivel, como *espacios de objetos* o un *sistema de mensajes de tipo publicación/suscripción*, la complejidad de gestionar los participantes la realiza el sistema. Una aplicación así implementada se puede acomodar a un incremento en el número de participantes sin una complejidad adicional.

Los *agentes móviles* son otro paradigma que permite a una aplicación escalar buena, mientras que el número de ordenadores participantes puede incrementarse sin un impacto significativo en la complejidad del programa basado en este paradigma.

Soporte multi-plataforma

Los paradigmas, al ser modelos abstractos, son inherentemente independientes de plataforma. Las herramientas basadas en los paradigmas, por otro lado, pueden ser dependientes de plataforma, y muchas de ellas lo son. Con el objeto de proporcionar generalidad, una herramienta que soporte plataformas heterogéneas necesariamente incrementa la complejidad, si se compara con una que soporte una única plataforma. Por esta razón, la sintaxis de programación también tiende a ser más pesada.

Muchas de las tecnologías Java, incluida el API Java RMI y los JavaSpaces, por omisión sólo ejecutan sobre máquinas virtuales Java. Como resultado, si se utilizan

estas tecnologías, todos los participantes en una aplicación deben estar escritos en lenguaje Java. De la misma forma, las tecnologías COM/DCOM sólo son desplegables sobre plataformas Microsoft.

Por contraste, CORBA es una arquitectura diseñada para dar soporte multi-plataforma; de forma que las herramientas basadas en esta tecnología pueden soportar programas escritos en diversos lenguajes y también procesos ejecutando sobre distintas plataformas.

Más allá de estas ventajas e inconvenientes, hay consideraciones de ingeniería de software que se deben considerar cuando se selecciona una herramienta. Algunas de ellas son:

- La madurez y estabilidad de la herramienta.
- La tolerancia a fallos ofrecida por la herramienta.
- La disponibilidad de herramientas de desarrollo.
- La mantenibilidad.
- La reutilización de código.

RESUMEN

Este capítulo ha repasado una amplia gama de paradigmas para aplicaciones distribuidas. Los paradigmas que se han presentado son:

- Paso de mensajes.
- Cliente-servidor.
- *Peer-to-peer*.
- Sistemas de mensajes:
  - — Punto a punto.
  - — Publicación/suscripción.
- Llamada a procedimientos remotos.
- Objetos distribuidos:
  - — Invocación de métodos remotos.
  - — *Object request brokers*.
  - — Espacio de objetos.
- Agentes móviles
- Servicios de red
- Aplicaciones colaborativas

A varios niveles, estos paradigmas proporcionan abstracción que aísla a los programadores de los detalles de comunicación entre procesos y la sincronización de eventos, permitiendo a quien desarrolla, concentrarse en la visión general de la aplicación completa.

En la elección de un paradigma o de una herramienta para una aplicación, existen ventajas e inconvenientes que se deben considerar, incluyendo la sobrecarga, escalabilidad, soporte multi-plataforma y consideraciones de ingeniería del software.

EJERCICIOS

1. Considere la implementación de una sala de *chat* sencilla donde los participantes acuden a un lugar de reunión virtual e intercambian mensajes entre todos los que están presentes en la sala.

   Explique cómo se podría aplicar cada uno de los paradigmas vistos en el capítulo en su implementación.

   Compare la adecuación de los paradigmas para esta aplicación. Desde el punto de vista del programador, ¿cuál o cuáles de los paradigmas parecen los más naturales? ¿Cuál o cuáles los menos naturales?
2. Considere las ventajas e inconvenientes que hemos visto:
   a. ¿Se le ocurre alguna comparativa adicional?
   b. Compare y contraste las fortalezas y debilidades de cada paradigma que se han visto, en términos de estas ventajas e inconvenientes adicionales.
3. Considere los paradigmas vistos. Para cada uno, describa una aplicación para la cual el paradigma es más apropiado. Explíquelas.
4. En muchos de los paradigmas que se han tratado hay implicado un *middleware*, un módulo software que sirve de intermediario entre los participantes de una aplicación:
   a. Considere el modelo de sistema de mensajes publicación/suscripción. ¿Cómo el *middleware* en este paradigma permite publicar y suscribir?
   b. ¿Cómo puede el *middleware* permitir el soporte multi-plataforma?
   c. ¿Cómo puede el *middleware* proporcionar comunicación asíncrona entre procesos?
   d. ¿Cuáles de los paradigmas necesitan un *middleware*? Explíquelo.
5. Suponga que está contruyendo un sistema software para tratar los registros de gastos. Usando el sistema, cada empleado puede transmitir una petición *on-line* de gasto, y posteriormente recibe una aprobación o denegación, también *on-line*. Un empleado puede también enviar un registro para un gasto una vez que éste ha sido cursado. Sin entrar en aspectos muy específicos, elija un paradigma para el sistema, justifique su elección, y describa cómo aplicaría dicho paradigma en la aplicación.

REFERENCIAS

1. David J. Barnes. ***Object-Oriented Programming***. Upper Saddle River, NJ: Prentice Hall, 1999.
2. Douglas E. Comer and David L. Stevens. ***Internetworking with TCP/IP***, Vol. 3: ***Client-Server Programming and Applications***. Upper Saddle River, NJ: Prentice Hall, 2001.
3. Elliotte Rusty Harold. ***Java Network Programming***. Sebastopol, CA: O'Reilly, 1997.
4. Welcome to Jini.org!, ***http://www.jini.org/***
5. Alan Dickman. ***Designing Applications with Msmq: Message Queuing for Developers***. Reading, MA: Addison-Wesley, 1998.
6. IBM MQ Series Family home page, ***http://www-4.ibm.com/software/ts/mqseries/***
7. John Wetherill, Messaging Systems and the Java™ Message Service, ***http://developer.java.sun.com/developer/technicalArticles/Networking/messaging/***

8. RFC1831: Especificación del Protocolo RFC, Remote Procedure Call Protocol Specification Version 2, August 1995, *http://www.ietf.org/rfc/rfc1831.txt*
9. DCE1.1; Remote Procedure Call, Open Group Standard, Document Number C706 August 1997, *http://www.opennc.org/public/pubs/catalog/c706.htm*
10. The Object Management Group homepage, *http://www.corba.org/*
11. Thomas J. Mowbray and Ron Zahavi. ***The Essential CORBA***. New York, NY: Wiley, 1995.
12. The Community Resource for Jini Technology, *http://jini.org/*
13. D'Agents: Mobile Agents at Dartmouth College, *http://agent.cs.dartmouth.edu/*
14. TACOMA—Operating system support for agents, *http://www.tacoma.cs.uit.no/*
15. JavaSpaces™ Technology, *http://java.sun.com/products/javaspaces/*
16. Concordia's welcome page, *http://www.meitca.com/HSL/Projects/Concordia/Welcome.html*
17. IBM Aglets Software Development Kit, *http://www.trl.ibm.co.jp/aglets/*
18. Java Shared Data Toolkit User Guide, *http://java.sun.com/products/java-media/jsdt/2.0/jsdt-guide/introduction.doc.html#15891* and *http://java.sun.com/products/java-media/jsdt/2.0/jsdt-guide/jsdtTOC.fm.html*
19. SMART Board Interactive Whiteboard, Software Features, *http://www.smarttech.com/products/smartboard/software.asp*
20. NetMeeting Home, *http://www.microsoft.com/windows/netmeeting/*
21. IBM Lotus Software—QuickPlace, *http://lotus.com/*
22. Microsoft Message Queueing, *http://msdn.microsoft.com/library/psdk/msmq/msmq_overview_4ilh.htm*
23. Java socket API, *http://java.sun.com/producsts/jdk/1.2/docs/api/index.html*
24. Winsock Development Information, *http://www.sockets.com/*
25. The World Wide Web Consortium (W3C), Simple Object Access Protocol (SOAP), *http://www.w3.org/TR/SOAP/*
26. The XNSORG homepage, *http://www.xns.org/*
27. Groove homepage, *http://www.groove.net/*
28. Mark Day, John F. Patterson, David Mitchell, The Notification Service Transfer Protocol (NSTP): Infrastructure for Synchronous Groupware, Lotus Technical Report 96-13, *http://www.scope.gmd.de/info/www6/technical/paper080/paper80.html*
29. jxta.org, *http://www.jxta.org/*, Project JXTA home site.
30. Jabber Software Foundation, *http://www.jabber.org/*, Jabber Software Foundation home site.

CAPÍTULO 4

El API de *sockets*

Este capítulo introduce la primera herramienta de programación para implementar comunicaciones entre procesos: el **API de *sockets***.

Como el lector podrá recordar del Capítulo 2, el API de *sockets* es un mecanismo que proporciona un nivel bajo de abstracción para IPC. Se presenta en este punto por su simplicidad. Aunque los programadores de aplicaciones apenas tienen que codificar en este nivel, la comprensión del API de *sockets* es importante al menos por dos razones. En primer lugar, los mecanismos de comunicación proprocionados en estratos superiores se construyen sobre el API de *sockets*; o sea, se implementan utilizando las operaciones proporcionadas por el API de *sockets*. En segundo lugar, para aquellas aplicaciones en las que es primordial el tiempo de respuesta o que se ejecutan sobre una plataforma con recursos limitados, el API de *sockets* puede ser el mecanismo de IPC más apropiado, o incluso el único disponible.

4.1. ANTECEDENTES

El API de *sockets* aparece por primera vez a principios de la década de los 80 como una biblioteca de programación que proporcionaba la funcionalidad de IPC en una versión del sistema operativo UNIX conocida como **Unix de Berkeley** (BSD 4.2). Actualmente los principales sistemas operativos dan soporte al API de *sockets*. En los sistemas basados en UNIX tales como BSD o Linux, el API es parte del núcleo, o *kernel*, del sistema operativo. En los sistemas operativos de computadores personales tales como MS-DOS, Windows NT (y sus variantes), Mac-OS y OS/2, el API se proporciona como bibliotecas de programación. (En los sistemas Windows, a esta API se

la conoce como ***Winsock***). Java, un lenguaje diseñado teniendo en cuenta la programación de aplicaciones en red, proporciona el API de *sockets* como parte de las clases básicas del lenguaje. Todas estas interfaces de programación de *sockets* comparten el mismo modelo de paso de mensajes y una sintaxis muy similar.

En este capítulo, se usará como caso representativo el API de *sockets* de Java.

Socket (en castellano, enchufe) es un término tomado del campo de las comunicaciones telefónicas. En los primeros días de la telefonía (anteriores al siglo XX), cuando una persona quería hacer una llamada a otra tenía que ser a través de un operador, el cual manualmente establecía una conexión introduciendo los dos extremos de un cable dentro de dos receptáculos específicos, cada uno asignado a uno de los dos interlocutores, sobre un panel de *sockets* (enchufes). La desconexión también la realizaba el operador manualmente. Esta metáfora fue la base del API de *sockets* para comunicación entre procesos.

4.2. LA METÁFORA DEL *SOCKET* EN IPC

Inspirándose en la terminología de la telefonía, el diseñador del API de *sockets* ha proporcionado una construcción de programación denominada *socket*. Cuando un proceso desea comunicarse con otro, debe crear una instancia de tal construcción (véase la Figura 4.1). Sin embargo, a diferencia de la telefonía primitiva, la comunicación entre los interlocutores puede ser orientada a conexión o sin conexión. Por claridad, primero se presentará el API de *sockets* sin conexión.

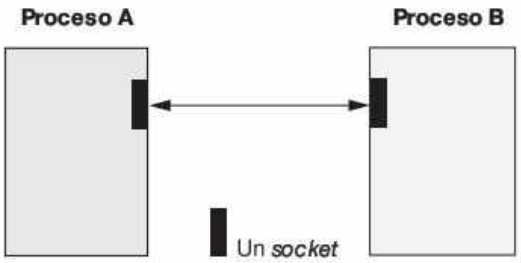


Figura 4.1. El modelo conceptual del API de *sockets*.

4.3. EL API DE *SOCKETS* DATAGRAMA

Como el lector podrá recordar del Capítulo 1 de este libro o de cualquier otro sitio, hay dos protocolos fundamentales en el nivel de transporte de la arquitectura de Internet: el protocolo de datagrama de usuario (**UDP**, *User Datagram Protocol*) y el protocolo de control de transmisión (**TCP**, *Transmission Control Protocol*).

El **protocolo de datagrama de usuario** (UDP) permite que un paquete se transporte (es decir, se envíe o se reciba en el nivel de transporte) utilizando comunicaciones sin conexión. El paquete de datos transportado de esta manera se denomina **datagrama**. Conforme a la comunicación sin conexión, cada datagrama transportado es dirigido y encaminado individualmente y puede llegar al receptor en cualquier orden. Por ejemplo, si el proceso 1 en la máquina A envía sucesivamente mensajes transportados en los datagramas m_1 y m_2 al proceso 2 en la máquina B, los datagramas

En la terminología de las redes de datos, un *paquete* es una unidad de datos transmitida por la red. Cada paquete contiene los datos (la carga, en inglés *payload*) y alguna información de control (la cabecera), que incluye la dirección de destino.

pueden transportarse sobre la red por diferentes rutas, y pueden llegar al proceso receptor en cualquiera de los dos órdenes posibles: m_1-m_2 o m_2-m_1.

El **protocolo de control de transmisión** (TCP) está orientado a conexión y transporta un flujo de datos sobre una conexión lógica establecida entre el emisor y el receptor. Gracias a la conexión, se garantiza que los datos mandados desde un emisor a un receptor van a ser recibidos en el mismo orden que se enviaron. Por ejemplo, si el proceso 1 en la máquina A manda sucesivamente mensajes transportados en m_1, y m_2 al proceso 2 que ejecuta en la máquina B, el proceso receptor puede asumir que los mensajes se le entregarán en el orden m_1-m_2 y no m_2-m_1.

El API de *sockets* de Java, como el resto de interfaces de programación de *sockets*, proporciona construcciones de programación de *sockets* que hacen uso tanto del protocolo UDP como TCP. Los *sockets* que utilizan UDP para el transporte son conocidos como ***sockets* datagrama**, mientras que los que usan TCP se denominan ***sockets stream*** Debido a su relativa simplicidad, en primer lugar se presentarán los *sockets* datagrama.

El *socket* datagrama sin conexión

Puede parecer sorprendente, pero los *sockets* datagrama pueden dar soporte tanto a una comunicación sin conexión como a una orientada a conexión en el nivel de aplicación (*véase* la Figura 4.2). Esto se debe a que, aunque los datagramas se envían o reciben sin la noción de conexiones en el nivel de transporte, el soporte en tiempo de ejecución del API de *sockets* puede crear y mantener conexiones lógicas para los datagramas intercambiados entre dos procesos, como se mostrará en la próxima sección.

En Java, el API de *sockets* datagrama proporciona dos clases:

1. La clase *DatagramSocket* para los *sockets*.
2. La clase *DatagramPacket* para los datagramas intercambiados.

El **soporte en tiempo de ejecución** de una API es un conjunto de software que está enlazado al programa durante su ejecución para dar soporte al API.

Un proceso que quiera mandar o recibir datos utilizando esta API debe instanciar un objeto *DatagramSocket*, o un *socket* para abreviar. Se dice que cada *socket* está **enlazado** a un puerto UDP de la máquina que es local al proceso (es decir, la máquina en la que se está ejecutando el proceso). Recuérdese del Capítulo 1 que en IPv4 los números de puerto válidos van desde el 0 al 65.535, estando reservados desde el 0 al 1023 para los servicios de carácter estándar, denominados «bien conocidos» (*well-known*) en la terminología de Internet.

Es muy recomendable que se desarrolle el hábito de consultar la documentación en línea del API de Java [java.sun.com, 1] para obtener la definición más actualizada de cada clase Java presentada. Se debería comprobar también en el API en línea cuál es la definición exacta y actual de un método o constructor.

Para mandar un datagrama a otro proceso (que ha instanciado presumiblemente su *socket* en una dirección local de, por ejemplo, la máquina *m* y el puerto *p*), un proceso debe crear un objeto que representa el datagrama en sí mismo. Este objeto puede crearse instanciando un objeto *DatagramPacket* que englobe (1) una referencia a un vector de octetos que contenga los datos de la carga, y (2) la dirección de destino (el ID de la máquina y el número de puerto al que el *socket* del receptor está enlazado, en este caso, *m* y *p*, respectivamente). Una vez que se crea el objeto *DatagramPacket* y en él se incluyen los datos de la carga y del destino, el proceso emisor realiza una llamada al método *send* del objeto *DatagramSocket*, especificando una referencia al objeto *DatagramPacket* como argumento.

Los **datos de la carga** se denominan de esta manera para diferenciarlos de los *datos de control*, que incluyen la dirección de destino y se transportan también en un datagrama.

En el proceso receptor, también se debe instanciar un objeto *DatagramSocket* y enlazarlo a un puerto local; el número de puerto debe coincidir con el especificado en el paquete datagrama del emisor. Para recibir los datagramas enviados al *socket*, el proceso crea un objeto *DatagramPacket* que hace referencia a un vector de octetos y llama a un método *receive* de su objeto *DatagramSocket*, especificando como argumento una referencia al objeto *DatagramPacket*.

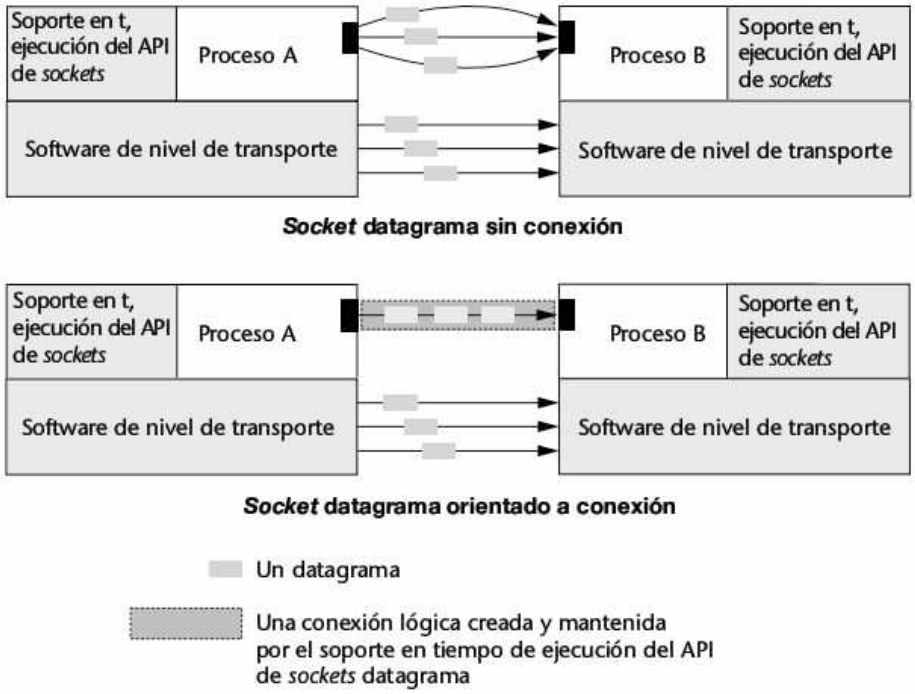


Figura 4.2. *Socket* datagrama sin conexión y orientado a conexión.

La Figura 4.3 ilustra las estructuras de datos usadas en los programas de los dos procesos, mientras que la Figura 4.4 ilustra el flujo de programa de los dos procesos.

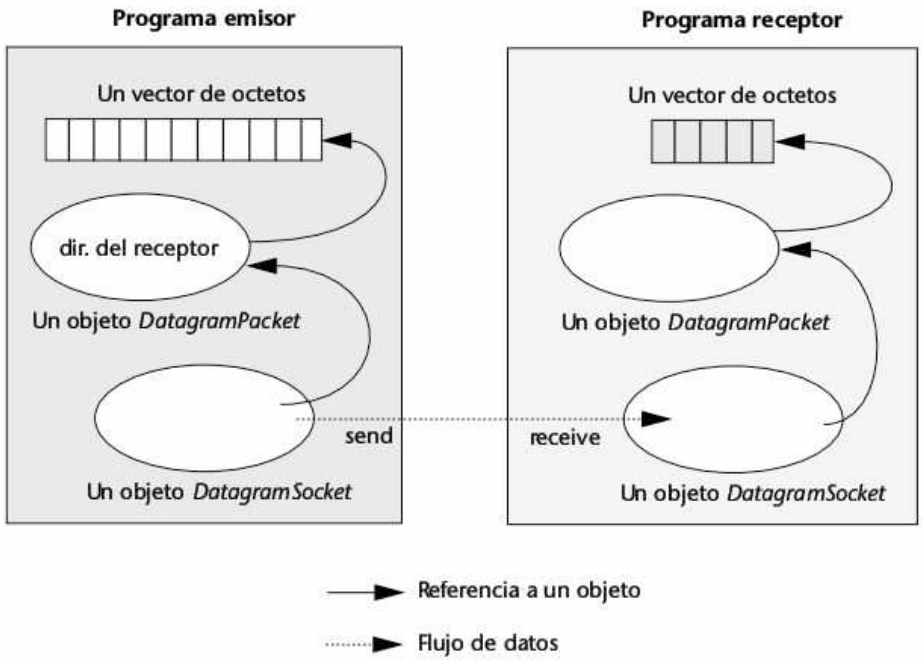


Figura 4.3. Las estructuras de datos en el programa emisor y en el receptor.

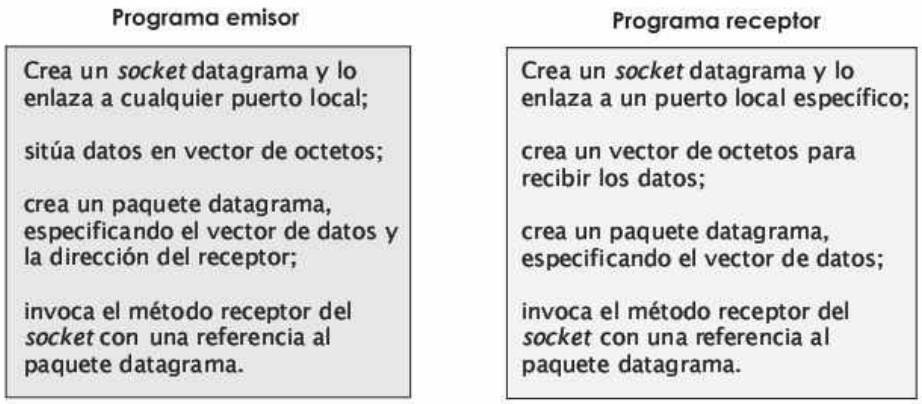


Figura 4.4. El flujo de programa en el proceso emisor y en el receptor.

Con los *sockets* sin conexión, un *socket* enlazado a un proceso puede utilizarse para mandar datagramas a diferentes destinos. Es también posible que múltiples procesos manden simultáneamente datagramas al mismo *socket* enlazado a un proceso receptor, en cuyo caso el orden de llegada de estos mensajes será impredecible, de acuerdo con el protocolo UDP subyacente. La Figura 4.5a ilustra un escenario donde un proceso, *A* utiliza un único *socket* sin conexión para comunicarse con otros dos procesos en una sesión. Por ejemplo, *A* puede recibir un datagrama m_1 de *B*, seguido por un datagrama m_2 de *C*, después m_3 y m_4 de *B*, seguidos de m_5 de *C*, y así sucesivamente. Alternativamente, es también posible que *A* abra un *socket* separado para cada proceso *B* y *C*, de manera que los datagramas de los dos procesos se puedan dirigir y recibir por los dos *sockets* separados (*véase* la Figura 4.5b).

La Tabla 4.1 resume los métodos principales y los constructores de la clase *DatagramPacket*, mientras que la Tabla 4.2 recopila los de la clase *DatagramSocket*. Recuerde que hay muchos más métodos de los que se presentan en esta tabla.

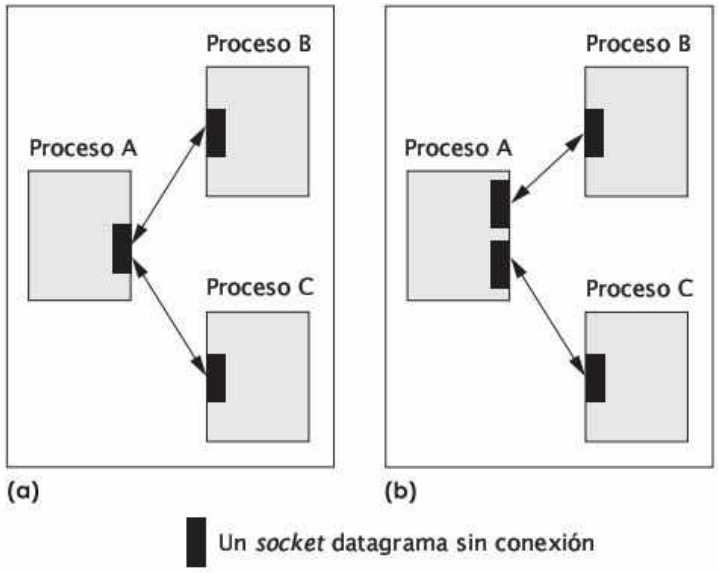


Figura 4.5. *Sockets* datagrama sin conexión.

Tabla 4.1. Métodos principales de la clase *DatagramPacket.*

Método/Constructor	Descripción
DatagramPacket(byte[] almacen, int longitud)	Construye un paquete datagrama para recibir paquetes de longitud *longitud*; los datos recibidos se almacenarán en el vector de octetos asociado a *almacén.*
DatagramPacket(byte[]almacen, int longitud, InetAddress direccion, int puerto) (*Nota*: La clase *InetAddress* representa una dirección IP.)	Construye un paquete datagrama para enviar paquetes de longitud *longitud* al *socket* enlazado al número de puerto y a la máquina especificados; los datos se almacenan en el vector de octetos asociado a *almacén.*
DatagramSocket()	Construye un *socket* datagrama y lo enlaza a cualquier puerto disponible en la máquina local; este constructor lo puede utilizar un proceso que manda datos y no necesita recibirlos.

Tabla 4.2. Métodos principales de la clase *DatagramSocket.*

Método/Constructor	Descripción
DatagramSocket(int puerto)	Construye un *socket* datagrama y lo enlaza al puerto especificado en la máquina local; este número de puerto se puede especificar después en un paquete datagrama destinado a este *socket.*
void close( )	Cierra este objeto *datagramSocket.*
void receive(DatagramPacket p)	Recibe un paquete datagrama utilizando este *socket.*
void send(DatagramPacket p)	Envía un paquete datagrama utilizando este *socket.*
void setSoTimeout(int plazo)	Fija un plazo máximo de espera en milisegundos para las operaciones de recepción bloqueantes realizadas con este *socket.*

La Figura 4.6 ilustra la sintaxis básica mediante un par de programas que se comunican utilizando *sockets* datagrama.

```
//Extracto de un programa receptor
Datagram Socket ds=
    new DatagramSocket(2345);
DatagramPacket dp=
   new DatagramPacket (buffer, MAXLON);
ds.receive(dp);
lon=dp.getLength( );
System.out.Println (lon + "octetosrecibidos");
Strings= newString(dp.getData( ),0,lon);
System.out.println(dp.getAddress( )+
   "en el puerto" + dp.get Port( ) + "dice" + s
```

```
//Extracto de un proceso emisor
InetAddress maquina Receptora=
   Inet Address.getByName("localHost");
DatagramSocketelSocket=new DatagramSocket();
String mensaje="¡Hola,mundo!";
byte[ ] datos=mensaje.getBytes( );
data = theLine.getBytes( );
DatagramPacket elPaquete=
   new DatagramPacket (datos, datos. length,
      maquinaReceptora,2345);
el Socket.send ( el Paquete);
```

Figura 4.6. Uso del API de *sockets* datagrama sin conexión en programas.

Sincronización de eventos en los *sockets* datagrama

En las interfaces de programación de *sockets* básicas, ya sean orientadas a conexión o sin conexión, las operaciones *send* son **no bloqueantes**, mientras que las operaciones *receive* son **bloqueantes**. Un proceso continuará con su ejecución después de realizar una llamada al método *send*. Sin embargo, una llamada al método *receive*, una vez invocada por un proceso, causará que el proceso se suspenda hasta que se reciba realmente un datagrama. Para evitar un bloqueo indefinido, el proceso receptor puede utilizar el método *setSoTimeout* para fijar un plazo máximo de tiempo de bloqueo, por ejemplo, 50 segundos. Si no se recibe ningún dato durante este plazo de tiempo, se activará una excepción Java (específicamente, ocurrirá una *java.io.InterruptedIOException*) que puede capturarse en el código para manejar la situación de manera apropiada.

La Figura 4.7 es un diagrama de eventos que muestra una sesión de un protocolo petición–respuesta utilizando *sockets* datagrama.

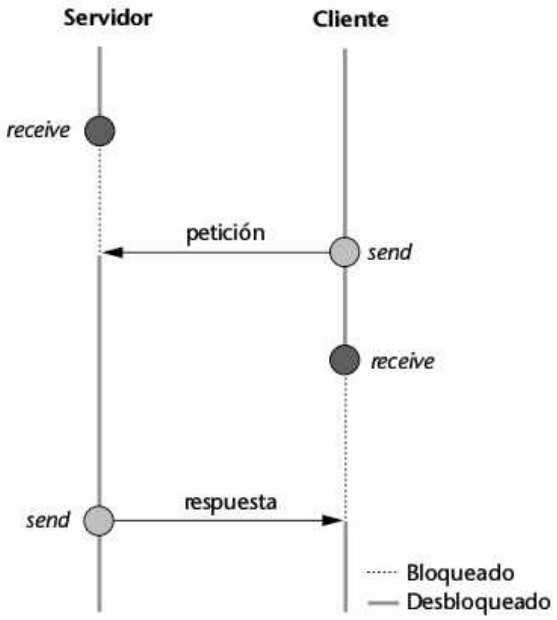


Figura 4.7. Sincronización de eventos con *sockets* sin conexión.

Ejemplo 1. Las Figuras 4.8 y 4.9 ilustran el código de dos programas que utilizan *sockets* datagrama para intercambiar una única cadena de datos. Por diseño, la lógica de los programas es la más sencilla posible para subrayar la sintaxis básica de las comunicaciones entre procesos. Nótese que el emisor crea un paquete datagrama que contiene una dirección de destino (*véanse* las líneas desde la 31 a la 33 en la Figura 4.8), mientras que el paquete datagrama del receptor no incluye una dirección de destino (*véanse* las líneas 31 y 32 en la Figura 4.9). Nótese también que el *socket* del emisor se enlaza a un número de puerto no especificado (*véase* la línea 28 de la Figura 4.8), mientras que el *socket* del receptor se enlaza a un número de puerto especificado (*véase* la línea 28 en la Figura 4.9) para que el emisor pueda especificar este número de puerto en su datagrama (*véase* la línea 33 en la Figura 4.8) como des-

tino. Se debería mencionar también que por simplicidad los programas de ejemplo utilizan una sintaxis rudimentaria (líneas 37-39 en *Ejemplo1Emisor* y 38-40 en *Ejemplo1Receptor*) para manejar excepciones. En una aplicación real, es necesario a menudo manejar las excepciones utilizando un código más refinado.

Figura 4.8. *Ejemplo1Emisor.java.*

```
1  import java.net.*;
2  import java.io.*;
3
4  /**
5  *     Este ejemplo ilustra las llamadas de método básicas para sockets
6  *     datagrama sin conexión
7  *     @author M. L. Liu
8  */
9  public class Ejemplo1Emisor {
10
11 //  Una aplicación que manda un mensaje utilizando un socket datagrama
12 //   sin conexión.
13 //   Se esperan tres argumentos de línea de mandato, en orden:
14 // <nombre del dominio o dirección IP del receptor>
15 // <número del puerto del socket del receptor>
16 // <mensaje, una cadena, para mandar>
17
18    public static void main(String[ ] args) {
19       if (args.length != 3)
20          System.out.println
21            ("Este programa requiere 3 argumentos de línea de mandato");
22       else {
23          try {
24            InetAddress maquinaReceptora = InetAddress.getByName(args[0]);
25             int puertoReceptor = Integer.parseInt(args[1]);
26             String mensaje = args[2];
27
28             // instancia un socket datagrama para mandar los datos
29             DatagramSocket miSocket = new DatagramSocket( );
30             byte[ ] almacen = mensaje.getBytes( );
31             DatagramPacket datagrama =
32                new DatagramPacket(almacen, almacen.length,
33                   maquinaReceptora, puertoReceptor);
34             miSocket.send(datagrama);
35             miSocket.close( );
36          } // fin de try
37          catch (Exception ex) {
38             ex.printStackTrace( );
39          } // fin de catch
40       } // fin de else
41    } // fin de main
42 } // fin de class
```

Figura 4.9. *Ejemplo1Receptor.java.*

```
1  import java.net.*;
2  import java.io.*;
3
4  /**
5  *     Este ejemplo ilustra las llamadas de método básicas para sockets
6  *     datagrama sin conexión.
7  *     @author M. L. Liu
8  */
9  public class Ejemplo1Receptor {
10
11 //  Una aplicación  que recibe un mensaje utilizando un socket datagrama
12 //   sin conexión.
13 //   Se espera un argumento de línea de mandato:
14 //   <número de puerto del socket del receptor>
15 //   Nota: se debería especificar el mismo número de puerto
16 //   en los argumentos de línea de mandato del emisor.
17
18    public static void main(String[] args) {
19       if (args.length != 1)
20          System.out.println
21            ("Este programa requiere un argumento de línea de mandato.");
22       else {
23          int puerto = Integer.parseInt(args[0]);
24          final int MAX_LON = 10;
25          // Esta es la longitud máxima asumida en octetos
26          // del datagrama que se va a recibir.
27          try {
28             DatagramSocket miSocket = new DatagramSocket(puerto);
29             // instancia un socket datagrama para recibir los datos
30             byte[ ] almacen = new byte[MAX_LON];
31             DatagramPacket datagrama =
32                new DatagramPacket(almacen, MAX_LON);
33             miSocket.receive(datagrama);
34             String mensaje = new String(almacen);
35             System.out.println(mensaje);
36             miSocket.close( );
37          } // fin de try
38          catch (Exception ex) {
39             ex.printStackTrace( );
40          } // fin de catch
41       } // fin de else
42    } // fin de main
43 } // fin de class
```

Dado que los datos se mandan en paquetes discretos en un modo sin conexión, hay algunas anomalías en el comportamiento de los *sockets* datagrama sin conexión:

- Si se manda un datagrama a un *socket* que el receptor todavía no ha creado, es posible que el datagrama sea desechado. En otras palabras, puede que el meca-

nismo de IPC no salve el datagrama para que se entregue al receptor cuando éste realice finalmente una llamada *receive*. En este caso, se pierden los datos y la llamada *receive* puede resultar bloqueada indefinidamente. Se puede experimentar con este comportamiento arrancando *Ejemplo1Emisor* antes de ejecutar *Ejemplo1Receptor*.

- Si el receptor especifica una zona de almacenamiento para el datagrama (esto es, el vector de octetos asociado al objeto *DatagramPacket*) con un tamaño *n*, un mensaje recibido con un tamaño en octetos mayor que *n* se truncará. Por ejemplo, si *Ejemplo1Emisor* manda un mensaje de 11 octetos, el último octeto en el mensaje (correspondiente al último carácter) no se mostrará en la salida de *Ejemplo1Receptor*, ya que el tamaño de la zona de almacenamiento para el datagrama del receptor es sólo de 10 octetos.

Ejemplo 2. En el ejemplo 1, la comunicación es *simplex*; o sea, unidireccional, desde el emisor al receptor. Es posible hacer la comunicación dúplex o bidireccional. Para hacerlo así, *Ejemplo1Emisor* necesitará enlazar su *socket* a una dirección específica para que *Ejemplo1Receptor* pueda mandar datagramas a esa dirección.

El código de ejemplo en las Figuras 4.10, 4.11 y 4.12 ilustra cómo puede llevarse a cabo la comunicación dúplex. En aras de la modularidad del código, se crea una clase llamada *MiSocketDatagrama* (Figura 4.10) como una subclase de *DatagramSocket*, con dos métodos de instancia para mandar y recibir un mensaje, respectivamente. El programa *Ejemplo2EmisorReceptor* (Figura 4.11) instancia un objeto *MiSocketDatagrama*, a continuación, llama a su método *enviaMensaje*, seguido por una llamada a su método *recibeMensaje*. El programa *Ejemplo2ReceptorEmisor* (Figura 4.12) instancia un objeto *MiSocketDatagrama*, a continuación, llama a su método *recibeMensaje*, seguido por una llamada a su método *enviaMensaje*.

Figura 4.10. *MiSocketDatagrama.java.*

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5  *     Una subclase de DatagramSocket que contiene
 6  *     métodos para mandar y recibir mensajes.
 7  *     @author M. L. Liu
 8  */
 9  public class MiSocketDatagrama extends DatagramSocket {
10    static final int MAX_LON = 10;
11    MiSocketDatagrama(int numPuerto) throws SocketException {
12      super(numPuerto);
13    }
14    public void enviaMensaje(InetAddress maquinaReceptora,
15         int puertoReceptor, String mensaje) throws IOException {
16
17      byte[ ] almacenEnvio = mensaje.getBytes( );
18      DatagramPacket datagrama =
19        new DatagramPacket(almacenEnvio, almacenEnvio.length,
20        maquinaReceptora, puertoReceptor);
```

(continúa)

```
21        this.send(datagrama);
22      } // fin de enviaMensaje
23
24      public String recibeMensaje( )
25           throws IOException {
26        byte[ ] almacenRecepcion = new byte[MAX_LON];
27        DatagramPacket datagrama =
28          new DatagramPacket(almacenRecepcion, MAX_LON);
29        this.receive(datagrama);
30        String mensaje = new String(almacenRecepcion);
31        return mensaje;
32      } // fin de recibeMensaje
33    } // fin de class
```

Figura 4.11. *Ejemplo2EmisorReceptor.java.*

```
1
2   import java.net.*;
3
4   /**
5   *     Este ejemplo ilustra un proceso que envía y después recibe
6   *     utilizando un socket datagrama.
7   *     @author M. L. Liu
8   */
9   public class Ejemplo2EmisorReceptor {
10  //  Una aplicación que manda y que después recibe un mensaje utilizando
11  //  un socket datagrama sin conexión.
12  //  Se esperan cuatro argumentos de línea de mandato, en orden:
13  //  <nombre de dominio o dirección IP del receptor>
14  //  <número de puerto del socket datagrama del receptor>
15  //  <número de puerto del socket datagrama de este proceso>
16  //  <mensaje, una cadena, para mandar>
17
18    public static void main(String[ ] args) {
19      if (args.length != 4)
20        System.out.println
21        ("Este programa requiere 4 argumentos de línea de mandato");
22      else {
23        try {
24          InetAddress maquinaReceptora = InetAddress.getByName(args[0]);
25           int puertoReceptor = Integer.parseInt(args[1]);
26           int miPuerto = Integer.parseInt(args[2]);
27           String mensaje = args[3];
28          MiSocketDatagrama miSocket = new MiSocketDatagrama(miPuerto);
29           // instancia un socket datagrama para enviar
```

(continúa)

```
30            // y recibir datos
31            miSocket.enviaMensaje( maquinaReceptora, puertoReceptor,
              mensaje);
32            // ahora espera recibir un datagrama por el socket
33            System.out.println(miSocket.recibeMensaje());
34            miSocket.close( );
35          } // fin de try
36          catch (Exception ex) {
37            ex.printStackTrace();
38          } // fin de catch
39        } // fin de else
40    } // fin de main
41
42  } // fin de class
```

Figura 4.12. *Ejemplo2ReceptorEmisor.java.*

```
1
2   import java.net.*;
3
4   /**
5   *     Este ejemplo ilustra un proceso que recibe un mensaje y
6   *     después lo envía utilizando un socket datagrama.
7   *     @author M. L. Liu
8   */
9   public class Ejemplo2ReceptorEmisor {
10  // Una aplicación que recibe un mensaje y después lo manda utilizando
11  // un socket datagrama sin conexión.
12  //  Se esperan cuatro argumentos de línea de mandato, en orden:
13  //  <nombre de dominio o dirección IP del receptor>
14  //  <número de puerto del socket datagrama del receptor>
15  //  <número de puerto del socket datagrama de este proceso>
16  //  <mensaje, una cadena, para mandar>
17
18    public static void main(String[ ] args) {
19      if (args.length != 4)
20        System.out.println
21        ("Este programa requiere 4 argumentos de línea de mandato");
22      else {
23        try {
24          InetAddress maquinaReceptora = InetAddress.getByName(args[0]);
25          int puertoReceptor = Integer.parseInt(args[1]);
26          int miPuerto = Integer.parseInt(args[2]);
```

(continúa)

```
27            String mensaje = args[3];
28            // instancia un socket datagrama para enviar
29            // y recibir datos
30           MiSocketDatagrama miSocket = new MiSocketDatagrama(miPuerto);
31            //  Primero espera a recibir un datagrama por el socket
32            System.out.println(miSocket.recibeMensaje());
33            // Ahora envía un mensaje al otro proceso.
34           miSocket.enviaMensaje(maquinaReceptora, puertoReceptor, mensaje);
35            miSocket.close( );
36          } // fin de try
37          catch (Exception ex) {
38            ex.printStackTrace( );
39          }// fin de catch
40        } // fin de else
41      } // fin de main
42
43    } // fin de class
```

Es también posible que múltiples procesos entablen una comunicación sin conexión de esta manera; es decir, se puede añadir un tercer proceso que también tenga un *socket* datagrama, de forma que pueda también mandar y recibir de los otros procesos.

En los ejercicios se tendrá la oportunidad de experimentar con el código de ejemplo para mejorar la comprensión del API de *sockets* datagrama.

El API de *sockets* datagrama orientados a conexión

A continuación se estudiará cómo usar los *sockets* datagramas para comunicaciones orientadas a conexión. Se debería mencionar aquí que es poco común emplear *sockets* datagrama para comunicaciones orientadas a conexión; la conexión proporcionada por esta API es rudimentaria y típicamente insuficiente para las aplicaciones. Los *sockets* en modo *stream*, que se presentarán más tarde en este capítulo, son más típicos y apropiados para la comunicación orientada a conexión.

La Tabla 4.3 describe dos métodos de la clase *DatagramSocket* que permiten crear y terminar una conexión. Para realizar una conexión con un *socket*, se especifica la dirección de un *socket* remoto. Una vez hecha tal conexión, el *socket* se utiliza para intercambiar paquetes de datagrama con el *socket* remoto. En una operación *send*, si la dirección del datagrama no coincide con la dirección del *socket* en el otro extremo, se activará *IllegalArgumentException*. Si se mandan los datos al *socket* desde una fuente que no corresponde con el *socket* remoto conectado, los datos se ignorarán. Así, una vez que se asocia una conexión a un *socket* datagrama, ese *socket* no estará disponible para comunicarse con otro *socket* hasta que la conexión se termine. Nótese que la conexión es unilateral; esto es, sólo se impone en un extremo. El *socket* en el otro lado está disponible para mandar y recibir datos a otros *sockets*, a menos que se realice una conexión con este *socket*.

Tabla 4.3. Llamadas de método para un *socket* datagrama orientado a conexión.

Método/Constructor	Descripción
void connect(InetAddress direccion, int puerto)	Crea una conexión lógica entre este *socket* y un *socket* en la dirección y puerto remotos.
void disconnect( )	Termina la conexión actual, si existe, de este *socket*.

Ejemplo 3. El código de *Ejemplo3*, mostrado en las Figuras 4.13 y 4.14, ilustra la sintaxis de uso de los *sockets* datagrama orientados a conexión. En la Figura 4.13, *Ejemplo3Emisor.java*, se crea una conexión entre el *socket* datagrama del proceso emisor y el del proceso receptor. Nótese que la conexión se hace en ambos lados. Una vez que se establece mutuamente una conexión, cada proceso está obligado a utilizar su *socket* para la IPC con otro proceso. (Sin embargo, esto no prohíbe a cada proceso crear otra conexión utilizando otro *socket*). En el ejemplo, el emisor manda sucesivamente por la conexión 10 copias del mismo mensaje. En el proceso receptor, se visualiza inmediatamente cada uno de los diez mensajes recibidos. El proceso receptor después manda un único mensaje de vuelta al proceso emisor para ilustrar que la conexión permite una comunicación bidireccional.

Figura 4.13. *Ejemplo3Emisor.java.*

```
 1  import java.net.*;
 2
 3
 4  /**
 5  *     Este ejemplo ilustra la sintaxis básica de los sockets datagrama
 6  *     orientados a conexión.
 7  *     @author M. L. Liu
 8  */
 9  public class Ejemplo3Emisor {
10
11  //  Una aplicación que utiliza un socket datagrama orientado a
12  //  conexión para mandar múltiples mensajes, después recibe uno.
13  //  Se esperan cuatro argumentos de línea de mandato, en orden:
14  //  <nombre de dominio o dirección IP del receptor>
15  //  <número de puerto del socket datagrama del otro proceso>
16  //  <número de puerto del socket datagrama de este proceso>
17  //  <mensaje, una cadena, para mandar>
18
19    public static void main(String[] args) {
20      if (args.length != 4)
21        System.out.println
22        ("Este programa requiere 4 argumentos de línea de mandato");
23      else {
24        try {
25          InetAddress maquinaReceptora = InetAddress.getByName(args[0]);
26          int puertoReceptor = Integer.parseInt(args[1]);
27          int miPuerto = Integer.parseInt(args[2]);
```

(continúa)

```
27            int miPuerto = Integer.parseInt(args[2]);
28            String mensaje = args[3];
29            // instancia una socket datagrama para la conexión
30           MiSocketDatagrama miSocket = new MiSocketDatagrama(miPuerto);
31            // hace la conexión
32            miSocket.connect(maquinaReceptora, puertoReceptor);
33            for (int i=0; i<10; i++)
34              miSocket.enviaMensaje(maquinaReceptora, puertoReceptor,
                    mensaje);
35            // ahora recibe un mensaje desde el otro extremo
36            System.out.println(miSocket.recibeMensaje( ));
37            // termina la conexión, después cierra el socket
38            miSocket.disconnect( );
39            miSocket.close( );
40          } // fin de try
41          catch (Exception ex) {
42            ex.printStackTrace( );
43          } // fin de catch
44        } // fin de else
45    } // fin de main
46  } // fin de class
```

Figura 4.14. *Ejemplo3Receptor.java.*

```
1   import java.net.*;
2
3
4   /**
5   *     Este ejemplo ilustra la sintaxis básica de los sockets datagrama
6   *     orientados a conexión.
7   *     @author M. L. Liu
8   */
9   public class Ejemplo3Receptor {
10
11  //  Una aplicación que utiliza un socket datagrama orientado a
12  //  conexión para recibir múltiples mensajes, después envía uno.
13  //  Se esperan cuatro argumentos de línea de mandato, en orden:
14  //  <nombre de dominio o dirección IP del emisor>
15  //  <número de puerto del socket datagrama del emisor>
16  //  <número de puerto del socket datagrama de este proceso>
17  //  <mensaje, una cadena, para mandar>
18
19    public static void main(String[ ] args) {
20      if (args.length != 4)
21        System.out.println
22        ("Este programa requiere 4 argumentos de línea de mandato");
```

(continúa)

```
23        else {
24          try {
25            InetAddress maquinaEmisora = InetAddress.getByName(args[0]);
26            int puertoEmisor = Integer.parseInt(args[1]);
27            int miPuerto = Integer.parseInt(args[2]);
28            String mensaje = args[3];
29            // instancia un socket datagrama para recibir los datos
30            MiSocketDatagrama miSocket = new
                MiSocketDatagrama(miPuerto);
31            // hace una conexión con el socket del emisor
32            miSocket.connect(maquinaEmisora, puertoEmisor);
33            for (int i=0; i<10; i++)
34              System.out.println(miSocket.recibeMensaje( ));
35            // ahora manda un mensaje al otro extremo
36             miSocket.enviaMensaje( maquinaEmisora, puertoEmisor,
                  mensaje);
37            miSocket.close( );
38          } // fin de try
39          catch (Exception ex) {
40            ex.printStackTrace( );
41          } // fin de catch
42        } // fin de else
43      } // fin de main
44    } // fin de class
```

Esto concluye la introducción a los *sockets* datagrama. Se tendrá ocasión de volver a estudiarlos en capítulos posteriores, pero en este punto se centrará la atención en otro modelo de API de *sockets*: El API de *sockets* en modo *stream*.

4.4. EL API DE *SOCKETS* EN MODO *STREAM*

Mientras que el API de *sockets* datagrama permite el intercambio de unidades **discretas** de datos (es decir, datagramas), el API de *sockets* en modo *stream* proporciona un modelo de transferencia de datos basado en la **E/S en modo *stream*** de los sistemas operativos Unix. Por definición, un *socket* en modo *stream* proporciona sólo comunicaciones orientadas a conexión.

En la entrada-salida en modo *stream*, los datos se transfieren utilizando el concepto de un flujo de datos continuo que fluye desde una fuente a un destino (también llamado sumidero). Los datos se insertan, o escriben, dentro de un flujo por un proceso que controla la fuente y los datos se extraen, o leen, del flujo por un proceso asociado al destino. Las Figuras 4.15 y 4.16 ilustran el concepto de un flujo de datos. Observe la naturaleza continua de un flujo que permite que los datos se inserten o extraigan del mismo a diferentes velocidades.

El API de *sockets* en modo *stream* (Figura 4.17) es una extensión del modelo de E/S en modo *stream*. Usando el API, cada uno de los dos procesos crea individualmente un *socket* en modo *stream*. A continuación, se forma una conexión entre los *sockets*. Los datos se escriben, como un flujo de caracteres, dentro del *socket* del emisor y, a continuación, el receptor puede leerlos a través de su *socket*. Esto es similar al API de *sockets*

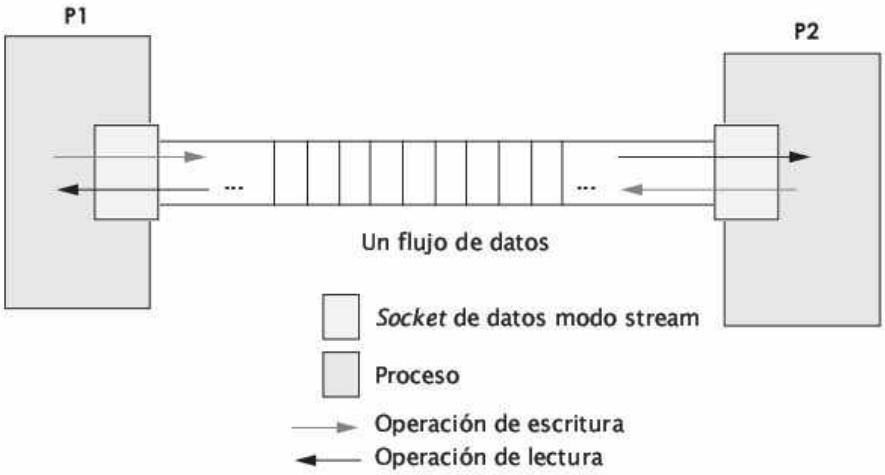


Figura 4.15. Uso de un *socket* en modo *stream* para transferencia de datos.

Las unidades de datos escritas y leídas no necesitan coincidir. Por ejemplo, 100 bytes de datos escritos utilizando una operación pueden leerse usando una operación de lectura de 20 bytes, seguida por otra operación *read* de 80 bytes.

Figura 4.16. E/S en modo *stream*.

Un servidor utiliza dos *sockets*: uno para aceptar las conexiones, otro para *send/receive*.

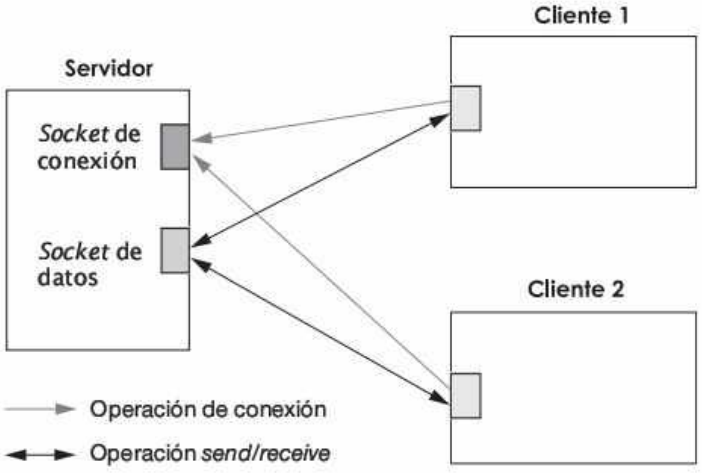


Figura 4.17. El API de *socket* en modo *stream*.

datagrama orientados a conexión que ya se ha estudiado anteriormente, excepto por la diferencia en la naturaleza discreta de los datos transportados en *sockets* datagrama.

En Java, las clases *ServerSocket* y *Socket* proporcionan el API de *sockets* en modo *stream*. El término *Server* proviene del paradigma cliente-servidor, para el que se diseñó el API (recuérdese que el paradigma cliente-servidor se presentó en el Capítulo 3; y se estudiará en detalle en el Capítulo 5). La sintaxis del API está estrechamente vinculada con el paradigma cliente-servidor. Por ahora, se estudiará el API de forma independiente al paradigma.

Hay dos tipos de *sockets* en el API en modo *stream*:

- La clase *ServerSocket* proporciona el primer tipo de *socket* y sirve para aceptar conexiones. Por claridad, en este capítulo se hará referencia a ellos como ***sockets* de conexión**.
- La clase *Socket* proporciona el otro tipo de *socket* que permite intercambiar datos. Por claridad, se hará referencia a ellos como ***sockets* de datos**.

Utilizando esta API, un proceso conocido como el **servidor** establece un *socket* de conexión y después se queda a la espera de las peticiones de conexión de otros procesos. Las peticiones de conexión se aceptan de una en una. A través del *socket* de datos, el proceso servidor puede leer y/o escribir del flujo de datos. Cuando se termina la sesión de comunicación entre los dos procesos, se cierra el *socket* de datos, y el servidor está preparado para aceptar la próxima petición de conexión a través del *socket* de conexión.

A un proceso que desea comunicarse con el servidor se le conoce como un **cliente**. Un cliente crea un *socket*; a continuación, a través del *socket* de conexión del servidor, solicita una conexión al servidor. Una vez que se acepta la petición, el *socket* del cliente se conecta al *socket* de datos del servidor de manera que el cliente pueda pasar a leer y/o escribir del flujo de datos. Cuando se completa la sesión de comunicación entre los dos procesos, se cierran los *sockets* de datos.

Operaciones y sincronización de eventos

Hay dos clases principales en el API de *sockets* en modo *stream*: la clase *ServerSocket* y la clase *Socket*. La clase *ServerSocket* permite el establecimiento de conexiones, mientras que la clase *Socket* sirve para la transferencia de datos. En las Tablas 4.4 y 4.5 se listan los métodos principales y los constructores de las dos clases, respectivamente.

Tabla 4.4. Métodos principales y constructores de la clase *ServerSocket* (*socket* de conexión).

Método/Constructor	Descripción
ServerSocket (int puerto)	Crea un *socket* de servidor en un puerto especificado.
Socket accept() throws IOException	Espera que se solicite una conexión a este *socket* y la acepta. El método bloquea hasta que se haga una conexión.
void close() throws IOException	Cierra este *socket*.
void setSoTimeout(int plazo) throws SocketException	Fija un plazo máximo de tiempo de espera (en milisegundos), de manera que una llamada *accept()* sobre este *socket* bloquee durante sólo esta cantidad de tiempo. Si el plazo expira, se activa una *java.io.interruptedIOException*.

Tabla 4.5. Métodos principales y constructores de la clase *Socket* (*socket* de datos).

Método/Constructor	Descripción
Socket(InetAddress dirección, int puerto)	Crea un *socket stream* y lo conecta al número de puerto y la dirección IP especificados.
void close() throws IOException	Cierra este *socket*.
InputStream getInputStream() throws IOException	Devuelve un flujo de entrada para que se puedan leer los datos de este *socket*.
OutputStream getOutputStream() throws IOException	Devuelve un flujo de salida para que se puedan escribir los datos en este *socket*.
Void setSoTimeout(int plazo) throws SocketException	Fija un periodo máximo de bloqueo de manera que una llamada *read()* en el *InputStream* asociado con este *socket* bloquee sólo durante esta cantidad de tiempo. Si el plazo de tiempo expira, se activa una *java.io.InterruptedIOException()*.

Con respecto a la sincronización de eventos, las siguientes operaciones son bloqueantes:

- *Accept* (aceptación de una conexión). Si no hay ninguna petición esperando, el proceso servidor se suspenderá hasta que llegue una petición de conexión.
- La lectura de un flujo de entrada asociado a un *socket* de datos. Si la cantidad de datos pedida no está actualmente presente en el flujo de datos, el proceso que solicita la lectura se bloqueará hasta que se haya escrito una cantidad de datos suficiente en el flujo de datos.

Nótese que no se proporcionan métodos *read* y *write* específicos, puesto que se deben utilizar los métodos asociados con las clases *InputStream* y *OutputStream* para realizar estas operaciones, como se verá en breve.

La Figura 4.18 ilustra los flujos de ejecución de un programa que espera una petición de conexión y de otro que solicita la conexión.

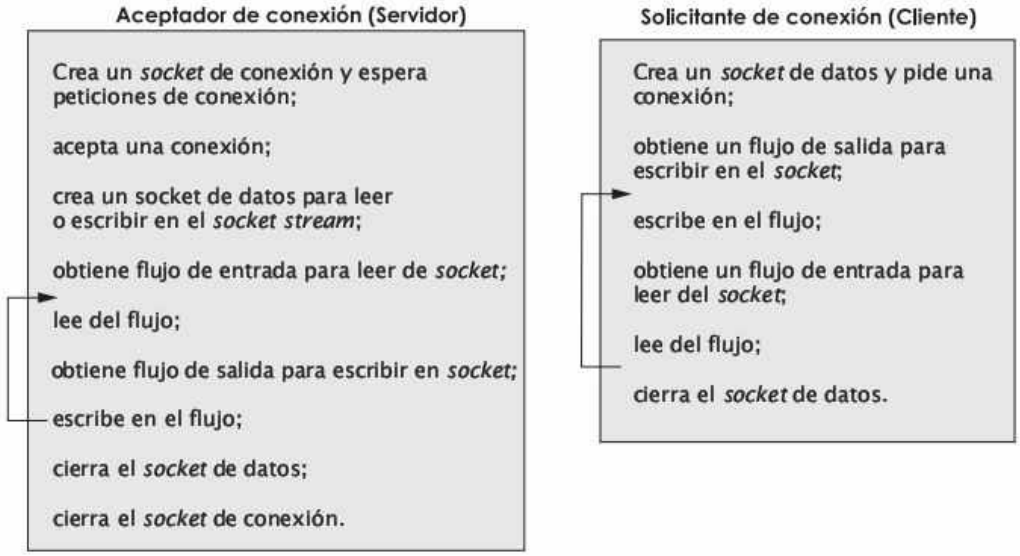


Figura 4.18. Flujos de ejecución de un programa que espera una petición de conexión y de otro que la solicita.

Ejemplo 4. Las Figuras 4.19 y 4.20 ilustran la sintaxis básica para los *sockets* en modo *stream*. *Ejemplo4AceptadorConexion*, como su nombre implica, acepta conexiones estableciendo un objeto *ServerSocket* en un puerto especificado (por ejemplo, 12345). *Ejemplo4SolicitanteConexion* crea un objeto *Socket*, especificando como argumentos el nombre de la máquina y el número de puerto (en este caso, 12345) del *aceptador*. Una vez que el *aceptador* ha aceptado la conexión, escribe un mensaje en el flujo de datos del *socket*. En el *solicitante*, el mensaje se lee del flujo de datos y se visualiza.

Figura 4.19. *Ejemplo4AceptadorConexion.java*.

```
1  import java.net.*;
2  import java.io.*;
3
4  /**
5  *     Este ejemplo ilustra la sintaxis básica del socket
6  *     en modo stream.
7  *     @author M. L. Liu
8  */
9  public class Ejemplo4AceptadorConexion {
10
11 //  Una aplicación que acepta una conexión y recibe un mensaje
       utilizando un socket en modo stream.
12 //  Se esperan dos argumentos de línea de mandato, en orden:
13 //  <número de puerto del socket de servidor utilizado en este proceso>
14 //  <mensaje, una cadena, para mandar>
15
16    public static void main(String[] args) {
17       if (args.length != 2)
18          System.out.println
19          ("Este programa requiere dos argumentos de línea de mandato");
20       else {
21          try {
22             int numPuerto = Integer.parseInt(args[0]);
23             String mensaje = args[1];
24             // instancia un socket para aceptar la conexión
25             ServerSocket socketConexion = new ServerSocket(numPuerto);
26 /**/        System.out.println("preparado para aceptar una conexión");
27             // espera una petición de conexión, instante en el cual
28             // se crea un socket de datos
29             Socket socketDatos = socketConexion.accept();
30 /**/        System.out.println("conexión aceptada");
31             // obtiene un flujo de salida para escribir en el socket
                  de datos
32             OutputStream flujoSalida = socketDatos.getOutputStream();
33             // crea un objeto PrintWriter para la salida en modo carácter
34             PrintWriter salidaSocket =
35                new PrintWriter(new OutputStreamWriter(flujoSalida));
36             // escribe un mensaje en el flujo de datos
37             salidaSocket.println(mensaje);
```

(continúa)

```
38            // La subsiguiente llamada al método flush es necesaria
39            // para que los datos se escriban en el flujo de datos
40            // del socket antes de que se cierre el socket.
41            salidaSocket.flush();
42  /**/      System.out.println("mensaje enviado");
43            socketDatos.close( );
44  /**/      System.out.println("socket de datos cerrado");
45            socketConexion.close( );
46  /**/      System.out.println("socket de conexión cerrado");
47          } // end try
48          catch (Exception ex) {
49              ex.printStackTrace( );
50          } // fin de catch
51        } // fin de else
52    } // fin de main
53  } // fin de class
```

Figura 4.20. *Ejemplo4SolicitanteConexion.java.*

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5  *     Este ejemplo ilustra la sintaxis básica del socket
 6  *     en modo stream.
 7  *     @author M. L. Liu
 8  */
 9  public class Ejemplo4SolicitanteConexion {
10
11  //  Una aplicación que solicita una conexión y manda un mensaje
          utilizando un socket en modo stream
12  //  Se esperan dos argumentos de línea de mandato, en orden:
13  //  <nombre de la máquina del aceptador de la conexión>
14  //  <número de puerto del aceptador de la conexión>
15    public static void main(String[] args) {
16      if (args.length != 2)
17        System.out.println
18        ("Este programa requiere dos argumentos de línea de mandato");
19      else {
20        try {
21          InetAddress maquinaAceptadora = InetAddress.getByName(args[0]);
22          int puertoAceptador = Integer.parseInt(args[1]);
23          // instancia un socket de datos
24          Socket miSocket = new Socket(maquinaAceptadora,
              puertoAceptador);
25  /**/    System.out.println("Solicitud de conexión concedida");
```

(continúa)

```
26            // obtiene un flujo de entrada para leer del socket de datos
27             InputStream flujoEntrada = miSocket.getInputStream();
28             // crea un objeto BufferedReader para la entrada en modo
                  carácter
29             BufferedReader socketInput =
30               new BufferedReader(new InputStreamReader(flujoEntrada));
31  /**/       System.out.println("esperando leer");
32             // lee una línea del flujo de datos
33             String mensaje = socketInput.readLine( );
34  /**/       System.out.println("Mensaje recibido:");
35             System.out.println("\t" + mensaje);
36             miSocket.close( );
37  /**/       System.out.println("socket de datos cerrado");
38           } // fin de try
39           catch (Exception ex) {
40             ex.printStackTrace( );
41           } // fin de catch
42         } // fin de else
43       } // fin de main
44   } // fin de class
```

Hay varios puntos reseñables en este ejemplo:

1. Debido a que se está tratando con un flujo de datos, se puede usar la clase *PrintWriter* de Java (línea 15 de la Figura 4.19) para escribir en un *socket* y *BufferedReader* (línea 29 de la Figura 4.20) para leer de un flujo. Los métodos usados con estas clases son los mismos que para escribir una línea de texto en la pantalla o leer una línea de texto del teclado.
2. Aunque el ejemplo muestra como emisor de datos al *Aceptador* y como receptor al *Solicitante*, los papeles se pueden intercambiar fácilmente. En ese caso, el *Solicitante* usará *getOutputStream* para escribir en el *socket*, mientras que el *Aceptador* utilizará *getInputStream* para leer del *socket*.
3. De hecho, cada proceso puede leer y escribir del flujo invocando *getInputStream* o *getOutputStream*, como se ilustra en el Ejemplo 5, que se verá más adelante en este capítulo.
4. Aunque el ejemplo lee y escribe una línea cada vez (utilizando los métodos *readLine()* y *println()*, respectivamente), es también posible leer y escribir parte de una línea en su lugar (usando *read()* y *print()* respectivamente). Sin embargo, para protocolos basados en texto donde los mensajes se intercambian como texto, lo habitual es leer y escribir una línea cada vez.
5. Cuando se utiliza *PrintWriter* para escribir en un *socket stream*, es necesario utilizar una llamada a *flush()* para «limpiar el flujo», de manera que se garantice que todos los datos se escriben desde la zona de almacenamiento de datos al flujo tan pronto como sea posible, antes de que el *socket* sea súbitamente cerrado (*véase* la línea 41 en la Figura 4.19).

La Figura 4.21 muestra el diagrama de eventos correspondiente a la ejecución de los programas del Ejemplo 4.

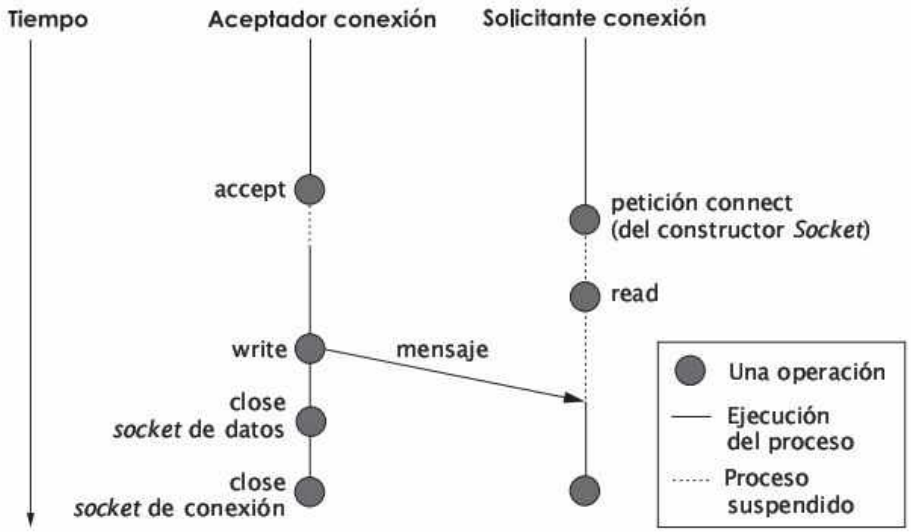


Figura 4.21. Diagrama de eventos de *Ejemplo4*.

El proceso *AceptadorConexion* comienza su ejecución en primer lugar. El proceso se suspende cuando se llama al método bloqueante *accept*, después se reanuda cuando recibe la petición de conexión del *Solicitante*. Una vez reanudada la ejecución, el *Aceptador* escribe un mensaje en el *socket* antes de cerrar tanto el *socket* de datos como el de conexión.

La ejecución del *SolicitanteConexion* se realiza de la siguiente forma: se instancia un objeto *Socket* y se hace una petición *connect* implícita al *Aceptador*. Aunque la petición *connect* no es bloqueante, el intercambio de datos a través de la conexión no puede llevarse a cabo hasta que el proceso en el otro extremo acepta (*accept*) la conexión. Una vez que se acepta la conexión, el proceso invoca una operación *read* para leer un mensaje del *socket*. Dado que la operación *read* es bloqueante, el proceso se suspende de nuevo hasta que se reciben los datos del mensaje, después de lo cual el proceso cierra (*close*) el *socket* y procesa los datos.

Se han insertado en los programas mensajes de diagnóstico (marcados con /**/), de manera que se pueda observar el progreso de la ejecución de los dos programas cuando se están ejecutando.

Para permitir la separación de la lógica de aplicación y la lógica de servicio en los programas, se emplea una subclase que esconde los detalles de los *sockets* de datos. La Figura 4.22 muestra el listado de código de la clase *MiSocketStream*, que proporciona métodos para leer y escribir de un *socket* de datos.

Figura 4.22. *MiSocketStream.java*, una subclase derivada de la clase *Socket* de Java.

```
1  import java.net.*;
2  import java.io.*;
3
4  /**
5  *     Una clase de envoltura de Socket que contiene
6  *     métodos para mandar y recibir mensajes.
```

(continúa)

```
 7  *     @author M. L. Liu
 8  */
 9  public class Misocketstream extends Socket {
10    private Socket socket;
11    private BufferedReader entrada;
12    private PrintWriter salida;
13
14    Misocketstream(String maquinaAceptadora,
15      int puertoAceptador ) throws SocketException,
16        IOException{
17      socket = new Socket(maquinaAceptadora, puertoAceptador );
18      establecerFlujos( );
19    }
20
21    Misocketstream(Socket socket) throws IOException {
22      this.socket = socket;
23      establecerFlujos( );
24    }
25
26    private void establecerFlujos( ) throws IOException{
27      // obtiene un flujo de salida para leer del socket de datos
28      InputStream flujoEntrada = socket.getInputStream();
29      entrada =
30        new BufferedReader(new InputStreamReader(flujoEntrada));
31      OutputStream flujoSalida = socket.getOutputStream();
32      // crea un objeto PrintWriter para salida en modo carácter
33      salida =
34        new PrintWriter(new OutputStreamWriter(flujoSalida));
35    }
36
37    public void enviaMensaje(String mensaje)
38      throws IOException {
39      salida.println(mensaje);
40      // La subsiguiente llamada al método flush es necesaria para que
41      // los datos se escriban en el flujo de datos del socket
42      // antes de que se cierre el socket.
43      salida.flush();
44    } // fin de enviaMensaje
45
46    public String recibeMensaje( )
47      throws IOException {
48      // lee una línea del flujo de datos
49      String mensaje = entrada.readLine( );
50      return mensaje;
51    } // fin de recibeMensaje
52
53    public void close( )
54      throws IOException {
55      socket.close( );
56    }
57  } //fin de class
```

Ejemplo 5. Las Figuras 4.23 y 4.24 son revisiones de los ficheros de código fuente presentados en las Figuras 4.19 (*AceptadorConexion*) y 4.20 (*SolicitanteConexion*), respectivamente, modificados para utilizar la clase ***MiSocketStream*** en vez de la clase ***Socket*** de Java.

Figura 4.23. *Ejemplo5AceptadorConexion.java.*

```
1   import java.net.*;
2   import java.io.*;
3
4   /**
5   *     Este ejemplo ilustra la sintaxis básica del socket
6   *     en modo stream.
7   *     @author M. L. Liu
8   */
9   public class Ejemplo5AceptadorConexion {
10
11  // Una aplicación que recibe un mensaje usando un socket en modo
        stream.
12  //   Se esperan dos argumentos de línea de mandato, en orden:
13  //   <número de puerto del socket de servidor utilizado en este proceso>
14  //   <mensaje, una cadena, para mandar>
15
16    public static void main(String[] args) {
17      if (args.length != 2)
18        System.out.println
19        ("Este programa requiere dos argumentos de línea de mandato");
20      else {
21        try {
22          int numPuerto = Integer.parseInt(args[0]);
23          String mensaje = args[1];
24          // instancia un socket para aceptar la conexión
25          ServerSocket socketConexion = new ServerSocket(numPuerto);
26  /**/    System.out.println("preparado para aceptar una conexión");
27          // espera una petición de conexión, instante en el cual
28          // se crea un socket de datos
29          Misocketstream socketDatos =
30            new MiSocketStream(socketConexion.accept());
31  /**/    System.out.println("conexión aceptada");
32          socketDatos.enviaMensaje(mensaje);
33
34  /**/    System.out.println("mensaje enviado");
35          socketDatos.close( );
36  /**/    System.out.println("socket de datos cerrado");
37          socketConexion.close( );
38  /**/    System.out.println("socket de conexión cerrado");
39        } // fin de try
40        catch (Exception ex) {
41          ex.printStackTrace( );
42        } // fin de catch
```

(continúa)

```
43        } // fin de else
44      } // fin de main
45  } // fin de class
```

Figura 4.24. *Ejemplo5SolicitanteConexion.java*.

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5  *     Este ejemplo ilustra la sintaxis básica del socket
 6  *     en modo stream.
 7  *     @author M. L. Liu
 8  */
 9  public class Ejemplo5SolicitanteConexion {
10
11  //  Una aplicación que manda un mensaje usando un socket en modo
        stream.
12  //  Se esperan dos argumentos de línea de mandato, en orden:
13  //
14  //  <nombre de la máquina del aceptador de la conexión>
15  //  <número de puerto del aceptador de la conexión>
16
17    public static void main(String[] args) {
18      if (args.length != 2)
19        System.out.println
20        ("Este programa requiere dos argumentos de línea de mandato");
21      else {
22        try {
23          String maquinaAceptadora = args[0];
24          int puertoAceptador = Integer.parseInt(args[1]);
25          // instancia un socket de datos
26          MiSocketStream miSocket =
27            new MiSocketStream(maquinaAceptadora, puertoAceptador);
28  /**/     System.out.println("Solicitud de conexión concedida");
29          String mensaje = miSocket.recibeMensaje( );
30  /**/     System.out.println("Mensaje recibido:");
31          System.out.println("\t" + mensaje);
32          miSocket.close( );
33  /**/     System.out.println("socket de datos cerrado");
34        } // fin de try
35        catch (Exception ex) {
36          ex.printStackTrace( );
37        }
38      } // fin de else
39    } // fin de main
40  } // fin de class
```

Utilizando la subclase *MiSocketStream*, es mucho más cómodo realizar entrada y salida en el *socket*, como se pedirá realizar en uno de los ejercicios al final de este capítulo.

Con esto termina la introducción al API de *sockets* en modo *stream*. En el próximo capítulo se volverá a estudiar este mecanismo de comunicación.

4.5. *SOCKETS* CON OPERACIONES DE E/S NO BLOQUEANTES

Como se mencionó anteriormente, las interfaces de programación introducidas en este capítulo son las básicas, que proporcionan operaciones *send* (no-bloqueantes) asíncronas y operaciones *receive* (bloqueantes) síncronas. Utilizando estas interfaces, un proceso que lee de un *socket* es susceptible de bloquearse. Para maximizar la concurrencia, se pueden utilizar hilos (*threads*), de manera que un hilo de espera realiza una operación de lectura bloqueante, mientras que otro hilo permanece activo para procesar otras tareas. Sin embargo, en algunas aplicaciones que requieren usar un elevado número de hilos, la sobrecarga incurrida puede ser perjudicial para el rendimiento o, peor todavía, para la viabilidad de la aplicación. Como una alternativa, hay interfaces de programación de *sockets* que proporcionan operaciones de E/S no bloqueantes. Utilizando una API de este tipo, ni *send* ni *receive* resultarán bloqueantes y, como se explicó en el Capítulo 2, será necesario que el proceso receptor utilice un manejador de eventos para que se le notifique de la llegada de los datos. Los *sockets* asíncronos están disponibles en Winsock y, a partir de la versión 1.4, Java también proporciona un nuevo paquete de E/S, **java.nio (NIO)**, que ofrece *sockets* con operaciones de E/S no bloqueantes. La sintaxis del nuevo paquete es considerablemente más compleja que la del API básica. Se recomienda a los lectores interesados que examinen [java.sun.com, 6].

4.6. EL API DE *SOCKETS* SEGUROS

Aunque los detalles quedan fuera del ámbito de este libro, el lector debería conocer la existencia de las interfaces de programación de **sockets seguros**, que son interfaces de *sockets* mejoradas con medidas de seguridad de datos.

Un analizador de protocolos es una herramienta que permite capturar y analizar los paquetes de datos para resolver problemas en la red.

Utilizando las interfaces de programación de *sockets* convencionales, los datos se transmiten como flujos de bits sobre los enlaces de la red. Estos flujos de bits, si se interceptan por medio de herramientas tales como analizadores de protocolos de red, pueden ser descodificados por alguien que tenga conocimientos de la representación de los datos intercambiados. Por ello, el riesgo de utilizar *sockets* para transmitir datos sensibles, como información de crédito y datos de autenticación. Para tratar el problema, se han introducido protocolos que protegen los datos transmitidos usando *sockets*. En los siguientes párrafos se describirán algunos de los protocolos más conocidos.

El nivel de *sockets* seguros

El **nivel de *sockets* seguros** (**SSL**, *Secure Sockets Layer*) [developer.netscape.com, 2] fue un protocolo desarrollado por Netscape Communications Corporation para transmitir documentos privados sobre Internet. (Esta descripción de SSL se basa en la de-

Las extensiones de *sockets* seguros de Java

Dr. Dobb's Journal, *febrero de 2001*
Autenticación y cifrado de conexiones.
Por Kirby W. Angell.
Reimpreso con el permiso del *Dr. Dobb's Journal*.

Uno se sienta delante de su computador, maravillándose de su aplicación Java distribuida. El código crea objetos *Socket* y *ServerSocket* como loco, mandando datos a través de Internet. Da gusto verlo, hasta que uno se da cuenta de que cualquiera puede interceptar los datos que se están leyendo, suplantar una de sus aplicaciones e inundar su sistema con datos falsos.

Tan pronto como se empieza a investigar sobre la autenticación y cifrado de las conexiones entre aplicaciones, uno se da cuenta de que ha entrado en una área compleja. Cuando uno trata con el cifrado, se tiene que preocupar por muchas cosas y no sólo de qué algoritmo se pretende utilizar. Los ataques al sistema pueden involucrar al algoritmo, al protocolo, a las contraseñas y a otros factores que uno ni siquiera podría considerar.

Afortunadamente, la mayoría de los detalles liosos de la autenticación y el cifrado del tráfico entre dos aplicaciones basadas en *sockets* se ha resuelto en la especificación del nivel de *sockets* seguros (SSL, *Secure Sockets Layer*). Sun Microsystems tiene una implementación de SSL en su paquete de extensión de *sockets* seguros de Java (JSSE, *Java Secure Sockets Extension*; http://java.sun.com/security/). JSSE y el entorno en tiempo de ejecución de Java (JRE, *Java Run-Time Environment*) proporcionan la mayoría de las herramientas necesarias para implementar SSL dentro de una aplicación Java en el caso de que se trate de un cliente comunicándose con servidores HTTPS. Dado que la documentación y las herramientas JSSE están principalmente orientadas hacia este fin, cuesta algo más de trabajo averiguar cómo utilizar el conjunto de herramientas dentro de una aplicación donde se necesitan crear tanto el lado del cliente de la conexión como el lado del servidor.

finición proporcionada por http://webopedia.internet.com). Una API de SSL tiene métodos o funciones similares al API de *sockets*, excepto en que los datos son **cifrados** antes de que se transmitan sobre una conexión SSL. Los navegadores modernos dan soporte a SSL. Cuando se ejecutan con el protocolo SSL seleccionado, estos navegadores transmitirán los datos cifrados utilizando el API de *sockets* SSL. Muchos sitios web también utilizan el protocolo para obtener información confidencial del usuario, tal como números de tarjeta de crédito. Por convención, un URL de una página web que requiere una conexión SSL comienza con **https:** en vez de **http:**.

La extensión de *sockets* seguros de Java

La extensión de *sockets* seguros de Java (JSSE, *Java Secure Socket Extension*) es un conjunto de paquetes de Java que posibilita las comunicaciones seguras en Internet. Implementa una versión de los protocolos SSL y TLS (*Transport Layer Security*) [ietf.org, 5] e incluye herramientas para el cifrado de datos, autenticación del servidor, integridad de mensajes y autenticación de cliente opcional. Utilizando JSSE [java.sun.com, 3; Angell, 4], los desarrolladores pueden proporcionar el tráfico de datos seguro entre dos procesos.

El API de JSSE se caracteriza por tener una sintaxis similar al API de *sockets* orientados a conexión presentada en este capítulo.

RESUMEN

En este capítulo, se introduce la interfaz básica de programación de aplicaciones de *sockets* para la comunicación entre procesos. El API de *sockets* está ampliamente disponible como una herramienta de programación para IPC en un nivel relativamente bajo de abstracción.

Utilizando las interfaces de *sockets* de Java, en el capítulo se han presentado dos tipos de *sockets*:

- Los *sockets* datagrama, que utilizan el protocolo de datagrama de usuario (UDP, *User Datagram Protocol*) en el nivel de transporte para mandar y recibir paquetes de datos discretos conocidos como datagramas.
- El *socket* en modo *stream*, que utiliza el protocolo de nivel de transporte (TCP, *Transmission Control Protocol*) en el nivel de transporte para mandar y recibir datos utilizando un flujo de datos.

Los aspectos fundamentales del API de *sockets* datagrama de Java son los siguientes:

- Permite tanto una comunicación sin conexión como una comunicación orientada a conexión.
- Cada proceso debe crear un objeto *DatagramSocket*.
- Cada datagrama se encapsula en un objeto *DatagramPacket*.
- En comunicación sin conexión, se puede utilizar un *socket* datagrama para mandar o recibir de cualquier otro *socket* datagrama; en comunicación orientada a conexión, un *socket* datagrama sólo puede usarse para mandar o recibir del *socket* datagrama asociado al otro extremo de la conexión.
- Los datos de un datagrama se sitúan en un vector de octetos; si un receptor proporciona un vector de octetos de insuficiente longitud, los datos recibidos se truncan.
- La operación *receive* es bloqueante; la operación *send* es no bloqueante.

Los aspectos fundamentales del API de *sockets* en modo *stream* son los siguientes:

- Soporta sólo una comunicación orientada a conexión.
- Un proceso juega un papel de aceptador de conexión y crea un *socket* de conexión utilizando la clave *ServerSocket*. A continuación, acepta las peticiones de conexión de otros procesos.
- Un proceso (un solicitante de conexión) crea un *socket* de datos utilizando la clase *Socket* y se realiza implícitamente una petición de conexión al aceptador de conexión.
- Cuando se concede una petición de conexión, el aceptador de conexión crea un *socket* de datos, de la clase *Socket*, para mandar y recibir datos del solicitante de conexión. El solicitante de conexión puede también mandar y recibir datos del aceptador de conexión utilizando su *socket* de datos.
- Las operaciones *receive* (leer) y *accept* (aceptar conexión) son bloqueantes; la operación *send* (escribir) es no bloqueante.
- La lectura y la escritura de datos en el *socket* de datos *stream* están desacopladas: pueden realizarse usando diferentes unidades de datos.

Hay interfaces de programación de *sockets* que proporcionan operaciones de E/S no bloqueantes, incluyendo Winsock y el NIOS de Java. La sintaxis de estas interfaces

es más compleja y requiere usar un manejador de eventos para gestionar las operaciones *bloqueantes*.

Los datos transmitidos por la red utilizando *sockets* son susceptibles a riesgos de seguridad. Para los datos sensibles, se recomienda emplear los *sockets* seguros. Entre las interfaces de *sockets* seguros disponibles se encuentran el nivel de *sockets* seguros (SSL, *Secure Sockets Layer*) y la extensión de *sockets* seguros de Java (JSSE, *Java Secure Sockets Extension*). Las interfaces de *sockets* seguros tienen métodos que son similares a las interfaces de *sockets* orientados a conexión.

EJERCICIOS

1. Usando sus propias palabras, escriba algunas frases para explicar cada uno de los siguiente términos:
   a. API (interfaz de programador de aplicaciones).
   b. El API de *sockets*.
   c. Winsock.
   d. La comunicación orientada a conexión frente a la comunicación sin conexión.
2. El proceso 1 manda sucesivamente 3 mensajes al proceso 2. ¿Cuál es el orden posible en el que pueden llegar los mensajes si:
   a. se utiliza un *socket* sin conexión para mandar cada mensaje?
   b. se utiliza un *socket* orientado a conexión para mandar cada mensaje?
3. En el método *setSoTimeout* de *DatagramSocket* (y de otras clases de *sockets*), ¿qué sucede si el periodo de plazo se fija en 0? ¿Significa que el plazo ocurre inmediatamente (ya que el periodo es 0)? Consulte el API en línea de Java para encontrar la respuesta.
4. Escriba un fragmento de código Java, que podría aparecer dentro de un método *main*, que abra un *socket* datagrama para recibir un datagrama de hasta 100 octetos, en un plazo máximo de 5 segundos. Si el plazo expira, debería visualizarse en la pantalla el siguiente mensaje: «agotado el plazo para recibir».
5. Este ejercicio guía al lector mediante experimentos con *sockets* datagrama sin conexión utilizando el código de *Ejemplo1*.

   Para empezar, se recomienda que se ejecuten ambos programas en una máquina, usando «localhost» como nombre de la máquina. Por ejemplo, se puede introducir el mandato «java Ejemplo1Emisor localhost 12345 hola» para ejecutar *Ejemplo1Emisor*. Opcionalmente, se podrían repetir los ejercicios ejecutando los programas en máquinas separadas, presuponiendo que se tiene acceso a tales máquinas.

   a. Compile los ficheros *.java*. A continuación, ejecute los dos programas (i) arrancando el receptor y, después, (ii) el emisor, teniendo cuidado de especificar los argumentos de línea de mandato apropiados en cada caso. El mensaje mandado no debería exceder la longitud máxima permitida en el receptor (esto es, 10 caracteres). Describa el resultado de la ejecución. *Nota*: Para ayudar a seguir la pista del resultado de la ejecución, se recomienda que se ejecute cada aplicación en una ventana separada en la pantalla, pre-

Para ayudar a seguir la pista del resultado de la ejecución, se recomienda que se ejecute cada aplicación en una ventana separada en la pantalla, preferiblemente fijando el tamaño y situando las ventanas de manera que se pueda ver una junto a otra.

feriblemente fijando el tamaño y las ventanas de manera que se pueda ver una junto a otra.

b. Vuelva a ejecutar las aplicaciones del apartado a, esta vez cambiando el orden de los pasos (i) y (ii). Describa y explique el resultado.

c. Repita el apartado a, esta vez mandando un mensaje de longitud más grande que la máxima longitud permitida (por ejemplo, «01234567890»). Describa y explique la salida producida.

d. Añada código al proceso receptor de manera que el plazo máximo de bloqueo del *receive* sea de 5 segundos. Arranque el proceso receptor pero no el proceso emisor. ¿Cuál es el resultado? Descríbalo y explíquelo.

e. Modifique el código original de *Ejemplo1* de manera que el receptor ejecute indefinidamente un bucle que repetidamente reciba y después muestre los datos recibidos. Vuelva a compilarlo. A continuación, (i) arranque el receptor, (ii) ejecute el emisor, mandando un mensaje «mensaje1», y (iii) en otra ventana, arranque otra instancia del emisor, mandando un mensaje «mensaje2». ¿El receptor recibe los dos mensajes? Capture el código y la salida. Describa y explique el resultado.

f. Modifique el código original de *Ejemplo1* de manera que el emisor utilice el mismo *socket* para mandar el mismo mensaje a dos receptores diferentes. Primero arranque los dos receptores, después el emisor. ¿Cada receptor recibe el mensaje? Capture el código y la salida. Describa y explique el resultado.

g. Modifique el código original de *Ejemplo1* de manera que el emisor utilice dos *sockets* distintos para mandar el mismo mensaje a dos receptores diferentes. En primer lugar, arranque los dos receptores y, después, el emisor. ¿Recibe el mensaje cada receptor? Capture el código y la salida. Describa y explique el resultado.

h. Modifique el código del último paso de modo que el emisor envíe repetidamente suspendiéndose el mismo durante 3 segundos entre cada envío. (Recuerde que en el Capítulo 1 se vio cómo utilizar *Thread.sleep()* para suspender un proceso durante un intervalo de tiempo especificado). Modifique el receptor de manera que ejecute un bucle que repetidamente reciba datos y luego los muestre. Compile y ejecute los programas durante unos pocos minutos antes de terminarlos (tecleando la secuencia «control-c»). Describa y explique el resultado.

i. Modifique el código original de *Ejemplo1* de modo que el emisor también reciba un mensaje del receptor. Se debería necesitar sólo un *socket* en cada proceso. Compile, ejecute y entregue su código; asegúrese de modificar los comentarios en consecuencia.

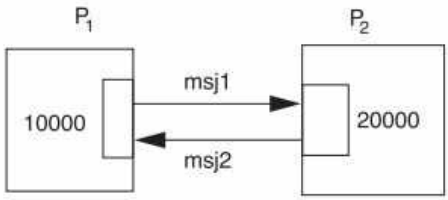
6. Este ejercicio guía al lector mediante experimentos con *socket* datagramas sin conexión mediante el código *Ejemplo2*.

a. Dibuje un diagrama de clases UML para ilustrar la relación entre las clases *DatagramSocket*, *MiSocketDatagrama*, *Ejemplo2EmisorReceptor* y *Ejemplo2ReceptorEmisor*. No se requiere especificar los atributos y métodos de la clase *DatagramSocket*.

b. Compile los ficheros *.java*. A continuación, arranque *Ejemplo2ReceptorEmisor*, seguido de *Ejemplo2EmisorReceptor*. Un ejemplo de los mandatos necesarios para ejecutar los programas es el siguiente:

```
java Ejemplo2ReceptorEmisor localhost 20000 10000 msj1
java Ejemplo2EmisorReceptor localhost 10000 20000 msj2
```

Describa el resultado. ¿Por qué es importante el orden de ejecución de los dos procesos?

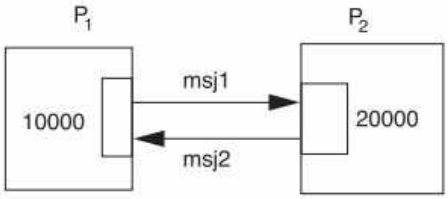


c. Modifique el código de manera que el proceso *EmisorReceptor* envíe y después reciba repetidamente, suspendiéndose a sí mismo durante 3 segundos entre cada iteración. Vuelva a compilar y repita la ejecución. Haga lo mismo con el *ReceptorEmisor*. Compile y ejecute los programas durante unos pocos minutos antes de ponerles fin (tecleando la sentencia «control-c»). Describa y explique el resultado.

7. Este ejercicio guía al lector mediante experimentos con un *socket* datagrama orientado a conexión utilizando el código *Ejemplo3*.

   a. Compile y ejecute los ficheros de código fuente de *Ejemplo3*. Describa la salida de la ejecución. Un ejemplo de los mandatos requeridos para ejecutar los programas es el siguiente:

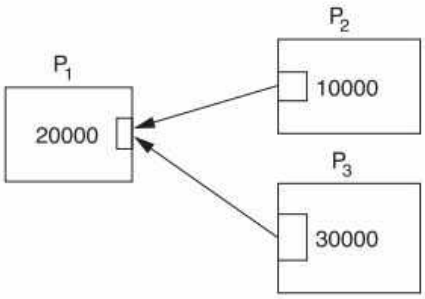
```
java Ejemplo3Receptor localhost 20000 10000 msj1
java Ejemplo3Emisor localhost 10000 20000 msj2
```



   b. Modifique el código de *Ejemplo3Emisor.java* de modo que la llamada al método *connect* especifique un número de puerto diferente del número de puerto del *socket* del receptor. (Se puede hacer simplemente añadiendo un 1 al número de puerto del receptor en la llamada al método de conexión). Cuando se vuelvan a ejecutar los programas (después de volver a compilar los ficheros de código fuente), el proceso emisor intentará ahora mandar datagramas cuya dirección de destino no coincide con la dirección especificada en la conexión del receptor. Describa y explique el resultado de la ejecución.

   c. Vuelva a ejecutar los programas originales. Esta vez arranque un segundo proceso emisor, especificándole la misma dirección del receptor. Un ejemplo de los mandatos necesarios para arrancar el árbol de procesos es:

```
java Ejemplo3Receptor localhost 1000 2000 msj1
java Ejemplo3Emisor localhost 2000 1000 msj2
java Ejemplo3Emisor localhost 2000 3000 msj3
```



Arranque los tres procesos en una sucesión rápida de manera que el segundo proceso emisor intente establecer una conexión con el proceso receptor cuando el *socket* de este último ya esté conectado al del primer proceso emisor. Describa y explique el resultado de la ejecución.

8. Este ejercicio guía al lector mediante experimentos con un *socket* en modo *stream* orientado a conexión utilizando los códigos *Ejemplo4* y *Ejemplo5*.
   a. Compile y ejecute *Ejemplo4*.java* (*Nota*: Se utiliza * como un carácter comodín, de modo que *Ejemplo4*.java* hace referencia a los ficheros cuyos nombres comienzan con «Ejemplo4» y terminan con «.java»). Arranque en primer lugar el *Aceptador* y, después, el *Solicitante*. Un ejemplo de los mandatos necesarios para ello es:

      ```
      java Ejemplo4AceptadorConexion localhost 12345 ¡Buenos dias!
      java Ejemplo4SolicitanteConexion localhost 12345
      ```

      Describa y explique el resultado.

   b. Repita el último apartado pero cambie el orden de ejecución de los programas:

      ```
      java Ejemplo4SolicitanteConexion localhost 12345
      java Ejemplo4AceptadorConexion localhost 12345 ¡Buenos dias!
      ```

      Describa y explique el resultado

   c. Añada un tiempo de retraso de 5 segundos en el proceso *AceptadorConexión* justo antes de que el mensaje se escriba en el *socket*, después repita el apartado a. Esto producirá el efecto de mantener al *Solicitante* bloqueado en la lectura durante 5 segundos adicionales de manera que se pueda observar visualmente el bloqueo. Muestre una traza de la salida de los procesos. ¿Están de acuerdo los mensajes de diagnóstico mostrados en la pantalla con el diagrama de eventos de la Figura 4.21?

   d. Modifique *Ejemplo4*.java* de manera que *AceptadorConexión* utilice *print()* para escribir un solo carácter cada vez en el *socket* antes de llevar a cabo un *println()* para escribir un final de línea. Vuelva a compilar y ejecutar los programas. ¿Se recibe el mensaje en su totalidad? Explíquelo.

   e. Compile y ejecute *Ejemplo5*.java*. Arranque en primer lugar el *Aceptador* y, después, el *Solicitante*. Un ejemplo de los mandatos necesarios para ello es:

      ```
      java Ejemplo5AceptadorConexion localhost 12345 ¡Buenos dias!
      java Ejemplo5SolicitanteConexion localhost 12345
      ```

      Debido a que el código es lógicamente equivalente a *Ejemplo4**, el resultado debería ser el mismo que el del apartado a. Modifique *Ejemplo5*.java* de modo que el proceso *AceptadorConexión* se convierta en el

emisor del mensaje y el *SolicitanteConexión* en el receptor del mensaje. (El lector podría querer borrar los mensajes de diagnóstico). Presente un diagrama de eventos, los listados de los programas y el resultado de la ejecución.

f. Modifique el *Ejemplo5*.java* de modo que el *SolicitanteConexión* mande un mensaje de respuesta al *AceptadorConexión* después de recibir un mensaje del *Aceptador*. El *Aceptador* debería visualizar el mensaje de respuesta. Presente un diagrama de eventos, los listados de los programas y el resultado de la ejecución.

9. ¿Hay interfaces de programación de *sockets* que proporcionen operaciones de E/S (lectura y escritura) no bloqueantes? Nombre alguno. ¿Cuáles son las ventajas de utilizar una API de este tipo en vez de las interfaces presentadas en este capítulo?
10. Examine el NIOS de Java en el JDK1.4. Escriba un informe describiendo cómo puede utilizarse la E/S de *sockets* no bloqueante en un programa de Java. Proporcione ejemplos de código para ilustrar la descripción.
11. ¿Qué es una API de *sockets* seguros? Nombre alguno de los protocolos de *sockets* seguros y una API que proporcione *sockets* seguros. Escriba un informe describiendo cómo puede utilizarse un *socket* seguro en un programa de Java. Proporcione ejemplos de código para ilustrar la descripción.

REFERENCIAS

1. Especificación del API de Java 2 plataforma v1.4, *http://java.sun.com/j2se/1.4/docs/api/index.html*
2. Introducción a SSL, *http://developer.netscape.com/docs/manuals/security/sslin/contents.htm.*
3. Extensión de *sockets* seguros de Java (TM), *http://java.sun.com./products/jsse/*
4. Kirby W. Angell, «The Java Socket Secure Extensions», *Dr. Dobb´s Journal*, febrero de 2001. *http://www.ddj.com/articles/2001/0102/0102a//0102a.htm?topic=security*
5. El protocolo TLS, RFC2246, *http://www.ietf.org/rfc/rfc2246.txt*
6. New I/O APIs, *http://java.sun.com/j2se/1.4/docs/guide/nio/index.html, java.sun.com*

CAPÍTULO 5

El paradigma cliente-servidor

En este capítulo se explorará en detalle el paradigma cliente-servidor. Se hará también uso del conocimiento de las interfaces de programación de *sockets* para examinar la implementación de las aplicaciones cliente-servidor que se muestran en este capítulo.

5.1. ANTECEDENTES

El término *cliente-servidor* tiene múltiples significados en informática. Puede referirse a una **arquitectura de red** donde los computadores en una red realizan diferentes papeles para compartir recursos. En una **arquitectura cliente-servidor**, a un computador se le denomina *servidor* si se dedica a gestionar recursos como impresoras o ficheros de manera que otros computadores, llamados *clientes*, puedan acceder a estos recursos a través del servidor. La arquitectura cliente-servidor, aunque relacionada con el paradigma cliente-servidor, no es el tema de este capítulo.

Arquitectura de red se refiere a cómo se conectan los computadores en una red. El término no se debe confundir con los modelos abstractos de arquitectura de red (el modelo OSI y el modelo de Internet), que se examinaron en el Capítulo 1.

En la informática distribuida, el paradigma cliente-servidor se refiere a un modelo de aplicaciones de red donde los procesos juegan uno de dos diferentes papeles: un **proceso servidor**, también llamado **servidor** para abreviar, se dedica a gestionar el acceso a algunos servicios de la red, mientras que los **procesos cliente**, llamados **clientes** para abreviar, acceden al servidor para obtener un servicio de red. Nótese que en la arquitectura cliente-servidor, los términos *cliente* y *servidor* están referidos a los **computadores**, mientras que en el paradigma de computación distribuida cliente-servidor, los términos se refieren a los **procesos**. En este libro, cuando se utiliza el término *cliente-servidor*, se usa para referirse al paradigma de computación distribuida.

La Figura 5.1 ilustra el concepto del modelo cliente-servidor. Un **proceso servidor** ejecuta en un computador conectado a la red, al cual nos referiremos como **máquina servidora**, para gestionar un servicio de red proporcionado por esa máquina. Nótese que es posible que una máquina servidora proporcione otros servicios de red que son gestionados por otros procesos. Un usuario, uno que típicamente esté usando otro computador, al que se llamará **máquina cliente**, utiliza un **proceso cliente** para acceder a un servicio particular. Es posible que otros procesos clientes (posiblemente

de otros servicios) ejecuten en la máquina cliente a la vez, pero debe usarse el proceso cliente apropiado para acceder a un servicio particular.

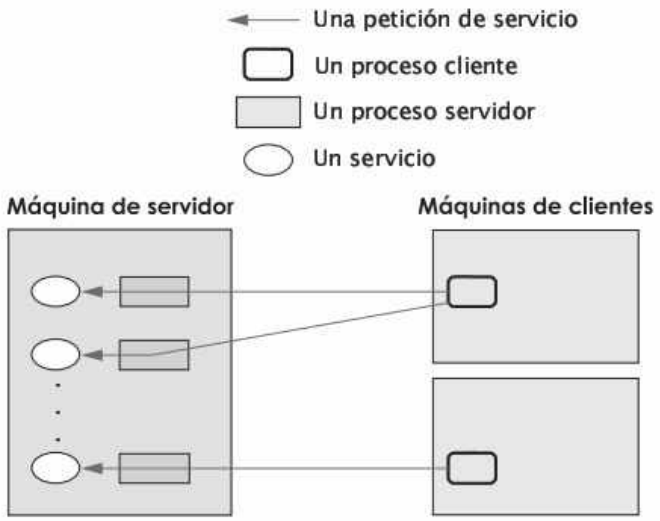


Figura 5.1. El paradigma de computación distribuida cliente-servidor.

El modelo cliente-servidor está diseñado para proporcionar servicios de red, los cuales fueron, y todavía son, la aplicación más popular de la computación distribuida. Por *servicio de red* se entiende un servicio proporcionado para permitir a los usuarios de la red compartir recursos. Tales recursos pueden ser tan triviales como la hora del día o tan complejos como ficheros en un sistema de ficheros de la máquina servidora o datos de un sistema de base de datos. A lo largo de los años, muchos de estos servicios de red se han estandarizado en Internet: *telnet*, que permite entrar (*logon*) de forma remota en una máquina servidora; *ftp*, para mandar y recibir ficheros de una máquina servidora; *Daytime*, que proporciona una marca de tiempo obtenida de una máquina servidora; y el *World Wide Web*, para buscar contenidos web en una máquina servidora.

5.2. CUESTIONES SOBRE EL PARADIGMA CLIENTE-SERVIDOR

Mientras que el concepto del paradigma es sencillo, en la **implementación** real hay un número de cuestiones que se deben afrontar. En esta sección se van a discutir estas cuestiones.

Una sesión de servicio

En el contexto del modelo cliente-servidor, se utilizará el término **sesión** para referirse a la interacción entre el servidor y el cliente. Como se muestra en la Figura 5.1, el servicio gestionado por un servidor puede ser accesible a múltiples clientes que quieran utilizar el servicio, algunas veces concurrentemente. Cada cliente entabla una sesión separada e independiente con el servidor, durante la cual el cliente conduce un diálogo con el servidor hasta que el cliente haya obtenido el servicio deseado.

La Figura 5.2 ilustra el flujo de ejecución del proceso servidor. Una vez que ha comenzado, un proceso servidor de red se ejecuta indefinidamente, realiza un bucle continuo que acepta peticiones de las sesiones de los clientes. Para cada cliente, el servidor conduce una sesión de servicio.

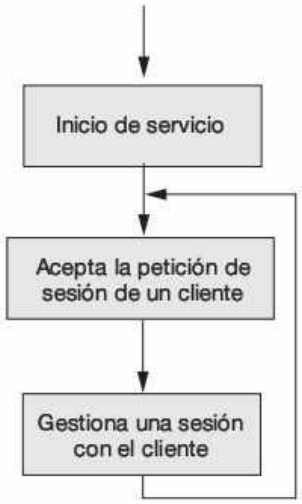


Figura 5.2. El flujo de ejecución del proceso servidor.

El protocolo de un servicio

Se necesita un protocolo para especificar las reglas que deben observar el cliente y el servidor durante una sesión de servicio. Dichas reglas incluyen especificaciones en cuestiones tales como (1) la manera de localizar el servicio, (2) la secuencia de comunicación entre los procesos, y (3) la representación e interpretación de los datos intercambiados.

Localización del servicio

Debe existir un mecanismo que permita que un proceso cliente localice un servidor de un determinado servicio. En el esquema más simple, la localización de un servicio es estática y se puede identificar utilizando la dirección del proceso servidor, en términos del nombre de la máquina y el número de puerto de protocolo asignado al proceso servidor. Este es el esquema usado para los servicios de Internet, donde a cada servicio de Internet se le asigna un número de puerto específico. En particular, a un servicio bien conocido, como FTP, HTTP, o *telnet*, se le asigna un número de puerto por defecto que se reserva en cada máquina de Internet para ese servicio. Por ejemplo, al servicio FTP se le asignan dos números de puerto de TCP: 20 y 21, y al HTTP el puerto de TCP 80.

En un sistema UNIX se puede consultar un fichero denominado *services* dentro del directorio /etc, o sea, /etc/services, para encontrar estas asignaciones de puerto.

En un nivel más alto de abstracción, un servicio puede identificarse utilizando un nombre lógico registrado en un directorio o en un registro. El nombre lógico necesitará traducirse a la ubicación física del proceso servidor. Si la traducción se realiza en tiempo de ejecución (es decir, cuando se ejecuta un proceso cliente), es posible que la localización del servicio sea dinámica, en cuyo caso se dice que el servicio es **transparente de la ubicación**.

Comunicaciones entre procesos y sincronización de eventos

En el modelo cliente-servidor, la interacción de los procesos sigue un patrón de petición-respuesta (*véase* la Figura 5.3). Durante una sesión de servicio, un cliente hace una petición al servidor, que contesta con una respuesta. El cliente puede hacer una petición subsiguiente, seguida por una respuesta del servidor. Este patrón se puede repetir indefinidamente hasta que concluya la sesión.

Por cada petición realizada, el cliente debe esperar por la respuesta del servidor antes de continuar. Por ejemplo, uno de los servicios más simples de la red es el servicio *Daytime*, mediante el cual un proceso cliente simplemente obtiene una marca de tiempo (la hora del día en la máquina servidora) del proceso servidor. Traducido a una hipotética conversación en español, el diálogo durante una sesión de servicio de este protocolo se llevaría a cabo de la siguiente forma:

El cliente: Hola, soy <dirección del cliente>. Por favor, ¿puede darme una marca de tiempo?

El servidor: Aquí la tiene: (en este punto estará la marca de tiempo).

De manera similar, el diálogo en una sesión web se realizaría de la siguiente manera:

El cliente: Hola, soy <dirección del cliente>.

El servidor: Vale. Soy un servidor web y entiendo el protocolo HTTP 1.0.

El cliente: Estupendo. Por favor, envíeme la página web *index.html* en la raíz de su árbol de documentos.

El servidor: Muy bien, aquí está lo que hay en la página (aquí aparecerá el contenido).

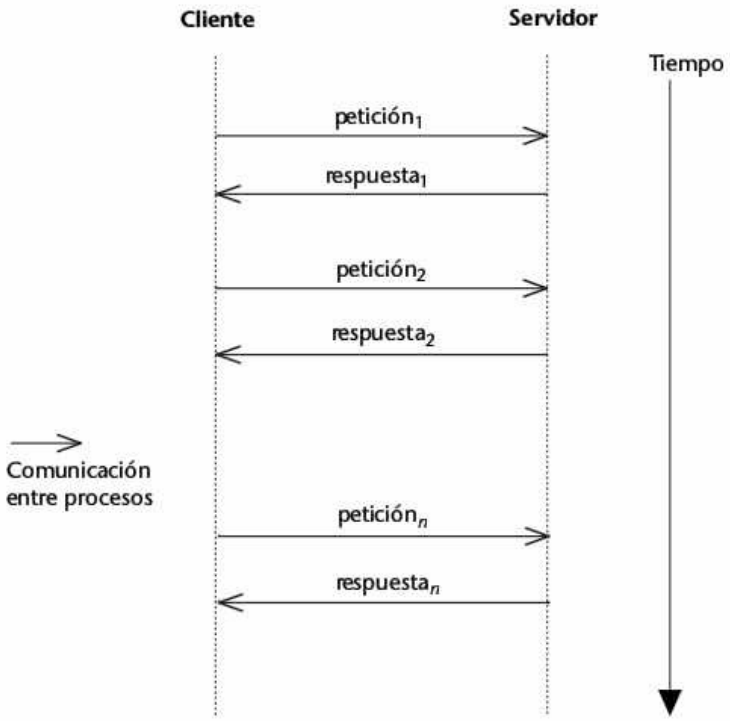


Figura 5.3. El patrón petición-respuesta de las comunicaciones entre procesos en el modelo cliente-servidor.

El diálogo en cada sesión sigue un patrón descrito en el protocolo especificado para el servicio. Por ejemplo, el servicio *Daytime* de Internet soporta el protocolo especificado en el RFC 867 de Internet [Postel, 1]. Cualquier implementación del programa cliente o servidor debe adherirse a la especificación del protocolo, incluyendo cómo debería proceder el diálogo de cada sesión. Entre otras cosas, la especificación define (1) la secuencia de intercomunicaciones entre el cliente y el servidor, (2) la sintaxis y la semántica de cada petición y respuesta, y (3) la acción esperada en cada lado al recibir una petición o respuesta determinada.

La **sintaxis** se refiere a la gramática y a la ortografía de un mensaje. La **semántica** corresponde con la interpretación y el significado de los mensajes.

Utilizando nuevamente como ejemplo el sencillo servicio *Daytime*, el protocolo especifica lo siguiente:

- No hay necesidad de ninguna sintaxis específica en las peticiones del cliente, puesto que contactar con el servidor ya supone automáticamente una solicitud de la hora del día.
- La respuesta es una marca de tiempo formateada como una cadena de caracteres de acuerdo con la especificación de protocolo. (Un toque humorístico: esta interacción no es muy diferente a la que ocurre cuando un padre, al recibir una llamada telefónica de un hijo que asiste a una universidad, responde inmediatamente, «Vale, el dinero ya está en camino»).

Un diagrama de secuencia es una buena manera de documentar las comunicaciones entre procesos durante una sesión de servicio. La Figura 5.4 muestra el diagrama de secuencia para una sesión del servicio *Daytime*. Nótese que el único mensaje intercambiado es la marca de tiempo.

Figura 5.4. El paradigma de computación distribuida cliente-servidor.

Representación de datos

La elección de la representación de datos depende de la naturaleza y la necesidad del protocolo. Para un servicio sencillo como el *Daytime*, es lógico utilizar una representación en modo texto, como se describe en el RFC 867 [Postel, 1]:

```
Sintaxis de Daytime

No hay una sintaxis específica para daytime. Se recomienda que se limite
a los caracteres ASCII imprimibles, el espacio, el retorno de carro y el
salto de línea. La fecha y hora deberían ocupar una sola línea.

Una sintaxis popular es:
   Día de la semana, mes día, año hora-zona horaria
   Ejemplo:
        Martes, febrero 22, 1982 17:37:43-PST
```

En esta especificación, el formato, o sintaxis, de la marca de tiempo se deja a criterio del implementador.

Elegir una representación de datos en modo texto para un protocolo tiene la ventaja de permitir que el diálogo sea legible para todos, ya que se puede utilizar E/S en modo texto estándar para mostrar los datos intercambiados.

5.3. INGENIERÍA DE SOFTWARE DE UN SERVICIO DE RED

En este punto el lector ya está preparado para examinar cómo construir el software necesario para proporcionar un servicio de red.

Hay dos conjuntos de software involucrados: uno para el proceso cliente y el otro para el proceso servidor. El conjunto de software que se necesita en la máquina cliente para proporcionar un servicio o aplicación, incluyendo el programa cliente y su entorno de apoyo en tiempo de ejecución, se denomina a veces software del **lado cliente**, en contraposición al software del **lado servidor**, que incluye al programa servidor y todos los entornos de apoyo en tiempo de ejecución que necesita. Suponiendo que el protocolo está bien definido y comprendido, es posible desarrollar separada e independientemente el software en ambos lados.

Se usará nuevamente el protocolo *Daytime* como un ejemplo del proceso de desarrollo de un servicio de red.

Arquitectura de software

En el Capítulo 1 se presentó una arquitectura de software de tres niveles para las aplicaciones de red, de manera que las funcionalidades de cada aplicación puedan dividirse en tres niveles: presentación, aplicación y servicio. Según esto, la arquitectura de software de una aplicación construida utilizando el modelo cliente-servidor, o una aplicación cliente-servidor, se puede describir de la siguiente manera:

- **Nivel de presentación.** En el lado del servidor, se necesita una interfaz de usuario (UI, *User Interface*) para permitir arrancar el proceso servidor; típicamente, basta con la ejecución de una línea de mandato. En el lado cliente, se precisa que el cliente proporcione una interfaz de usuario. Mediante dicha interfaz, un usuario en la máquina cliente puede solicitar el servicio y recibir la respuesta del servidor. En el código de ejemplo, se utilizará una interfaz de usuario en modo texto por simplicidad, pero en su lugar puede utilizarse una interfaz de usuario gráfica (GUI, *Graphic User Interface*) o una página web. En el Capítulo 9 se examinará cómo construir una página web que acepte y visualice datos.
- **Nivel de lógica de aplicación.** En el lado del servidor, necesita obtenerse la hora del día del sistema y mandarla a la máquina cliente. En el lado del cliente, hay que enviar al servidor la petición del usuario y mostrarle la respuesta del servidor.
- **Nivel de servicio.** Los servicios requeridos para dar soporte a la aplicación son (1) en el lado del servidor, una lectura del reloj de la máquina servidora (para la marca de tiempo); y (2) en ambos lados, un mecanismo de IPC.

Cada computador tiene un reloj y la mayoría de los lenguajes de programación proporcionan una API para obtener el valor del reloj.

Las funcionalidades de los tres niveles deben estar presentes en el software conjunto. Algunas funcionalidades pertenecen al cliente y otras al servidor. Para aplicaciones de gran escala, es aconsejable diseñar el software de manera que las funcionalidades de los tres niveles se encapsulen en módulos de software separados. *Véase* la Figura 5.5.

Figura 5.5. Arquitectura de software para una aplicación cliente-servidor.

Mecanismo de IPC

Hay que considerar el mecanismo de IPC que se utilizará en la lógica de la aplicación; el mecanismo seleccionado afectará en cómo las aplicaciones transmiten datos.

Dado el repertorio limitado de mecanismos de IPC aprendido hasta ahora, se presenta la posibilidad de elegir entre (1) un *socket* datagrama sin conexión, (2) un *socket* datagrama orientado a conexión, o (3) un *socket* en modo *stream*.

Con la arquitectura de software apropiada, los detalles del mecanismo se esconderán completamente de la lógica de presentación y afectarán únicamente a la lógica de aplicación en la sintaxis de los mandatos de IPC.

Se comenzará examinando un código de ejemplo que utiliza el *socket* datagrama sin conexión. Se verá posteriormente cómo la arquitectura de software permite adaptar el software al API de *socket* en modo *stream* orientado a conexión.

No hay que alarmarse por el gran número de clases de Java utilizadas en el ejemplo. Cada clase es simple y representa un módulo de software que implementa un nivel de la lógica en la arquitectura de software.

Cliente-servidor *Daytime* usando *sockets* datagrama sin conexión

Software del lado cliente

Lógica de presentación. La Figura 5.6 presenta la clase *ClienteDaytime1.java*, que encapsula la lógica de presentación del lado cliente; esto es, proporciona la interfaz de usuario del proceso cliente. Se apreciará que el código en esta clase se ocupa meramente de obtener la entrada (la dirección del servidor) del usuario y de mostrar la salida (la marca de tiempo) al mismo. Para obtener la marca de tiempo, se realiza una llamada de método a una clase auxiliar, *ClienteDaytimeAuxiliar1.java*. Este método esconde los detalles de la lógica de la aplicación y del servicio subyacente. Como resultado, el programador de *ClienteDaytime1.java* no necesita conocer qué tipos de *sockets* se utilizan en la IPC.

Lógica de aplicación. La clase *ClienteDaytimeAuxiliar1.java* (Figura 5.7) encapsula la lógica de aplicación del lado cliente. Este módulo realiza la IPC para mandar una petición y recibir una respuesta utilizando una subclase de *DatagramSocket*, *MiSocketDatagramaCliente*. Nótese que a este módulo se le esconden los detalles de la utilización de *sockets* datagrama. En particular, este módulo no necesita tratar con el vector de octetos que lleva los datos de la carga.

Lógica de servicio. La clase *MiSocketDatagramaCliente.java* (Figura 5.8) proporciona los detalles del servicio de IPC, en este caso utilizando el API de *socket* datagrama.

Hay al menos dos ventajas significativas en separar los tres niveles de la lógica de la aplicación en módulos de software diferentes:

- Cada módulo pueden desarrollarlo personas que estén especialmente capacitadas para centrarse en ese módulo. Los ingenieros de software especializados en interfaces de usuario pueden concentrarse en el desarrollo de los módulos de la lógica de presentación, mientras que aquellos especializados en la lógica de aplicación y de servicio pueden dedicarse al desarrollo de los otros módulos.
- La separación permite que se hagan modificaciones en la lógica de un nivel sin requerir que se hagan cambios en otros niveles. Por ejemplo, se puede cambiar la interfaz de usuario de modo texto a modo gráfico sin necesitar cambios en la lógica de aplicación o la de servicio. De manera similar, los cambios hechos en la lógica de aplicación deberían ser transparentes al nivel de presentación.

La Figura 5.9 es un diagrama de clases UML que describe las clases utilizadas en la implementación del programa *ClienteDaytime1*.

Software del lado servidor

Lógica de presentación. Típicamente, hay muy poca lógica de presentación en el lado del servidor. En este caso, la única entrada de usuario corresponde con el puerto del servidor que, por simplificar, se maneja utilizando un argumento de línea de mandato.

Lógica de aplicación. La clase *ServidorDaytime1.java* (Figura 5.10) encapsula la lógica de aplicación del lado servidor. Este módulo ejecuta un bucle infinito, esperando una petición de un cliente y después dirige una sesión de servicio para ese cliente. El módulo realiza la IPC para recibir una petición y manda una respuesta utilizando una subclase de *DatagramSocket*, *MiSocketDatagramaServidor*. Nótese que a este módulo se le esconden los detalles de la utilización de los *sockets* datagrama. En particular, este módulo no necesita tratar con el vector de octetos que lleva los datos de la carga.

Lógica de servicio. La clase *MiSocketDatagramaServidor* (Figura 5.11) proporciona los detalles del servicio de IPC, en este caso utilizando el API de *socket* datagrama. Esta clase es similar a la clase *MiSocketDatagramaCliente*, con la excepción de que el método *recibeMensaje* devuelve un objeto de la clase *MensajeDatagrama* (Figura 5.12), que contiene la dirección del emisor, así como el mensaje en sí mismo. El servidor necesita la dirección del emisor para mandar una petición al cliente. ésta es una idiosincrasia del *socket* sin conexión: el servidor no tiene manera de conocer dónde mandar una respuesta en caso contrario. Los métodos empleados para obtener la dirección del emisor de un datagrama recibido son *getAddress* y *getHost*, cuyas descripciones se muestran en la Tabla 5.1. Estos dos métodos no se mencionaron en el último capítulo cuando se presentó el API de *sockets* datagrama.

Tabla 5.1. Los métodos *getAddress* y *getPort* de la clase *DatagramPacket*.

Método	Descripción
public InetAddress getAddress()	Devuelve la dirección IP de la máquina remota de un *socket* desde el que fue recibido el datagrama.
public int getPort()	Devuelve el número de puerto en la máquina remota de un *socket* desde el que fue recibido el datagrama.

Figura 5.6. *ClienteDaytime1.java*.

```
1  import java.io.*;
2
3
4  /**
5  * Este módulo contiene la lógica de presentación de un ClienteDaytime.
6  * @author M. L. Liu
7  */
8  public class ClienteDaytime1 {
9    public static void main(String[] args) {
10     InputStreamReader is = new InputStreamReader(System.in);
11     BufferedReader br = new BufferedReader(is);
12     try {
13       System.out.println("Bienvenido al cliente Daytime.\n" +
14         "¿Cuál es el nombre de la máquina servidora?");
15       String nombreMaquina = br.readLine( );
16       if (nombreMaquina.length() == 0) // si el usuario no
                                          // introduce un nombre
17         nombreMaquina = "localhost"; // usa el nombre de máquina
                                        // por defecto
18       System.out.println("¿Cuál es el nº de puerto de la máquina
           servidora?");
19       String numPuerto = br.readLine();
20       if (numPuerto.length () == 0)
21         numPuerto = "13"; // número de puerto por defecto
22       System.out.println("Marca de tiempo recibida del servidor"
23         + ClienteDaytimeAuxiliar1.obtenerMarcatiempo(nombreMaquina,
             numPuerto));
24     } // fin de try
25     catch (Exception ex) {
26       ex.printStackTrace( );
27     } // fin de catch
28   } // fin de main
29 } // fin de class
```

(continúa)

Figura 5.7. *ClienteDaytimeAuxiliar1.java.*

```
1
2  import java.net.*;
3
4  /**
5  * Esta clase es un módulo que proporciona la lógica de aplicación
6  * de un cliente de Daytime.
7  * @author M. L. Liu
8  */
9  public class ClienteDaytimeAuxiliar1 {
10
11   public static String obtenerMarcatiempo(String nombreMaquina,
12     String numPuerto) {
13
14     String marcaTiempo = "";
15     try {
16       InetAddress serverHost = InetAddress.getByName(nombreMaquina);
17       int serverPort = Integer.parseInt(numPuerto);
18       // instancia un socket datagrama para tanto los datos de
19       // emisión como los de recepción
20       MiSocketDatagramaCliente miSocket = new
           MiSocketDatagramaCliente();
21       miSocket.enviaMensaje( serverHost, serverPort, "");
22       // ahora recibe la marca de tiempo
23       marcaTiempo = miSocket.recibeMensaje();
24       miSocket.close( );
25
26     } // fin de try
27     catch (Exception ex) {
28       ex.printStackTrace( );
29     } // fin de catch
30     return marcaTiempo;
31   } //fin de main
32 } // fin de class
```

* Es mejor técnica de desarrollo de software activar en este método una excepción en caso de errores causados por llamadas de métodos; esta técnica no se ha utilizado aquí para evitar la complejidad de crear otra clase adicional para las excepciones.

Figura 5.8. *MiSocketDatagramaCliente.java*

```
1  import java.net.*;
2  import java.io.*;
3
4  /**
5  * Una subclase de DatagramSocket que contiene
6  * métodos para mandar y recibir mensajes
7  * @author M. L. Liu
8  */
9  public class MiSocketDatagramaCliente extends DatagramSocket {
10   static final int MAX_LON = 100;
```

(continúa)

```
11      MiSocketDatagramaCliente( ) throws SocketException{
12        super( );
13      }
14
15      MiSocketDatagramaCliente(int numPuerto) throws SocketException{
16        super(numPuerto);
17      }
18
19      public void enviaMensaje(InetAddress maquinaReceptora,
20           int puertoReceptor, String mensaje) throws IOException {
21        byte[ ] almacenEnvio = mensaje.getBytes( );
22        DatagramPacket datagrama =
23          new DatagramPacket(almacenEnvio, almacenEnvio.length,
24        maquinaReceptora, puertoReceptor);
25        this.send(datagrama);
26      } // fin de enviaMensaje
27
28      public String recibeMensaje( )
29           throws IOException {
30        byte[ ] almacenRecepcion = new byte[MAX_LON];
31        DatagramPacket datagram =
32          new DatagramPacket(almacenRecepcion, MAX_LON);
33        this.receive(datagram);
34        String mensaje = new String(almacenRecepcion);
35        return mensaje;
36      } // fin de recibeMensaje
37    } // fin de class
```

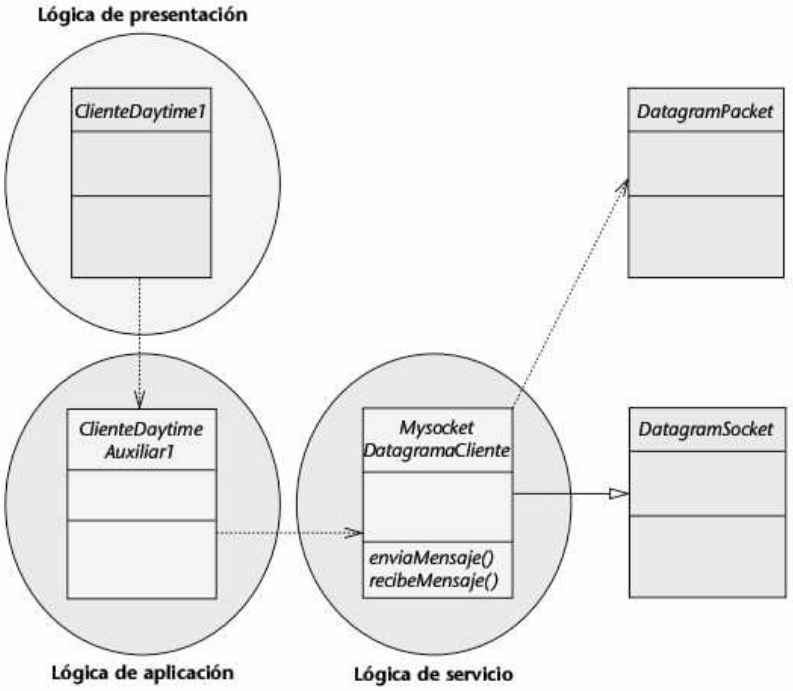


Figura 5.9. Diagrama de clases UML de *ClienteDaytime1* (no se muestran todos los atributos).

Figura 5.10. *ServidorDaytime1.java.*

```
1  import java.io.*;
2  import java.util.Date; // para obtener una marca de tiempo
3
4  /**
5  * Este módulo contiene la lógica de aplicación de un servidor Daytime
6  * que utiliza un socket datagrama para la comunicación entre procesos.
7  * Se requiere un argumento de línea de mandato para el puerto del
     servidor.
8  * @author M. L. Liu
9  */
10 public class ServidorDaytime1 {
11   public static void main(String[] args) {
12     int puertoServidor = 13; // puerto por defecto
13     if (args.length == 1 )
14       puertoServidor = Integer.parseInt(args[0]);
15     try {
16       // instancia un socket datagrama para tanto mandar como
17       // recibir datos
18       MiSocketDatagramaServidor miSocket =
            new MiSocketDatagramaServidor(puertoServidor);
19       System.out.println("El servidor Daytime está listo.");
20       while (true) { // bucle infinito
21         MensajeDatagrama peticion =
              miSocket.recibeMensajeYEmisor();
22         System.out.println("Petición recibida");
23         // no es importante el mensaje recibido; es la dirección
24         // del emisor lo que se necesita para responder..
25         // Ahora obtiene una marca de tiempo del sistema.
26         Date marcaTiempo = new Date ( );
27         System.out.println("marca de tiempo enviada:" +
              marcaTiempo.toString());
28         // Ahora manda la respuesta al solicitante.
29         miSocket.enviaMensaje(peticion.obtieneDireccion( ),
30           peticion.obtienePuerto( ), marcaTiempo.toString( ));
31       } // fin de while
32     } // fin de try
33     catch (Exception ex) {
34       ex.printStackTrace( );
35     } // fin de catch
36   } // fin de main
37 } // fin de class
```

Figura 5.11. *MiSocketDatagramaServidor.java.*

```
1  import java.net.*;
2  import java.io.*;
3
4
```

(continúa)

```
/**
* Una subclase de DatagramSocket que contiene
* métodos para mandar y recibir mensajes
* @author M. L. Liu
*/
public class MiSocketDatagramaServidor extends DatagramSocket {
  static final int MAX_LON = 100;
  MiSocketDatagramaServidor(int numPuerto) throws SocketException{
    super(numPuerto);
  }
  public void enviaMensaje(InetAddress maquinaReceptora,
       int puertoReceptor,
       String mensaje)
       throws IOException {
    byte[ ] almacenEnvio = mensaje.getBytes( );
    DatagramPacket datagrama =
      new DatagramPacket(almacenEnvio, almacenEnvio.length,
    maquinaReceptora, puertoReceptor);
    this.send(datagrama);
  } // fin de enviaMensaje

  public String recibeMensaje( )
       throws IOException {
    byte[ ] almacenRecepcion = new byte[MAX_LON];
    DatagramPacket datagrama =
      new DatagramPacket(almacenRecepcion, MAX_LON);
    this.receive(datagrama);
    String mensaje = new String(almacenRecepcion);
    return mensaje;
  } //fin de recibeMensaje

  public MensajeDatagrama recibeMensajeYEmisor( )
       throws IOException {
    byte[ ] almacenRecepcion = new byte[MAX_LON];
    DatagramPacket datagrama =
      new DatagramPacket(almacenRecepcion, MAX_LON);
    this.receive(datagrama);
    // crea un objeto MensajeDatagrama, para contener
    // el mensaje recibido y la dirección del emisor
    MensajeDatagrama valorDevuelto = new MensajeDatagrama( );
    valorDevuelto.fijaValor(new String(almacenRecepcion),
      datagrama.getAddress( ),
      datagrama.getPort( ));
    return valorDevuelto;
  } //fin de recibeMensaje
} //fin de class
```

Figura 5.12. *MensajeDatagrama.java.*

```
1  import java.net.*;
2  /**
3  * Una clase para utilizar con MiSocketDatagramaServidor para
4  * devolver un mensaje y la dirección del emisor
5  * @author M. L. Liu
6  */
7  public class MensajeDatagrama{
8     private String mensaje;
9     private InetAddress direccionEmisor;
10    private int puertoEmisor;
11    public void fijaValor(String mensaje, InetAddress dir,
        int puerto) {
12       this.mensaje = mensaje;
13       this.direccionEmisor = dir;
14       this.puertoEmisor = puerto;
15    }
16
17    public String obtieneMensaje( ) {
18       return this.mensaje;
19    }
20
21    public InetAddress obtieneDireccion( ) {
22       return this.direccionEmisor;
23    }
24
25    public int obtienePuerto( ) {
26       return this.puertoEmisor;
27    }
28 } //fin de class
```

La Figura 5.13 es un diagrama de clases UML que describe las clases utilizadas en la implementación del programa *ServidorDaytime1*.

Cliente-servidor *Daytime* usando *sockets* en modo *stream*

En la sección previa se vio cómo el servicio *Daytime* puede implementarse utilizando el *socket* datagrama sin conexión para el mecanismo de IPC.

Supóngase que se quiere implementar el mismo servicio utilizando en su lugar *sockets* orientados a conexión. Dado que este cambio sólo afecta básicamente a la lógica de servicio, sólo se necesitarían hacer modificaciones importantes en las clases en el nivel de la lógica de servicio, como se mostrará en esta sección. La lógica de aplicación, específicamente la clase *auxiliar*, se necesitará ajustar en consecuencia.

Software del lado cliente

Lógica de presentación La Figura 5.14 presenta la clase *ClienteDaytime2* que es la misma que *ClienteDaytime1*, excepto por un cambio en el nombre de la clase auxiliar, *ClienteDaytimeAuxiliar2*. (De hecho, *ClienteDaytime2* puede ser exactamente igual que *ClienteDaytime1* si simplemente se remplaza el cuerpo de la clase

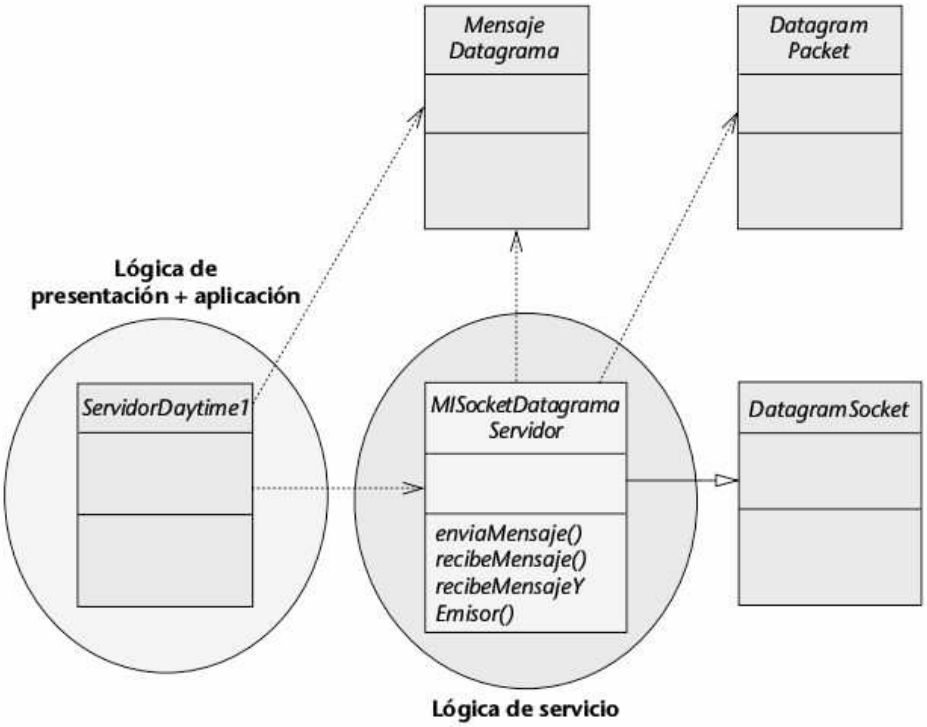


Figura 5.13. El diagrama de clases UML de *ServidorDaytime1* (no muestra todos los atributos).

ClienteDaytimeAuxiliar1 por el de *ClienteDaytimeAuxiliar2*.) El método *obtenerMarcaTiempo* en *ClienteDaytimeAuxiliar2* ahora utiliza el API de *sockets* en modo *stream*, pero los detalles son transparentes a *ClienteDaytime2*.

Lógica de aplicación. La clase *ClienteDaytimeAuxiliar2* (Figura 5.15), que encapsula la lógica de aplicación del lado cliente, es similar a la de la clase *ClienteDaytimeAuxiliar1*, excepto en que se utiliza un *socket* en modo *stream* en lugar de un *socket* datagrama. Nótese que ya no se requiere que el cliente envíe un mensaje nulo (que transporte la dirección de respuesta) para una petición, ya que la dirección de respuesta está encapsulada en la conexión).

Lógica de servicio. La clase *MiSocketStream* (Figura 5.17) proporciona los detalles del servicio de IPC, en este caso utilizando el API de *sockets* en modo *stream*. La clase *MiSocketStream* es una **clase de envoltura**, en el sentido de que «envuelve» a la clase Socket (eso es, contiene una variable de instancia que es una referencia a un objeto *Socket*) y proporciona métodos para mandar un mensaje a un *socket* y recibir un mensaje desde éste.

Software del lado servidor

Lógica de presentación. El código de *ServidorDaytime2* es idéntico al de *ServidorDaytime1*. La única entrada de usuario corresponde con el puerto servidor, el cual, para simplificar, se maneja utilizando un argumento de línea de mandato.

Lógica de aplicación. El código de *ServidorDaytime2* (Figura 5.16) usa el API de *sockets* en modo *stream* para aceptar una conexión. La referencia de *socket* devuelta

(que corresponde con el *socket* de datos) se utiliza después para instanciar un objeto *MiSocketStream*, cuyo método *enviaMensaje* se emplea para transmitir una marca de tiempo al cliente en el otro extremo de la conexión.

Lógica de servicio. La misma clase de envoltura utilizada en el cliente, *MiSocketStream* (Figura 5.17), se utiliza también en el servidor, ya que contiene los métodos necesarios para la IPC en modo *stream*. Nótese que es posible usar una clase diferente, o incluso un mecanismo distinto, para proporcionar la lógica de servicio si el software de servidor se desarrollase independientemente del software de cliente.

Para insistir en el tema, el cambio para pasar a utilizar el API de *sockets* en modo *stream*, requiere sólo modificaciones significativas en los módulos que proporcionan la lógica de servicio en cada lado.

Figura 5.14. *ClienteDaytime2.java.*

```
1  import java.io.*;
2
3
4  /**
5  * Este módulo contiene la lógica de presentación de un ClienteDaytime.
6  * @author M. L. Liu
7  */
8  public class ClienteDaytime2 {
9     public static void main(String[ ] args) {
10       InputStreamReader is = new InputStreamReader(System.in);
11       BufferedReader br = new BufferedReader(is);
12       try {
13          System.out.println("Bienvenido al cliente Daytime.\n" +
14             "¿Cuál es el nombre de la máquina servidora?");
15          String nombreMaquina = br.readLine();
16          if (nombreMaquina.length() == 0) // si el usuario no
                                             // introduce un nombre
17             nombreMaquina = "localhost"; // usa el nombre de máquina
                                            // por defecto
18          System.out.println("¿Cuál es el nº puerto de la máquina
               servidora?");
19          String numPuerto = br.readLine();
20          if (numPuerto.length () == 0)
21             numPuerto = "13"; // número de puerto por defecto
22          System.out.println("Marca de tiempo recibida del servidor"
23             + ClienteDaytimeAuxiliar2.obtenerMarcaTiempo(nombreMaquina,
                  numPuerto));
24       } // fin de try
25       catch (Exception ex) {
26          ex.printStackTrace( );
27       } // fin de catch
28    } // fin de main
29 } // fin de class
```

Figura 5.15. *ClienteDaytimeAuxiliar2.java.*

```
1
2  import java.net.*;
3
4  /**
5  * Esta clase es un módulo que proporciona la lógica de aplicación
6  * para un cliente Daytime que utiliza un socket en modo stream
     para IPC.
7  * @author M. L. Liu
8  */
9  public class ClienteDaytimeAuxiliar2 {
10
11   public static String obtenerMarcaTiempo(String nombreMaquina,
12     String numPuerto) throws Exception {
13
14
15     String marcaTiempo = "";
16
17     int puertoServidor = Integer.parseInt(numPuerto);
18     // instancia un socket en modo stream y espera a que se haga
19     // una conexión al puerto servidor
20   /**/System.out.println("Petición de conexión realizada");
21     MiSocketStream miSocket =
22       new MiSocketStream(nombreMaquina, puertoServidor);
23     // ahora espera hasta recibir la marca de tiempo
24     marcaTiempo = miSocket.recibeMensaje();
25     miSocket.close( ); // implica la desconexión
26     return marcaTiempo;
27   } // fin
28 } // fin de class
```

Figura 5.16. *ServidorDaytime2.java.*

```
1  import java.io.*;
2  import java.net.*;
3  import java.util.Date; // para obtener una marca de tiempo
4
5  /**
6  * Este módulo contiene la lógica de aplicación de un servidor Daytime
7  * que utiliza un socket orientado a conexión para IPC.
8  * Se requiere un argumento de línea de mandato para el puerto del
     servidor.
9  * @author M. L. Liu
10 */
11 public class ServidorDaytime2 {
12   public static void main(String[] args) {
13     int puertoServidor = 13; // puerto por defecto
14     if (args.length == 1 )
15       puertoServidor = Integer.parseInt(args[0]);
```

(continúa)

```
16        try {
17          // instancia un socket stream para aceptar
18          // las conexiones
19          ServerSocket miSocketConexion =
20            new ServerSocket(puertoServidor);
21          System.out.println("El servidor Daytime está listo.");
22          while (true) { // bucle infinito
23            // espera para aceptar una conexión
24  /**/       System.out.println("Espera una conexión.");
25             MiSocketStream miSocketDatos = new MiSocketStream
26             (miSocketConexion.accept( ));
27             // Nota: no hay necesidad de leer una petición - la
28             // petición es implícita.
29  /**/       System.out.println("Un cliente ha hecho una conexión.");
30             Date marcaTiempo = new Date ( );
31  /**/       System.out.println("marca de tiempo enviada: "+
                  marcaTiempo.toString());
32             // ahora manda la respuesta al solicitante.
33             miSocketDatos.enviaMensaje(marcaTiempo.toString( ));
34             miSocketDatos.close( );
35          } // fin de while
36        } // fin de try
37        catch (Exception ex) {
38          ex.printStackTrace( );
39        }
40      } // fin de main
41  } // fin de class
```

Figura 5.17. *MiSocketStream.java.*

```
 1  import java.net.*;
 2  import java.io.*;
 3
 4  /**
 5  *     Una clase de envoltura de Socket que contiene
 6  *     métodos para mandar y recibir mensajes.
 7  *     @author M. L. Liu
 8  */
 9  public class MiSocketStream extends Socket {
10    private Socket socket;
11    private BufferedReader entrada;
12    private PrintWriter salida;
13
14    MiSocketStream(String maquinaAceptadora,
15       int puertoAceptador ) throws SocketException,
16          IOException{
17       socket = new Socket(maquinaAceptadora, puertoAceptador );
18       establecerFlujos( );
```

(continúa)

```
19
20      }
21
22      MiSocketStream(Socket socket) throws IOException {
23        this.socket = socket;
24        establecerFlujos( );
25      }
26
27      private void establecerFlujos( ) throws IOException{
28        // obtiene un flujo de salida para leer del socket de datos
29        InputStream flujoEntrada = socket.getInputStream();
30        entrada =
31          new BufferedReader(new InputStreamReader(flujoEntrada));
32        OutputStream flujoSalida = socket.getOutputStream();
33        // crea un objeto PrintWriter para salida en modo carácter
34        salida =
35          new PrintWriter(new OutputStreamWriter(flujoSalida));
36      }
37
38      public void enviaMensaje(String mensaje)
39        throws IOException {
40        salida.println(mensaje);
41        // La subsiguiente llamada al método flush es necesaria para
42        // que los datos se escriban en el flujo de datos del socket
43        // antes de que se cierre el socket.
44        salida.flush();
45      } // fin de enviaMensaje
46
47      public String recibeMensaje( )
48        throws IOException {
49        // lee una línea del flujo de datos
50        String mensaje = entrada.readLine( );
51        return mensaje;
52      } // fin de recibeMensaje
53
54    } //fin de class
```

Las Figuras 5.18 y 5.19 son los diagramas de clases UML describiendo las relaciones de clases entre *ClienteDaytime2* y *ServidorDaytime2*, respectivamente.

Se acaba de presentar un servicio de red basado en el protocolo *Daytime* implementado de dos maneras diferentes, una (**Daytime*1) utilizando un mecanismo de IPC sin conexión, la otra (*Daytime**2) utilizando un mecanismo orientado a conexión. Un servidor, como *ServidorDaytime1*, que utiliza un mecanismo de IPC sin conexión se llama un **servidor sin conexión**, mientras que uno como *ServidorDaytime2* se denomina **servidor orientado a conexión.**

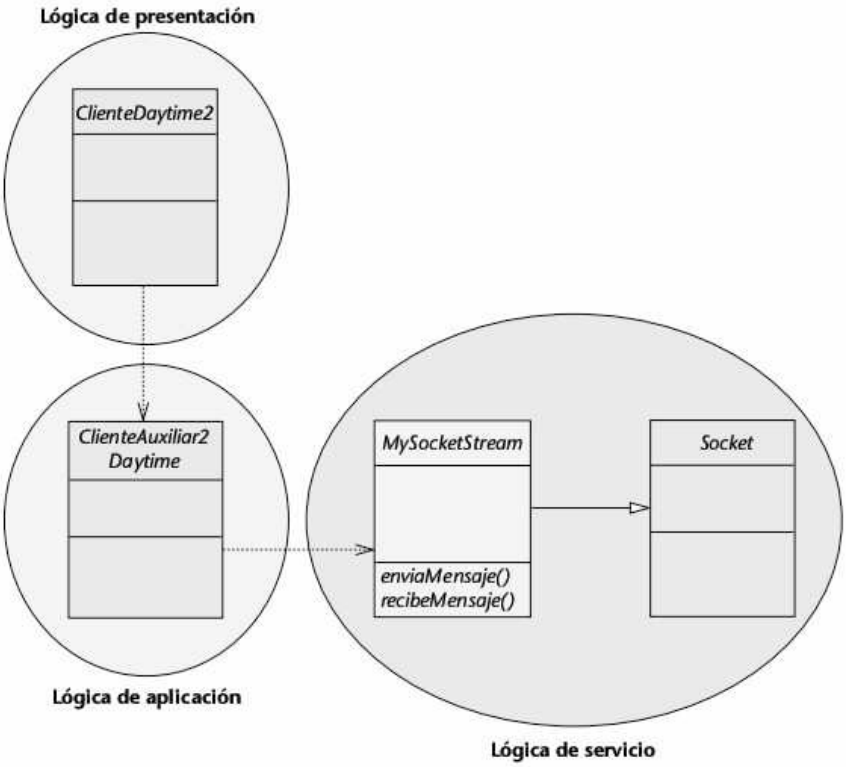


Figura 5.18. Diagrama de clases UML de *ClienteDaytime2* (no se muestran todos los atributos).

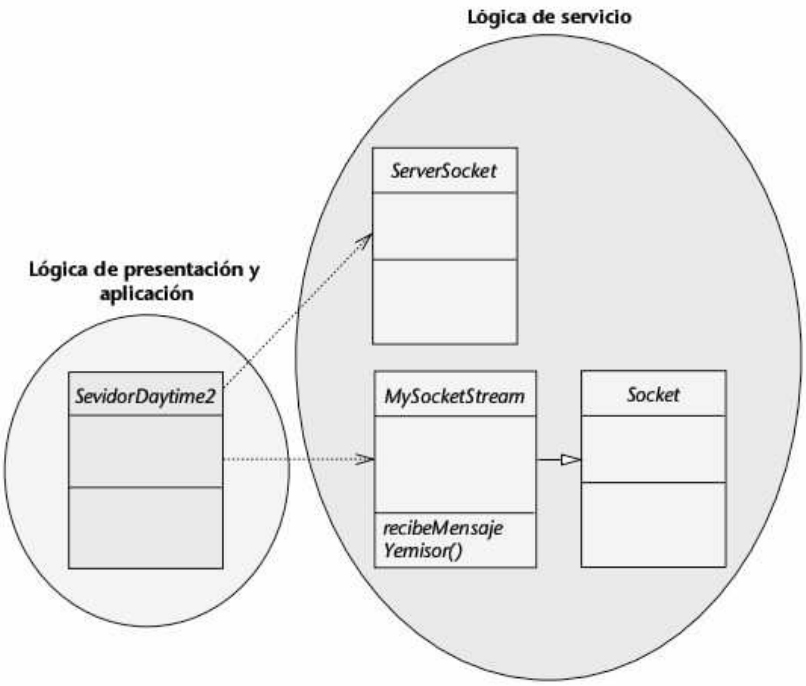


Figura 5.19. Diagrama de clases UML de *ServidorDaytime2* (no se muestran todos los atributos).

Prueba de un servicio de red

Debido a su complejidad inherente, es notoriamente difícil probar el buen funcionamiento del software de red. Incluso un servicio de red sencillo como *Daytime* plantea un reto para los principiantes. Seguidamente se exponen algunos consejos que pueden ser útiles:

- Utilice la arquitectura de software de tres niveles y organice en módulos cada nivel tanto en el lado del cliente como en el del servidor.
- Use una estrategia gradual o paso a paso en el desarrollo de cada módulo. Comience con resguardos (*stubs*) para cada método, y compile y pruebe cada módulo después de introducir detalles adicionales.
- Desarrolle en primer lugar el cliente. A veces es útil emplear un servidor de *Echo* (que se presentará en la siguiente sección), cuya corrección sea ya conocida y que use un mecanismo de IPC compatible, para probar el buen funcionamiento del cliente independientemente del servidor. Haciéndolo de esta manera, se puede desarrollar el cliente separadamente del servidor.
- Utilice mensajes de diagnóstico (similares a los incluidos en el código de ejemplo presentado en este capítulo) dentro del código fuente para informar del progreso del programa durante su ejecución.
- Compruebe el conjunto cliente-servidor en una única máquina antes de ejecutar los programas en máquinas separadas.

Un resguardo (***stub***) es un método definido con un código mínimo que permite que el programa compile y ejecute inicialmente.

Los ejemplos cliente-servidor de *Daytime* son útiles como una introducción al desarrollo de un servicio de red. La simplicidad del protocolo es tal que la sesión de servicio involucra meramente una ronda de intercambio de mensajes entre los dos procesos. A continuación, se examinará un protocolo más complejo donde la noción de una sesión de servicio es más significativa.

5.4. SERVIDORES ORIENTADOS A CONEXIÓN Y SIN CONEXIÓN

El protocolo *Echo* de Internet [Postel, 2] es la base del bien conocido servicio de Internet con el mismo nombre. Este protocolo permite a un cliente simplemente mandar líneas de texto al servidor, de una en una, recibiendo del servidor un eco de cada una de ellas. En la práctica, el protocolo es útil ya que puede usarse el servidor *Echo* por defecto (que ejecuta en el puerto 7) de cualquier máquina en Internet como un servidor sustituto temporal cuando un ingeniero de software está desarrollando un cliente para otro protocolo. Para objetivos pedagógicos, el protocolo es interesante porque puede implicar múltiples rondas de intercambio de datos entre un cliente y un servidor. Una vez más, se examinarán dos implementaciones del servicio: una que utiliza un servidor sin conexión y otra que usa uno orientado a conexión.

Cliente-servidor *Echo* sin conexión

Las Figuras 5.20, 5.21 y 5.22 presentan una implementación del servicio *Echo* utilizando el API de *sockets* datagrama sin conexión.

El cliente *Echo*

La lógica de presentación del cliente se encapsula en la clase *ClienteEcho1* (*véase* la Figura 5.20), que proporciona la interfaz de usuario que solicita, en primer lugar, la información del servidor y después, utilizando un bucle, las líneas de texto que se van a mandar al servidor *Echo*. El envío de una cadena de texto y la recepción del eco devuelto se maneja mediante un método, *obtenerEco*, de la clase *ClienteEchoAuxiliar1* (Figura 5.21). La clase *ClienteEchoAuxiliar1* (Figura 5.21) proporciona la lógica de aplicación del cliente. Se crea una instancia de esta clase por cada proceso cliente (Figura 5.20), que mantiene la dirección de la máquina servidora, así como una referencia al *socket* utilizado por el cliente de IPC. El método *obtenerEco* utiliza el *socket* para mandar una línea al servidor y después recibir una línea de éste. Finalmente, el método *close* cierra el *socket*.

El servidor *Echo*

ServidorEcho1.java (Figura 5.22) combina la lógica de presentación y la lógica de aplicación para el servidor. En cada iteración de un bucle infinito, el servidor lee una línea del *socket* y después escribe la línea de vuelta al *socket*, dirigiendo la respuesta al emisor. Dado que no hay ninguna conexión involucrada, es posible que el servidor interaccione con diferentes clientes en iteraciones sucesivas, resultando sesiones de servicio concurrentes intercaladas. La Figura 5.23 ilustra un escenario en el que dos clientes concurrentes de esta implementación, A y B, intercalan sus interacciones con una instancia de *ServidorEcho1*.

Figura 5.20. *ClienteEcho1.java.*

```
 1 import java.io.*;
 2
 3  /**
 4  * Este módulo contiene la lógica de presentación de un cliente Echo.
 5  * @author M. L. Liu
 6  */
 7  public class ClienteEcho1 {
 8     static final String mensajeFin = ".";
 9     public static void main(String[ ] args) {
10       InputStreamReader is = new InputStreamReader(System.in);
11       BufferedReader br = new BufferedReader(is);
12       try {
13          System.out.println("Bienvenido al cliente Echo.\n" +
14             "¿Cuál es el nombre de la máquina servidora?");
15          String nombreMaquina = br.readLine();
16          if (nombreMaquina.length() == 0) // si el usuario no
                                             // introdujo un nombre
17             nombreMaquina = "localhost"; // usa el nombre de máquina
                                            // por defecto
18          System.out.println("Introduzca el nº puerto de la máquina
               servidora.");
19          String numPuerto = br.readLine( );
20          if (numPuerto.length() == 0)
```

(continúa)

```
21            numPuerto = "7"; // número de puerto por defecto
22          ClienteEchoAuxiliar1 auxiliar =
23            new ClienteEchoAuxiliar1(nombreMaquina, numPuerto);
24          boolean hecho = false;
25          String mensaje, eco;
26          while (!hecho) {
27            System.out.println("Introduzca una línea para recibir el eco"
28              + "eco del servidor, o un único punto para terminar.");
29            mensaje = br.readLine( );
30            if ((mensaje.trim()).equals(mensajeFin)){
31              hecho = true;
32              auxiliar.hecho( );
33            }
34            else {
35              eco = auxiliar.obtenerEco(mensaje);
36              System.out.println(eco);
37            }
38          } // fin de while
39        } // fin de  try
40        catch (Exception ex) {
41          ex.printStackTrace( );
42        }
43      } //fin de main
44    } // fin de class
```

Figura 5.21. *ClienteEchoAuxiliar1.java.*

```
1   import java.net.*;
2   import java.io.*;
3
4   /**
5   * Esta clase es un módulo que proporciona la lógica de aplicación
6   * para un cliente Echo utilizando un socket datagrama sin conexión.
7   * @author M. L. Liu
8   */
9   public class ClienteEchoAuxiliar1 {
10    private MiSocketDatagramaCliente miSocket;
11    private InetAddress maquinaServidora;
12    private int puertoServidor;
13
14    ClienteEchoAuxiliar1(String nombreMaquina, String numPuerto)
15      throws SocketException, UnknownHostException {
16      this.maquinaServidora = InetAddress.getByName(nombreMaquina);
17      this.puertoServidor = Integer.parseInt(numPuerto);
18      // instancia un socket datagrama para mandar
19      // y recibir datos
20      this.miSocket = new MiSocketDatagramaCliente();
```

(continúa)

```
21     }
22
23     public String obtenerEco(String mensaje)
24       throws SocketException, IOException {
25       String eco = "";
26       miSocket.enviaMensaje( maquinaServidora, puertoServidor, mensaje);
27         // ahora recibe el eco
28         eco = miSocket.recibeMensaje();
29         return eco;
30       } // fin de obtenerEco
31
32     public void hecho( ) throws SocketException {
33       miSocket.close( );
34     } // fin de hecho
35
36   } // fin de class
```

Figura 5.22. *ServidorEcho1.java.*

```
1   import java.io.*;
2
3
4   /**
5   * Este módulo contiene la lógica de aplicación de un servidor Echo
6   * que utiliza un socket datagrama sin conexión para la comunicación
7   * entre procesos.
8   * Se requiere un argumento de línea de mandato para el puerto del
      servidor.
9   * @author M. L. Liu
10  */
11  public class ServidorEcho1 {
12    public static void main(String[] args) {
13      int puertoServidor = 7; // número de puerto por defecto
14      if (args.length == 1 )
15        puertoServidor = Integer.parseInt(args[0]);
16      try {
17        // instancia un socket datagrama para mandar
18        // y recibir datos
19        MiSocketDatagramaServidor miSocket =
20          new MiSocketDatagramaServidor(puertoServidor);
21        System.out.println("Servidor Echo listo.");
22        while (true) { // bucle infinito
23          MensajeDatagrama peticion = miSocket.recibeMensajeYEmisor();
24          System.out.println("Petición recibida");
25          String mensaje = peticion.obtieneMensaje( );
26          System.out.println("mensaje recibido: "+ mensaje);
27          // Ahora manda el eco al solicitador
28          miSocket.enviaMensaje(peticion.obtieneDireccion( ),
```

(continúa)

```
29            peticion.obtienePuerto( ), mensaje);
30          } //fin de  while
31        } // fin de try
32        catch (Exception ex) {
33          ex.printStackTrace( );
34        }
35      } //fin de main
36    } // fin de class
```

Cliente-servidor *Echo* orientado a conexión

Las Figuras 5.24, 5.25 y 5.26 presentan una implementación de un cliente y un servidor del servicio *Echo* utilizando el API de *sockets* en modo *stream*. De nuevo, la implementación de la lógica de presentación (en *ClienteEcho2* y *ServidorEcho2*) es igual que **Echo1*. Sin embargo, la lógica de aplicación (en *ClienteEchoAuxiliar2* y *ServidorEcho2*), así como la lógica de servicio (en *MiSocketStream*) son diferentes (ya que se utiliza un *socket* en modo *stream* en lugar de un *socket* datagrama).

Nótese que en *ClienteEchoAuxiliar2* la conexión al servidor se realiza en el constructor, mientras que el método *obtenerEco* proporciona cada ronda de intercambio de mensajes. Se utiliza un método, *hecho*, para transmitir el mensaje de fin de sesión (uno que contiene sólo un punto) al servidor antes de que se cierre el *socket* del lado cliente.

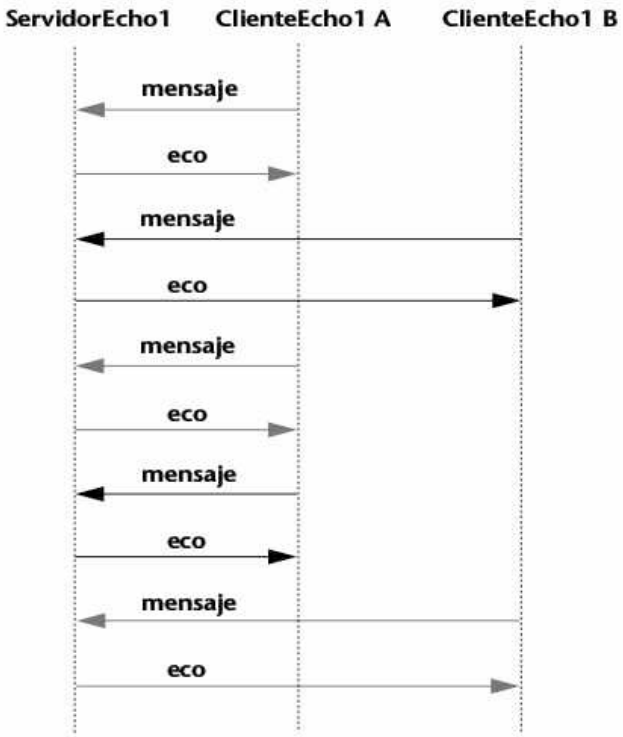


Figura 5.23. Un diagrama de secuencia que ilustra dos sesiones intercaladas con *ServidorEcho1*.

En *ServidorEcho2*, se crea en primer lugar un *socket* de conexión para aceptar las conexiones. Por cada conexión aceptada, el servidor recibe continuamente un mensaje y hace eco del mismo por el *socket* de datos asociado a la conexión, hasta que se recibe el mensaje de fin de sesión. Al final de la sesión, se cierra el *socket* de datos del cliente actual y se termina la conexión. El servidor entonces espera hasta aceptar otra conexión.

A lo largo de una sesión, el servidor mantiene su conexión con el cliente e intercambia datos con el mismo usando un *socket* de datos dedicado a ese cliente. Si otro cliente se conecta con el servidor mientras éste está todavía ocupado con una sesión, el cliente no será capaz de intercambiar datos con el servidor hasta que el servidor haya completado la sesión actual. La Figura 5.27 muestra el diagrama de secuencia de dos sesiones cuando el cliente 2 intenta conectar con el servidor mientras que se le está sirviendo al cliente 1. Nótese que no hay intercalado de sesiones en este caso.

Figura 5.24. *ClienteEcho2.java.*

```
1  import java.io.*;
2
4  /**
5  * Este módulo contiene la lógica de presentación de un cliente Echo.
6  * @author M. L. Liu
7  */
8  public class ClienteEcho2 {
9    static final String mensajeFin = ".";
9    public static void main(String[ ] args) {
10     InputStreamReader is = new InputStreamReader(System.in);
11     BufferedReader br = new BufferedReader(is);
12     try {
13       System.out.println("Bienvenido al cliente Echo.\n" +
14         "¿Cuál es el nombre de la máquina servidora?");
15       String nombreMaquina = br.readLine( );
16       if (nombreMaquina.length() == 0) // si el usuario no
                                          // introdujo un nombre
17         nombreMaquina = "localhost"; // utiliza nombre de máquina
                                        // por defecto
18       System.out.println("¿Cuál es el nº puerto de la máquina
         servidora?");
19       String numPuerto = br.readLine();
20       if (numPuerto.length() == 0)
21         numPuerto = "7"; // número de puerto por defecto
22       ClienteEchoAuxiliar2 auxiliar =
23         new ClienteEchoAuxiliar2(nombreMaquina, numPuerto);
24       boolean hecho = false;
25       String mensaje, eco;
26       while (!hecho) {
27         System.out.println("Introduzca una línea para recibir el eco "
28           + " del servidor, o un único punto para terminar.");
29         mensaje = br.readLine( );
30         if ((mensaje.trim( )).equals (".")){
31           hecho = true;
```

(continúa)

```
32              auxiliar.hecho( );
33            }
34            else {
35              eco = auxiliar.obtenerEco(mensaje);
36              System.out.println(eco);
37            }
38          } // fin de while
39        } // fin de try
40        catch (Exception ex) {
41          ex.printStackTrace( );
42        } // fin de catch
43      } // fin de main
44    } // fin de class
```

Figura 5.25. *ClienteEchoAuxiliar2.java.*

```
1
2   import java.net.*;
3   import java.io.*;
4
5   /**
6   * Esta clase es un módulo que proporciona la lógica de aplicación
7   * para un cliente Echo utilizando un socket en modo stream.
8   * @author M. L. Liu
9   */
10  public class ClienteEchoAuxiliar2 {
11
12    static final String mensajeFin = ".";
13    private MiSocketStream miSocket;
14    private InetAddress maquinaServidora;
15    private int puertoServidor;
16
17    ClienteEchoAuxiliar2(String nombreMaquina,
18      String numPuerto) throws SocketException,
19      UnknownHostException, IOException {
20
21      this.maquinaServidora = InetAddress.getByName(nombreMaquina);
22      this.puertoServidor = Integer.parseInt(numPuerto);
23      // instancia un socket en modo stream y espera por una
        // conexión.
24      this.miSocket = new MiSocketStream(nombreMaquina,
25        this.puertoServidor);
26  /**/ System.out.println("Petición de conexión hecha");
27    } // fin de constructor
28
29    public String obtenerEco( String mensaje) throws SocketException,
30      IOException {
31      String eco = "";
```

(continúa)

```
32        miSocket.enviaMensaje( mensaje);
33        // ahora recibe el eco
34        eco = miSocket.recibeMensaje();
35        return eco;
36     } //fin de obtenerEco
37
38     public void hecho( ) throws SocketException,
39        IOException{
40        miSocket.enviaMensaje(mensajeFin);
41        miSocket.close( );
42     } // fin de hecho
43  } // fin de class
```

Figura 5.26. *ServidorEcho2.java.*

```
 1  import java.io.*;
 2  import java.net.*;
 3
 4  /**
 5  * Este módulo contiene la lógica de aplicación de un servidor Echo
 6  * que utiliza un socket en modo stream para comunicarse entre procesos.
 7  * Se requiere un argumento de línea de mandato para el puerto del
      servidor.
 8  * @author M. L. Liu
 9  */
10  public class ServidorEcho2 {
11     static final String mensajeFin = ".";
12     public static void main(String[] args) {
13        int puertoServidor = 7; // puerto por defecto
14        String mensaje;
15
16        if (args.length == 1 )
19           puertoServidor = Integer.parseInt(args[0]);
20        try {
21           // instancia un socket stream para aceptar
22           // las conexiones
23           ServerSocket miSocketConexion =
24              new ServerSocket(puertoServidor);
25    /**/   System.out.println("Servidor Echo listo.");
26           while (true) { // bucle infinito
27              // espera para aceptar una conexión
28    /**/      System.out.println("Espera una conexión.");
29              MiSocketStream miSocketDatos = new MiSocketStream
30                 (miSocketConexion.accept( ));
31    /**/      System.out.println("conexión aceptada");
32              boolean hecho = false;
```

(continúa)

```
33            while (!hecho) {
34              mensaje = miSocketDatos.recibeMensaje( );
35   /**/       System.out.println("mensaje recibida: "+ mensaje);
36              if ((mensaje.trim()).equals (mensajeFin)){
37                // la sesión se termina, cierra el socket de datos.
38   /**/         System.out.println("Final de la sesión.");
39                miSocketDatos.close( );
40                hecho = true;
41              } // fin de if
42              else {
43                // Ahora manda el eco al solicitante
44                miSocketDatos.enviaMensaje(mensaje);
45              } // fin de else
46            } // fin de while !hecho
47          } // fin de while infinito
48        } // fin de try
49        catch (Exception ex) {
50          ex.printStackTrace( );
51        } // fin de catch
52      } // fin de main
51    } // fin de class
```

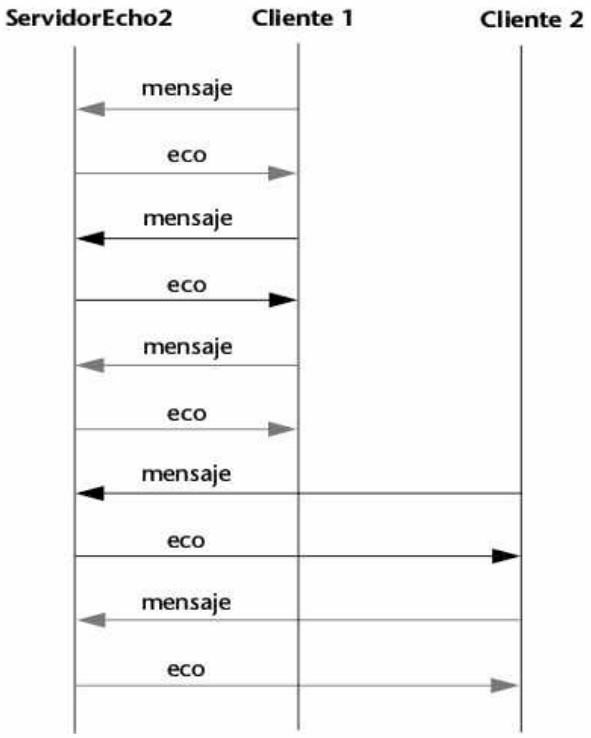


Figura 5.27. *ServidorEcho2* no permitirá sesiones intercaladas.

5.5. SERVIDOR ITERATIVO Y SERVIDOR CONCURRENTE

Como el lector habrá podido apreciar, con un servidor orientado a conexión tal como *ServidorDaytime2* y *ServidorEcho2*, no hay solapamiento de sesiones de cliente, ya que el servidor se limita a intercambiar datos con un cliente cuya conexión ha aceptado (pero no ha desconectado). A este tipo de servidor se le denomina **servidor iterativo**, ya que sirve a un cliente cada vez. Cuando se solicita un servicio a un servidor iterativo popular, un cliente se bloqueará hasta que se sirvan todos los clientes precedentes. El resultado puede ser un tiempo de bloqueo significativo si las sesiones de servicio son largas, tal como en el caso de un protocolo de transferencia de ficheros. Supóngase que cada sesión puede durar t unidades de tiempo, y que en un tiempo dado, n clientes han pedido conexiones. Obviando otros retrasos en aras de la simplicidad, el próximo cliente que solicita conexión puede contar con que estará bloqueado durante al menos $n * t$ unidades de tiempo. Si t es largo, el retraso será inaceptable.

Como se examinó en el Capítulo 1, en un sistema con un único procesador, la ejecución concurrente se lleva a cabo compartiendo el tiempo (*time-sharing*) del procesador y, por tanto, la concurrencia no es real.

La solución a este problema es introducir concurrencia en el servidor, dando lugar a un **servidor concurrente**. Un servidor concurrente es capaz de gestionar múltiples sesiones de cliente en paralelo. Para proporcionar un servidor concurrente, se pueden utilizar hilos o, alternativamente, operaciones de IPC asíncronas. (La programación concurrente con hilos se trató en el Capítulo 1, y las operaciones de IPC asíncronas en el Capítulo 2). La utilización de hilos es la técnica convencional y tiene la virtud de ser relativamente sencilla. Sin embargo, los requisitos de escala y de rendimiento de las aplicaciones actuales han hecho necesaria, en algunos casos, la utilización de una IPC asíncrona.

En esta presentación se tratará la técnica de utilizar hilos para construir un servidor concurrente. De manera similar a los servidores iterativos, un servidor concurrente utiliza un único *socket* de conexión para escuchar las conexiones. Sin embargo, un servidor concurrente crea un nuevo hilo para aceptar cada conexión y dirigir una sesión de servicio con el cliente conectado; de manera que el hilo termina cuando concluye la sesión.

Las Figuras 5.28 y 5.29 presentan el servidor concurrente y la clase que define el hilo que utiliza, respectivamente. El método *run* de la clase que define el hilo lleva a cabo la lógica de una sesión de cliente. Nótese que no se necesita ningún cambio en el código del lado del cliente: puede utilizarse *ClienteEcho2* para acceder a *ServidorEcho3* sin ningún cambio.

La Figura 5.30 muestra el diagrama de secuencia de dos sesiones concurrentes. Se estima de interés que el lector compare este diagrama de secuencia con los dos presentados anteriormente (Figuras 5.23 y 5.27). Con un servidor concurrente, un cliente no tendrá que esperar mucho tiempo hasta que se acepte su conexión; el único retraso que el cliente experimentará será el resultante de su propia sesión de servicio.

Figura 5.28. *ServidorEcho3.java.*

```
1  import java.io.*;
2  import java.net.*;
3
4
5  /**
```

(continúa)

```
 6  * Este módulo contiene la lógica de aplicación de un servidor Echo
 7  * que utiliza un socket en modo stream para comunicación entre procesos.
 8  * A diferencia de ServidorEcho2, los clientes se sirven
       concurrentemente.
 9  * Se requiere un argumento de línea de mandato para el puerto del
       servidor.
10  * @author M. L. Liu
11  */
12  public class ServidorEcho3 {
13    public static void main(String[] args) {
14      int puertoServidor = 7; // Puerto por defecto
15      String mensaje;
16
17      if (args.length == 1 )
18        puertoServidor = Integer.parseInt(args[0]);
19      try {
20        // instancia un socket stream para aceptar
21        // las conexiones
22        ServerSocket miSocketConexion =
23          new ServerSocket(puertoServidor);
24   /**/  System.out.println("Servidor Echo listo.");
25        while (true) { // bucle infinito
26          // espera para aceptar una conexión
27   /**/    System.out.println("Espera una conexión.");
28          MiSocketStream miSocketDatos = new MiSocketStream
29            (miSocketConexion.accept( ));
30   /**/    System.out.println("conexión aceptada");
31          // Arranca un hilo para manejar la sesión de cliente
32          Thread elHilo =
33            new Thread(new HiloServidorEcho(miSocketDatos));
34          elHilo.start();
35          // y continúa con el siguiente cliente
36        } // fin de while infinito
37      } // fin de try
38      catch (Exception ex) {
39        ex.printStackTrace( );
40      } // fin de catch
41    } // fin de main
42  } // fin de class
```

Figura 5.29. *HiloServidorEcho.java.*

```
1
2  import java.io.*;
3  /**
4  * Este módulo está diseñado para usarse con un servidor Echo
       concurrente.
5  * Su método run lleva a cabo la lógica de una sesión de cliente.
```

(continúa)

```
 6  * @author M. L. Liu
 7  */
 8  class HiloServidorEcho implements Runnable {
 9    static final String mensajeFin = ".";
10    MiSocketStream miSocketDatos;
11
12    HiloServidorEcho (MiSocketStream miSocketDatos) {
13      this.miSocketDatos = miSocketDatos;
14    } // fin de constructor
15
16    public void run( ) {
17      boolean hecho = false;
18      String mensaje;
19      try {
20        while (!hecho) {
21          mensaje = miSocketDatos.recibeMensaje( );
22   /**/     System.out.println("mensaje recibido: "+ mensaje);
23          if ((mensaje.trim()).equals (mensajeFin)){
24            // se termina la sesión; cierra el socket de datos
25   /**/       System.out.println("Final de la sesión.");
26            miSocketDatos.close( );
27            hecho = true;
28          } //fin de if
29          else {
30            // Ahora manda el eco al solicitante
31            miSocketDatos.enviaMensaje(mensaje);
32          } // fin de else
33        } // fin de while !hecho
34      } // fin de try
35      catch (Exception ex) {
36        System.out.println("Excepción capturada en hilo: " + ex);
37      } // fin de catch
38    } // fin de run
39  } // fin de class
```

Los servidores concurrentes son los que se usan normalmente en los servicios de red modernos. Aunque los servidores iterativos son sólo aceptables para los protocolos más simples, el estudio de servidores de este tipo es todavía una buena experiencia pedagógica.

5.6. SERVIDORES CON ESTADO

Tanto el protocolo *Daytime* como el *Echo* pertenecen a una categoría de protocolos conocida como **protocolos sin estado**, en contraste con los **protocolos con estado**. Un protocolo es *sin estado* cuando no necesita mantener ninguna información de estado en el servidor. Tal es el caso del protocolo *Daytime*, donde el servidor simplemente envía a cada cliente una marca de tiempo obtenida del sistema, así como del protocolo *Echo*, que sólo requiere que el servidor haga eco de cada mensaje que re-

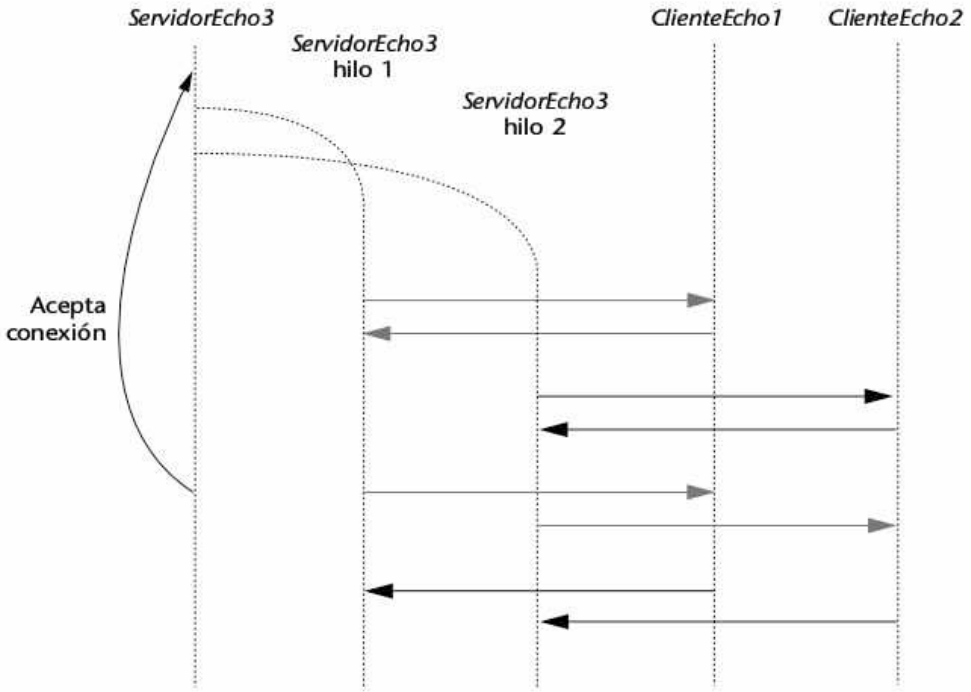


Figura 5.30. *ServidorEcho3* permite sesiones de cliente concurrentes.

cibe. En ambos protocolos, la tarea realizada por el servidor es independiente de su estado, de aspectos tales como, por ejemplo, cuánto tiempo lleva en servicio el servidor o qué cliente está siendo servido.

Un servidor sin estado es aquel que proporciona un servicio siguiendo un protocolo sin estado y por tanto no tiene que mantener ninguna información de estado, como en el caso del servidor *Daytime* o del servidor *Echo*. Un **servidor con estado**, como su nombre implica, es uno que debe mantener alguna información de estado para proporcionar su servicio.

¿Qué significa exactamente información de estado? Hay dos tipos: información de estado global e información de estado de sesión.

Información de estado global

El servidor mantiene este tipo de información para todos los clientes durante la vida del servidor. Por ejemplo, supóngase un protocolo, cuyo nombre sea *contador* (no se trata de un protocolo estándar de Internet), que requiere que un servidor mantenga un contador, iniciado a 0. Cada vez que un cliente contacta con el servidor, éste incrementa el contador en 1 y envía el valor actual del contador al cliente. Para proporcionar este servicio, el servidor debe tener el valor del contador situado en algún tipo de almacenamiento de donde se pueda recuperar y actualizar durante la ejecución del servicio. Para un contador sencillo, el almacenamiento puede implementarse utilizando una variable de un tipo de datos primitivo de Java. Para una información de estado más compleja, el almacenamiento puede implementarse utilizando un fichero, una base de datos u otros mecanismos. La Figura 5.31 muestra el código de un servidor que implementa el protocolo *contador*. Nótese que se utiliza una variable estática para

mantener el contador cuya actualización se sincroniza para asegurar exclusión mutua. Las Figuras 5.32 y 5.33 muestran el programa cliente.

Figura 5.31. *ServidorContador1.java.*

```
1  import java.io.*;
2
3  /**
4  * Este módulo contiene la lógica de aplicación de un servidor
     Contador
5  * que utiliza un socket datagrama para la comunicación entre
     procesos.
6  * Se requiere un argumento de línea de mandato para el puerto del
     servidor.
7  * @author M. L. Liu
8  */
9  public class ServidorContador1 {
10
11   /* información de estado */
12   static int contador = 0;
13
14   public static void main(String[] args) {
15     int puertoServidor = 12345; // puerto por defecto
16     if (args.length == 1 )
17       puertoServidor = Integer.parseInt(args[0]);
18     try {
19       // instancia un socket datagrama para enviar
20       // y recibir
21       MiSocketDatagramaServidor miSocket =
22         new MiSocketDatagramaServidor(puertoServidor);
23  /**/  System.out.println("Servidor Contador listo.");
24       while (true) { // bucle infinito
25         MensajeDatagrama peticion =
26           miSocket.recibeMensajeYEmisor();
27         System.out.println("Petición recibida");
28         // no es importante el mensaje recibido; es la dirección
29         // del emisor lo que se necesita para responder.
30         // Incrementa el contador, después manda su valor al cliente
31         incremento();
32  /**/   System.out.println("contador enviado "+ contador);
33         // Ahora manda la respuesta al solicitante
34         miSocket.enviaMensaje(peticion.obtieneDireccion(),
35           peticion.obtienePuerto(), String.valueOf(contador));
36       } // fin de while
37     } // fin de try
38     catch (Exception ex) {
39       ex.printStackTrace( );
40     } // fin de catch
41   } // fin de main
42
```

(continúa)

```
43    static private synchronized void incremento(){
44      contador++;
45    }
46
47  } // fin de class
```

Figura 5.32. *ClienteContador1.java.*

```
 1  import java.io.*;
 2
 3
 4  /**
 5  * Este módulo contiene la lógica de presentación de un cliente
      Contador.
 6  * @author M. L. Liu
 7  */
 8  public class ClienteContador1 {
 9    public static void main(String[] args) {
10      InputStreamReader is = new InputStreamReader(System.in);
11      BufferedReader br = new BufferedReader(is);
12      try {
13        System.out.println("Bienvenido al cliente Contador.\n" +
14          "¿Cuál es el nombre de la máquina servidora?");
15        String nombreMaquina = br.readLine();
16        if (nombreMaquina.length() == 0) // si el usuario no
                                           // introdujo un nombre
17          nombreMaquina = "localhost"; // utiliza nombre de máquina
                                         // por defecto
18        System.out.println("¿Cuál es el nº de puerto de la máquina
            servidora?");
19        String numPuerto = br.readLine();
20        if (numPuerto.length() == 0)
21          numPuerto = "12345"; // número de puerto por defecto
22        System.out.println("Contador recibido del servidor: "
23          + ClienteContadorAuxiliar1.obtenerContador(nombreMaquina,
            numPuerto));
24      } // fin de try
25      catch (Exception ex) {
26        ex.printStackTrace( );
27      }
28    } //fin de main
29  } // fin de class
```

Figura 5.33. *ClienteContadorAuxiliar1.java.*

```
1
2  import java.net.*;
```

(continúa)

```
 3
 4  /**
 5  * Esta clase es un módulo que proporciona la lógica de aplicación
 6  * para un cliente Contador.
 7  * @author M. L. Liu
 8  */
 9  public class ClienteContadorAuxiliar1 {
10
11    public static int obtenerContador(String nombreMaquina,
12         String numPuerto)
13    {
14       int contador = 0;
15       String mensaje = "";
16       try {
17          InetAddress maquinaServidora =
18             InetAddress.getByName(nombreMaquina);
19          int puertoServidor = Integer.parseInt(numPuerto);
20          // instancia un socket datagrama para enviar
21          // y recibir datos.
22          MiSocketDatagramaCliente miSocket = new
            MiSocketDatagramaCliente();
23          miSocket.enviaMensaje( maquinaServidora, puertoServidor, "");
24          // ahora recibe el valor del contador
25          mensaje = miSocket.recibeMensaje();
26    /**/  System.out.println("Mensaje recibido: " + mensaje);
27          contador = Integer.parseInt(mensaje.trim());
28          miSocket.close( );
29       } // fin de try
30       catch (Exception ex) {
31          ex.printStackTrace( );
32       } // fin de catch
33       return contador;
34    } // fin de main
35  } // fin de class
```

Información de estado de sesión

Algunos protocolos o aplicaciones, tienen la necesidad de mantener información específica para una sesión de cliente.

Considérese un servicio de red tal como *ftp*. Un fichero es típicamente transferido en bloques, requiriéndose varias rondas de intercambios de datos para completar la transferencia del fichero. La información de estado de cada sesión incluye:

1. El nombre del fichero con el que se está trabajando.
2. El número del bloque actual.
3. La acción (traer, llevar, etc.) que se está realizando sobre el fichero.

Hay al menos dos esquemas para mantener los datos de estado de sesión.

Servidor sin estado. En este esquema, el cliente puede mantener la información de estado de sesión de manera que cada petición contenga la información de estado de sesión, permitiendo al servidor procesar cada petición de acuerdo con los datos de estado enviados en la petición. El diálogo de una sesión se llevará a cabo aproximadamente de la siguiente forma:

Cliente: Por favor, envíeme el bloque 0 del fichero *arch* del directorio *dir*.

Servidor: Vale. Aquí tiene ese bloque del fichero.

Cliente: Por favor, envíeme el bloque 1 del fichero *arch* del directorio *dir*.

Servidor: Vale. Aquí tiene ese bloque del fichero.

...

Cliente: Por favor, envíeme el bloque *n* del fichero *arch* del directorio *dir*.

Servidor: Vale. Aquí tiene ese bloque del fichero.

Al requerir que el cliente mantenga los datos de sesión, este esquema permite al servidor procesar cada petición de la misma manera, y por eso reduce la complejidad de la lógica de aplicación. A tal tipo de servidor se le conoce como **servidor sin estado**.

Servidor con estado. En el segundo esquema, el servidor debe mantener la información del estado de la sesión, en cuyo caso el diálogo de una sesión se llevará a cabo aproximadamente de la siguiente forma:

Cliente: Por favor, envíeme el fichero *arch* del directorio *dir*.

Servidor: Vale. Aquí tiene el bloque 0 del fichero *arch*.

Cliente: Lo tengo.

Servidor: Vale. Aquí tiene el bloque 1 del fichero *arch*.

Cliente: Lo tengo.

...

Servidor: Vale. Aquí tiene el bloque *n* del fichero *arch*.

Cliente: Lo tengo.

Con este esquema, el servidor controla el progreso de la sesión manteniendo el estado de la misma. A este tipo de servidor se le conoce como un **servidor con estado**. La Figura 5.34 ilustra la diferencia entre un servidor con estado y uno sin estado.

Los servidores con estado son más complejos de diseñar e implementar. Además de la lógica requerida para mantener los datos del estado, se debe prever la necesidad de salvaguardar la información de estado en el caso de que se interrumpa una sesión. Si la máquina donde ejecuta un servidor con estado falla temporalmente en medio de una sesión, es importante que la sesión interrumpida se reanude en el estado correcto. En caso contrario, es posible que, en el ejemplo planteado, el cliente reciba un bloque erróneo del fichero después de que la sesión se recupere del fallo.

Otro ejemplo de protocolo con estado corresponde con una aplicación del tipo «cesta de la compra». Cada sesión debe mantener los datos de estado que hacen el seguimiento de la identidad del comprador y de los contenidos acumulados en su cesta de compras.

En la implementación real, un servidor puede ser sin estado, con estado, o híbrido. En este último caso, los datos de estado pueden distribuirse entre el servidor y el cliente. El tipo de servidor que se utiliza es una cuestión de diseño.

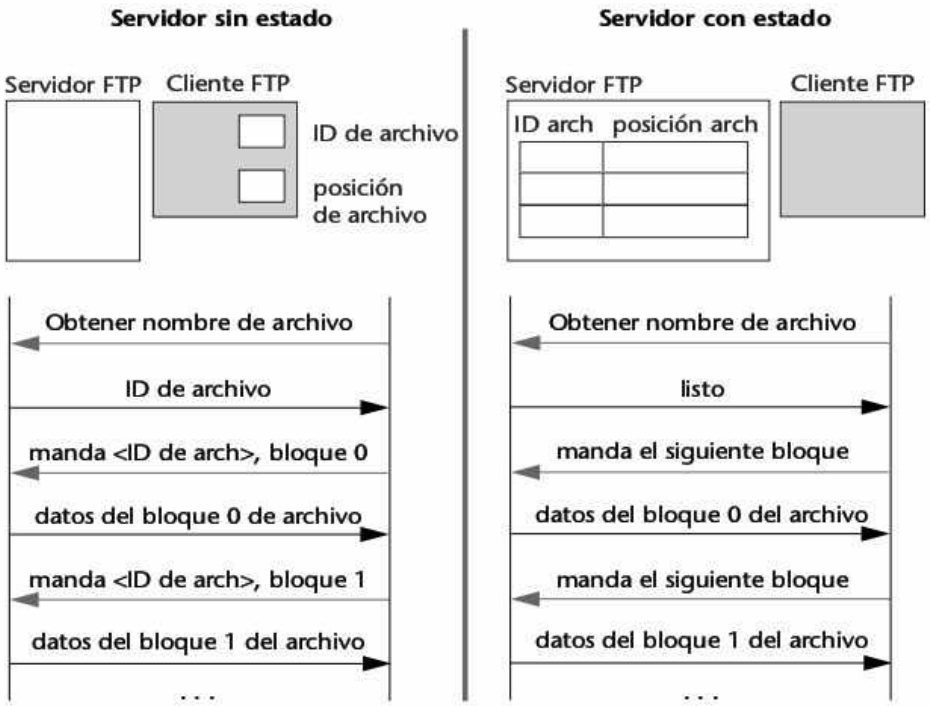


Figura 5.34. La diferencia entre un servidor sin estado y otro con estado.

La Sociedad de Internet

Comunicado de prensa (extracto).
La «Estrella Polar» que definió Internet.
Reimpreso con el permiso de la sociedad de Internet (http://www.isoc.org/internet/).

Reston, VA, 19 de octubre de 1998. La comunidad de Internet llora la pérdida de un líder y del primer miembro particular de la Sociedad de Internet. Jonathan B. Postel falleció el viernes 16 de octubre. Durante treinta años, Postel sirvió a todos los usuarios de Internet en una variedad de papeles fundamentales, aunque pocos fuera de la comunidad técnica conocen su nombre. Mucho antes de que aparecieran libros sobre Internet, había sólo una serie de documentos técnicos conocidos como Petición de comentarios o RFC (*Request for Comments*). Jon editó y organizó este material que ayudó a establecer los primeros estándares de Internet.

«Jon ha sido nuestra estrella polar durante décadas, iluminando brillante e incesantemente, proporcionando confort y un sentido de seguridad mientras todo lo demás estaba cambiando», dijo Vint Cerf, el presidente actual del comité de la Sociedad de Internet. «Él fue el Boswell de Internet y su conciencia técnica. Su pérdida será profundamente sentida, no sólo por su capacidad, sino porque la comunidad ha perdido un querido y muy apreciado amigo».

Postel comenzó su carrera en la red en 1969 mientras se graduaba en UCLA trabajando en el actualmente famoso proyecto ARPANET (precursor de Internet) como un ayudante investigador del profesor Leonard Kleinrock quien dijo: «Las contribuciones fundamentales de Jon en nuestro trabajo durante aquellos críticos primeros días de ARPANET todavía no son suficientemente reconocidas. Mientras estábamos abriendo nuevos horizontes con el naci-

miento de Internet en 1969, recuerdo a Jon como un joven programador de nuestro equipo, brillante y profundamente dedicado. En aquella época había fronteras y Jon era verdaderamente un pionero con visión de futuro. En esos días se convirtió en el editor de la serie Petición de comentarios, que aún continúa en la actualidad. La dedicación de Jon al crecimiento y al bienestar de Internet continuó desde aquellos tiempos embriagadores durante el resto de su vida. Y por todo esto, él no buscaba ni reconocimiento ni alabanza. La muerte de Jon es una trágica pérdida, que será sentida por todos aquellos cuyas vidas tienen que ver con Internet, pero especialmente por aquellos de nosotros que recorrimos el camino con este hombre gentil y tranquilo durante muchos, muchos años».

Durante mucho tiempo, Postel fue la Autoridad de Números Asignados de Internet (IANA, *Internet Assigned Numbers Authority*) que supervisa la reserva y asignación de los nombres de dominio y direcciones de Internet. Finalmente, la tarea creció tanto que hubo que formar un pequeño equipo de personas para ayudar en este trabajo. Su temprano reconocimiento de la importancia de la documentación cuidadosa parece verdaderamente clarividente con la retrospectiva de hoy en día. Todo el trabajo técnico en ARPANET y más tarde en Internet, así como la teoría y la práctica requerida por la administración de nombres y direcciones, es recogida ampliamente en la actualidad por los historiadores debido a la dedicación de Jon. En la época en que la red era sólo un experimento estuvo plenamente comprometido en proporcionar un refugio seguro y a salvo para la información que hace posible que funcione Internet.

Él fue directamente responsable de la gestión del nombre de dominio .US. Además, sirvió como un miembro del comité de Arquitectura de Internet (*Internet Architecture Board*) desde su creación en 1983, continuando hasta el presente. Postel desempeñó muchos otros papeles incluyendo el ser parte fundamental en la fundación de la Sociedad de Internet. También fundó los servicios de red Los Nettos en el área de Los Angeles.

Postel ofreció algo que difícilmente se puede encontrar en cualquier momento: colaboración continua, competente y discreta. Consideró sus responsabilidades como una especie de responsabilidad pública. Nunca recibió ningún beneficio personal del gran negocio de Internet, prefiriendo permanecer fuera del frenesí del negocio, de las Ofertas Públicas Iniciales (IPO, *Initial Public Offerings*) y de toda la parafernalia de Wall Street.

Stephen D. Crocker, amigo y compañero de universidad de Postel, lideró el desarrollo de los protocolos de ARPANET entre máquinas finales. Crocker comenzó la serie RFC y Jon instantáneamente se ofreció voluntariamente a editarlas. «Lo impensable ha sucedido. Todos hemos perdido a un gran amigo y a un pilar principal de apoyo y cordura en nuestro mundo peculiar y surrealista», dijo Crocker. «Para aquellos de nosotros involucrados en la red humana de ingenieros que diseñan y desarrollan Internet, fue una persona fundamental que hizo que la red siguiera funcionando. Trabajó incansable y desinteresadamente. Él siempre estaba allí».

«No puedo creer que haya fallecido, Jon era un héroe para mí y para muchos otros en Internet. Fue un gran mediador: siempre sonriendo y preparado para considerar una nueva idea, sin ningún plan particular excepto promover las grandezas de Internet alrededor del mundo», dijo Jean Amour Polly, anterior miembro directivo (*trustee*) de la Sociedad de Internet.

Mientras trabajaba en ARPANET en UCLA, Postel siguió un programa de investigación de doctorado bajo la dirección de los profesores David Faber de UC Irvine y Gerald Estrin de UCLA. El profesor Faber recuerda: «Jon era mi segundo estudiante de doctorado. Yo fui el principal consejero de su tesis junto con Jerry Estrin y recuerdo con cariño los meses que trabajamos hombro con hombro con Jon mientras su impaciente mente desarrollaba las ideas en las que se basó su tesis pionera que fundó el área de la verificación de protocolos. Dado que yo estaba en UC Irvine y Jon en UCLA solíamos quedar por la mañana, antes de mi paseo hasta UCI, en una crepería en Santa Mónica para desayunar y trabajar duro desarrollando la tesis. Yo adquirí un gran respeto por Jon, además de 10 libras de peso».

Jerry Estrin recuerda a Jon Postel como un ser humano sin egoísmo, gentil y maravillosamente encantador que se preocupaba realmente de la gente y sus contribuciones personales. «En los 70, no tuvo miedo de aplicar un modelo de grafos novedoso para verificar los protocolos complejos de ARPANET. Nunca olvidaré a Jon dirigiéndose a mí durante un seminario de graduación y preguntándome amablemente si podía abstenerme de fumar en pipa durante la clase. Mostraba la misma preocupación con respecto a los efectos tóxicos del tabaco como sobre el potencial impacto positivo de las redes de computadores».

Postel trabajó en muchos puestos durante su largo contacto con Internet. Trabajó con la leyenda industrial Doug Engelbart en SRI Internacional en Menlo Park, CA, donde proporcionó un importante soporte al oNLine System, en muchos aspectos un predecesor de la World Wide Web, incluyendo la característica del hiperenlace que a todos resulta tan familiar en la actualidad. Se trasladó al área de Washington para dar apoyo a la Agencia de Proyectos de Investigación Avanzada (ARPA, *Advanced Research Projects Agency*) durante un tiempo y después, fue al *Information Sciences Institute* de USC donde se convirtió en una estrella permanente en el paraíso de Internet, sirviendo de guía a todos los internautas cuando exploraban el extenso océano en el que se ha convertido Internet.

Postel fue elegido para el comité directivo de la Sociedad de Internet en 1993 y reelegido para un segundo periodo de tres años en 1996. En los dos últimos años, trabajó sin descanso ayudando a facilitar la migración tanto del IANA, financiado por el gobierno estadounidense, como del sistema general de gestión de nombres de dominio a una empresa internacional del sector privado sin ánimo de lucro. Poco antes de su fallecimiento, se creó la Corporación de Internet para Nombres y Números Asignados (ICANN, *Internet Corporation for Assigned Names and Numbers*), que se ha propuesto como sucesora del sistema financiado por el gobierno estadounidense que ha dado servicio durante cerca de 30 años a la comunidad de Internet.

En el sitio web de la Sociedad de Internet (ISOC) se pueden encontrar tributos a Jon Postel de gente que le conocía, ya sea personalmente o a través de su trabajo, http://www.isoc.org/postel.

RESUMEN

Este capítulo ha presentado el paradigma cliente-servidor de computación distribuida. Los temas tratados incluyen:

- La diferencia entre la arquitectura de sistema cliente-servidor y el paradigma de computación distribuida cliente-servidor.
- Una definición del paradigma y una explicación de por qué motivo se ha utilizado ampliamente en los servicios y aplicaciones de red.
- Los temas de sesiones de servicio, protocolos, localización de servicios, comunicaciones entre procesos, representación de datos y sincronización de eventos en el contexto del paradigma cliente-servidor.
- La arquitectura de software de tres niveles de las aplicaciones cliente-servidor: lógica de presentación, lógica de aplicación y lógica de servicios.
- Servidor sin conexión frente a servidor orientado a conexión.
- Servidor iterativo frente a servidor concurrente y el efecto que tiene sobre una sesión de cliente.
- Servidor con estado frente a sin estado.

- En el caso de un servidor con estado, la información de estado global frente a la información de estado de sesión.

EJERCICIOS

1. En el contexto de la computación distribuida, describa el paradigma cliente-servidor. ¿Por qué es especialmente apropiado este paradigma para los servicios de red?
2. Describa la arquitectura de software de tres niveles para el software cliente-servidor. Explique brevemente las funcionalidades de cada nivel en cada lado. ¿Por qué es ventajoso encapsular la lógica de distintos niveles en módulos de software separados?
3. Este ejercicio trata con *ServidorDaytime1* y *ClienteDaytime1*, que usan *sockets* datagrama sin conexión. Durante este conjunto de ejercicios, es suficiente con que ejecute los procesos cliente y servidor en una sola máquina y especifique el nombre de la máquina como localhost.
   a. ¿Por qué es necesario que el servidor use el método *recibeMensajeYEmisor* para aceptar las peticiones de cliente en vez del método *recibeMensaje*?
   b. Compile **Daytime1.java («javac *Daytime1.java.»)*. Después ejecute los programas:
      i. Comenzando en primer lugar por el cliente y, a continuación, el servidor (no olvide especificar un número de puerto como argumento de línea de mandato). ¿Qué sucede? Descríbalo y explíquelo.
      ii. Comenzando en primer lugar por el servidor (no olvide especificar un número de puerto como argumento de línea de mandato y a continuación, el cliente. ¿Qué sucede? Descríbalo y explíquelo.
   c. Modifique la constante *MAX_LON* en *MiSocketDatagramaCliente.java* para que valga 10. Vuelva a compilar y ejecute de nuevo los programas, comenzando primero por el servidor. Observe el mensaje recibido por el cliente. Describa y explique la salida.
   d. Restaure el valor original de *MAX_LON*. Vuelva a compilar y a ejecutar los programas, comenzado primero por el servidor. Arranque otro cliente, preferiblemente en una máquina diferente. Describa y explique la salida. Dibuje un diagrama de tiempo-eventos que describa la secuencia de eventos de las interacciones entre el servidor y los dos clientes.
4. Este ejercicio incluye *ServidorDaytime2* y *ClienteDaytime2*, que utilizan *sockets stream*.
   a. Describa con sus propias palabras la diferencia entre este conjunto de clases y las clases de **Daytime1*.
   b. Compile **Daytime2.java* (*javac *Daytime2.java)*. Después ejecute los programas:
      i. Comenzando primero por el cliente y a continuación, el servidor (no olvide especificar un número de puerto como un argumento de línea de mandato). ¿Qué sucede? Descríbalo y explíquelo.

ii. Comenzando primero por el servidor (no olvide especificar un número de puerto como argumento de línea de mandato) y a continuación, el cliente. ¿Qué sucede? Descríbalo y explíquelo.

c. Para experimentar con el efecto de una conexión, añada un retraso (utilizando *Thread.sleep(3000)* en *ServidorDaytime2.java* después de aceptar una conexión y antes de obtener una marca de tiempo del sistema). Añadir el retraso tiene el efecto de aumentar artificialmente 3 segundos la duración de cada sesión de servicio. Recompile y arranque el servidor, después arranque dos clientes en dos pantallas separadas de su sistema. ¿Cuánto tiempo tardó el segundo cliente en hacer una conexión con el servidor? Descríbalo y explíquelo, teniendo en cuenta que el servidor es iterativo.

5. Asumiendo que la implementación es correcta, el lector debería ser capaz de utilizar cualquiera de los dos programas *ClienteDaytime* de los ejemplos presentados (Figuras 5.6 y 5.14, respectivamente) para obtener una marca de tiempo de cualquier máquina de Internet que proporcione el servicio *Daytime* estándar en el puerto 13 de TCP/UDP. Trate de hacerlo con una máquina conocida, utilizando tanto *ClienteDaytime1* como *ClienteDaytime2*. Tenga en cuenta que en algunos sistemas se rechazará el acceso al servicio *Daytime* en el puerto 13 por razones de seguridad. Describa el experimento y la salida generada, teniendo en cuenta que la mayoría de los servidores se implementaron hace tiempo utilizando el lenguaje C.

6. Este ejercicio utiliza *ServidorEcho1* y *ClienteEcho1*, que utilizan *sockets* datagrama para el servicio *Echo*.

   Compile **Echo1.java (javac *Echo1.java)* y a continuación, realice las siguientes operaciones:

   a. Ejecute los programas comenzando por el servidor (no olvide especificar el número de puerto como argumento de línea de mandato) y a continuación, el cliente. Realice una sesión y observe los mensajes de diagnóstico visualizados en ambos lados. Describa sus observaciones sobre la secuencia de eventos.

   b. Con el servidor en ejecución, arranque dos clientes en ventanas separadas. ¿Se pueden realizar las sesiones de los dos clientes en paralelo? Describa y explique sus observaciones. ¿Están de acuerdo con el diagrama de secuencia de la Figura 5.23?

   c. Puede sorprenderse de lo que sucede cuando un cliente manda datos a un servidor sin conexión que ya está ocupado sirviendo a otro cliente. Realice el siguiente experimento: añada un retraso de 10 segundos (10.000 milisegundos) en el servidor antes de que se envíe el eco. A continuación, repita el apartado b. Describa y explique sus observaciones. ¿El servidor recibe el dato del segundo cliente?

7. Este ejercicio trata con *ServidorEcho2* y *ClienteEcho2*. Recuerde que *ServidorEcho2* es un servidor orientado a conexión e iterativo.

   Compile **Echo2.java (javac *Echo2.java)* y a continuación, realice las siguientes operaciones:

   a. Ejecute los programas comenzando por el servidor (no olvide especificar el número de puerto como argumento de línea de mandato) y a continuación, el cliente. Realice una sesión y observe los mensajes de diagnóstico visualizados en ambos lados. Escriba sus observaciones sobre la secuencia de eventos.

b. Con el servidor en ejecución, arranque dos clientes en ventanas separadas. ¿Se pueden realizar las dos sesiones en paralelo? Describa y explique sus observaciones. ¿Están de acuerdo con el diagrama de secuencia de la Figura 5.27?

8. Este ejercicio trata con *ServidorEcho3*. Recuerde que *ServidorEcho3* es un servidor orientado a conexión y concurrente. Compile **Echo3.java* y a continuación, realice las siguientes operaciones:

   a. Ejecute los programas comenzando por el servidor *ServidorEcho3* (no olvide especificar el número de puerto como argumento de línea de mandato) y a continuación, el cliente *ClienteEcho2*. Realice una sesión y observe los mensajes de diagnóstico visualizados en ambos lados. Describa sus observaciones sobre la secuencia de eventos.

   b. Con el servidor en ejecución, arranque dos clientes en ventanas separadas. ¿Se pueden realizar las dos sesiones de cliente en paralelo? Describa y explique sus observaciones. ¿Están de acuerdo con el diagrama de secuencia de la Figura 5.30?

9. Con sus propias palabras, describa las diferencias, desde el punto de vista del cliente, entre un servidor iterativo y un servidor concurrente para un servicio, tal como *Echo*, que involucre múltiples rondas de intercambio de mensajes. La descripción debería tratar aspectos tales como (i) la lógica del software (recuerde la arquitectura de tres niveles) y (ii) el rendimiento en tiempo de ejecución (en términos del retraso o demora experimentado por el cliente).

10. Este ejercicio trata con servidores con estado que mantienen información de estado global.

    a. Compile *ServidorContador1.java* y *ClienteContador1.java* («*javac*Contador1.java*»). Ejecute el servidor y a continuación, varias veces un cliente. ¿Se incrementa el contador con cada cliente?

    b. Modifique *ServidorContador1.java* y *ClienteContador1.java* de manera que el contador se incremente en 2 por cada cliente. Vuelva a ejecutar el cliente y el servidor y compruebe la salida resultante.

    c. Proporcione el código de un servidor orientado a conexión y un cliente para el protocolo *contador*.

11. Utilizando la arquitectura de software de tres niveles presentada en este capítulo, diseñe e implemente un conjunto cliente-servidor para el protocolo siguiente (no se trata de un servicio conocido): cada cliente envía un nombre al servidor. El servidor acumula los nombres recibidos por sucesivos clientes (añadiendo cada uno, terminado con un carácter de nueva línea, '\n', a una cadena estática de caracteres). Al recibir un nombre, el servidor envía los nombres que ha recogido al cliente. El cliente, a continuación, visualiza todos los nombres que recibe del servidor. La Figura 5.35 ilustra el diagrama de secuencia del protocolo con tres sesiones de cliente concurrentes.

    a. ¿Se trata de un servidor con estado? Si es así, ¿qué tipo de información de estado mantiene (global o de sesión)?

    b. Cree una o más versiones del protocolo:

       i. Cliente y servidor sin conexión.

       ii. Cliente y servidor orientado a conexión e iterativo.

       iii. Cliente y servidor orientado a conexión y concurrente.

Por cada versión, debe entregar: (A) los listados de los programas y (B) una descripción de cómo se realiza la arquitectura de tres niveles utilizando módulos de software separados (las clases Java).

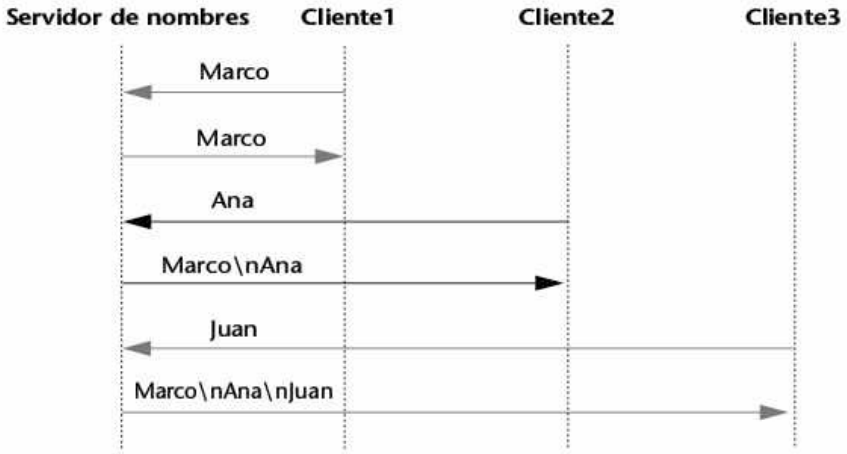


Figura 5.35. Un diagrama de secuencia del protocolo que muestra tres sesiones de cliente concurrentes.

12. Considere el siguiente protocolo, que se llamará protocolo *CuentaAtrás*. Cada cliente contacta con el servidor mediante un mensaje inicial que especifica un valor entero *n*. El cliente, a continuación, realiza un bucle para recibir *n* mensajes del servidor, tal que los mensajes contienen los valores *n, n-1, n-2, ..., 1* sucesivamente.
    a. ¿Se trata de un servidor con estado? Si es así, ¿qué tipo de información de estado mantiene (global o de sesión)?
    b. Cree una o más versiones del protocolo:
        i. Cliente y servidor sin conexión.
        ii. Cliente y servidor orientado a conexión e iterativo.
        iii. Cliente y servidor orientado a conexión y concurrente.

    Por cada versión, debe entregar: (A) los listados de los programas y (B) una descripción de cómo se realiza la arquitectura de tres niveles utilizando módulos de software separados (las clases Java).
13. Considere un servicio de red de las siguiente características: un cliente solicita que el usuario introduzca mediante el teclado un entero *n* y lo envía al servidor. A continuación, recibe del servidor una cadena de caracteres que contiene el valor *n+1*.
    a. Escriba una especificación independiente de la implementación para este protocolo, con la descripción de (1) la secuencia de eventos (utilizando un diagrama de secuencias) y la (2) representación de datos.
    b. Proporcione el código de las tres siguientes versiones del conjunto cliente-servidor:

    — Sin conexión (utilizando paquetes datagrama).

— Orientado a conexión (utilizando *sockets stream*) y servidor iterativo.
— Orientado a conexión (utilizando *sockets stream*) y servidor concurrente.

REFERENCIAS

1. Jon Postel, El protocolo DayTime, RFC 867, *http://www.ietf.org/rfc/rfc0867.txt?number=867*
2. Jon Postel, El protocolo Echo, RFC 862, *http://www.ietf.org/rfc/rfc0862.txt?number=862*

CAPÍTULO 6

Comunicación de grupo

En los capítulos previos, se ha presentado la comunicación entre procesos (IPC, *Interprocess Communication*) como el intercambio de información entre dos procesos. En este capítulo, se examinará la IPC entre un grupo de procesos, o **comunicación de grupo**.

6.1. UNIDIFUSIÓN FRENTE A MULTIDIFUSIÓN

En la IPC que se ha presentado hasta ahora, los datos se mandan desde un proceso de origen, el emisor, a un proceso de destino, el receptor. Esta forma de IPC se denomina **unidifusión**, el envío de información a un único receptor, en contraste con **multidifusión**, el envío de información a múltiples receptores. La unidifusión proporciona una IPC de uno-a-uno, mientras que la multidifusión provee una IPC de uno-a-muchos. *Véase* la Figura 6.1.

Mientras que la mayoría de servicios y aplicaciones de red utilizan unidifusión para IPC, la multidifusión es útil para aplicaciones tales como mensajes inmediatos,

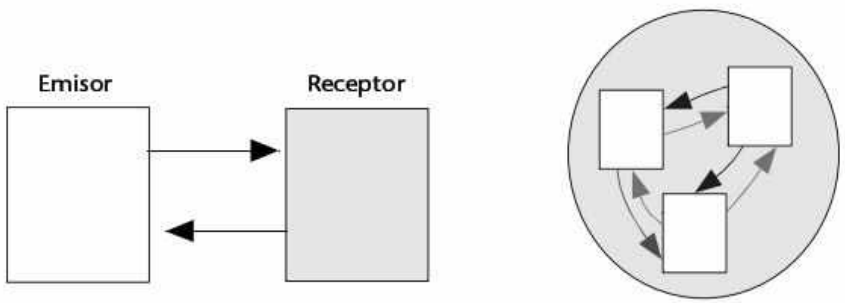


Figura 6.1. IPC de unidifusión y de multidifusión.

La **replicación** de un servicio consiste en mantener duplicados de ese servicio. Una técnica común para mejorar la disponibilidad de un servicio ante la presencia de fallos es duplicar los datos y el sistema que proporciona ese servicio.

La **tolerancia de fallos** se refiere a la capacidad que tiene una aplicación para tolerar fallos hasta cierto punto.

groupware, conferencias en línea y aprendizaje interactivo, y puede utilizarse para aplicaciones tales como subastas en línea en tiempo real. La multidifusión es también útil para la replicación de servicios con el propósito de obtener tolerancia a fallos.

En aplicaciones o servicios de red que hacen uso de multidifusión, un conjunto de procesos forma un grupo, llamado **grupo de multidifusión**. Cada proceso en un grupo puede mandar y recibir mensajes. Un mensaje enviado por cualquier proceso del grupo puede recibirlo cada proceso que forma parte del grupo. Un proceso puede también decidir abandonar un grupo de multidifusión.

En una aplicación como puede ser una conferencia en línea, un grupo de procesos interaccionan utilizando multidifusión para intercambiar audio, vídeo y/o datos de texto.

Como en ocasiones anteriores, la discusión se centrará en el nivel de servicio, específicamente en el mecanismo de IPC, de la aplicación.

6.2. UNA API DE MULTIDIFUSIÓN ARQUETÍPICA

Una interfaz de programación de aplicaciones que provea multidifusión debe proporcionar las siguientes operaciones primitivas:

- **Incorporación**. Esta operación permite a un proceso unirse a un **grupo de multidifusión** específico. Un proceso que se ha unido a un grupo de multidifusión es un **miembro** del grupo, lo que le da derecho a recibir todos los mensajes dirigidos al grupo. Un proceso puede ser miembro de uno o más grupos de multidifusión en un momento dado. Nótese que para esta y otras operaciones de multidifusión se necesita un esquema de denominación que permita identificar de forma única un grupo de multidifusión.
- **Abandono**. Esta operación permite que un proceso deje de formar parte de un grupo de multidifusión. Un proceso que ha dejado un grupo de multidifusión ya no es miembro del mismo y por tanto, no tiene derecho a recibir ninguna multidifusión dirigida al grupo, aunque el proceso pueda seguir siendo miembro de otros grupos de multidifusión.
- **Envío**. Esta operación permite que un proceso mande un mensaje a todos los procesos que actualmente forman parte de un grupo de multidifusión.
- **Recepción**. Esta operación permite que un proceso miembro reciba mensajes enviados a un grupo de multidifusión.

En la Sección 6.5 se examinarán el API de multidifusión de Java y algunos programas de ejemplo que usan esta API. En ese momento se verá cómo se proporcionan estas operaciones primitivas utilizando la sintaxis de Java. Antes de esto, sin embargo, se explorarán algunas de las cuestiones interesantes específicas de la multidifusión, que surgen de la naturaleza uno-a-muchos de la multidifusión.

6.3. MULTIDIFUSIÓN SIN CONEXIÓN FRENTE A ORIENTADA A CONEXIÓN

El mecanismo de multidifusión básico es sin conexión. La razón es obvia si se considera la naturaleza uno-a-muchos de la multidifusión. En un grupo de n procesos, si se establece una conexión entre el emisor y cada uno de los otros procesos del gru-

po, se necesitará un total de *n-1* conexiones. Además, cada uno de los *n* procesos puede ser potencialmente un emisor, de manera que cada proceso debe mantener una conexión con cada uno de los otros procesos, dando lugar a un total de $n * (n-1)$ o redondeando, n^2 conexiones. Si *n-1* es grande, el número total de conexiones será prohibitivamente elevado.

Además, la IPC sin conexión es apropiada para un tipo común de aplicaciones de multidifusión: aplicaciones donde se transmiten datos de audio y vídeo entre procesos en tiempo real. (Que los datos se transmiten en tiempo real, significa que la latencia entre el emisor y el receptor debería ser cercana a cero). Cuando se transmiten los datos de audio o vídeo, la reducción en la **latencia** proporcionada por la comunicación sin conexión prevalece sobre las ventajas ofrecidas por la comunicación orientada a conexión. Cuando se envían datos para una animación, por ejemplo, es más aceptable que el receptor experimente una distorsión ocasional en la imagen de un fotograma (un acontecimiento probable con una comunicación sin conexión) que un retraso frecuente perceptible entre fotogramas consecutivos (un acontecimiento probable con una comunicación orientada a conexión).

La **latencia** se refiere al retraso o demora en la transmisión de datos.

6.4. MULTIDIFUSIÓN FIABLE FRENTE A NO FIABLE

Cuando un proceso envía un mensaje de multidifusión, el soporte en tiempo de ejecución del mecanismo de multidifusión es responsable de entregar los mensajes a cada proceso que pertenezca actualmente al grupo de multidifusión. Dado que cada proceso implicado puede residir en una máquina diferente, la entrega de los mensajes requiere la cooperación de mecanismos que están ejecutando independientemente en esos sistemas. Debido a factores como fallos en enlaces de la red, defectos en las máquinas de la red, demoras en el encaminamiento y diferencias en el software y hardware, el tiempo desde que se manda un mensaje de multidifusión hasta que se recibe puede variar en los distintos procesos receptores. Mientras que las diferencias en la entrega del mensaje entre distintas máquinas puede ser insignificante si las máquinas están ubicadas geográficamente próximas, puede no ser así si las máquinas están distribuidas en una red de área amplia.

Asimismo, puede ocurrir que uno o más procesos no reciban un mensaje, debido a errores y/o fallos en la red, en las máquinas o en el soporte en tiempo de ejecución.

Mientras que algunas aplicaciones, tales como conferencias de vídeo, pueden tolerar un fallo ocasional o el desorden en los mensajes recibidos, hay aplicaciones, como las aplicaciones de base de datos, en las que tales anomalías no son aceptables.

Por tanto, cuando se emplea un mecanismo de multidifusión para una aplicación, es importante que se elija uno con las características apropiadas para dicha aplicación. En caso contrario, se necesitarán proporcionar medidas en la lógica de aplicación para manejar las anomalías que pueden ocurrir en la entrega de los mensajes. Por esta razón, es útil clasificar los mecanismos de multidifusión en términos de sus características de entrega de mensajes.

Multidifusión no fiable

Básicamente, el sistema de multidifusión hará un intento bienintencionado de entregar los mensajes a cada proceso que forma parte del grupo, pero no se garantiza la

entrega del mensaje en su forma correcta a cada proceso. De esta manera, cualquier mensaje enviado por un proceso puede ser recibido por cero o más procesos. En el mejor de los casos, todos los procesos reciben el mensaje en su forma correcta. En el peor de los casos, ningún proceso recibe correctamente el mensaje. En otros casos, algunos procesos pueden recibir el mensaje, pero otros procesos no, o algunos procesos pueden recibir el mensaje corrupto, o más de una vez. Se dice que un sistema de este tipo proporciona **multidifusión no fiable**.

Multidifusión fiable

Un sistema de multidifusión que garantice que finalmente se entrega correctamente cada mensaje a cada proceso del grupo se dice que proporciona **multidifusión fiable**. En este sistema, se puede asumir que cada mensaje que envía un proceso va a ser finalmente entregado sin estar corrupto a todos los procesos del grupo.

La definición de multidifusión fiable requiere que cada proceso participante reciba exactamente una copia de cada mensaje enviado. Sin embargo, la definición no establece ninguna restricción sobre el orden en que se entregan los mensajes a cada proceso individual: cada proceso puede recibir los mensajes en una secuencia que corresponde con cualquier permutación del orden en que fueran mandados. En aplicaciones donde el orden de entrega del mensaje es significativo, es útil clasificar a su vez a los sistemas de multidifusión fiable como se describe a continuación.

Sin orden

Un sistema de multidifusión fiable sin orden garantiza la entrega segura de cada mensaje, pero no proporciona ninguna garantía sobre el orden de entrega de los mensajes.

Por ejemplo, considere que tres procesos P_1, P_2 y P_3 han formado un grupo de multidifusión. Además, supóngase que se envían tres mensajes, m_1, m_2 y m_3, al grupo. En ese caso, un sistema de multidifusión fiable sin orden puede entregar los mensajes a cada uno de los tres procesos en cualquiera de las 3! (factorial de 3)= 6 permutaciones (m_1-m_2-m_3, m_1-m_3-m_2, m_2-m_1-m_3, m_2-m_3-m_1, m_3-m_1-m_2 o m_3-m_2-m_1).

Nótese que es posible que cada participante reciba los mensajes en un orden diferente al de entrega de los mensajes a otros participantes. En el ejemplo, es posible que a P_1 se le entreguen los mensajes en el orden m_1-m_2-m_3, mientras que que a P_2 se le entreguen en el orden m_2-m_1-m_3, y a P_3 en el orden m_1-m_3-m_2. Por supuesto, es también posible que los mensajes se entreguen a P_1, P_2 y P_3 en el mismo orden, m_1-m_2-m_3, pero no se puede hacer esta suposición en una aplicación si se emplea un mecanismo de multidifusión sin orden.

Multidifusión FIFO

Un sistema que garantice que la entrega de los mensajes se adhiere a la siguiente condición se dice que proporciona una multidifusión **FIFO** (primero en entrar-primero en salir, *first in-first out*), o en orden-de-envío:

> *Si el proceso P manda los mensajes m_i y m_j, en ese orden, a un grupo de multidifusión, entonces a cada proceso en el grupo de multidifusión se le entregarán los mensajes en ese orden.*

Para ilustrar esta definición, se examinará un ejemplo. Supóngase que P_1 manda los mensajes m_1, m_2, y m_3, en este orden, a un grupo de multidifusión G. En este caso, con una multidifusión FIFO, se garantiza que a cada proceso de G se le entregarán los mensajes en ese mismo orden: m_1, m_2 y a continuación, m_3. Nótese que esta definición no establece ninguna restricción en el orden de entrega de los mensajes mandados por procesos diferentes. Para ilustrar este punto, se utilizará un ejemplo simplificado de un grupo de multidifusión de dos procesos: P_1 y P_2. Supóngase que P_1 envía los mensajes m_{11} y después m_{12}, mientras que P_2 manda los mensaje m_{21} y, a continuación, m_{22}. En este caso, un sistema de multidifusión FIFO puede entregar los mensajes a cada uno de los procesos en cualquiera de los siguientes órdenes:

m_{11}-m_{12}-m_{21}-m_{22}, m_{11}-m_{21}-m_{12}-m_{22}, m_{11}-m_{21}-m_{22}-m_{12},

m_{21}-m_{11}-m_{12}-m_{22}, m_{21}-m_{11}-m_{22}-m_{12}, m_{21}-m_{22}-m_{11}-m_{12}.

Nótese que mientras que los mensajes que manda P_1 se entregan a cada proceso en el orden de secuencia m_{11}-m_{12}, y los enviados por P_2 se entregan en el orden de secuencia m_{21}-m_{22}, las dos secuencias pueden intercalarse de cualquier manera.

Multidifusión en orden causal

Un sistema de multidifusión se dice que proporciona una multidifusión **causal** si la entrega de mensajes satisface el siguiente criterio:

> *Si un mensaje m_i causa (tiene como consecuencia) la existencia del mensaje m_j, entonces m_i se entregará a cada proceso antes que m_j. Los mensajes m_i y m_j tiene una relación llamada relación* ***sucede-antes*** *o* ***causal****, denotada como $m_i \rightarrow m_j$.*

La relación sucede-antes es transitiva: si $m_i \rightarrow m_j$ y $m_j \rightarrow m_k$, entonces $m_i \rightarrow m_j \rightarrow m_k$. En este caso, un sistema de multidifusión en orden causal garantiza que estos tres mensajes se entreguen a cada proceso en el orden de m_i, m_j y a continuación, m_k.

Como ilustración, supóngase que tres procesos (P_1, P_2 y P_3) están en un grupo de multidifusión. P_1 manda un mensaje m_1, al que P_2 responde con un mensaje m_2. Dado que m_2 es desencadenado por m_1, comparten una relación causal de $m_1 \rightarrow m_2$. Supóngase que la recepción de m_2 causa a su vez que P_3 envíe un mensaje de multidifusión m_3, es decir, $m_2 \rightarrow m_3$. Los tres mensajes, por tanto, comparten la relación causal de $m_1 \rightarrow m_2 \rightarrow m_3$. Un sistema de mensajes de multidifusión en orden causal garantiza que estos tres mensajes se entregan a cada uno de los tres procesos en el orden m_1-m_2-m_3. Nótese que en este caso no habría ninguna restricción en el orden de entrega de los mensajes si el sistema de multidifusión fuera FIFO en lugar de causal.

Haciendo una variación del ejemplo anterior, supóngase que P_1 multidifunde un mensaje m_1, al que P_2 responde con un mensaje de multidifusión m_2 y, de forma independiente, P_3 contesta a m_1 con un mensaje de multidifusión m_3. Los tres mensajes comparten ahora la relación causal: $m_1 \rightarrow m_2$ y $m_1 \rightarrow m_3$. Un sistema de multidifusión en orden causal puede entregar estos mensajes a los procesos participantes en cualquiera de las dos siguientes órdenes:

m_1-m_2-m_3

m_1-m_3-m_2

En este ejemplo, no es posible que se entreguen los mensajes a cualquiera de los procesos en cualquier otra permutación de los tres mensajes, tales como m_2-m_1-m_3 o m_3-m_1-m_2. La primera de estas permutaciones viola la relación causal $m_1 \rightarrow m_2$, mientras que la segunda lo hace con la relación causal $m_1 \rightarrow m_3$.

Multidifusión en orden atómico

En un sistema de **multidifusión en orden atómico**, se garantiza que todos los mensajes son entregados a cada participante exactamente en el mismo orden. Nótese que el orden de entrega no tiene porque ser FIFO o causal, pero debe ser el mismo para todo proceso.

Ejemplo:

P_1 manda m_1, P_2 manda m_2, y P_3 manda m_3.

Un sistema atómico garantizará que los mensajes se entregarán a cada proceso en sólo uno de los seis siguientes órdenes: m_1-m_2-m_3, m_1-m_3-m_2, m_2-m_1-m_3, m_2-m_3-m_1, m_3-m_1-m_2, m_3-m_2-m_1.

Ejemplo:

P_1 manda m_1 y a continuación, m_2.

P_2 responde a m_1 mandando m_3.

P_3 responde a m_3 enviando m_4.

Aunque la multidifusión atómica no impone ningún orden en estos mensajes, la secuencia de eventos dicta que P_1 debe entregar m_1 antes de enviar m_2. De manera similar, P_2 debe recibir m_1 y a continuación, m_3, mientras que P_3 debe recibir m_3 antes de m_4. Por ello, cualquier orden de entrega atómica debe preservar el orden m_1-m_3-m_4. El mensaje restante m_2 puede, sin embargo, intercalarse con estos mensajes de cualquier manera. Por tanto, una multidifusión atómica dará como resultado que los mensajes se entreguen a cada uno de los procesos en uno de los siguientes órdenes: m_1-m_2-m_3-m_4, m_1-m_3-m_2-m_4 o m_1-m_3-m_4-m_2. Por ejemplo, los mensajes pueden entregarse a cada proceso en este orden: m_1-m_3-m_2-m_4.

6.5. EL API DE MULTIDIFUSIÓN BÁSICA DE JAVA

En el nivel de transporte, la multidifusión básica soportada por Java es una extensión del Protocolo Datagrama de Usuario (UDP, *User Datagram Protocol*). que, como se recordará, es sin conexión y no fiable. En el nivel de red de la arquitectura de red, los paquetes de multidifusión se transmiten a través de la red utilizando el encaminamiento de multidifusión de Internet [cisco.com, 5] proporcionado por encaminadores (conocidos como *mrouters*) capaces de encaminar la multidifusión además de la unidifusión. La multidifusión en una red local (una interconectada sin un encaminador) se lleva a cabo utilizando la multidifusión proporcionada por el protocolo de red de área local (como la multidifusión de *Ethernet*). El encaminamiento y entrega de mensajes de multidifusión son temas fuera del ámbito de este libro. Afortunadamente, tales asuntos son transparentes a un programador de aplicaciones que utiliza una API de multidifusión.

El API de multidifusión básica de Java proporciona un conjunto de clases que están estrechamente relacionadas con las clases del API de *sockets* de datagrama que se examinó en el Capítulo 4. Hay tres clases principales en el API, las dos primeras ya se han visto en el contexto de los *sockets* datagrama.

1. *InetAddress*. En el API de *sockets* datagrama, esta clase representa la dirección IP del emisor o del receptor. En la multidifusión, esta clase puede utilizarse para identificar un grupo de multidifusión (como se explicará en la próxima sección «Direcciones IP de multidifusión»).

2. *DatagramPacket*. Como en el caso de los *sockets* datagrama, un objeto de esta clase representa un datagrama real; en multidifusión, un objeto de *DatagramPacket* representa un paquete de datos enviado a todos los participantes o recibidos por cada participante de un grupo de multidifusión.
3. *MulticastSocket*. La clase *MulticastSocket* se deriva de la clase *DatagramSocket*, con capacidades adicionales para incorporarse o abandonar un grupo de multidifusión. Un objeto de las clase *socket* datagrama de multidifusión puede utilizarse para mandar y recibir paquetes IP de multidifusión.

Direcciones IP de multidifusión

Recuérdese que en el API de *socket* de unidifusión de Java, un emisor identifica a un receptor especificando el nombre de la máquina del proceso receptor, así como el puerto del protocolo al que está ligado dicho proceso.

Considérese por un momento a quién necesita dirigirse un emisor de multidifusión. En lugar de a un único proceso, un datagrama de multidifusión está destinado a ser recibido por todos los procesos que son miembros actuales de un grupo de multidifusión específico. Por tanto, cada datagrama necesita dirigirse a un grupo de multidifusión en vez de a un proceso individual.

El API de multidifusión de Java utiliza las direcciones de multidifusión del Protocolo de Internet (IP, *Internet Protocol*) para identificar los grupos de multidifusión.

En IPv4, un grupo de multidifusión se especifica mediante (1) una dirección IP de la clase D combinada con (2) un número de puerto estándar de UDP. (Nótese que en IPv6 el direccionamiento de multidifusión es significativamente diferente; *véase* [faqs.org, 4] para más detalles). Recuerde del Capítulo 1 que las direcciones IP de la clase D son aquellas que comienzan con la cadena de bits 1110 y por tanto, estas direcciones están en el rango desde el 224.0.0.0 al 239.255.255.255, inclusive. Excluyendo los cuatro bits del prefijo, hay 32-4 = 28 bits restantes, lo que da lugar a un tamaño del espacio de direcciones de 2^{28}, es decir, están disponibles, aproximadamente, 268 millones de direcciones de clase D, aunque se reserva la dirección 224.0.0.0 y no debería utilizarse por ninguna aplicación. Las direcciones de multidifusión IPv4 las gestiona y asigna la Autoridad de Números Asignados de Internet (IANA, *Assigned Numbers Authority*) [rfc-editor.org, 3].

Una aplicación que utiliza el API de multidifusión de Java debe especificar al menos una dirección de multidifusión para la aplicación. A la hora de seleccionar una dirección de multidifusión para una aplicación, se presentan las siguientes opciones:

1. Obtener una dirección de multidifusión estática permanentemente asignada de IANA. Las direcciones permanentes están limitadas a aplicaciones de Internet globales y bien conocidas y su asignación está muy restringida. En IANA [iana.org, 2], se puede encontrar la lista de direcciones actualmente asignadas. A continuación, se presenta una muestra de algunas de las direcciones asignadas más interesantes:

```
224.0.0.1 Todos los sistemas en esta subred
224.0.0.11 Agentes móviles
224.0.1.16 MUSIC-SERVICE
224.0.1.17 SEANET-TELEMETRY
224.0.1.18 SEANET-IMAGE
224.0.1.41 gatekeeper
224.0.1.84 jini-announcement
```

```
224.0.1.85 jini-request
224.0.1.115 Multidifusión simple
224.0.6.0-224.0.6.127 Proyecto ISIS de Cornell
224.0.7.0-224.0.7.255 Where-Are-You
224.0.8.0-224.0.8.255 INTV
224.0.9.0-224.0.9.255 Invisible Worlds
224.0.11.0-224.0.11.255 NCC.NET Audio
224.0.12.0-224.0.12.063 Microsoft y MSNBC
224.0.17.0-224.0.17.031 Mercantile & Commodity Exchange
224.0.17.064-224.0.17.127 ODN-DTV
224.0.18.0-224.0.18.255 Dow Jones
224.0.19.0-224.0.19.063 Walt Disney Company
224.0.22.0-224.0.22.255 WORLD MCAST
224.2.0.0-224.2.127.253 Llamadas de conferencias multimedia
```

2. Elegir una dirección arbitraria, suponiendo que la combinación de la dirección y el número de puerto elegidos de manera aleatoria no estará probablemente en uso.
3. Obtener una dirección de multidifusión transitoria en tiempo de ejecución; esta dirección puede recibirla una aplicación mediante el Protocolo de Anuncio de Sesión (*Session Announcement Protocol*) [faqs.org, 6].

La tercera opción queda fuera del ámbito de este capítulo. En los ejercicios y ejemplos se utilizará a menudo la dirección estática 224.0.0.1, con un nombre de dominio de ALL-SYSTEMS.MCAST.NET, para procesos que ejecutan en todas las máquinas de la red de área local, como las que tendrá el lector en su laboratorio. Alternativamente, se puede utilizar una dirección arbitraria que presumiblemente no haya sido asignada, como por ejemplo, un número en el rango de 239.*.*.* (por ejemplo, 239.1.2.3).

En el API de Java, un objeto *MulticastSocket* se asocia a una dirección de puerto como por ejemplo, 3456, y los métodos del objeto permiten unirse a una dirección de multidifusión como por ejemplo, 239.1.2.3 y dejarla.

Incorporación a un grupo de multidifusión

Para incorporarse a un grupo de multidifusión en una dirección IP *m* y un puerto UDP *p*, se debe instanciar un objeto *MulticastSocket* con *p* y a continuación, se puede invocar el método *joinGroup*, especificando la dirección *m*:

```
// incorporación a un grupo de multidifusión en la dirección IP
// 224.0.0.1 y en el puerto 3456
InetAddress grupo = InetAddress.getByName("224.0.0.1");
MulticastSocket s = new MulticastSocket(3456);
s.joinGroup(grupo);
```

Envío a un grupo de multidifusión

Se puede mandar un mensaje de multidifusión utilizando una sintaxis similar a la del API de *sockets* datagrama. Concretamente, se debe crear un paquete datagrama con la especificación de una referencia a un vector de octetos que contenga los datos, la longitud del vector, la dirección de multidifusión y el número de puerto. A continua-

ción, se puede invocar el método *send* del objeto *MulticastSocket* (heredado de la clase *DatagramSocket*) para mandar los datos.

No es necesario que un proceso se una a un grupo de multidifusión para mandar mensajes al mismo, aunque debe hacerlo para poder recibir los mensajes. Cuando se manda un mensaje a un grupo de multidifusión, todos los procesos que se han unido al grupo de multidifusión, que puede incluir al emisor, deberían recibir el mensaje, aunque no se garantiza.

El siguiente segmento de código ilustra la sintaxis requerida para enviar un mensaje a un grupo de multidifusión:

```
String msj = "Este es un mensaje de multidifusión.";
InetAddress grupo = InetAddress.getByName("239.1.2.3");
MulticastSocket s = new MulticastSocket(3456);
s.joinGroup(grupo); // opcional
DatagramPacket hola = new DatagramPacket(msj.getBytes( ),
  msj.length( ), grupo, 3456);
s.send(hola);
```

Recepción de mensajes mandados a un grupo de multidifusión

Un proceso que se ha unido a un grupo de multidifusión puede recibir mensajes enviados al grupo utilizando una sintaxis similar a la usada para recibir datos mediante una API de *sockets* datagrama. El siguiente segmento de código ilustra la sintaxis usada para recibir mensajes enviados a un grupo de multidifusión.

```
byte[] almacen = new byte[1000];
InetAddress grupo = InetAddress.getByName("239.1.2.3");
MulticastSocket s = new MulticastSocket(3456);
s.joinGroup(grupo);
DatagramPacket recibido = new DatagramPacket(almacen, almacen.length);
s.receive(recibido);
```

Abandono de un grupo de multidifusión

Un proceso puede dejar un grupo de multidifusión invocando el método *leaveGroup* de un objeto *MulticastSocket*, especificando la dirección de multidifusión del grupo:

```
s.leaveGroup(grupo);
```

Ajuste del «tiempo-de-vida»

El soporte en tiempo de ejecución de una API de multidifusión a menudo emplea una técnica conocida como propagación de mensajes, en la que se propaga un paquete desde una máquina a una máquina vecina usando un algoritmo que, cuando se ejecuta apropiadamente, entregará finalmente el mensaje a todos los participantes. Sin embargo, ante alguna circunstancia anómala, es posible que el algoritmo que controla la propagación de mensajes no termine apropiadamente, dando como resultado que un paquete circule por la red indefinidamente. Este fenómeno no es deseable, ya que causa una sobrecarga innecesaria en los sistemas y en la red. Para evitar que esto ocurra, se recomienda que se fije un parámetro «tiempo-de-vida» en cada datagrama de multidifusión. El parámetro de tiempo-de-vida (ttl, *time-to-live*), cuando se fija, limi-

ta el número de enlaces de red, o saltos, a través de los que se retransmitirá el paquete en la red.

En el API de Java, este parámetro se puede fijar invocando el método *setTimeToLive* del *MulticastSocket* del emisor de la siguiente manera:

```
String msj = "¡Hola a todos!";
InetAddress grupo = InetAddress.getByName("224.0.0.1");
MulticastSocket s = new MulticastSocket(3456);
s.setTimeToLive(1); // ajusta time-to-live a 1 salto – un valor apropiado
                    // para multidifusión en máquinas locales
DatagramPacket hola = new DatagramPacket(msj.getBytes( ),
  msj.length( ),grupo, 3456);
s.send(hola);
```

El valor especificado para el ttl debe estar en el rango $0 \leq ttl \leq 255$; en caso contrario, se activará una *IllegalArgumentException*.

Las valores de ttl recomendados [Harold, 12] son:

- 0 si la multidifusión está restringida a procesos en la misma máquina.
- 1 si la multidifusión está restringida a procesos en la misma subred.
- 32 si la multidifusión está restringida a procesos en la misma zona.
- 64 si la multidifusión está restringida a procesos en la misma región.
- 128 si la multidifusión está restringida a procesos en el mismo continente.
- 255 si la multidifusión no está restringida.

Ejemplo 1. Las Figuras 6.2 y 6.3 ilustran el código de un ejemplo simple de aplicación de multidifusión, que se presenta aquí básicamente para ilustrar la sintaxis del API. Cuando se ejecuta, cada proceso receptor (Figura 6.3) se subscribe al grupo de multidifusión 239.1.2.3 en el puerto 1234 y espera la llegada de un mensaje. El proceso emisor (Figura 6.2), por otro lado, no es un miembro del grupo de multidifusión (aunque podría serlo); dicho proceso manda un único mensaje al grupo de multidifusión 239.1.2.3 en el puerto 1234 antes de cerrar su *socket* de multidifusión.

Figura 6.2. *Ejemplo1Emisor.java.*

```
1  import java.io.*;
2  import java.net.*;
3
4  /**
5  * Este ejemplo ilustra la sintaxis de la multidifusión básica.
6  * @author M. L. Liu
7  */
8  public class Ejemplo1Emisor {
9
10 // Una aplicación que usa un socket de multidifusión para enviar
11 // un único mensaje a un grupo de multidifusión.
12 // El mensaje se especifica como un argumento de línea de mandato.
13
14   public static void main(String[ ] args) {
15     MulticastSocket s;
16     InetAddress grupo;
```

(continúa)

```
17      if (args.length != 1)
18        System.out.println
19          ("Este programa requiere un argumento de línea de mandato");
20        else {
21          try {
22            // crea el socket de multidifusión
23            grupo = InetAddress.getByName("239.1.2.3");
24            s = new MulticastSocket(3456);
25            s.setTimeToLive(32); // restringe multidifusión a proc. que
26                                 // ejecutan en máquinas en la misma zona.
27            String msj = args[0];
28            DatagramPacket paquete =
29              new DatagramPacket(msj.getBytes(), msj.length(),
30              grupo, 3456);
31            s.send(paquete);
32            s.close( );
33          }
34          catch (Exception ex) { // llega aquí si ocurre un error
35            ex.printStackTrace( );
36          } // fin de catch
37        } //fin de else
38      } // fin de main
39    } // fin de class
```

Figura 6.3. *Ejemplo1Receptor.java.*

```
1   import java.io.*;
2   import java.net.*;
3
4
5   /**
6   * Este ejemplo ilustra la sintaxis de la multidifusión básica.
7   * @author M. L. Liu
8   */
9   public class Ejemplo1Receptor {
10
11  // Una aplicación que se une a un grupo de multidifusión y
12  // recibe el único mensaje enviado al grupo.
13    public static void main(String[ ] args) {
14      MulticastSocket s;
15      InetAddress grupo;
16      try {
17        // se une a un grupo de multidifusión y espera recibir un
          // mensaje
18        grupo = InetAddress.getByName("239.1.2.3");
```

(continúa)

```
19          s = new MulticastSocket(3456);
20          s.joinGroup(grupo);
21          byte[] almacen = new byte[100];
22          DatagramPacket recibido = new DatagramPacket(almacen,
               almacen.length);
23          s.receive(recibido);
24          System.out.println(new String(almacen));
25          s.close( );
26        }
27        catch (Exception ex) { // llega aquí si ocurre un error
28          ex.printStackTrace( );
29        } // fin de catch
30      } // fin de main
31  } // fin de class
```

Ejemplo 2. Como otra ilustración del API de multidifusión de Java, se presenta un ejemplo donde cada proceso de un grupo de multidifusión manda un mensaje, y, de forma independiente, cada proceso también visualiza todos los mensajes que recibe como miembro del grupo de multidifusión.

Ejemplo2EmisorReceptor.java (Figura 6.4) es el código del ejemplo. En el método *main* se crea un hilo para recibir y visualizar los mensajes (*véase* la línea 39). Para asegurarse de que cada proceso está listo para recibir, se realiza una pausa (véase las líneas desde la 40 a la 43) antes de que el proceso envíe su mensaje.

Figura 6.4. *Ejemplo2EmisorReceptor.java.*

```
1   // Este programa ilustra el envío y recepción usando multidifusión
2
3   import java.io.*;
4   import java.net.*;
5   /**
6   * Este ejemplo ilustra el uso de múltiples hilos para enviar y
7   * recibir multidifusión en un proceso.
8   * @author M. L. Liu
9   */
10  public class Ejemplo2EmisorReceptor{
11
12  // Una aplicación que usa un socket de multidifusión para enviar
13  // un único mensaje a un grupo de multidifusión, y utiliza un hilo
14  // independiente que usa otro socket de multidifusión para recibir
15  // mensajes enviados al mismo grupo.
16  // Se requieren tres argumentos de línea de mandato:
17  // <dirección IP de multidifusión>,<puerto de
    // multidifusión>,<mensaje>
18
19    public static void main(String[ ] args) {
20
21      InetAddress grupo = null;
```

(continúa)

```
22        int puerto = 0;
23        MulticastSocket socket = null;
24        String caracteres;
25        byte[] datos = null;
26
27        if (args.length !=3)
28          System.out.println("Se requieren 3 args. de línea de mandato");
29        else {
30          try {
31            grupo = InetAddress.getByName(args[0]);
32            puerto = Integer.parseInt(args[1]);
33            caracteres = args[2];
34            datos = caracteres.getBytes();
35            DatagramPacket paquete =
36              new DatagramPacket(datos, datos.length, grupo, puerto);
37            Thread elHilo =
38              new Thread(new HiloLector(grupo, puerto));
39            elHilo.start();
40            System.out.println("Pulse Intro cuando listo para enviar:");
41            InputStreamReader is = new InputStreamReader(System.in);
42            BufferedReader br = new BufferedReader(is);
43            br.readLine();
44            socket = new MulticastSocket(puerto);
45            socket.setTimeToLive(1);
46            socket.send(paquete);
47            socket.close( );
48          }
49          catch (Exception se) {
50            se.printStackTrace( );
51          } // fin de catch
52        } // fin de else
53      } // fin de main
54
55    } // fin de class
```

Figura 6.5. *HiloLector.java.*

```
1   import java.net.*;
2   import java.io.*;
3   /**
4   * Esta clase es utilizada por Ejemplo2EmisorReceptor para
5   * leer mensajes de multidifusión mientras el hilo principal envía
6   * un mensaje de multidifusión. Hace eco en la pantalla de cada
7   * mensaje leído.
8   * @author M. L. Liu
9   */
10  class HiloLector implements Runnable {
```

(continúa)

```
11
12      static final int MAX_LON = 30;
13      private InetAddress grupo;
14      private int puerto;
15
16      public HiloLector(InetAddress grupo, int puerto) {
17        this.grupo = grupo ;
18        this.puerto = puerto;
19      }
20
21      public void run( ) {
22
23        try {
24
25          MulticastSocket socket = new MulticastSocket(puerto);
26          socket.joinGroup(grupo);
27          while (true) {
28            byte[ ] datos = new byte[MAX_LON];
29            DatagramPacket paquete =
30              new DatagramPacket(datos, datos.length, grupo, puerto);
31            socket.receive(paquete);
32            String s = new String(paquete.getData());
33            System.out.println(s);
34          } // fin de while
35        } // fin de try
36        catch (Exception exception) {
37          exception.printStackTrace( );
38        } // fin de catch
39      } // fin de run
40
41  } //fin de class
```

El API de multidifusión de Java y mecanismos similares pueden emplearse para dar soporte a la lógica de servicio de una aplicación. Nótese que una aplicación puede utilizar una combinación de unidifusión y multidifusión para su IPC.

Una aplicación que hace uso de la multidifusión a veces se denomina aplicación **consciente de la multidifusión** (*multicast-aware*).

Para los interesados en una sala de *chat* implementada utilizando multidifusión, consúltese la referencia [Hughes, 7].

6.6. EL API DE MULTIDIFUSIÓN FIABLE

El API de multidifusión de Java que se ha explorado en este capítulo es una extensión del API de *sockets* datagrama. Por ello, comparte una característica fundamental de los datagramas: la entrega no fiable. En particular, no se garantiza que se entreguen los mensajes a los procesos receptores. Por tanto, esta API proporciona una **multidifusión no fiable**. Sin embargo, el lector debe tener en cuenta a la hora de realizar los ejercicios que, si se ejecutan los procesos en una máquina o en máquinas de

una subred, no se observará ninguna pérdida de mensajes o perturbación en el orden de entrega de los mismos. Estas anomalías son más probables cuando las máquinas involucradas están conectadas de forma remota, debido a los fallos en la red o a los retrasos en el encaminamiento.

Como se mencionó previamente, hay aplicaciones para las que la multidifusión no fiable es inaceptable. Para tales aplicaciones, hay un número de paquetes disponibles que proporcionan una API de multidifusión fiable, entre los que se incluyen:

- El **servicio de Multidifusión Fiable de Java (Servicio JRM,** ***Java Reliable Multicast Service*****)** [Rosenzweig, Kandansky y Hanna, 8; Bischof, 9] es un paquete que mejora el API de multidifusión básica de Java proporcionándole la capacidad de que un receptor pueda reparar los datos de multidifusión que se hayan perdido o dañado, así como medidas de seguridad para proteger la privacidad de los datos.
- El sistema **Totem** [alpha.ece.ucsb.edu, 10], desarrollado por la universidad de California, en Santa Bárbara, «proporciona una entrega de mensajes fiable y totalmente ordenada para procesos incluidos en grupos de procesos sobre una red de área local, o sobre múltiples redes de área local interconectadas por pasarelas».
- El **Entorno de Multidifusión Fiable (RMF,** ***Reliable Multicast Framework***) de TASC [tascnets.com, 11] proporciona multidifusión fiable y en orden de envío (FIFO).

El uso de estos paquetes queda fuera del ámbito de este libro. Se anima a los lectores interesados a que consulten las referencias para detalles adicionales.

RESUMEN

Este capítulo proporciona una introducción al uso de la comunicación de grupo en la computación distribuida.

- La multidifusión difiere de la unidifusión: la unidifusión es una comunicación uno-a-uno, mientras que la multidifusión es una comunicación uno-a-muchos.
- Una API de multidifusión arquetípica debe proporcionar operaciones para incorporarse a un grupo de multidifusión, abandonar un grupo de multidifusión, enviar a un grupo y recibir los mensajes de multidifusión enviados al grupo.
- La multidifusión básica es sin conexión y no fiable: en un sistema de multidifusión no fiable, no está garantizado que los mensajes se entreguen sin contratiempos a cada participante.
- Un sistema de multidifusión fiable asegura que cada mensaje mandado a un grupo de multidifusión se entregue correctamente a cada participante. La multidifusión fiable puede clasificarse a su vez por el orden de entrega de mensajes que proporciona distinguiéndose las siguientes categorías:
  - — Multidifusión sin orden, que puede entregar los mensajes a cada participante en cualquier orden.
  - — Multidifusión FIFO, que mantiene el orden de los mensajes mandados por cada emisor.
  - — Multidifusión causal, que preserva las relaciones causales entre los mensajes.

— Multidifusión atómica, que entrega los mensajes a cada participante en el mismo orden.

- La dirección de multidifusión utiliza una combinación de una dirección de clase D y un número de puerto de UDP. Las direcciones IP de clase D las gestiona y asigna IANA. Una aplicación de multidifusión puede utilizar una dirección de clase D estática, una dirección transitoria obtenida en tiempo de ejecución o una dirección arbitraria que no esté asignada.
- El API de multidifusión básica de Java proporciona una multidifusión no fiable. Se crea un *MulticastSocket* con la especificación de un número de puerto. Los métodos *joinGroup* y *leaveGroup* de la clase *MulticastSocket* se pueden invocar para unirse o abandonar un grupo de multidifusión específico. Los métodos *send* y *receive* se pueden invocar para mandar y recibir un datagrama de multidifusión. Se necesita también la clase *DatagramPacket* para crear los datagramas.
- Existen paquetes que proporcionan multidifusión fiable, incluyendo el servicio de Multidifusión Fiable de Java (JRM, *Java Reliable Multicast*).

EJERCICIOS

1. Supóngase que un grupo de multidifusión actualmente tiene dos procesos, P_1 y P_2, que forman parte del mismo. Suponga que P_1 multidifunde m_{11} y, a continuación, m_{12}; P_2 multidifunde m_{21}, y luego m_{22}. Además, suponga que no se pierde ningún mensaje en la entrega.

   a. Teóricamente, ¿en cuántas diferentes ordenaciones pueden entregarse los cuatro mensajes a cada proceso si los mensajes no están relacionados?

   b. Teóricamente, ¿en cuántas diferentes ordenaciones pueden entregarse los cuatro mensajes a cada proceso si los mensajes están relacionados de forma causal de la siguiente manera $m_{11} \rightarrow m_{12} \rightarrow m_{21} \rightarrow m_{22} \rightarrow$

   c. ¿Cuáles son los posibles órdenes de entrega de los mensajes a cada proceso si los mensajes no están relacionados y la multidifusión es FIFO, causal y atómica?

   d. ¿Cuáles son los posibles órdenes de entrega de los mensajes a cada proceso si los mensajes están relacionados de forma causal de la siguiente manera $m_{11} \rightarrow m_{21} \rightarrow m_{12} \rightarrow m_{22}$ y la multidifusión es FIFO, causal y atómica?

2. Supóngase que tienen lugar los siguientes eventos en orden cronológico en un grupo de multidifusión del que forman parte tres procesos P_1, P_2, y P_3:

   P_1 multidifunde m_1.

   P_2 responde a m_1 mediante la multidifusión de m_2.

   P_3 multidifunde m_3 espontáneamente.

   P_1 responde a m_3 mediante la multidifusión de m_4.

   P_3 responde a m_2 mediante la multidifusión de m_5.

   P_2 multidifunde m_6 espontáneamente.

   Para cada una de las siguientes situaciones, especifique en la entrada correspondiente de la tabla que aparece más adelante si el modo de multidifusión permite o no esa situación:

a. A todos los procesos se les entrega m_1, m_2, m_3, m_4, m_5, m_6, en ese orden.

b. A P_1 y P_2 se les entrega m_1, m_2, m_3, m_4, m_5, m_6.
   A P_3 se le entrega m_2, m_3, m_1, m_4, m_5, m_6.

c. A P_1 se le entrega m_1, m_2, m_5, m_3, m_4, m_6.
   A P_2 se le entrega m_1, m_3, m_5, m_4, m_2, m_6.
   A P_3 se le entrega m_3, m_1, m_4, m_2, m_5, m_6.

d. A P_1 se le entrega m_1, m_2, m_3, m_4, m_5, m_6.
   A P_2 se le entrega m_1, m_4, m_2, m_3, m_6, m_5.
   A P_3 se le entrega m_1, m_3, m_6, m_4, m_2, m_5.

e. A P_1 se le entrega m_1, m_2, m_3, m_4, m_5, m_6.
   A P_2 se le entrega m_1, m_3, m_2, m_5, m_4, m_6.
   A P_3 se le entrega m_1, m_2, m_6, m_5, m_3, m_4.

f. A P_1 se le entrega m_2, m_1, m_6.
   A P_2 se le entrega m_1, m_2, m_6.
   A P_3 se le entrega m_6, m_2, m_1.

g. No se entrega ningún mensaje a alguno de los procesos.

Situación	Multidifusión fiable	Multidifusión FIFO	Multidifusión causal	Multidifusión atómica
a				
b				
c				
d				
e				
f				
g				

3. Este ejercicio se basa en *Ejemplo1* presentado en este capítulo.

   a. Compile los programas *Ejemplo1*.java*. A continuación, ejecútelos en cada una de las siguientes secuencias. Describa y explique cada una de las salidas resultantes:

      i. Arranque primero dos o más procesos receptores, después un proceso emisor con el mensaje que se desee.

      ii. Arranque primero un proceso emisor con el mensaje que se desee, después dos o más procesos receptores.

   b. Basado en el *Ejemplo1Receptor.java*, cree un programa, *Ejemplo1aReceptor.java*, que se una a un grupo de multidifusión con una dirección IP diferente (por ejemplo, 239.1.2.4) pero con el mismo puerto. Compile *Ejemplo1aReceptor.java*. Arranque primero dos o más procesos *Ejemplo1Receptor*, a continuación, un proceso *Ejemplo1aReceptor* y por último, un proceso emisor con el mensaje que se desee. ¿Recibe el proceso *Ejemplo1aReceptor* el mensaje? Describa y explique la salida.

c. Basado en el *Ejemplo1Receptor.java*, cree un programa, *Ejemplo1bReceptor.java*, que se una a un grupo de multidifusión con la misma dirección IP pero con un puerto diferente. Compile *Ejemplo1bReceptor.java*. Arranque primero dos o más procesos *Ejemplo1Receptor*, luego, un proceso *Ejemplo1bReceptor* y, por último, un proceso emisor con el mensaje que se desee. ¿Recibe el proceso *Ejemplo1bReceptor* el mensaje? Describa y explique la salida.

d. Basado en el *Ejemplo1Emisor.java*, cree un programa, *Ejemplo1EmisorReceptor.java*, que se una al grupo de multidifusión, mande un mensaje, y después espere (reciba) un mensaje de multidifusión antes de cerrar el *socket* de multidifusión y terminar. Compile el programa y, a continuación, arranque dos o más programas receptores antes de comenzar el proceso *EmisorReceptor*. Describa la salida. Entregue el listado de *EmisorReceptor.java*.

e. Basado en *Ejemplo1Emisor.java*, cree un programa, *Ejemplo1bEmisor.java*, que mande un mensaje a la dirección de multidifusión del programa *Ejemplo1bReceptor.java*. Compile el programa, y luego arranque un proceso *Ejemplo1Receptor*, un proceso *Ejemplo1bReceptor*, un proceso *Ejemplo1Emisor* y, por último, un proceso *Ejemplo1bEmisor*. Describa y explique el mensaje o los mensajes recibidos por cada proceso.

f. Basado en *Ejemplo1Receptor.java* y *Ejemplo1bReceptor.java*, cree un programa, *Ejemplo1cReceptor.java*, que utilice dos hilos (incluyendo el hilo principal). Cada hilo se debe unir a uno de los dos grupos de multidifusión y recibir, y después visualizar, un mensaje antes de abandonar el grupo. Puede resultar útil el ejemplo *HiloLector.java*.

Compile y ejecute *Ejemplo1cReceptor.java*, y después arranque un proceso *Ejemplo1bEmisor*. ¿El proceso receptor visualiza ambos mensajes? Entregue el listado del programa *Ejemplo1cReceptor.java* y su clase que define el hilo de ejecución.

4. Este ejercicio se basa en *Ejemplo2* presentado en este capítulo.

a. Compile *Ejemplo2EmisorReceptor.java*, y luego arranque dos o más procesos del programa, especificando para cada uno un mensaje diferente. Un ejemplo de los mandatos necesarios para ello es:

```
java Ejemplo2EmisorReceptor 239.1.2.3 1234 msj1
java Ejemplo2EmisorReceptor 239.1.2.3 1234 msj2
java Ejemplo2EmisorReceptor 239.1.2.3 1234 msj3
```

En este ejemplo, cada uno de los tres procesos visualizaría en la pantalla los mensajes msj1, msj2 y msj3. Asegúrese de arrancar todos los procesos antes de permitir que cada uno mande su mensaje. Describa las salidas resultantes.

b. Modifique *Ejemplo2EmisorReceptor.java* de manera que cada proceso mande su mensaje diez veces. Compile y ejecute. Describa las salidas resultantes y entregue los listados de programas.

5. Escriba su propia aplicación de multidifusión. Desarrolle una aplicación de manera que múltiples procesos utilicen comunicación de grupo para llevar a cabo una elección. Hay dos candidatos: Sánchez y Pérez. Cada proceso multidifunde su voto en un mensaje que le identifica a sí mismo y su voto. Cada proceso lleva la cuenta de cuántos votos tiene cada candidato, incluyendo el propio. Al final de la elección (cuando todo el mundo en el grupo ha votado),

cada proceso hace el recuento de votos de forma independiente y visualiza la salida en su pantalla (por ejemplo, Sánchez 10, Pérez 5).

Entregue el listado de la aplicación y responda a estas preguntas:

a. ¿De qué manera su diseño permite a los participantes unirse a un grupo de multidifusión?

b. ¿Cómo se sincroniza en su diseño el principio de la elección de manera que cada proceso esté listo para recibir cualquier difusión de voto por parte de un miembro del grupo?

c. Al ejecutar la aplicación, ¿han coincidido los recuentos de votos en todas las máquinas? ¿Puede asumir que los recuentos siempre coincidirán en todas las máquinas? Explíquelo.

REFERENCIAS

1. Java 2 Platform SE v1.3.1: Clase MulticastSocket, *http://java.sun.com/j2se/1.3/docs/api/java/net/MulticastSocket.html*
2. Direcciones de multidifusión de IANA, *http://www.iana.org/assignments/multicast-addresses*
3. RFC 3171, IANA Guidelines for IPv4 Multicast Address Allocation, *http://www.rfc-editor.org/rfc/rfc3171.txt*
4. RFC 2375-IPv6 Multicast Address Assignments, *http://www.faqs.org/rfcs/rfc2375.html*
5. Cisco-Multicast Routing, *http://www.cisco.com/warp/public/614/17.html*
6. RFC 2974-Session Announcement Protocol, *http://www.faqs.org/rfcs/rfc2974.html*
7. Merlin Hughes, Multicast the Chatwaves - JavaWorld, Octubre 1999, *http://www.javaworld.com/javaworld/jw-10-1999/jw-10-step_p.html*
8. Phil Rosenzweig, Miriam Kadansky y Steve Hanna, *The Java Reliable Multicast Service: A Reliable Multicast Library, http://www.sun.com/research/techrep/1998/smli_tr-98-68.pdf*
9. Hans-Peter Bischof, JRMS Tutorial, Department of Computer Science, Rochester Institute of Technology, *http://www.cs.rit.edu/~hpb/JRMS/Tutorial/*
10. Robust Distributed Systems for Real-Time Applications, *http://alpha.ece.ucsb.edu/project_totem.html*
11. Reliable Multicast Framework (RMF), *http://www.tascnets.com/newtascnets/Software/RMF/*, Litton TASC.
12. Elliotte Rusty Harold, *Java Network Programming*, Sebastopol, CA: O'Reilly Press, 2000.

CAPÍTULO 7

Objetos distribuidos

Hasta ahora este libro se ha centrado en el uso del paradigma de paso de mensajes en la computación distribuida. A través del paradigma de paso de mensajes, los procesos intercambian datos y, mediante el uso de determinados protocolos, colaboran en la realización de tareas. Las interfaces de programación de aplicaciones basadas en este paradigma, tales como el API de sockets de unidifusión y multidifusión de Java, proporcionan una abstracción que permite esconder los detalles de la comunicación de red a bajo nivel, así como escribir código de comunicación entre procesos (IPC), utilizando una sintaxis relativamente sencilla. Este capítulo introduce un paradigma que ofrece aún una mayor abstracción, los objetos distribuidos.

7.1. PASO DE MENSAJES FRENTE A OBJETOS DISTRIBUIDOS

El paradigma de paso de mensajes es un modelo natural para la computación distribuida, en el sentido de que imita la comunicación entre humanos. Se trata de un paradigma apropiado para los servicios de red, puesto que estos procesos interactúan a través del intercambio de mensajes. Pero este paradigma no proporciona la abstracción necesaria para algunas aplicaciones de red complejas, por los siguientes motivos:

- El paso de mensaje básico requiere que los procesos participantes estén **fuertemente acoplados**. A través de esta interacción, los procesos deben comunicarse directamente entre ellos. Si la comunicación se pierde entre los procesos (debido a fallos en el enlace de comunicación, en el sistema o en uno de los procesos), la colaboración falla. Por ejemplo, considérese una sesión del protocolo *Echo*: si la comunicación entre el cliente y el servidor es interrumpida, la sesión no puede continuar.

- El paradigma de paso de mensajes está **orientado a datos**. Cada mensaje contiene datos con un formato mutuamente acordado, y se interpreta como una petición o respuesta de acuerdo al protocolo. La recepción de cada mensaje desencadena una acción en el proceso receptor.

  Por ejemplo, en el protocolo *Echo*, el receptor de un mensaje del proceso *p* solicita esta acción al servidor *Echo*: un mensaje conteniendo los mismos datos se envía al proceso *p*. En el mismo protocolo, la recepción de un mensaje del servidor *Echo* por el proceso *p* desencadena esta acción: un nuevo mensaje es solicitado por el usuario y el mensaje se envía al servidor *Echo*.

  Mientras que el hecho de que el paradigma sea orientado a datos es apropiado para los servicios de red y aplicaciones de red sencillas, no es adecuado para aplicaciones complejas que impliquen un gran número de peticiones y respuestas entremezcladas. En dichas aplicaciones, la interpretación de los mensajes se puede convertir en una tarea inabordable.

El **paradigma de objetos distribuidos** es un paradigma que proporciona mayor abstracción que el modelo de paso de mensajes. Como su nombre indica, este paradigma está basado en objetos existentes en un sistema distribuido. En programación orientada a objetos, basada en un lenguaje de programación orientado a objetos, tal como Java, los objetos se utilizan para representar entidades significativas para la aplicación. Cada objeto encapsula

- el **estado** o datos de la entidad: en Java, dichos datos se encuentran en las **variables de instancia** de cada objeto;
- las **operaciones** de la entidad, a través de las cuales se puede acceder o modificar el estado de la entidad: en Java, estas operaciones se denominan **métodos**.

Para ilustrar estos conceptos, considérese los objetos de la clase *MensajeDatagrama* presentada en la Figura 5.12 (en el Capítulo 5). Cada objeto instanciado de esta clase contiene tres variables de estado: un mensaje, la dirección del emisor y el número de puerto del emisor. Además, cada objeto contiene tres operaciones: (1) un método *fijaValor*, que permite modificar los valores de las variables de estado, (2) un método *obtieneMensaje*, que permite obtener el valor actual del mensaje, y (3) un método *obtieneDireccion*, que permite obtener la dirección del emisor.

Aunque en este libro se han utilizado objetos en capítulos anteriores, tales como el objeto *MensajeDatagrama*, se trata de objetos **locales**, en lugar de objetos **distribuidos**. Los objetos locales son objetos cuyos métodos sólo se pueden invocar por un **proceso local**, es decir, un proceso que se ejecuta en el mismo computador del objeto. Un objeto distribuido es aquel cuyos métodos pueden invocarse por un **proceso remoto**, es decir, un proceso que se ejecuta en un computador conectado a través de una red al computador en el cual se encuentra el objeto. En un paradigma de objetos distribuidos, los recursos de la red se representan como objetos distribuidos. Para solicitar un servicio de un recurso de red, un proceso invoca uno de sus métodos u operaciones, pasándole los datos como parámetros al método. El método se ejecuta en la máquina remota, y la respuesta es enviada al proceso solicitante como un valor de salida. Comparado con el paradigma de paso de mensajes, el paradigma de objetos distribuidos es **orientado a acciones**: hace hincapié en la invocación de las operaciones, mientras que los datos toman un papel secundario (como parámetros y valores de retorno). Aunque es menos intuitivo para los seres humanos, el paradigma de objetos distribuidos es más natural para el desarrollo de software orientado a objetos.

La Figura 7.1 ilustra el paradigma. Un proceso que se ejecuta en la máquina A realiza una llamada a un método de un objeto distribuido de la máquina B, pasando los

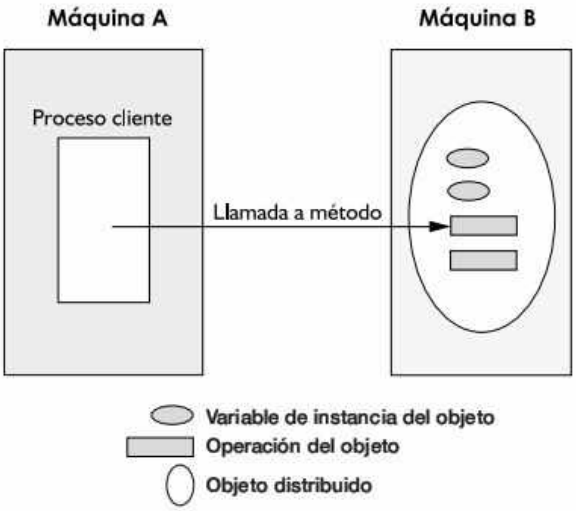


Figura 7.1. El paradigma de objetos distribuidos.

datos necesarios, en caso de existir, mediante argumentos. La llamada al método invoca una acción realizada por el método en la máquina A, y un valor de salida, en caso de que exista, se pasa desde la máquina A a la máquina B. Un proceso que utiliza objetos distribuidos se dice que es un **proceso cliente** de ese objeto, y los métodos del objeto se denominan **métodos remotos** (por contraposición a los métodos locales, o métodos pertenecientes a un objeto local) del proceso cliente.

En el resto del capítulo se va a proceder a presentar una arquitectura genérica que da soporte al paradigma de objetos distribuidos; a continuación se explorará un ejemplo de este tipo de arquitecturas: **Java RMI (*Remote Method Invocation*)**.

7.2. UNA ARQUITECTURA TÍPICA DE OBJETOS DISTRIBUIDOS

La premisa de todo sistema de objetos distribuidos es minimizar las diferencias de programación entre las invocaciones de métodos remotos y las llamadas a métodos locales, de forma que los métodos remotos se puedan invocar en una aplicación utilizando una sintaxis similar a la utilizada en la invocación de los métodos locales. Realmente existen diferencias, porque la invocación de métodos remotos implica una comunicación entre procesos independientes, y por tanto deben tratarse aspectos tales como el empaquetamiento de los datos (*marshaling*), así como la sincronización de los eventos. Estas diferencias quedan ocultas en la arquitectura.

La Figura 7.2 presenta una arquitectura típica de una utilidad que dé soporte al paradigma de objetos distribuidos.

Al objeto distribuido proporcionado o **exportado** por un proceso se le denomina **servidor de objeto**. Otra utilidad, denominada **registro de objetos**, o simplemente **registro**, debe existir en la arquitectura para registrar los objetos distribuidos.

Para acceder a un objeto distribuido, un proceso, el **cliente de objeto**, busca en el registro para encontrar una **referencia** al objeto. El cliente de objeto utiliza esta referencia para realizar llamadas a los métodos del objeto remoto, o **métodos remotos**. Lógicamente, el cliente de objeto realiza una llamada directamente al método remoto. Re-

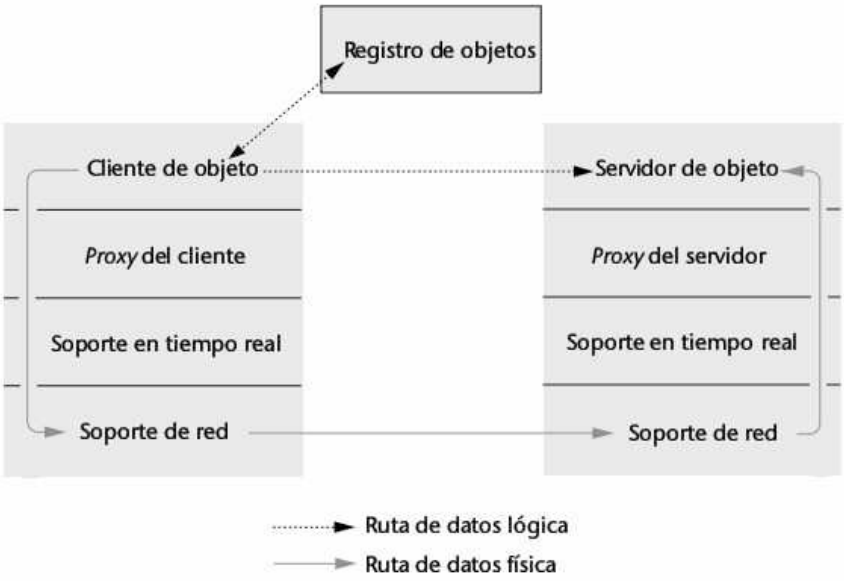


Figura 7.2. Un sistema de objetos distribuidos típico.

En un lenguaje de programación, una referencia es un «manejador» de un objeto; es decir, la representación a través de la cual se puede localizar dicho objeto en el computador donde reside.

El término *proxy*, en el contexto de la computación distribuida, se refiere a un componente software que sirve como intermediario de otros componentes software.

almente, un componente software se encarga de gestionar esta llamada. Este componente se denomina ***proxy* de cliente** y se encarga de interactuar con el software en la máquina cliente con el fin de proporcionar **soporte en tiempo de ejecución** para el sistema de objetos distribuidos. De esta forma, se lleva a cabo la comunicación entre procesos necesaria para transmitir la llamada a la máquina remota, incluyendo el empaquetamiento de los argumentos que se van a transmitir al objeto remoto.

Una arquitectura similar es necesaria en la parte del servidor, donde el soporte en tiempo de ejecución para el sistema de objetos distribuidos gestiona la recepción de los mensajes y el desempaquetado de los datos, enviando la llamada a un componente software denominado ***proxy* de servidor**. El *proxy* de servidor invoca la llamada al método local en el objeto distribuido, pasándole los datos desempaquetados como argumentos. La llamada al método activa la realización de determinadas tareas en el servidor. El resultado de la ejecución del método es empaquetado y enviado por el *proxy* de servidor al *proxy* de cliente, a través del soporte en tiempo de ejecución y el soporte de red de ambas partes de la arquitectura.

7.3. SISTEMAS DE OBJETOS DISTRIBUIDOS

El paradigma de objetos distribuidos se ha adoptado de forma extendida en las aplicaciones distribuidas, para las cuales existe un gran número de herramientas disponibles basadas en este paradigma. Entre las herramientas más conocidas se encuentran:

- Java RMI (*Remote Method Invocation*),
- sistemas basados en CORBA (*Common Object Request Broker Architecture*),
- el modelo de objetos de componentes distribuidos o DCOM (*Distributed Component Object Model*), y
- herramientas y API para el protocolo SOAP (*Simple Object Access Protocol*).

De todas estas herramientas, la más sencilla es Java RMI [java.sun.com/products, 7; java.sun.com/doc, 8; developer.java.sun.com, 9; java.sun.com/marketing, 10], que se describe detalladamente en este capítulo. CORBA [corba.org, 1] y sus implementaciones son detalladas en el Capítulo 9. SOAP [w3.org, 2] es un protocolo basado en la Web y se introducirá en el Capítulo 11, al analizar las aplicaciones basadas en la Web. DCOM [microsoft.com, 3; Grimes, 4] se encuentra fuera del ámbito de este libro; los lectores interesados en DCOM deben consultar las referencias.

No es posible cubrir todas las utilidades de objetos distribuidos existentes, y además seguramente seguirán emergiendo nuevas herramientas que den soporte a este paradigma. Familiarizarse con el API de Java RMI proporciona los fundamentos y prepara al lector para aprender los detalles de utilidades similares.

7.4. LLAMADAS A PROCEDIMIENTOS REMOTOS

RMI tiene su origen en un paradigma denominado **Llamada a procedimientos remotos** o **RPC** (***Remote Procedure Call***).

La **programación procedimental** precede a la programación orientada a objetos. En la programación procedimental, un procedimiento o función es una estructura de control que proporciona la abstracción correspondiente a una acción. La acción de una función se invoca a través de una llamada a función. Para permitir usar diferentes variables, una llamada a función puede ir acompañada de una lista de datos, conocidos como argumentos. El valor o la referencia de cada argumento se le pasa a la función, e incluso podría determinar la acción que realiza dicha función. La llamada a procedimiento convencional es una llamada a un procedimiento que reside en el mismo sistema que el que la invoca y por tanto, se denomina **llamada a procedimiento local**.

En el modelo de llamada a procedimiento remoto, un proceso realiza una llamada a procedimiento de otro proceso, que posiblemente resida en un sistema remoto. Los datos se pasan a través de argumentos. Cuando un proceso recibe una llamada, se ejecuta la acción codificada en el procedimiento. A continuación, se notifica la finalización de la llamada al proceso que invoca la llamada y, si existe un valor de retorno o salida, se le envía a este último proceso desde el proceso invocado. La Figura 7.3 muestra el paradigma RPC.

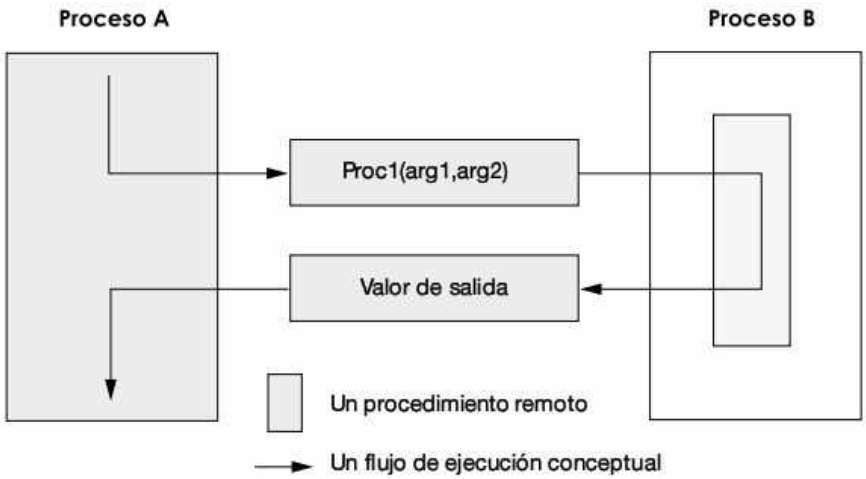
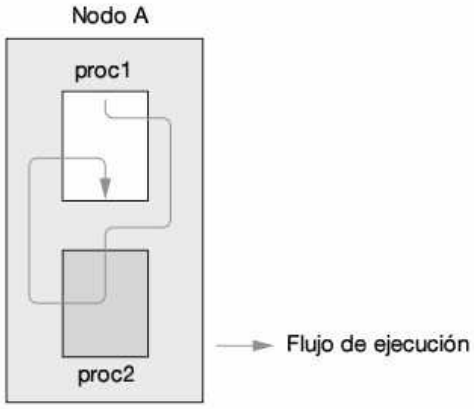


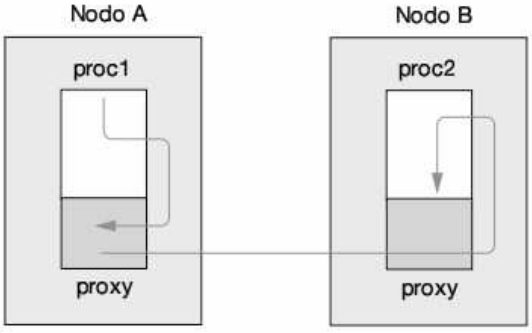
Figura 7.3. El paradigma de llamada a procedimiento remotos.

Basándose en el modelo RPC, han aparecido un gran número de interfaces de programación de aplicaciones. Estas API permiten realizar llamadas a procedimientos remotos utilizando una sintaxis y una semántica similar a las de las llamadas a procedimientos locales. Para enmascarar los detalles de la comunicación entre procesos, cada llamada a procedimiento remoto se transforma mediante una herramienta denominada **rpcgen** en una llamada a procedimiento local dirigida a un módulo software comúnmente denominado **resguardo** (***stub***) o, más formalmente, ***proxy***. Los mensajes que representan la llamada al procedimiento y sus argumentos se pasan a la máquina remota a través del *proxy*.

En el otro extremo, un *proxy* recibe el mensaje y lo transforma en una llamada a procedimiento local al procedimiento remoto. La Figura 7.4 muestra paso a paso la transformación de una llamada a procedimiento remoto en una llamada a procedimiento local y el paso de mensajes necesario.



Llamada a procedimiento local



1. proc1 en el nodo A realiza una llamada a proc2 en nodo B.
2. El soporte en tiempo real convierte la llamada en una llamada al proxy del nodo A.
3. El proxy empaqueta los datos y realiza una llamada IPC al proxy del nodo B.

7. El proxy recibe el valor de salida, desempaqueta los datos y envía el valor de salida a proc2, que continúa su flujo de ejecución.

4. El proxy en el nodo B desempaqueta los datos recibidos y realiza una llamada a proc2.
5. Se ejecuta el código de proc2 y se devuelve el resultado al proxy del nodo B.
6. El proxy empaqueta el valor de salida y realiza una llamada IPC al proxy del nodo A.

Llamada a procedimiento remoto
(no se muestra la ruta de ejecución de vuelta)

Figura 7.4. Llamada a procedimiento local frente a llamada a procedimiento remoto.

Hay que destacar que hay que emplear un *proxy* a cada lado de la comunicación para proporcionar el soporte en tiempo de ejecución necesario para la comunicación entre procesos, llevándose a cabo el correspondiente empaquetado y desempaquetado de datos, así como las llamadas a *sockets* necesarias.

Desde su introducción a principios de los años 80, el modelo RPC se ha utilizado ampliamente en las aplicaciones de red. Existen dos API que prevalecen en este paradigma. La primera, el API ***Open Network Computing Remote Procedure Call*** [ietf.org, 5], es una evolución del API de RPC que desarrolló originalmente Sun Microsystems, Inc., a principios de los años 80. La otra API popular es ***Open Group Distributed Computing Environment*** (DCE) RPC [opennc.org, 6]. Ambas interfaces proporcionan una herramienta, **rpcgen**, para transformar las llamadas a procedimientos remotos en llamadas a procedimientos locales al resguardo.

A pesar de su importancia histórica, este libro no analiza en detalle RPC, por los siguientes motivos:

- RPC, como su nombre indica, es orientado a procedimiento. Las API de RPC emplean una sintaxis que permite realizar llamadas a procedimiento o función. Por tanto, son más adecuadas para programas escritos en un lenguaje procedimental, tal como C. Sin embargo, no son adecuadas para programas escritos en Java, el lenguaje orientado a objetos adoptado en este libro.
- En lugar de RPC, Java proporciona el API RMI, que es orientado a objetos y tiene una sintaxis más accesible que RPC.

7.5. RMI (*REMOTE METHOD INVOCATION*)

RMI es una implementación orientada a objetos del modelo de llamada a procedimientos remotos. Se trata de una API exclusiva para programas Java, aunque debido a su relativa simplicidad, se trata de un buen comienzo para los estudiantes que estén aprendiendo a utilizar objetos distribuidos en aplicaciones de red.

En RMI, un servidor de objeto exporta un objeto remoto y lo registra en un servicio de directorios. El objeto proporciona métodos remotos, que pueden invocar los programas clientes.

Sintácticamente, un objeto remoto se declara como una **interfaz remota**, una extensión de la interfaz Java. El servidor de objeto implementa la interfaz remota. Un cliente de objeto accede al objeto mediante la invocación de sus métodos, utilizando una sintaxis similar a las invocaciones de los métodos locales.

El resto del capítulo se encargará de explorar en detalle el API de Java RMI.

7.6. LA ARQUITECTURA DE JAVA RMI

La Figura 7.5 muestra la arquitectura del API de Java RMI. Al igual que las API de RPC, la arquitectura de Java RMI utiliza módulos de software *proxy* para dar el soporte en tiempo de ejecución necesario para transformar la invocación del método remoto en una llamada a un método local y gestionar los detalles de la comunicación entre procesos subyacente. Cada uno de los extremos de la arquitectura, cliente y servidor, está formado por tres capas de abstracción. A continuación se describen los dos extremos de la arquitectura.

Parte cliente de la arquitectura

1. La **capa resguardo o *stub*.** La invocación de un método remoto por parte de un proceso cliente es dirigida a un objeto *proxy*, conocido como **resguardo**.

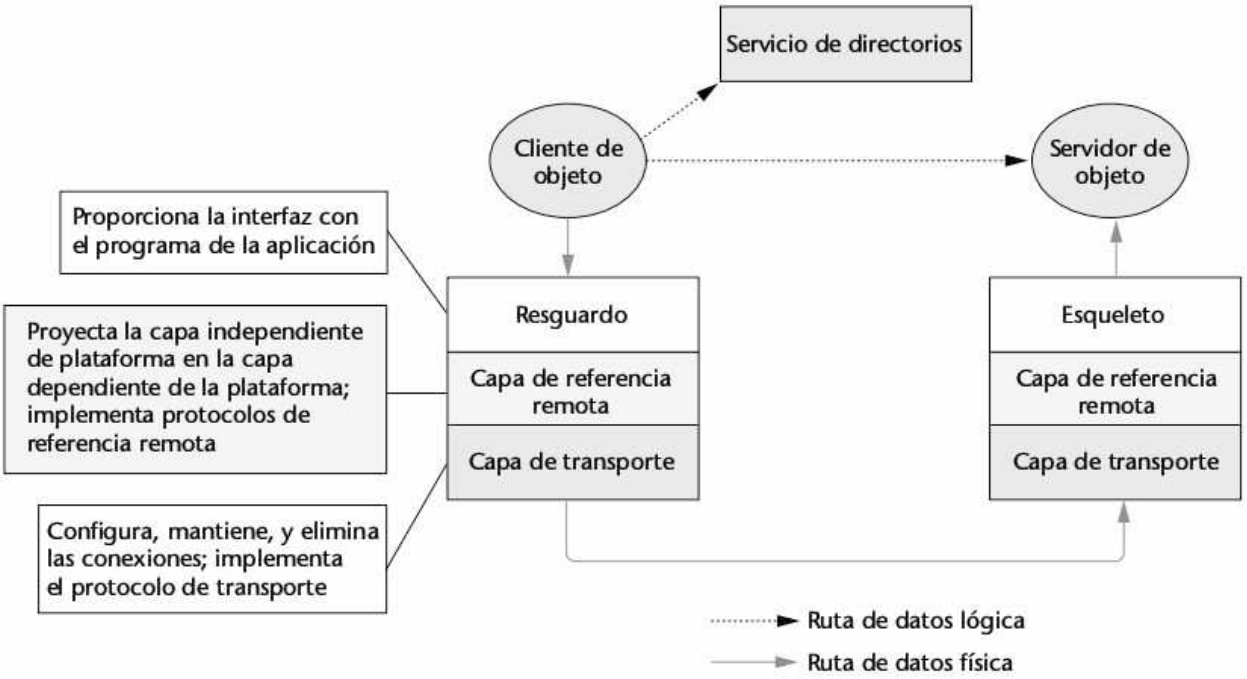


Figura 7.5. La arquitectura de Java RMI.

Esta capa se encuentra debajo de la capa de aplicación y sirve para interceptar las invocaciones de los métodos remotos hechas por los programas clientes; una vez interceptada la invocación es enviada a la capa inmediatamente inferior, la capa de referencia remota.

2. La **capa de referencia remota** interpreta y gestiona las referencias a los objetos de servicio remoto hechas por los clientes, e invoca las operaciones entre procesos de la capa siguiente, la capa de transporte, a fin de transmitir las llamadas a los métodos a la máquina remota.
3. La **capa de transporte** está basada en TCP y por tanto, es orientada a conexión. Esta capa y el resto de la arquitectura se encargan de la conexión entre procesos, transmitiendo los datos que representan la llamada al método a la máquina remota.

Parte servidora de la arquitectura

El esqueleto, el *proxy* de la parte servidora está «*deprecated*», es decir, no se utiliza en las nuevas versiones, a partir de la versión Java 1.2. Su funcionalidad queda reemplazada por el uso de una técnica conocida como «reflexión». Este libro continúa incluyendo el esqueleto en la arquitectura como concepto de la misma.

Conceptualmente, la parte servidora de la arquitectura también está formada por tres capas de abstracción, aunque la implementación varía dependiendo de la versión de Java.

1. La **capa esqueleto o *skeleton*** se encuentra justo debajo de la capa de aplicación y se utiliza para interactuar con la capa resguardo en la parte cliente. Como se cita en [java.sun.com/products, 7],

   «La capa esqueleto se encarga de conversar con la capa resguardo; lee los parámetros de la invocación al método del enlace, realiza la llamada al objeto que implementa el servicio remoto, acepta el valor de retorno, y a continuación devuelve dicho valor al resguardo».

2. La **capa de referencia remota**. Esta capa gestiona y transforma la referencia remota originada por el cliente en una referencia local, que es capaz de comprender la capa esqueleto.
3. La **capa de transporte**. Al igual que en la parte cliente, se trata de una capa de transporte orientada a conexión, es decir, TCP en la arquitectura de red TCP/IP.

Registro de los objetos

El API de RMI hace posible el uso de diferentes servicios de directorios para registrar un objeto distribuido. Uno de estos servicios de directorios es la **interfaz de nombrado y directorios de Java** (JNDI, *Java Naming and Directory Interface)*, que es más general que el registro RMI que se utilizará en este capítulo, en el sentido de que lo pueden utilizar aplicaciones que no usan el API RMI. El registro RMI, *rmiregistry*, es un servicio de directorios sencillo proporcionado por el *kit* de desarrollo de software Java (SDK, *Java Software Development Kit*). El registro RMI es un servicio cuyo servidor, cuando está activo, se ejecuta en la **máquina del servidor del objeto**. Por convención, utiliza el puerto TCP 1099 por defecto.

El SDK de Java se puede instalar en una máquina local. Permite el uso de bibliotecas de clases Java y herramientas, tales como el compilador de Java *javac*.

Lógicamente, desde el punto de vista del desarrollador de software, las invocaciones a métodos remotos realizadas en un programa cliente interactúan directamente con los objetos remotos en un programa servidor, de la misma forma que una llamada a un método local interactúa con un objeto local. Físicamente, las invocaciones del método remoto se transforman en llamadas a los resguardos y esqueletos en tiempo de ejecución, dando lugar a la transmisión de datos a través de la red. La Figura 7.6 es un diagrama de tiempos y eventos que describe la interacción entre el resguardo y el esqueleto.

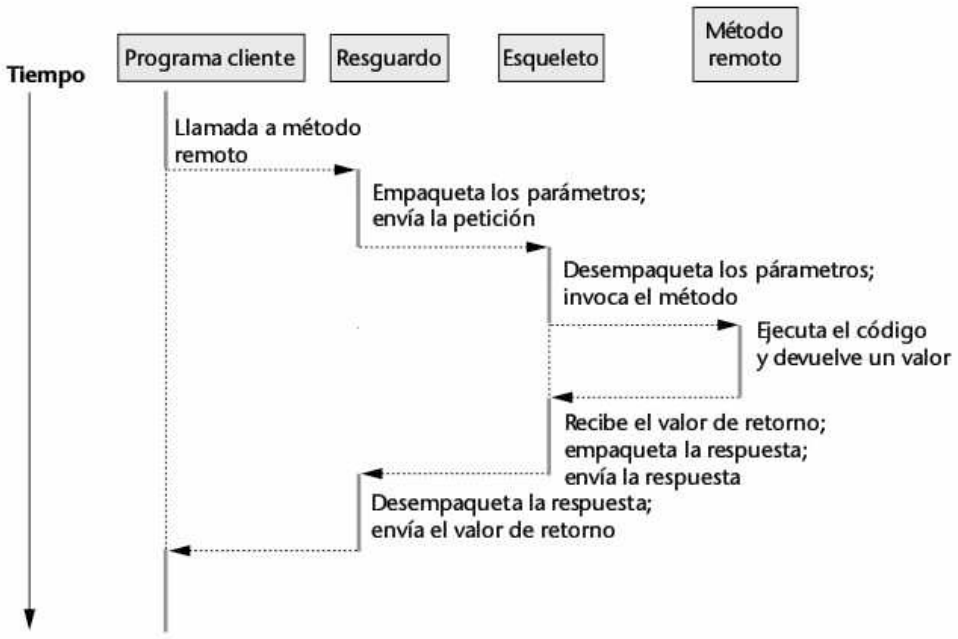


Figura 7.6. Interacciones entre el resguardo RMI y el esqueleto RMI (basado en un diagrama de [java.sun.com/docs, 8]).

7.7. API DE JAVA RMI

En esta sección se introduce un subconjunto del API de Java RMI. (Por simplicidad, la presentación de este capítulo no cubre los **gestores de seguridad**, que es muy recomendable utilizarlos en todas las aplicaciones RMI. Los gestores de seguridad se explican en la Sección 8.3 del próximo capítulo.) Esta sección cubre las siguientes áreas: la **interfaz remota**, el **software de la parte servidora** y el **software de la parte cliente**.

La interfaz remota

En el API de RMI, el punto inicial para crear un objeto distribuido es una **interfaz remota** Java. Una interfaz Java es una clase que se utiliza como plantilla para otras clases: contiene las declaraciones de los métodos que deben implementar las clases que utilizan dicha interfaz.

Una interfaz remota Java es una interfaz que hereda de la clase Java ***remote***, que permite implementar la interfaz utilizando sintaxis RMI. Aparte de la extensión que se hace de la clase *remote* y de que todas las declaraciones de los métodos deben especificar la excepción *RemoteException*, una interfaz remota utiliza la misma sintaxis que una interfaz Java local. A fin de mostrar la sintaxis básica, la Figura 7.7 muestra un ejemplo de interfaz remota.

Figura 7.7. Un ejemplo de interfaz remota Java.

```
 1  // fichero: InterfazEjemplo.java
 2  // implementada por una clase servidor Java RMI.
 3
 4  import java.rmi.*
 5
 6  public interface InterfazEjemplo extends Remote {
 7    // cabecera del primer método remoto
 8    public String metodoEj1( )
 9      throws java.rmi.RemoteException;
10    // cabecera del segundo método remoto
11    public int metodoEj2(float parametro)
12       throws java.rmi.RemoteException;
13    // cabeceras de otros métodos remotos
14 } // fin interfaz
```

En este ejemplo, se declara una interfaz denominada *InterfazEjemplo*. La interfaz extiende o hereda la clase Java *remote* (línea 6), convirtiéndose de este modo en una interfaz remota.

Dentro del bloque que se encuentra entre las llaves (líneas 6-14) se encuentran las declaraciones de los dos **métodos remotos**, que se llaman *metodoEj1* (líneas 8-9) y *metodoEj2* (líneas 11-12), respectivamente.

Como se puede observar, *metodoEj1* no requiere argumentos (de ahí la lista de parámetros vacía detrás del nombre del método) y devuelve un objeto de tipo *String*. El método *metodoEj2* requiere un argumento de tipo *float* y devuelve un valor de tipo *int*.

Obsérvese que un objeto *serializable*, tal como un objeto *String* o un objeto de otra clase, puede ser un argumento o puede ser devuelto por un método remoto. Al

método remoto se le pasa una copia del elemento (específicamente, una copia del objeto), sea éste un tipo primitivo o un objeto. El valor devuelto se gestiona de la misma forma, pero en la dirección contraria.

Un objeto *serializable* es un objeto de una clase que puede «aplanarse», de forma que se puede empaquetar para su transmisión a través de la red.

Cada declaración de un método debe especificar la excepción *java.rmi.RemoteException* en la sentencia *throws* (líneas 9 y 12). Cuando ocurre un error durante el procesamiento de la invocación del método remoto, se lanza una excepción de este tipo, que debe ser gestionada en el programa del método que lo invoca. Las causas que originan este tipo de excepción incluyen los errores que pueden ocurrir durante la comunicación entre los procesos, tal como fallos de acceso y fallos de conexión, así como problemas asociados exclusivamente a la invocación de métodos remotos, como por ejemplo no encontrar el objeto, el resguardo o el esqueleto.

Software de la parte servidora

Un servidor de objeto es un objeto que proporciona los métodos y la interfaz de un objeto distribuido. Cada servidor de objeto debe (1) implementar cada uno de los métodos remotos especificados en la interfaz, y (2) registrar en un servicio de directorios un objeto que contiene la implementación. Se recomienda que las dos partes se realicen en clases separadas, como se ilustra a continuación.

La implementación de la interfaz remota

Se debe crear una clase que implemente la interfaz remota. La sintaxis es similar a una clase que implementa una interfaz local. La Figura 7.8 muestra un esquema o plantilla de la implementación.

Figura 7.8. Sintaxis de un ejemplo de implementación de interfaz remota.

```
1   import java.rmi.*;
2   import java.rmi.server.*;
3
4   /**
5   Esta clase implementa la interfaz remota InterfazEjemplo.
6   */
7
8   public class ImplEjemplo extends UnicastRemoteObject
9      implements InterfazEjemplo {
10
11     public ImplEjemplo( ) throws RemoteException {
12        super( );
13     }
14
15     public metodoEj1() throws RemoteException {
16        // código del método
17     }
18
19     public metodoEj2() throws RemoteException {
20        // código del método
21     }
22
23  } // fin clase
```

Las sentencias de importación (*import*) (líneas 1-2) son necesarias para que el código pueda utilizar las clases *UnicastRemoteObject* y *RemoteException*.

La cabecera de la clase (línea 8) debe especificar (1) que es una subclase de la clase Java *UnicastRemoteObject*, y (2) que implementa una interfaz remota específica, llamada *InterfazEjemplo* en la plantilla. (Nota: Un objeto *UnicastRemoteObject* da soporte a RMI *unicast*, es decir, RMI utilizando comunicación unidifusión entre procesos. Presumiblemente, se puede implementar una clase *MulticastRemoteObject*, que dé soporte a RMI con comunicación multidifusión.)

Se debe definir un constructor de la clase (líneas 11-13). La primera línea del código debe ser una sentencia (la llamada *super()*) que invoque al constructor de la clase base. Puede aparecer código adicional en el constructor si se necesita.

A continuación, debe aparecer la implementación de cada método remoto (líneas 15-21). La cabecera de cada método debe coincidir con la cabecera de dicho método en el fichero de la interfaz.

La Figura 7.9 muestra un diagrama UML para la clase *ImplEjemplo*.

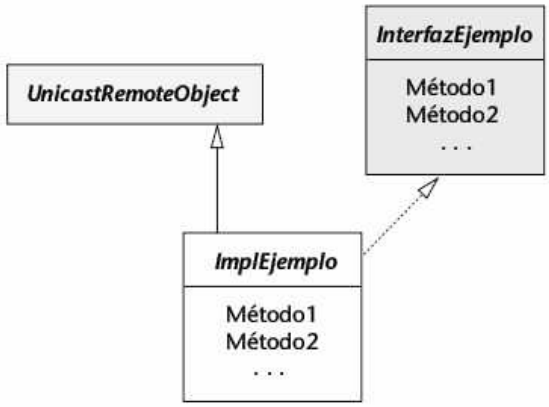


Figura 7.9. El diagrama de clases UML para *ImplEjemplo*.

Generación del resguardo y del esqueleto

En RMI, un objeto distribuido requiere un *proxy* por cada uno de los servidores y clientes de objeto, conocidos como esqueleto y resguardo del objeto, respectivamente. Estos *proxies* se generan a partir de la implementación de una interfaz remota utilizando una herramienta del SDK de Java: el compilador RMI ***rmic***. Para utilizar esta herramienta, se debe ejecutar el siguiente mandato en una interfaz de mandatos UNIX o Windows:

```
rmic <nombre de la clase de la implementación de la interfaz remota>
```

Por ejemplo:

```
rmic ImplEjemplo
```

Si la compilación se realiza de forma correcta, se generan dos ficheros *proxy*, cada uno de ellos con el prefijo correspondiente al nombre de la clase de la implementación. Por ejemplo, *ImplEjemplo_skel.class* y *ImplEjemplo_stub.class*.

El fichero del resguardo para el objeto, así como el fichero de la interfaz remota deben compartirse con cada cliente de objeto: estos ficheros son imprescindibles para que el programa cliente pueda compilar correctamente. Una copia de cada fichero debe colocarse manualmente en la parte cliente (es decir, debe colocarse una copia del fichero en un directorio apropiado del cliente). Adicionalmente, Java RMI dispone de una característica denominada **descarga de resguardo**, que consiste en que el cliente obtiene de forma dinámica el fichero de resguardo. Esta característica se estudiará en el Capítulo 8, donde se analizan varios temas avanzados de RMI.

El servidor de objeto

La clase del servidor de objeto instancia y exporta un objeto de la implementación de la interfaz remota. La Figura 7.10 muestra una plantilla para la clase del servidor de objeto.

Figura 7.10. Sintaxis de un ejemplo de un servidor de objeto.

```
1   import java.rmi.*;
2   import java.rmi.server.*;
3   import java.rmi.registry.Registry;
4   import java.rmi.registry.LocateRegistry;
5   import java.net.*;
6   import java.io.*;
7
8   /**
9   * Esta clase representa el servidor de un objeto
10  * distribuido de la clase ImplEjemplo, que implementa la
11  * interfaz remota InterfazEjemplo.
12  */
13
14  public class ServidorEjemplo {
15     public static void main(String args[]) {
16        String numPuertoRMI, URLRegistro;
17        try{
18           // código que permite obtener el valor del número de puerto
19           ImplEjemplo objExportado = new ImplEjemplo();
20           arrancarRegistro(numPuertoRMI);
21           // registrar el objeto bajo el nombre "ejemplo"
22           URLRegistro = "rmi://localhost:" + numPuerto + "/ejemplo";
23           Naming.rebind(URLRegistro, objExportado);
24           System.out.println("Servidor ejemplo preparado.");
25        } // fin try
26        catch (Exception excr) {
27           System.out.printl(
28           "Excepción en ServidorEjemplo.main: " + excr);
29        } // fin catch
30     } // fin main
31
32     // Este método arranca un registro RMI en la máquina
33     //local, si no existe en el número de puerto especificado
```

(continúa)

```
34      private static void arrancarRegistro (int numPuertoRMI)
35              throws RemoteException {
36          try {
37              Registry registro = LocateRegistry.getRegistry(numPuertoRMI);
38              registro.list();
39              // El método anterior lanza una excepción
40              // si el registro no existe.
41          }
42          catch (RemoteException exc) {
43              //No existe un registro válido en este puerto.
44              System.out.println(
45                 "El registro RMI no se puede localizar en el puerto:
46                 + RMIPortNum);
47              Registry registro =LocateRegistry.createRegistry(numPuertoRMI);
48              System.out.println(
49                 "Registro RMI creado en el puerto " + RMIPortNum);
50        } // fin catch
51      } // fin arrancarRegistro
52
53    } // fin clase
```

En los siguientes párrafos se analizan las diferentes partes de esta plantilla.

Creación de un objeto de la implementación de la interfaz remota. En la línea 19, se crea un objeto de la clase que **implementa** la interfaz remota; a continuación, se **exportará** la referencia a este objeto.

Exportación del objeto. Las líneas 20-23 de la plantilla exportan el objeto. Para exportar el objeto, se debe registrar su referencia en un servicio de directorios. Como ya se ha mencionado anteriormente, en este capítulo se utilizará el servicio *rmiregistry* del SDK Java. Un servidor *rmiregistry* debe ejecutarse en el nodo del servidor de objeto para poder registrar objetos RMI.

Cada registro RMI mantiene una lista de objetos exportados y posee una interfaz para la búsqueda de estos objetos. Todos los servidores de objetos que se ejecutan en la misma máquina pueden compartir un mismo registro. Alternativamente, un proceso servidor individual puede crear y utilizar su propio registro si lo desea, en cuyo caso múltiples servidores *rmiregistry* pueden ejecutarse en diferentes números de puertos en la misma máquina, cada uno con una lista diferente de objetos exportados.

Una colisión de nombres se produce cuando se intenta exportar un objeto cuyo nombre coincide con el nombre de otro objeto ya existente en el registro.

En un sistema de producción, debería existir un servidor *rmiregistry*, ejecutando de forma continua, presumiblemente en el puerto por defecto 1099. Para los ejemplos de este libro, se supone que existe un registro RMI siempre disponible, aunque se utiliza código para arrancar una copia del servidor bajo demanda en un puerto de la elección del lector, de forma que cada estudiante pueda utilizar una copia diferente del registro para sus experimentos y no existan colisiones de nombres.

En la plantilla del servidor de objeto, se implementa el método estático *arrancarRegistro()* (líneas 34-51), que arranca un servidor de registro RMI si no está actualmente en ejecución, en un número de puerto especificado por el usuario (línea 20):

```
arrancarRegistro(numPuertoRMI);
```

En un sistema de producción donde se utilice el servidor de registro RMI por defecto y esté ejecutando continuamente, la llamada *arrancarRegistro* y, por tanto, el método *arrancarRegistro*, puede omitirse.

En la plantilla del servidor de objeto, el código para exportar un objeto (líneas 22-23) se realiza del siguiente modo:

```
// registrar el objeto con el nombre "ejemplo"
URLRegistro = "rmi://localhost:" + numPuerto + "/ejemplo";
Naming.rebind(URLRegistro, objExportado);
```

La clase *Naming* proporciona métodos para almacenar y obtener referencias del registro. En particular, el método *rebind* permite almacenar en el registro una referencia a un objeto con un URL de la forma:

```
rmi://<nombre máquina>:<número puerto>/<nombre referencia>
```

El método *rebind* sobreescribe cualquier referencia en el registro asociada al nombre de la referencia. Si no se desea sobreescribir, existe un método denominado *bind*.

El nombre de la máquina debe corresponder con el nombre del servidor, o simplemente se puede utilizar «*localhost*». El nombre de la referencia es un nombre elegido por el programador y debe ser único en el registro.

El código del ejemplo comprueba primero si se está ejecutando actualmente un registro RMI en el puerto por defecto. Si no es así, se activa un registro RMI.

Alternativamente, se puede activar un registro RMI manualmente utilizando la utilidad *rmiregistry*, que se encuentra en el SDK, a través de la ejecución del siguiente mandato en el intérprete de mandatos:

```
rmiregistry <número puerto>
```

donde el número de puerto es un número de puerto TCP. Si no se especifica ningún puerto, se utiliza el puerto por defecto 1099.

Cuando se ejecuta un servidor de objeto, la exportación de los objetos distribuidos provoca que el proceso servidor comience a escuchar por el puerto y espere a que los clientes se conecten y soliciten el servicio del objeto. Un servidor de objeto RMI es un servidor concurrente: cada solicitud de un cliente de objeto se procesa a través de un hilo independiente del servidor. Dado que las invocaciones de los métodos remotos se pueden ejecutar de forma concurrente, es importante que la implementación de un objeto remoto sea *thread-safe*. Los lectores pueden revisar este tema en «Programación concurrente» en el Capítulo 1.

Software de la parte cliente

La clase cliente es como cualquier otra clase Java. La sintaxis necesaria para hacer uso de RMI supone localizar el registro RMI en el nodo servidor y buscar la referencia remota para el servidor de objeto; a continuación se realizará un *cast* de la referencia a la clase de la interfaz remota y se invocarán los métodos remotos. La Figura 7.11 presenta una plantilla para el cliente de objeto.

Figura 7.11. Plantilla para un cliente de objeto.

```
1  import java.io.*;
2  import java.rmi.*;
3  import java.rmi.registry.Registry;
4  import java.rmi.registry.LocateRegistry;
5
6  /**
```

(continúa)

```
 7  * Esta clase representa el cliente de un objeto
 8  * distribuido de la clase ImplEjemplo, que implementa la
 9  * interfaz remota InterfazEjemplo.
10  */
11
12  public class ClienteEjemplo {
13    public static void main(String args[ ]) {
14      try {
15        int puertoRMI;
16        String nombreNodo;
17        String numPuerto;
18        // Código que permite obtener el nombre del nodo y
19        el número de puerto del registro
20
21       // Búsqueda del objeto remoto y cast de la
22       // referencia con la correspondiente clase
23       // de la interfaz remota – reemplazar "localhost por el
            nombre del nodo del objeto remoto.
24       String URLRegistro =
25         "rmi://localhost:" + numPuerto + "/ejemplo";
26       InterfazEjemplo h =
27         (InterfazEjemplo) Naming.lookup(URLRegistro);
28       // invocar el o los métodos remotos
29       String mensaje = h.metodoEj1();
30       System.out.println(mensaje);
31       // el método metodoEj2 puede invocarse del mismo modo
32      }// fin try
33      catch (Exception exc) {
34        exc.printStackTrace();
35       } // fin catch
36     } // fin main
37    // Posible definición de otros métodos de la clase
38  } // fin clase
```

Las sentencias de importación. Las sentencias de importación (líneas 1-4) se necesitan para que el programa pueda compilar.

Búsqueda del objeto remoto. El código entre las líneas 24 y 27 permite buscar el objeto remoto en el registro. El método *lookup* de la clase *Naming* se utiliza para obtener la referencia al objeto, si existe, que previamente ha almacenado en el registro el servidor de objeto. Obsérvese que se debe hacer un *cast* de la referencia obtenida a la clase de la interfaz remota (no a su implementación).

```
String URLRegistro = "rmi://localhost:" + numPuerto + "/ejemplo";
InterfazEjemplo h = (InterfazEjemplo)Naming.lookup(URLRegistro);
```

Invocación del método remoto. Se utiliza la referencia a la interfaz remota para invocar cualquiera de los métodos de dicha interfaz, como se muestra en las líneas 29-30 del ejemplo:

```
String mensaje = h.metodoEj1();
System.out.println(mensaje);
```

Obsérvese que la sintaxis utilizada para la invocación de los métodos remotos es igual que la utilizada para invocar métodos locales.

7.8. UNA APLICACIÓN RMI DE EJEMPLO

Desde la Figura 7.12 hasta la Figura 7.15 se muestra la lista completa de ficheros necesarios para crear la aplicación RMI *HolaMundo*. El servidor exporta un objeto que contiene un único método, denominado *decirHola*. Se recomienda al lector identificar en el código las diferentes partes que se han descrito en la sección anterior.

Figura 7.12. *HolaMundoInt.java.*

```
1  // Un ejemplo sencillo de interfaz RMI - M. Liu
2  import java.rmi.*;
3
4  /**
5  * Interfaz remota.
6  * @author M. L. Liu
7  */
8
9  public interface HolaMundoInt extends Remote {
10 /**
11 * Este método remoto devuelve un mensaje.
12 * @para name – una cadena de caracteres con un nombre.
13 * @return - una cadena de caracteres.
14 */
15    public String decirHola(String nombre)
16       throws java.rmi.RemoteException;
17
18    }
```

Figura 7.13. *HolaMundoImpl.java.*

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3
4  /**
5  * Esta clase implementa la interfaz remota
6  * HolaMundoInt.
7  * @author M. L. Liu
8  */
9
10 public class HolaMundoImpl extends UnicastRemoteObject
11 implements HolaMundoInt {
12
13    public HolaMundoImpl() throws RemoteException {
14       super();
15    }
16
```

(continúa)

```
17      public String decirHola(String nombre)
18           throws RemoteException {
19         return "Hola mundo" + nombre;
20      }
21   } //fin clase
```

Figura 7.14. *HolaMundoServidor.java.*

```
 1   import java.rmi.*;
 2   import java.rmi.server.*;
 3   import java.rmi.registry.Registry;
 4   import java.rmi.registry.LocateRegistry;
 5   import java.net.*;
 6   import java.io.*;
 7
 8   /**
 9   * Esta clase representa el servidor de un objeto
10   * de la clase HolaMundo, que implementa la
11   * interfaz remota HolaMundoInterfaz.
12   * @author M. L. Liu
13   */
14
15   public class HolaMundoServidor {
16        public static void main (String args[]) {
17           InputStreamReader ent = new InputStreamReader(System.in));
18           BufferedReader buf = new BufferedReader(ent);
19           String numPuerto, URLRegistro;
20           try {
21              System.out.println("Introducir el número de puerto del
                 registro RMI:");
22              numPuerto = buf.readLine().trim();
23              int numPuertoRMI = Integer.parseInt(numPuerto);
24              arrancarRegistro(numPuertoRMI);
25              HolaMundoImpl objExportado = new HolaMundoImpl();
26              URLRegistro = "rmi://localhost:"+ numPuerto + "/holaMundo";
27              Naming.rebind(URLRegistro, objExportado);
28      /**/    System.out.println
29      /**/    ("Servidor registrado. El registro contiene actualmente:");
30      /**/    // lista de los nombres que se encuentran en el registro
                   actualmente
31      /* */   listaRegistro(URLRegistro);
32              System.out.println("Servidor HolaMundo preparado.");
33           } // fin try
34           catch (Exception excr) {
35              System.out.println("Excepción en HolaMundoServidor.main:
                 " + excr);
36           } // fin catch
37        } // fin main
38
```

(continúa)

```
39        // Este método arranca un registro RMI en la máquina
40        // local, si no existe en el número de puerto especificado.
41        private static void arrancarRegistro(int numPuertoRMI)
42             throws RemoteException {
43          try {
44            Registry registro = LocateRegistry.getRegistry(numPuertoRMI);
45            registro.list(); // Esta llamada lanza
46            // una excepción si el registro no existe
47          }
48          catch (RemoteException e) {
49            // Registro no válido en este puerto
50  /**/      System.out.println
51  /**/        ("El registro RMI no se puede localizar en el puerto "
52  /**/        + numPuertoRMI);
53            Registry registro =
54              LocateRegistry.createRegistry(numPuertoRMI);
55  /**/      System.out.println(
56  /**/        "Registro RMI creado en el puerto " + numPuertoRMI);
57          } // fin catch
58      } // fin arrancarRegistro
59
60      // Este método lista los nombres registrados con un objeto Registry
61      private static void listaRegistro(String URLRegistro)
62           throws RemoteException, MalformedURLException {
63        System.out.println("Registro " + URLRegistro * " contiene: ");
64        String [] nombres = Naming.list(URLRegistro);
65        for (int i=0; i<nombres.length; i++)
66          System.out.println(names[i]);
67      } // fin listaRegistro
68
69  } // fin clase
```

Figura 7.15. *HolaMundoCliente.java.*

```
1   import java.io.*;
2   import java.rmi.*;
3
4   /**
5   * Esta clase representa el cliente de un objeto
6   * distribuido de clase HolaMundo, que implementa la
7   * interfaz remota HolaMundoInterfaz.
8   * @author M. L. Liu
9   */
10
11  public class HolaMundoCliente {
12
13    public static void main(String args[]) {
14      try {
```

(continúa)

```
15          int numPuertoRMI;
16          String nombreNodo;
17          InputStreamReader ent = new InputStreamReader(System.in);
18          BufferedReader buf = new BufferedReader(ent);
19          System.out.println("Introducir el nombre del nodo del
            registro RMI:");
20          nombreNodo = buf.readLine();
21          System.out.println("Introducir el numero de puerto del
            registro RMI:");
22          String numPuerto = buf.readLine();
23          numPuertoRMI = Integer.parseInt(numPuerto);
24       String URLRegistro =
25          "rmi://" + nombreNodo + ":" + numPuerto + "/holaMundo";
26       // Busqueda del objeto remoto y cast del objeto de la interfaz
27          HolaMundoInterfaz h =
28             (HolaMundoInterfaz)Naming.lookup(URLRegistro);
29          System.out.println("Busqueda completa");
30          // Invocar el método remoto
31          String mensaje = h.decirHola("Pato Donald");
32          System.out.println("HolaMundoCliente: " + mensaje);
33       } // fin try
34       catch (Exception e) {
35          System.out.println("Excepcion en HolaMundoCliente: " + e);
36       } // fin catch
37    } // fin main
38 } // fin clase
```

Una vez comprendida la estructura básica del ejemplo de aplicación RMI presentado, el lector debería ser capaz de utilizar esta sintaxis como plantilla para construir cualquier aplicación RMI, modificando la presentación y la lógica de la aplicación; la lógica del servicio (utilizando RMI) queda inalterada.

La tecnología RMI es un buen candidato como componente software en la capa de servicio. Un ejemplo de aplicación industrial es un sistema de informe de gastos de una empresa, que se muestra en la Figura 7.16 y que se describe en [java.sun.com/marketing, 8]. En la aplicación mostrada, el servidor de objeto proporciona métodos remotos que permiten a los clientes de objeto buscar o actualizar los datos en una base de datos de gastos. Los programas clientes del objeto proporcionan la lógica de aplicación o negocio necesaria para procesar los datos y la lógica de presentación para la interfaz de usuario.

Java RMI posee un gran número de características. Este capítulo ha presentado un conjunto muy básico de estas características, como ejemplo de un sistema de objetos distribuidos. Algunas de las características avanzadas de RMI más interesantes se describirán en el siguiente capítulo.

7.9. PASOS PARA CONSTRUIR UNA APLICACIÓN RMI

Una vez vistos algunos de los aspectos del API de RMI, se va a pasar a describir paso a paso el procedimiento para construir una aplicación RMI, de forma que el lector

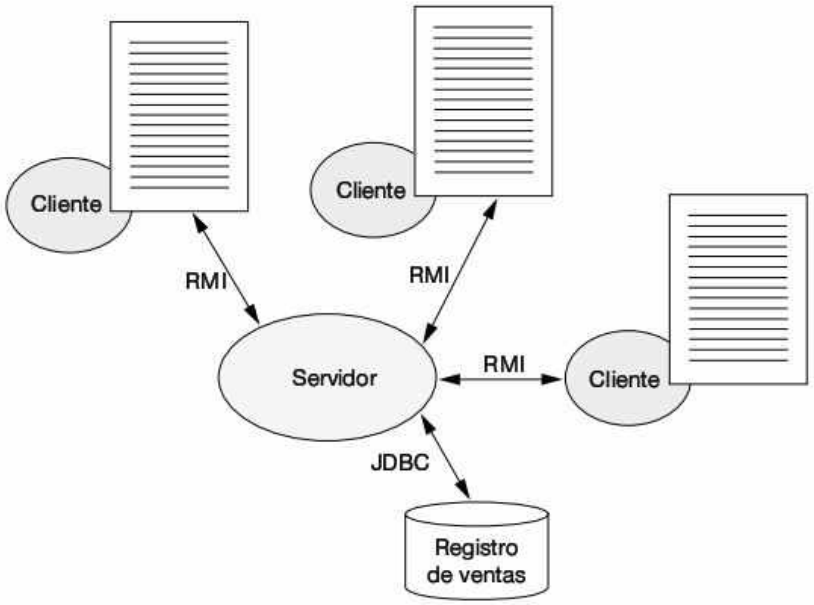


Figura 7.16. Un ejemplo de aplicación RMI.

pueda experimentar con dicho paradigma. Se describe tanto la parte del servidor de objeto como del cliente de objeto. Hay que tener en cuenta que en un entorno de producción es probable que el desarrollo de software de ambas partes sea independiente.

El algoritmo es expresado en términos de una aplicación denominada *Ejemplo*. Los pasos se aplicarán para la construcción de cualquier aplicación, reemplazando el nombre *Ejemplo* por el nombre de la aplicación.

Algoritmo para desarrollar el software de la parte servidora

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota del servidor en *InterfazEjemplo.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
3. Implementar la interfaz en *ImplEjemplo.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
4. Utilizar el compilador de RMI **rmic** para procesar la clase de la implementación y generar los ficheros de resguardo y esqueleto para el objeto remoto:

   ```
   rmic ImplEjemplo
   ```

   Los ficheros generados se encontrarán en el directorio como *ImplEjemplo_Skel.class* e *ImplEjemplo_Stub.class*. Se deben repetir los pasos 3 y 4 cada vez que se realice un cambio a la implementación de la interfaz.
5. Crear el programa del servidor de objeto *ServidorEjemplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
6. Activar el servidor de objeto.

   ```
   java ServidorEjemplo
   ```

Algoritmo para desarrollar el software de la parte cliente

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Obtener una copia del fichero *class* de la interfaz remota. Alternativamente, obtener una copia del fichero fuente de la interfaz remota y compilarlo utilizando ***javac*** para generar el fichero *class* de la interfaz.
3. Obtener una copia del fichero de resguardo para la implementación de la interfaz, *ImplEjemplo_Stub.class*.
4. Desarrollar el programa cliente *ClienteEjemplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
5. Activar el cliente.

```
java ClienteEjemplo
```

La Figura 7.17 muestra la colocación de los ficheros de una aplicación en las partes cliente y servidora. Los ficheros class de la interfaz remota y del resguardo para cada objeto remoto deben estar en la máquina donde se encuentra el cliente de objeto, junto con la clase del cliente de objeto. En la parte servidora se deben encontrar los fichero class de la interfaz, del servidor de objeto, de la implementación de la interfaz y del esqueleto para el objeto remoto.

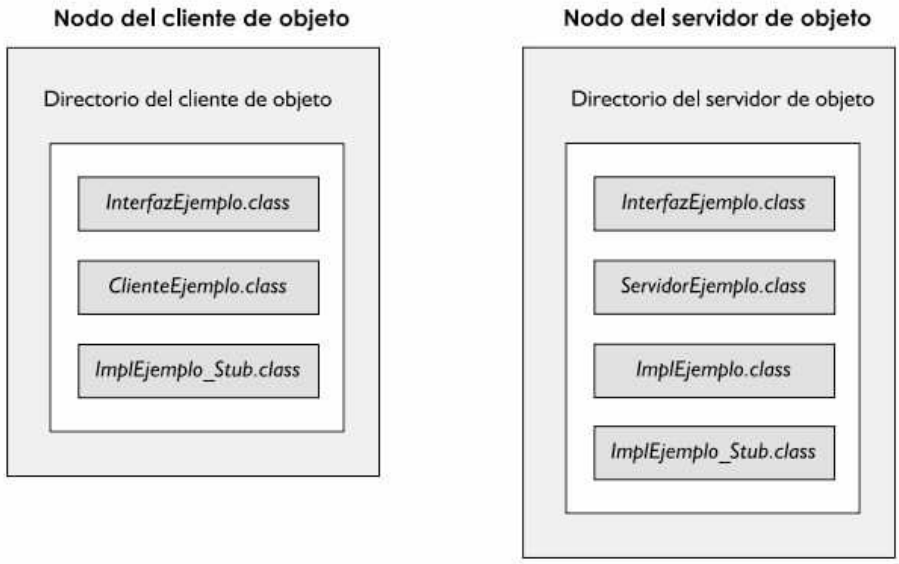


Figura 7.17. Colocación de los ficheros en una aplicación RMI.

7.10. PRUEBAS Y DEPURACIÓN

Como cualquier tipo de programación de red, las pruebas y depuración de los procesos concurrentes no son triviales. Es recomendable utilizar los siguientes pasos incrementales a la hora de desarrollar una aplicación RMI:

1. Construir una plantilla para un programa RMI básico. Empezar con una interfaz remota que sólo contenga la declaración de un método, su implementación uti-

lizando un resguardo, un programa servidor que exporte el objeto y un programa cliente con código que sólo invoque al método remoto. Probar la plantilla en una máquina hasta que se pueda ejecutar correctamente el método remoto.

2. Añadir una declaración cada vez a la interfaz. Con cada adición, modificar el programa cliente para que invoque al método que se ha añadido.
3. Rellenar la definición de cada método remoto uno a uno. Probar y depurar de forma detallada cada método añadido antes de incluir el siguiente.
4. Después de que todos los métodos remotos se han probado detalladamente, crear la aplicación cliente utilizando una técnica incremental. Con cada incremento, probar y depurar los programas.
5. Distribuir los programas en máquinas separadas. Probarlos y depurarlos.

7.11. COMPARACIÓN ENTRE RMI Y EL API DE *SOCKETS*

El API de RMI, como API representativa del paradigma de objetos distribuidos, es una herramienta eficiente para construir aplicaciones de red. Puede utilizarse en lugar del API de *sockets* (que representa el paradigma de paso de mensajes) para construir una aplicación de red rápidamente. Sin embargo, debería tenerse en cuenta que además de ventajas, también existen desventajas en esta opción.

Algunas de estas ventajas y desventajas se enumeran a continuación:

- El API de *sockets* está más cercano al sistema operativo, por lo que tiene menos sobrecarga de ejecución. RMI requiere soporte software adicional, incluyendo los *proxies* y el servicio de directorio, que inevitablemente implican una sobrecarga en tiempo de ejecución. Para aplicaciones que requieran alto rendimiento, el API de *sockets* puede ser la única solución viable.
- Por otro lado, el API de RMI proporciona la abstracción necesaria para facilitar el desarrollo de software. Los programas desarrollados con un nivel más alto de abstracción son más comprensibles y por tanto más sencillos de depurar.
- Debido a que el API de *sockets* opera a más bajo nivel, se trata de una API típicamente independiente de plataforma y lenguaje. Puede no ocurrir lo mismo con RMI. Java RMI, por ejemplo, requiere soportes de tiempo de ejecución específicos de Java. Como resultado, una aplicación implementada con Java RMI debe escribirse en Java y sólo se puede ejecutar en plataformas Java.

La elección de un paradigma y una API apropiados es una decisión clave en el diseño de una aplicación. Dependiendo de las circunstancias, es posible que algunas partes de la aplicación utilicen un paradigma o API y otras partes otro.

En ingeniería de software, un prototipo es una versión inicial que se realiza de forma rápida para mostrar la interfaz de usuario y las funciones de una aplicación propuesta.

Debido a la relativa facilidad con la que las aplicaciones de red pueden desarrollarse utilizando RMI, RMI es un buen candidato para el desarrollo rápido de un prototipo de una aplicación.

7.12. PARA PENSAR

El modelo de computación distribuida orientada a objetos presentado en este capítulo está basado en la visión de que, desde el punto de vista del programador, no hay una distinción esencial entre los objetos locales y los objetos remotos. Aunque esta visión es ampliamente aceptada como la base de los sistemas de objetos distribuidos,

existen detractores de la misma. Los autores de [research.sun.com, 11], por ejemplo, afirman que esta visión, aunque conveniente, es inapropiada porque ignora las diferencias inherentes entre los objetos locales y remotos. El Ejercicio 11 al final del capítulo pide al lector profundizar más en el estudio de este argumento.

RESUMEN

Este capítulo ha introducido el paradigma de objetos distribuidos. A continuación se resumen algunos de los puntos claves:

- El paradigma de objetos distribuidos posee un nivel de abstracción más alto que el paradigma de paso de mensajes.
- Mediante el uso de este paradigma, un proceso invoca métodos de un objeto remoto, pasando los datos como argumentos y recibiendo un valor de retorno con cada llamada, de forma similar a las llamadas a los métodos locales.
- En un sistema de objetos distribuidos, un servidor de objeto consiste en un objeto distribuido que posee métodos que un cliente de objeto puede invocar. Cada extremo de la comunicación requiere un *proxy*, que interactúa con el soporte en tiempo de ejecución del sistema para llevar a cabo la comunicación entre procesos necesaria. Adicionalmente, debe existir un registro de objetos que permita a los objetos distribuidos registrarse y poder hacer búsquedas.
- Entre los protocolos de los sistemas de objetos distribuidos más conocidos se encuentran Java RMI (***Remote Method Invocation***), el modelo de objetos de componentes distribuidos DCOM, la arquitectura CORBA (***Common Object Request Broker Architecture***), y el protocolo SOAP (***Simple Object Access Protocol***).
- Java RMI es un sistema representativo de los sistemas de objetos distribuidos. Algunos de los temas relacionados con RMI que se han tratado en este capítulo son:
  - — La arquitectura del API Java RMI incluye tres capas en ambos extremos, el cliente y el servidor. En la parte cliente, la capa resguardo acepta una invocación a un método remoto y la transforma en mensajes, que envía a la parte servidora. En la parte servidora, la capa esqueleto recibe los mensajes y los transforma en llamadas locales al método remoto. Para registrar un objeto, se debe utilizar un servicio de directorios, como JNDI o el registro RMI.
  - — El software de una aplicación RMI incluye una interfaz remota, software en la parte servidora, y software en la parte cliente. Este capítulo presentó la sintaxis y los algoritmos recomendados para desarrollar este software.
- Se analizaron las diferencias entre el API de *sockets* y el API de Java RMI.

EJERCICIOS

1. Compare y contraste el paradigma de paso de mensajes y el paradigma de objetos distribuidos.
2. Compare y contraste una llamada a procedimiento local y una llamada a procedimiento remoto.
3. Describa la arquitectura de Java RMI. ¿Cuál es el papel del registro RMI?

4. Considérese una aplicación sencilla, donde un cliente envía dos valores enteros a un servidor, que suma los valores y devuelve el resultado al cliente.
   a. Describa cómo se implementaría la aplicación mediante el uso del API de *sockets*. Describa los mensajes intercambiados y las acciones correspondientes a cada mensaje.
   b. Describa cómo se implementaría la aplicación mediante el uso del API RMI. Describa la interfaz, los métodos remotos, y la invocación de los métodos remotos en el programa cliente.
5. Este ejercicio utiliza el ejemplo *HolaMundo*.
   a. Cree un directorio para este ejercicio. Coloque los ficheros fuente del ejemplo *HolaMundo* en el directorio.
   b. Compile *HolaMundoInterfaz.java* y *HolaMundoImpl.java*.
   c. Utilice *rmic* para compilar *HolaMundoImpl*. Mire la carpeta para comprobar que se han generado las clases del *proxy*. ¿Cuáles son sus nombres?
   d. Compile *HolaMundoServidor.java*. Compruebe el contenido de la carpeta.
   e. Ejecute el servidor, especificando un número de puerto aleatorio para el registro RMI. Compruebe los mensajes que se muestran, incluyendo la lista de nombres actualmente registrados. ¿Se puede ver el nombre bajo el cual se ha registrado el objeto remoto (tal y como se especifica en el programa)?
   f. Compile y ejecute *HolaMundoCliente.java*. Cuando se solicite, especifique «localhost» como nombre de la máquina y el número de puerto RMI previamente introducido. ¿Qué sucede? Explíquelo.
   g. Ejecute el programa cliente en una máquina separada. ¿La ejecución fue correcta?
6. Cree una nueva carpeta. Copie todos los ficheros fuente del ejemplo *HolaMundo* a la carpeta. Añada código al método *decirHola* de *HolaMundoImpl.java*, de forma que haya un retardo de 5 segundos antes de que el método termine. Esto tiene el efecto de alargar artificialmente la latencia de cada invocación del método. Compile y arranque el servidor.

   En pantallas separadas, arranque dos o más clientes. Observe la secuencia de eventos en las pantallas. ¿Se puede afirmar si el servidor de objeto ejecuta las llamadas a los métodos concurrente o iterativamente? Explíquelo.
7. Cree una nueva carpeta. Copie todos los ficheros fuente del ejemplo *HolaMundo* a la carpeta. Modifique el método *decirHola* de forma que se le pase un argumento, una cadena de caracteres con un nombre, y que la cadena que devuelve sea la cadena «Hola Mundo» concatenada con el nombre pasado como argumento.
   a. Muestre las modificaciones del código.
   b. Recompile y ejecute el servidor y a continuación el cliente. Describa y explique lo que sucede.
   c. Ejecute *rmic* de nuevo para generar los nuevos *proxies* de la interfaz modificada, y a continuación ejecute el servidor y el cliente.

   Entregue los listados fuentes y las salidas de la ejecución.
8. Utilice RMI para implementar un servidor y cliente *Daytime*.

9. Utilizando RMI, escriba una aplicación para un prototipo de un sistema de consultas de opinión. Asúmase que sólo se va a encuestar un tema. Los entrevistados pueden responder *sí*, *no* o *ns/nc*. Escriba una aplicación servidora, que acepte los votos, guarde el recuento (en memoria), y proporcione el recuento actual a aquellos que estén interesados.
   a. Escriba el fichero de interfaz primero. Debería proporcionar métodos remotos para aceptar una respuesta a la encuesta, proporcionando el recuento actual (ejemplo, 10 sí, 2 no, 5 ns/nc) sólo cuando el cliente lo requiera.
   b. Diseñe e implemente un servidor que (i) exporte los métodos remotos, y (ii) mantenga información de estado (los recuentos). Obsérvese que las actualizaciones de los recuentos deben protegerse con exclusión mutua.
   c. Diseñe e implemente una aplicación cliente que proporcione una interfaz de usuario para aceptar una respuesta y/o una petición, y para interactuar con el servidor apropiadamente a través de la invocación de métodos remotos.
   d. Pruebe la aplicación ejecutando dos o más clientes en máquinas diferentes (preferiblemente en plataformas diferentes).
   e. Entregue los listados de los ficheros, que deben incluir los ficheros fuente (el fichero de interfaz, los ficheros del servidor y los ficheros del cliente), y un fichero LEEME que explique los contenidos y las interrelaciones de los ficheros fuente, así como el procedimiento para ejecutar el trabajo.
10. Cree una aplicación para gestionar unas elecciones. El servidor exporta dos métodos:
   — *emitirVoto*, que acepta como parámetro una cadena de caracteres que contiene un nombre de candidato (Gore o Bush), y no devuelve nada, y
   — *obtenerResultado*, que devuelve, en un vector de enteros, el recuento actual para cada candidato.

   Pruebe la aplicación ejecutando todos los procesos en una máquina. A continuación pruebe la aplicación ejecutando el proceso cliente y servidor en máquinas separadas.

   Entregue el código fuente de la interfaz remota, el servidor y el cliente.
11. Lea la referencia [research.sun.com, 11]. Resuma las razones por las que los autores encuentran fallos en el modelo de objetos distribuidos, que minimiza las diferencias de programación entre las invocaciones local y remoto de métodos. ¿Está de acuerdo con los autores? ¿Cuál podría ser un modelo alternativo para objetos distribuidos que resolviera los problemas identificados por los autores? (*Pista*: Busque información sobre la tecnología *Jini* [sun.com, 12].)

REFERENCIAS

1. Object Management Group. Website de OMG Corba, http://www.corba.org
2. World Wide Web Consortium. SOAP versión 1.2 Parte 0: Primero, http://www.w3.org/TR/soap12-part0/.
3. Modelo de Objetos de Componentes Distribuidos (DCOM) – Documentos, especificaciones, ejemplos y recursos de Microsoft DCOM, http://www.microsoft.com/com/tech/DCOM.asp, Microsoft.
4. Richard T. Grimes. *Professional DCOM Programming*. Chicago, IL: Wrox Press, Inc., 1997.

5. RFC 1831: Especificación del protocolo Llamada a Procedimientos Remotos, versión 2, Agosto 1995, http://www.ietf.org/rfc/rfc1831.txt
6. The Open Group, DCE 1.1: Remote Procedure Call, http://www.opennc.org/public/pubs/catalog/c706.htm
7. Java Remote Method Invocation, http://java.sun.com/products/jdk/rmi
8. RMI – Tutorial de Java, http://java.sun.com/docs/books/tutorial/rmi
9. Introducción a la computación distribuida con RMI, http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html
10. Java Remote Method Invocation – Computación distribuida para Java, http://java.sun.com/marketing/collateral/javarmi.html
11. Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. *«A Note on Distributed Computing»*, Informe TR-94-29, Sun Microsystems Laboratories, 1994, http://research.sun.com/research/techrep/1994/smli_tr-94-29.pdf
12. Jini Network Technology, *«An Executive Overview»*, white paper. Sun Microsystems, Inc., 2001, http://www.sun.com/software/jini/whitepapers/jini-execoverview.pdf

CAPÍTULO 8

RMI avanzado

En el último capítulo, Java RMI se describió como ejemplo de un sistema de objetos distribuidos. En dicho capítulo sólo se mostraron las características de diseño más básicas de RMI, aunque se mencionó que el API poseía un extenso conjunto de características. El lector puede ignorar este capítulo si no está interesado en explorar de forma más detallada RMI. Sin embargo, es muy recomendable el uso de los gestores de seguridad (*véase El gestor de seguridad de RMI*) en todas las aplicaciones RMI.

Este capítulo analizará algunas de las características avanzadas de RMI más interesantes, a saber, **descarga de resguardo, gestores de seguridad**, y ***callback* de cliente**. Aunque no se trata de características inherentes del paradigma de objetos distribuidos, se trata de mecanismos que pueden ser útiles para los desarrolladores de aplicaciones. Adicionalmente, el estudio de estos temas permite al lector reforzar su conocimiento del paradigma de objetos distribuidos en general, y del API de RMI en particular.

8.1. *CALLBACK* DE CLIENTE

Considérese una aplicación RMI donde un servidor de objeto debe notificar a los procesos participantes la ocurrencia de algún evento. Como ejemplos, en un *chat*, cuando un nuevo participante entra, se avisa al resto de los participantes de este hecho; en un sistema de subastas en tiempo real, cuando empiezan las ofertas, se debe avisar a los procesos participantes. Esta característica también es útil en un juego en red cuando se informa a los jugadores de la actualización del estado del juego. Dentro del entorno del API básica de RMI presentada en el capítulo anterior, es imposible que

el servidor inicie una llamada al cliente para transmitirle alguna clase de información que esté disponible, debido a que una llamada a método remoto es unidireccional (del cliente al servidor). Una forma de llevar a cabo la transmisión de información es que cada proceso cliente realice un **sondeo** al servidor de objeto, invocando de forma repetida un método remoto, que supóngase que se llama *haComenzadoOferta*, hasta que el método devuelva el valor booleano verdadero:

```
InterfazServidor h =
   (InterfazServidor) Naming.lookup(URLRegistro);
while (!(h.haComenzadoOferta()) {;}
   // comienza la oferta
```

El **sondeo** (***polling***) es de hecho una técnica empleada en muchos programas de red. Pero se trata de una técnica muy costosa en términos de recursos del sistema, ya que cada invocación de un método remoto implica un hilo separado en la máquina servidora, además de los recursos de sistema que su ejecución conlleva. Una técnica más eficiente se denomina ***callback***: permite que cada cliente de objeto interesado en la ocurrencia de un evento se registre a sí mismo con el servidor de objeto, de forma que el servidor inicie una invocación de un método remoto del cliente de objeto cuando dicho evento ocurra. La Figura 8.1 compara las dos técnicas: sondeo y *callback*.

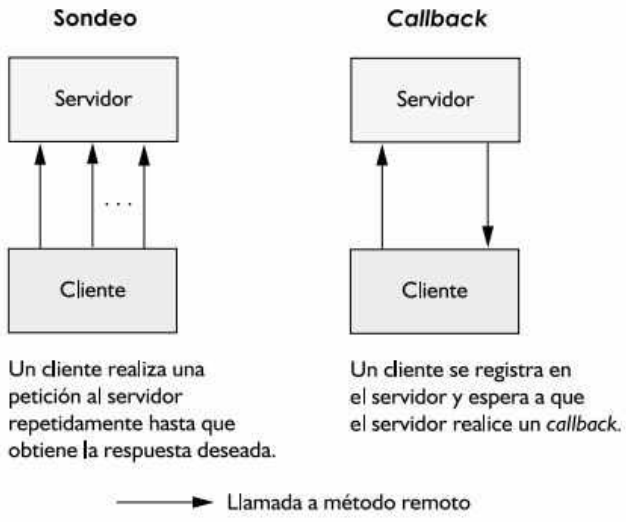


Figura 8.1. Sondeo (*polling*) frente a *callback*.

En RMI, el ***callback* de cliente** es una característica que permite a un cliente de objeto registrarse a sí mismo con un servidor de objeto remoto para ***callbacks***, de forma que el servidor pueda llevar a cabo una invocación del método del cliente cuando el evento ocurra. Hay que observar que con los *callbacks* de clientes, las invocaciones de los métodos remotos se convierten en bidireccionales, o ***dúplex***, desde el cliente al servidor y viceversa. Debido a que el API de RMI básica, introducida en el capítulo anterior, sólo permite la invocación de métodos remotos de clientes en servidores de objetos, se necesita claramente sintaxis adicional para dar soporte a esta nueva característica.

Cuando un servidor de objeto realiza un *callback*, los papeles de los dos procesos se invierten: el servidor de objeto se convierte en cliente del cliente de objeto, debido a que el primero inicia una invocación de método remoto en el segundo.

La Figura 8.2 muestra la arquitectura de RMI con *callback* de cliente. Comparada con la arquitectura básica de RMI, se puede observar que en este caso se necesitan dos conjuntos de *proxies*, uno para la interfaz remota del servidor, como en la arquitectura básica de RMI, y otro para una interfaz adicional, la interfaz remota del cliente. La interfaz remota del cliente proporciona un método remoto que puede invocar el servidor a través del *callback*.

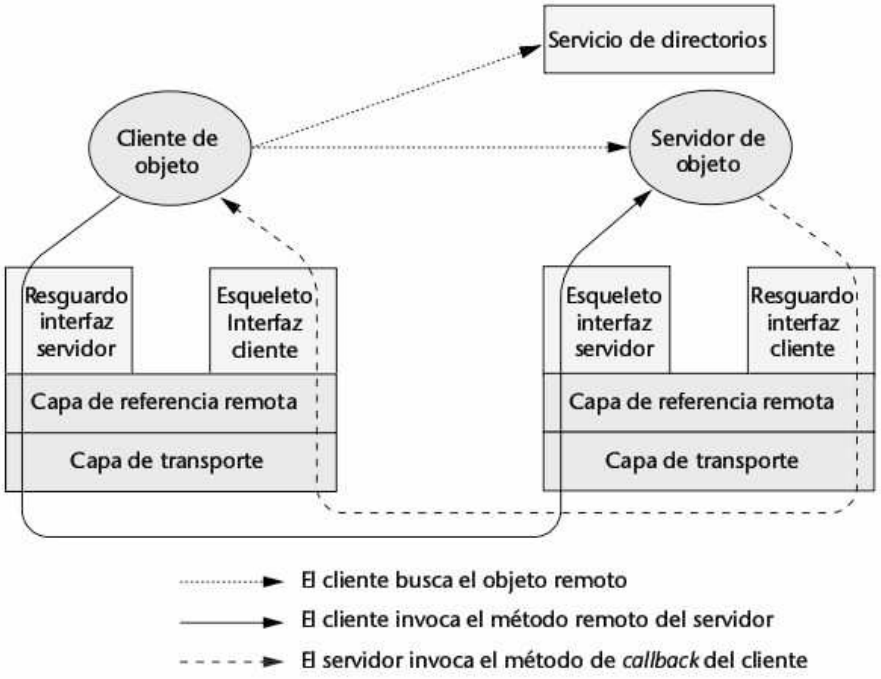


Figura 8.2. La arquitectura de RMI con *callback* de cliente.

Como ejemplo, se extenderá la aplicación *HolaMundo*, presentada en el capítulo anterior, de forma que el cliente del objeto se registre con el servidor para *callback* y entonces se le notifique cualquier registro de otro cliente de objeto para *callback* con el servidor.

Extensión de la parte cliente para *callback* de cliente

Para el *callback*, el cliente debe proporcionar un método remoto que permita al servidor notificarle el evento correspondiente. Esto puede hacerse de una forma similar a los métodos remotos del servidor de objeto. En la siguiente subsección se describe la sintaxis necesaria para llevar a cabo esto.

La interfaz remota de cliente

Es importante recordar que el servidor de objeto proporciona una interfaz remota que declara los métodos que un cliente de objeto puede invocar. Para el *callback*, es ne-

cesario que el cliente de objeto proporcione una interfaz remota similar. Se le denominará interfaz remota de cliente (por ejemplo, *InterfazCallbackCliente*), por oposición a la interfaz remota de servidor (por ejemplo, *InterfazCallbackServidor*). La interfaz remota de cliente debe contener al menos un método que será invocado por el servidor en el *callback*. Como ejemplo, se describe la siguiente interfaz remota de cliente:

```
public interfaz InterfazCallbackCliente
   extends java.rmi.Remote {
   // Este método remoto es invocado por un servidor
   // que realice un callback al cliente que implementa
   // esta interfaz.
   // El parámetro es una cadena de caracteres que
   // contiene información procesada por el cliente
   // una vez realizado el callback.
   // Este método devuelve un mensaje al servidor.
   public String notificame(String mensaje)
      throws java.rmi.RemoteException;

} // final de la interfaz
```

El servidor debe invocar el método *notifícame* cuando realiza el *callback*, pasando como argumento una cadena de caracteres (*String*). Una vez recibido el *callback*, el cliente utiliza esta cadena para componer otra cadena que devuelve al servidor.

La implementación de la interfaz remota de cliente

Al igual que la interfaz remota de servidor, es necesario implementar la interfaz remota de cliente en una clase, denominada *ImplCallbackCliente* en el ejemplo, tal y como se muestra a continuación:

```
import java.rmi.*;
import java.rmi.server.*;
public class ImplCallbackCliente extends UnicastRemoteObject
   implements InterfazCallbackCliente {
   public ImplCallbackCliente() throws RemoteException {
      super();
   }
   public String notificame(String mensaje) {
      String mensajeRet ="Callback recibido: " + mensaje;
      System.out.println(mensajeRet);
      return mensajeRet;
   }
} // final clase ImplCallbackCliente
```

En este ejemplo el método de *callback notifícame* simplemente imprime la cadena de caracteres que le pasa el servidor como argumento, y devuelve otra cadena a dicho servidor.

Al igual que la interfaz remota de servidor, se debe utilizar el compilador **rmic** con la implementación de la interfaz remota de cliente para generar los *proxies* necesarios en tiempo de ejecución.

Extensión de la clase cliente

En la clase del cliente de objeto, se necesita añadir código al cliente para que instancie un objeto de la implementación de la interfaz remota de cliente. A continuación, se registra con el servidor una referencia al objeto utilizando un método remoto proporcionado por el servidor (*véase* la próxima sección, «Extensión de la parte servidora para *callback* de cliente»). Un ejemplo de cómo debe realizarse esto se muestra a continuación:

```
InterfazCallbackServidor h =
  (InterfazCallbackServidor) Naming.lookup(URLRegistro);
InterfazCallbackCliente objCallback =
  new ImplCallbackCliente();
// registrar el objeto para callback
h.registrarCallback(objCallback);
```

Las Figuras entre la 8.3 hasta la 8.5 presentan el código del software de la parte cliente para la aplicación *HolaMundo* modificada.

Figura 8.3. Fichero *InterfazCallbackCliente.java* de la aplicación *HolaMundo* modificada.

```
 1  import java.rmi.*;
 2
 3  /**
 4  * Esto es una interfaz remota para ilustrar el
 5  * callback de cliente.
 6  * @author M. L. Liu
 7  */
 8
 9  public interface InterfazCallbackCliente
10        extends java.rmi.Remote{
11    // Este método remoto se invoca mediante callback
12    // de servidor, de forma que realiza un callback a
13    // un cliente que implementa esta interfaz.
14    // @message una cadena de caracteres que contiene
15    // información procesada por el cliente.
16
17    public void notificame(String mensaje)
18        throws java.rmi.RemoteException;
19
20  } // fin interfaz
```

Figura 8.4. Fichero *ImplCallbackCliente.java* de la aplicación *HolaMundo* modificada.

```
 1  import java.rmi.*;
 2  import java.rmi.server.*;
 3
 4  /**
 5  * Esta clase implementa la interfaz remota
 6  * InterfazCallbackCliente.
 7  * @author M. L. Liu
```

(continúa)

```
 8  */
 9
10  public class ImplCallbackCliente extends UnicastRemoteObject
11       implements InterfazCallbackCliente {
12
13    public ImplCallbackCliente () throws RemoteException {
14      super( );
15    }
16
17    public String notificame(String mensaje){
18      String mensajeRet = "Callback recibido: " + mensaje;
19      System.out.println(mensajeRet);
20        return mensajeRet;
21    }
22
23  } // fin clase ImplCallbackCliente
```

Figura 8.5. Fichero *ClienteEjemplo.java* de la aplicación *HolaMundo* modificada.

```
 1  import java.io.*;
 2  import java.rmi.*;
 3
 4  /**
 5  * Esta clase representa el cliente de objeto para un
 6  * objeto distribuido de la clase ImplCallbackServidor,
 7  * que implementa la interfaz remota
 8  * InterfazCallbackServidor. También acepta callbacks
 9  * del servidor.
10  *
11  *
12  *
13  * @author M. L. Liu
14  */
15
16  public class ClienteEjemplo {
17
18    public static void main(String args[]) {
19      try {
20        int puertoRMI;
21        String nombreNodo;
22        InputStreamReader ent =
23          new InputStreamReader(System.in);
24        BufferedReader buf= new BufferedReader(ent);
25        System.out.println(
26          "Introduce el nombre de nodo del registro RMI:");
27        nombreNodo = buf.readLine();
28        System.out.println(
```

(continúa)

```
29            "Introduce el número de puerto del registro RMI:");
30          String numPuerto = buf.readLine();
31          puertoRMI = Integer.parseInt(numPuerto);
32          System.out.println(
33            "Introduce cuantos segundos va a permanecer registrado:");
34          String duracionTiempo = buf.readLine();
35          int tiempo = Integer.parseInt(duracionTiempo);
36          String URLRegistro =
37            "rmi://localhost:"+ numPuerto + "/callback";
38          // Búsqueda del objeto remoto y cast al objeto
39          // de la interfaz
40          InterfazCallbackServidor h =
41            (InterfazCallbackServidor) Naming.lookup(URLRegistro);
42          System.out.println("Búsqueda completa ");
43          System.out.println("El servidor dice " + h.decirHola());
44          InterfazCallbackCliente objCallback =
45            new ImplCallbackCliente();
46          // registrar para callback
47          h.registrarCallback(objCallback);
48          System.out.println("Registrado para callback.");
49          try {
50            Thread.sleep(tiempo*1000);
51          }
52          catch (InterruptedException exc) { // sobre el método sleep
53            h.eliminarRegistroCallback(objCallback);
54            System.out.println("No registrado para callback.");
55          }
56        } // fin try
57        catch (Exception e) {
58          System.out.println(
59            "Excepción en ClienteEjemplo: " + e);
60        }
61      } // fin main
62    } // fin clase
```

Extensión de la parte servidora para *callback* de cliente

En la parte del servidor, se necesita añadir un método remoto para que el cliente pueda registrarse para *callback*. En el caso más sencillo, la cabecera del método puede ser análoga a la siguiente:

```
public void registrarCallback(
// Se puede elegir el nombre de método deseado
InterfazCallbackCliente objCallbackCliente
) throws java.rmi.RemoteException;
```

Como argumento se pasa una referencia a un objeto que implementa la interfaz remota de cliente (*InterfazCallbackCliente*, no *ImplCallbackCliente*). También se pue-

de proporcionar un método *eliminarRegistroCallback*, para que un cliente pueda cancelar el registro (de forma que no reciba más *callbacks*). La implementación de estos métodos, así como la implementación de un método local *hacerCallbacks* (para realizar los *callbacks*) se muestra en la Figura 8.7. La Figura 8.6 muestra el fichero con la interfaz del servidor, que contiene las cabeceras de los métodos adicionales. La Figura 8.8 muestra el código para el servidor de objeto, que queda sin modificar respecto a la anterior versión, presentada en el capítulo anterior.

Figura 8.6. Fichero *InterfazCallbackServidor.java* de la aplicación *HolaMundo* modificada.

```
1  import java.rmi.*;
2
3  /**
4  * Esto es una interfaz remota para ilustrar el
5  * callback de cliente.
6  * @author M. L. Liu
7  */
8
9  public interface InterfazCallbackServidor extends Remote{
10
11    public String decirHola( )
12      throws java.rmi.RemoteException;
13
14      // Este método remoto permite a un cliente
15      // de objeto registrarse para callback
16      // @param objClienteCallback es una referencia
17      // al cliente de objeto; el servidor lo
18      // utiliza para realizar los callbacks
19
20    public void registrarCallback(
21         InterfazCallbackCliente objCallbackCliente)
22      throws java.rmi.RemoteException;
23
24      // Este método remoto permite a un cliente
25      // de objeto cancelar su registro para callback
26
27    public void eliminarRegistroCallback(
28         InterfazCallbackCliente objCallbackCliente)
29      throws java.rmi.RemoteException;
30 }
```

Figura 8.7. Fichero *ImplCallbackServidor.java* de la aplicación *HolaMundo* modificada.

```
1  import java.rmi.*;
2  import java.rmi.server.*;
3  import java.util.Vector;
4
5  /**
6  * Esta clase implementa la interfaz remota
```

(continúa)

```
 7  * InterfazCallbackServidor.
 8  * @author M. L. Liu
 9  */
10
11  public class ImplCallbackServidor extends UnicastRemoteObject
12       implements InterfazCallbackServidor {
13
14    private Vector listaClientes;
15
16
17    public ImplCallbackServidor () throws RemoteException {
18      super( );
19      listaClientes = new Vector();
20    }
21
22    public String decirHola()
23         throws java.rmi.RemoteException {
24      return("Hola Mundo");
25    }
26
27    public void registrarCallback(
28         InterfazCallbackCliente objCallbackCliente)
29         throws java.rmi.RemoteException {
30      // almacena el objeto callback en el vector
31      if (!(listaClientes.contains(objCallbackCliente))) {
32        listaClientes.addElement(objCallbackCliente);
33        System.out.println("Nuevo cliente registrado ");
34        hacerCallbacks();
35      } // fin if
36    }
37
38    // Este método remoto permite a un cliente de objeto
39    // cancelar su registro para callback
40    // @param id es un identificador para el cliente;
41    // el servidor lo utiliza únicamente para identificar al cliente
         registrado.
42    public synchronized void eliminarRegistroCallback(
43         InterfazCallbackCliente objCallbackCliente)
44         throws java.rmi.RemoteException{
45      if (listaClientes.removeElement(objCallbackCliente)){
46        System.out.println("Cliente no registrado ");
47      } else {
48        System.out.println(
49          "eliminarRegistro: el cliente no fue registrado."
50      }
51    }
52
```

Los métodos *registrarCallback* y *eliminarRegistroCallback* modifican una estructura común (el objeto *Vector* que contiene referencias a los *callback* de clientes). Dado que estos métodos se pueden invocar concurrentemente, es importante que se protejan con exclusión mutua. En este ejemplo la exclusión mutua se consigue a través del uso de un método sincronizado (*synchronized*).

(continúa)

```
53      private synchronized void hacerCallbacks( ) throws
        java.rmi.RemoteException {
54        // realizar callback de un cliente registrado
55        System.out.println(
56           ************************************\n" +
57           "Callback iniciado — ");
58        for (int i=0; i<listaClientes.size(); i++) {
59           System.out.println("haciendo callback número"+ i "\n");
60           // convertir el objeto vector a un objeto callback
61           InterfazCallbackCliente proxCliente =
62              (InterfazCallbackCliente) listaClientes.elementAt(i);
63           // invocar el método de callback
64           proxCliente.notificame("Número de clientes registrados="
65              + listaClientes.size());
66        } // fin for
67        System.out.println("************************************\n"
68           + "Servidor completo callbacks —");
69      } // fin hacerCallbacks
70
71    } // fin clase ImplCallbackServidor
```

Figura 8.8. Fichero *ServidorEjemplo.java* de la aplicación *HolaMundo* modificada.

```
 1  import java.rmi.*;
 2  import java.rmi.server.*;
 3  import java.rmi.registry.Registry;
 4  import java.rmi.registry.LocateRegistry;
 5  import java.net.*;
 6  import java.io.*;
 7
 8  /**
 9  * Esta clase representa el servidor de objeto para un
10  * objeto distribuido de la clase Callback, que
11  implementa la interfaz remota InterfazCallback.
12  * @author M. L. Liu
13  */
14
15  public class ServidorEjemplo {
16    public static void main(String args[]) {
17      InputStreamReader ent =
18        new InputStreamReader(System.in);
19      BufferedReader buf= new BufferedReader(ent);
20      String numPuerto, URLRegistro;
21      try {
22        System.out.println(
23        "Introducir el número de puerto del registro RMI:");
24        numPuerto=(buf.readLine()).trim();
25        int numPuertoRMI = Integer.parseInt(numPuerto);
```

(continúa)

```
26          arrancarRegistro(numPuertoRMI);
27          ImplCallbackServidor objExportado =
28            new ImplCallbackServidor( );
29          URLRegistro =
30            "rmi://localhost:" + numPuerto + "/callback";
31          Naming.rebind(URLRegistro, objExportado);
32          System.out.println("Servidor callback preparado.");
33        } // fin try
34        catch (Exception exc) {
35          System.out.println(
36          "Excepción en ServidorEjemplo.main: " + exc);
37        } // fin catch
38      } // fin main
39
40      // Este método arranca un registro RMI en el nodo
41      // local, si no existe en el número de puerto especificado.
42      private static void arrancarRegistro(int numPuertoRMI)
43        throws RemoteException {
44        try {
45          Registry registro =
46          LocateRegistry.getRegistry(numPuertoRMI);
47          registro.list();
48          // Esta llamada lanza una excepción
49          // si el registro no existe
50        }
51        catch (RemoteException e) {
52          // No existe registro válido en el puerto
53          Registry registro =
54          LocateRegistry.createRegistry(numPuertoRMI);
55        }
56      } // fin arrancarRegistro
57
58    } // fin clase
```

El servidor necesita emplear una estructura de datos que mantenga una lista de las referencias a la interfaz de cliente registradas para *callbacks*. En el código de ejemplo, un objeto *Vector* es utilizado para este propósito, aunque se puede sustituir por cualquier otra estructura de datos apropiada. Cada llamada a *registrarCallback* implica añadir una referencia al vector, mientras que cada llamada a *eliminarRegistroCallback* supone borrar una referencia del vector.

En el ejemplo, el servidor realiza un *callback* (mediante el método *hacerCallbacks*) siempre que se lleva a cabo una llamada a *registrarCallback*, donde se envía al cliente a través de *callback* el número de clientes actualmente registrados. En otras aplicaciones, los *callbacks* se pueden activar por otros eventos y pueden gestionarse a través de un manejador de eventos.

En el ejemplo, un cliente elimina su registro después de un determinado periodo de tiempo. En las aplicaciones reales, la cancelación del registro se puede realizar al

final de la sesión del cliente (tal como en el caso de una sesión de *chat* o en una sesión de subastas).

Pasos para construir una aplicación RMI con *callback* de cliente

En las siguientes páginas se presenta una descripción revisada del procedimiento para construir una aplicación RMI paso a paso, permitiendo *callback* de cliente.

Algoritmo para desarrollar el software de la parte del servidor

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota de servidor en *InterfazCallbackServidor.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
3. Implementar la interfaz en *ImplCallbackServidor.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
4. Utilizar el compilador RMI *rmic* para procesar la clase de la implementación y generar los ficheros resguardo y esqueleto para el objeto remoto:

   ```
   rmic ImplCallbackServidor
   ```

   Los ficheros generados se pueden encontrar en el directorio como *ImplCallbackServidor_Skel.class* y *ImplCallbackServidor_Stub.class*. Los pasos 3 y 4 deben repetirse cada vez que se cambie la implementación de la interfaz.
5. Obtener una copia del fichero *class* de la interfaz remota del cliente. Alternativamente, obtener una copia del fichero fuente para la interfaz remota y compilarlo utilizando *javac* para generar el fichero *class* de la interfaz *InterfazCallbackCliente.class*.
6. Crear el programa correspondiente al servidor de objeto *ServidorEjemplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
7. Obtener una copia del fichero resguardo de la interfaz remota del cliente *ImplCallbackCliente_Stub.class*.
8. Activar el servidor de objeto

   ```
   java ServidorEjemplo
   ```

Algoritmo para desarrollar el software de la parte cliente

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota de cliente en *InterfazCallbackCliente.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
3. Implementar la interfaz en *ImplCallbackCliente.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
4. Utilizar el compilador RMI *rmic* para procesar la clase de la implementación *ImplCallbackCliente.class* y generar los ficheros resguardo y esqueleto

ImplCallbackCliente_Skel.class y *ImplCallbackCliente_Stub.class* para el objeto remoto:

```
rmic ImplCallbackCliente
```

Los ficheros generados se pueden encontrar en el directorio como *ImplCallbackCliente_Skel.class* y *ImplCallbackCliente_Stub.class*. Los pasos 3 y 4 deben repetirse cada vez que se cambie la implementación de la interfaz.

5. Obtener una copia del fichero *class* de la interfaz remota del servidor. Alternativamente, obtener una copia del fichero fuente para la interfaz remota y compilarlo utilizando *javac* para generar el fichero *class* de la interfaz *Interfaz*.
6. Crear el programa correspondiente al cliente de objeto *ClienteEjemplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
7. Obtener una copia del fichero resguardo de la interfaz remota del servidor *ImplCallbackServidor_Stub.class*. Activar el cliente de objeto

```
java ClienteEjemplo
```

La Figura 8.9 muestra los ficheros que se necesitan en los dos extremos, cliente y servidor, cuando se utiliza *callback* de cliente. (Como se mencionó en el capítulo anterior, desde la versión 1.2 de Java no se requieren clases esqueleto en las aplicaciones RMI. Las funciones de las clases esqueleto se realizan a través de una técnica denominada **reflexión**.)

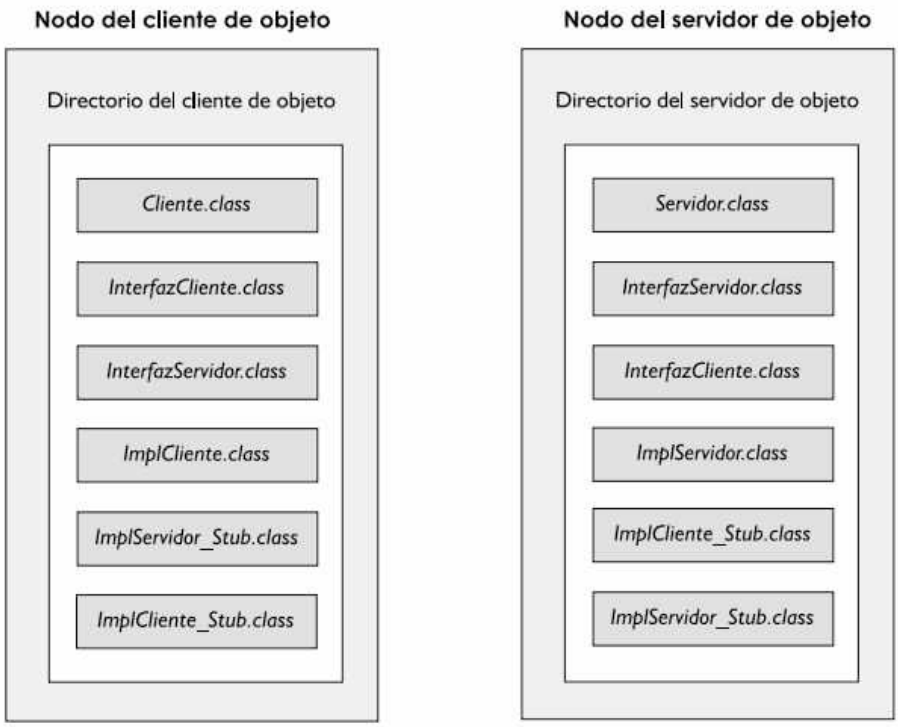


Figura 8.9. Colocación de los ficheros en una aplicación RMI con *callback* de cliente.

8.2. DESCARGA DE RESGUARDO

En la arquitectura de un sistema de objetos distribuidos se requiere un *proxy* para interactuar con la llamada a un método remoto de un cliente de objeto. En Java RMI, este *proxy* o intermediario es el resguardo de la interfaz remota del servidor. En el capítulo anterior se describieron la forma en la que se generan los *proxies* de la interfaz remota del servidor (ambos el resguardo y el esqueleto) mediante el compilador RMI *rmic*. La clase resguardo generada debe estar en la máquina cliente en tiempo de ejecución cuando un programa cliente se ejecute. Esto se puede resolver colocando manualmente el fichero *class* del resguardo en el mismo paquete o directorio que el programa del cliente de objeto.

En Java, un **paquete** es un conjunto de clases, interfaces u otros paquetes relacionados y que están declarados.

Java RMI proporciona un mecanismo que permite que los clientes obtengan dinámicamente los resguardos necesarios [developer.java.sun.com, 2]. Mediante **descarga dinámica de resguardo**, no se necesita una copia de la clase del resguardo en la máquina cliente. Por el contrario, éste se transmite bajo demanda desde un servidor web a la máquina cliente cuando se activa dicho cliente.

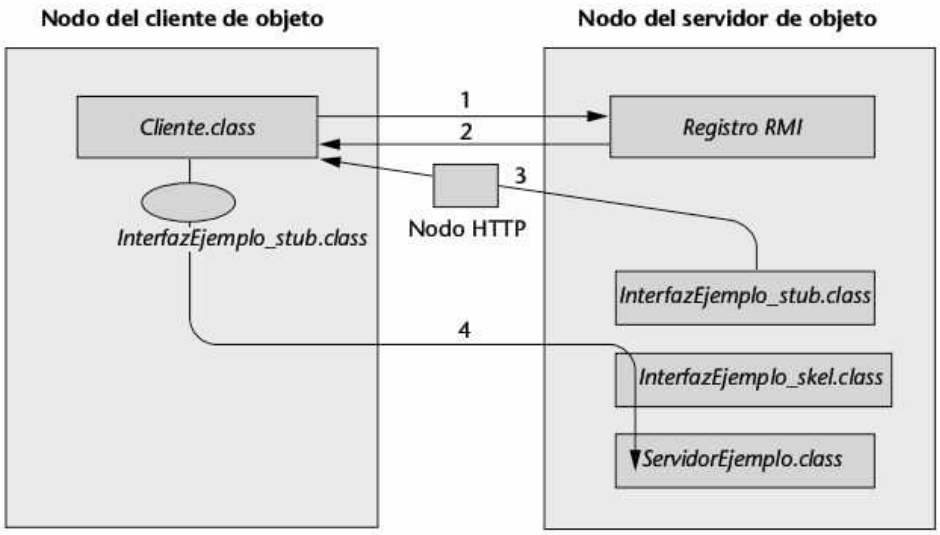
Esta técnica utiliza la «habilidad de descargar dinámicamente software Java de cualquier URL a una máquina virtual Java (JVM, *Java Virtual Machine*) ejecutándose en un proceso separado, normalmente en un sistema físico diferente» [java.sun.com/products, 1]. Mediante el uso de descarga dinámica, el desarrollador almacena una clase resguardo en un servidor web como un documento web, que puede ser descargado (utilizando HTTP) cuando un cliente de objeto se ejecuta, de la misma forma que se lleva a cabo la descarga de ***applets***. El uso de HTTP para descargar *applets* se discutirá en el Capítulo 11.

Al igual que antes, un servidor exporta un objeto contactando con el registro RMI y registrando una referencia remota al objeto, especificando un nombre simbólico para la referencia. Si se desea utilizar descarga de resguardo, el servidor debe también indicar al registro el URL donde se encuentra almacenada la clase resguardo. Los mecanismos para realizar esto se presentarán en una sección posterior.

De la misma forma que antes, un cliente que desee invocar un método remoto de un objeto exportado contacta con el registro RMI en la máquina servidora para traer la referencia remota a través del nombre. Sin descarga de resguardo, el objeto resguardo (un fichero *class* Java) debe colocarse en la máquina cliente manualmente y la máquina virtual Java debe poder localizarlo. Si se utiliza descarga de resguardo (es decir, se han realizado los pasos necesarios descritos en el párrafo anterior con el servidor de objeto), entonces se puede obtener dinámicamente la clase resguardo de un servidor HTTP de forma que puede interactuar con el cliente de objeto y el soporte en tiempo real de RMI. La clase resguardo descargada no es persistente, es decir, no se almacena de forma permanente en la máquina cliente, sino que por el contrario el sistema libera la clase correspondiente cuando la sesión del cliente finaliza. Si no se utiliza cache en el servidor web, cada ejecución de la clase cliente requiere la descarga del resguardo del servidor web.

Caching de Web es una técnica que emplea la técnica más general de *caching* para evitar transferir un mismo documento repetidas veces.

La Figura 8.10 muestra la interacción entre el cliente de objeto, el servidor de objeto y el registro RMI cuando se utiliza descarga de resguardo. Pronto se describirán los algoritmos necesarios para ejecutar una aplicación mediante el uso de descarga de resguardo. Antes, se debe presentar un tema relacionado: el gestor de seguridad de RMI.



1. El cliente busca el objeto de la interfaz en el registro RMI del nodo servidor.
2. El registro RMI devuelve una referencia remota al objeto de la interfaz.
3. Si el resguardo del objeto de la interfaz no está en el nodo cliente y el servidor está configurado para ello, se descarga el resguardo de un servidor HTTP.
4. A través del resguardo del servidor, el proceso cliente interactúa con el esqueleto del objeto de la interfaz para acceder a los métodos del objeto servidor.

Figura 8.10. Descarga de resguardo.

8.3. EL GESTOR DE SEGURIDAD DE RMI

Aunque la descarga de resguardo es una característica útil, su uso supone un problema para el sistema de seguridad. Este problema no está asociado al uso de RMI, sino a la descarga de objetos en general. Cuando un objeto como un resguardo RMI se transfiere desde un nodo remoto, su ejecución entraña el riesgo de ataques maliciosos al nodo local. Debido a que un objeto descargado procede de un origen desconocido, la ejecución de su código, si no se restringe, podría causar estragos en el nodo local, provocando daños similares a los causados por un virus de computador [cert.org, 3].

Para evitar los problemas de seguridad del uso de descarga de resguardo, Java proporciona una clase denominada *RMISecurityManager*. Un programa RMI puede instanciar un objeto de esta clase. Una vez instanciado, el objeto supervisa durante la ejecución del programa todas las acciones que puedan suponer un riesgo de seguridad. Estas acciones incluyen el acceso a ficheros locales y la realización de conexiones de red, ya que dichas acciones podrían suponer modificaciones de los recursos locales no deseadas o mal uso de los recursos de red. En particular, el soporte en tiempo real de RMI requiere que un proceso servidor instale un **gestor de seguridad** antes de exportar cualquier objeto que requiera descarga de resguardo, y que un cliente instale un gestor de seguridad antes de que puede realizar la descarga del resguardo.

Aunque la noción de gestor de seguridad no se introdujo en el capítulo anterior, se recomienda su uso en todas las aplicaciones RMI, independientemente de que se

utilice descarga de resguardo o no. Por defecto, un gestor de seguridad RMI es muy restrictivo: no permite acceso a los ficheros y sólo permite conexiones a la máquina origen. (Esta restricción de acceso también se utiliza en las descargas de *applets*). Esta restricción, sin embargo, no permite a un cliente de objeto RMI contactar con el **registro RMI** del servidor de objeto y tampoco le permite llevar a cabo descarga de resguardo. Es posible relajar estas condiciones instalando un fichero especial conocido como **fichero de política de seguridad**, cuya sintaxis especifica el tipo de restricción que un gestor de seguridad, incluyendo los gestores de seguridad RMI, debe utilizar. Por defecto, existe un fichero de política de seguridad en un directorio especial de cada sistema que utiliza Java. Las restricciones especificadas en el fichero de políticas de seguridad del sistema (las restricciones por defecto anteriormente mencionadas) serán empleadas por el gestor de seguridad a menos que se sobreescriban mediante el uso de un fichero de políticas alternativo. Alternativamente, una aplicación puede especificar un fichero de políticas de seguridad, de forma que las restricciones las impone la propia aplicación. Para los ejercicios realizados por el lector, se recomienda que se especifique un fichero de seguridad con cada aplicación que se ejecute, de forma que se tenga control exclusivamente sobre las restricciones impuestas en la aplicación del lector, sin afectar a las restricciones de otros programas. En algunos sistemas, puede ocurrir que un usuario normal no tenga privilegio de acceso para modificar el fichero por defecto de políticas de seguridad de Java.

A continuación, se describe cómo una aplicación utiliza el gestor de seguridad de RMI.

Instanciación de un gestor de seguridad en un programa RMI

La clase *RMISecurityManager* se puede instanciar tanto en el cliente de objeto como en el servidor de objeto utilizando la siguiente sentencia:

```
System.setSecurityManager(new RMISecurityManager());
```

Esta sentencia debería aparecer antes del código de acceso al registro RMI. Las Figuras 8.11 y 8.12 muestran los ejemplos del programa *HolaMundo*, presentados en el capítulo anterior, pero instanciando un gestor de seguridad.

Figura 8.11. *HolaMundoServidor.java* haciendo uso de un gestor de seguridad.

```
 1  import java.rmi.*;
 2  import java.rmi.server.*;
 3  import java.rmi.registry.Registry;
 4  import java.rmi.registry.LocateRegistry;
 5  import java.net.*;
 6  import java.io.*;
 7
 8  /**
 9  * Esta clase representa el servidor de objeto para un
10  * objeto distribuido de la clase HolaMundo, que
11  implementa la interfaz remota InterfazHolaMundo. Se
12  instala un gestor de seguridad para realizar descarga de resguardo
    seguro.
13  * @author M. L. Liu
```

(continúa)

```
14  */
15
16  public class HolaMundoServidor {
17    public static void main(String args[]) {
18      InputStreamReader ent =
19        new InputStreamReader(System.in);
20      BufferedReader buf= new BufferedReader(ent);
21      String numPuerto, URLRegistro;
22      try {
23        System.out.println(
24        "Introducir el numero de puerto del registro RMI:");
25        numPuerto=(buf.readLine()).trim();
26        int numPuertoRMI = Integer.parseInt(numPuerto);
27
28        // arrancar un gestor de seguridad - esto es
29        // necesario si se utiliza descarga de resguardo
30        System.setSecurityManager(
31          new RMISecurityManager());
32
33        arrancarRegistro(numPuertoRMI);
34        ImplHolaMundo objExportado = new ImplHolaMundo();
35        URLRegistro =
36        "rmi://localhost:" + numPuerto + "/holaMundo";
37        Naming.rebind(URLRegistro, objExportado);
38        System.out.println(
39        "Servidor registrado. El registro contiene:");
40        // listar los nombres registrados actualmente
41        listarRegistro(URLRegistro);
42        System.out.println("Servidor Hola Mundo preparado.");
43      } // fin try
44      catch (Exception exc) {
45        System.out.println(
46        "Excepción en HolaMundoServidor.main: " + exc);
47      } // fin catch
48    } // fin main
49
50    // Este método arranca un registro RMI en el nodo
51    // local, si no existe en el número de puerto especificado.
52    private static void arrancarRegistro(int numPuertoRMI)
53      throws RemoteException {
54      try {
55        Registry registro =
56        LocateRegistry.getRegistry(numPuertoRMI);
57        registro.list(); // Esta llamada lanza
58        // una excepción si el registro no existe
59      }
60      catch (RemoteException e) {
```

(continúa)

```
61          // No existe registro válido en el puerto
62          System.out.println(
63          "El registro RMI no se localiza en este puerto "
64          + numPuertoRMI);
65          Registry registro =
66          LocateRegistry.createRegistry(numPuertoRMI);
67          System.out.println(
68          "Registro RMI creado en el puerto "+ numPuertoRMI);
69        }
70      } // fin arrancarRegistro
71
72      // Este método lista los nombres registrados en RMI
73      private static void listarRegistro(String URLRegistro)
74        throws RemoteException, MalformedURLException {
75        System.out.println(
76        "Registro " + URLRegistro + " contiene: ");
77        String [] nombres = Naming.list(URLRegistro);
78        for (int i=0; i< nombres.length; i++)
79          System.out.println(nombres[i]);
80      } // fin listarRegistro
81
82  } // fin clase
```

Figura 8.12. *HolaMundoCliente.java* haciendo uso de un gestor de seguridad.

```
1   import java.io.*;
2   import java.rmi.*;
3
4   /**
5   * Esta clase representa el cliente de objeto para un
6   * objeto distribuido de la clase HolaMundo, que
7   * implementa la interfaz remota InterfazHolaMundo.
8   * Se instala un gestor de seguridad para realizar descarga de
      resguardo segura.
9   * @author M. L. Liu
10  */
11
12  public class HolaMundoCliente {
13
14    public static void main(String args[]) {
15      try {
16        int puertoRMI;
17        String nombreNodo;
18        InputStreamReader ent =
19          new InputStreamReader(System.in);
20        BufferedReader buf= new BufferedReader(ent);
21        System.out.println(
```

(continúa)

```
22          "Introduce el nombre de nodo del registro RMI:");
23          nombreNodo = buf.readLine();
24          System.out.println(
25          "Introduce el número de puerto del registro RMI:");
26          String numPuerto = buf.readLine();
27          puertoRMI = Integer.parseInt(numPuerto);
28
29          // arrancar un gestor de seguridad – esto es
30          // necesario si se utiliza descarga de resguardo
31          System.setSecurityManager(new RMISecurityManager());
32
33          String URLRegistro =
34          "rmi://localhost:"+ numPuerto + "/holaMundo";
35          // Búsqueda del objeto remoto y cast al
36          // objeto de la interfaz
37          InterfazHolaMundo h =
38          (InterfazHolaMundo)Naming.lookup(URLRegistro);
39          System.out.println("Búsqueda completa ");
40          // invocar el método remoto
41          String mensaje=h.decirHola());
42          System.out.println("HolaMundoCliente: " + mensaje);
43        } // fin try
44        catch (Exception e) {
45          System.out.println(
46          "Excepción en HolaMundoCliente: " + e);
47        }
48      } // fin main
49    } // fin clase
```

La sintaxis de un fichero de políticas de seguridad de Java

Un fichero de políticas de seguridad de Java es un fichero de texto que contiene códigos que permiten especificar la concesión de permisos específicos. A continuación se muestra un fichero típico *java.policy* para una aplicación RMI.

```
grant {
   // Este permiso permite a los clientes RMI realizar
   // conexiones de sockets a los puertos públicos de
   // cualquier computador.
   // Si se arranca un puerto en el registro RMI en este
   // rango, no existirá una violación de acceso de
   // conexión.
   // permission java.net.SocketPermission "*:1024-65535",
   //   "connect,accept,resolve";
   // Este permiso permite a los sockets acceder al puerto
   // 80, el puerto por defecto HTTP que el cliente
```

(continúa)

```
    // necesita para contactar con el servidor HTTP para
    // descarga de resguardo
    permission java.net.SocketPermission "*:80", "connect";
};
```

Se recomienda al lector que cuando realice los ejercicios, haga una copia de este fichero para la aplicación con el nombre *java.policy* en el mismo directorio tanto en la máquina del cliente de objeto como en la máquina del servidor de objeto.

Cuando se active el cliente, hay que utilizar la opción del mandato que permite especificar que el proceso cliente debe tener los permisos definidos en el fichero de políticas, de la siguiente forma:

```
java –Djava.security.policy=java.policy ClienteEjemplo
```

Del mismo modo, el servidor debe activarse del siguiente modo:

```
java –Djava.security.policy=java.policy ServidorEjemplo
```

Estos dos mandatos asumen que el fichero de políticas se llama *java.policy* y está disponible en el directorio actual de la parte servidora y cliente.

Una descripción detallada de las políticas de seguridad Java, incluyendo una explicación de la sintaxis utilizada en este fichero, se puede encontrar en [java.sun.com/marketing, 4].

Uso de descarga de resguardo y un fichero de políticas de seguridad

1. Si debe descargarse el resguardo de un servidor HTTP, transfiera la clase resguardo a un directorio apropiado del servidor HTTP, por ejemplo, al directorio **resguardos** de la máquina www.miempresa.com, y asegúrese de que el permiso de acceso del fichero es de lectura para todos los usuarios.
2. Cuando se activa el servidor, se debe especificar las siguientes opciones del mandato:

```
java –Djava.rmi.server.codbase=<URL>
      -Djava.security.policy=
      <ruta completa del fichero de políticas de seguridad>
```

donde

<URL> es el URL del directorio donde se encuentra la clase resguardo; por ejemplo, http://www.miempresa.com/resguardos/.

Obsérvese la barra del final del URL, que indica que el URL especifica un directorio, no un fichero.

<ruta completa del fichero de políticas de seguridad> especifica el fichero de políticas de seguridad de la aplicación; por ejemplo, java.security, si el fichero java.security se encuentra en el directorio actual.

Por ejemplo,

```
java –Djava.rmi.server.codebase=http://www.miempresa.com/resguardos/
-Djava.security.policy=java.security HolaMundoServidor
```

(todo en una línea)

arrancará la aplicación *HolaMundoServidor* y permitirá realizar descarga de resguardo del directorio *resguardos* del servidor web de www.miempresa.com.

La Figura 8.13 muestra el conjunto de ficheros que se necesitan para una aplicación RMI y dónde se deben colocar, suponiendo descarga dinámica de resguardo. (Por simplicidad, se asume que la aplicación no usa *callback* de cliente. Se podrían añadir los ficheros necesarios para realizar *callback* de cliente, si se deseara.)

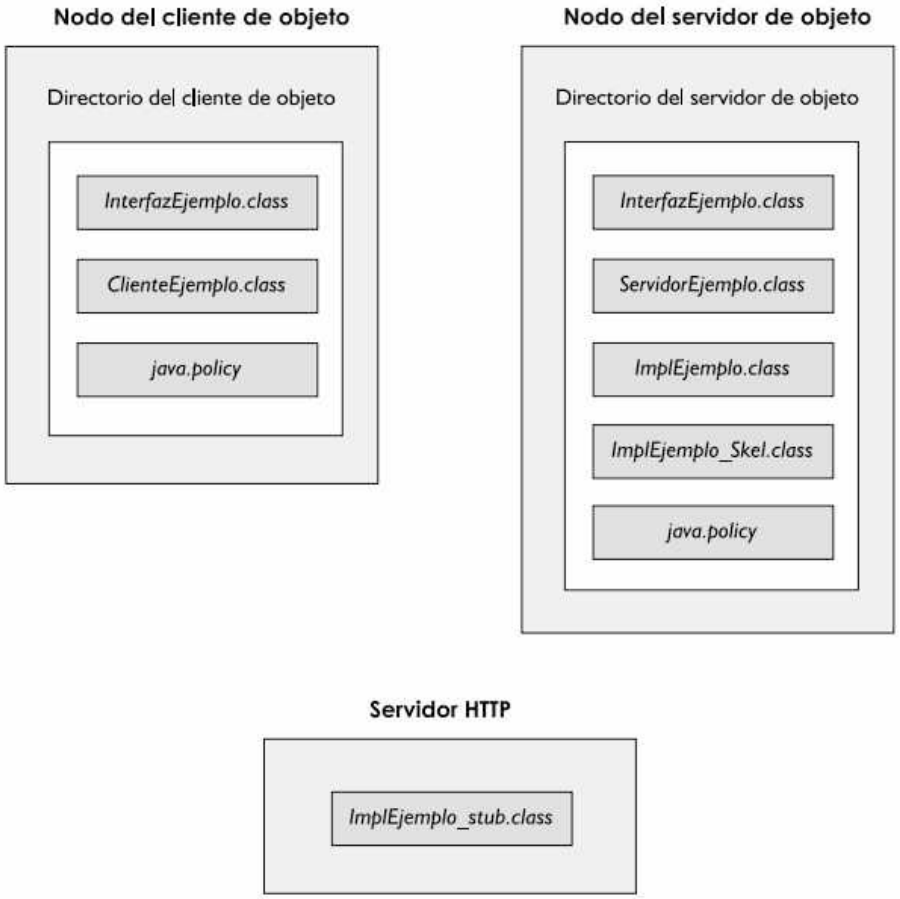


Figura 8.13. Colocación de los ficheros RMI en una aplicación que utiliza descarga de resguardo.

En la parte del servidor, los ficheros necesarios son los ficheros *class* del servidor, la interfaz remota, la implementación de la interfaz (generada por *javac*), el fichero *class* del resguardo (generado por *rmic*), la clase del esqueleto (generado por *rmic*), y el fichero de políticas de seguridad de la aplicación. En la parte cliente, los ficheros que se necesitan son el fichero *class* del cliente, el fichero *class* de la interfaz remota del servidor y el fichero de políticas de seguridad de la aplicación. Finalmente, el fichero *class* del resguardo se debe almacenar en el nodo HTTP del cual se descarga el resguardo.

Algoritmos para construir una aplicación RMI, que permita descarga de resguardo

A continuación se realiza una descripción del procedimiento paso a paso para la construcción de una aplicación RMI, teniendo en cuenta el uso de descarga de resguardo. De nuevo, por cuestiones de simplicidad, se han obviado los detalles para el *callback* de cliente.

Algoritmo para desarrollar el software de la parte del servidor

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Especificar la interfaz remota de servidor en *InterfazEjemplo.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
3. Implementar la interfaz en *ImplEjemplo.java*. Compilarla y revisarla hasta que no exista ningún error de sintaxis.
4. Utilizar el compilador RMI *rmic* para procesar la clase de la implementación y generar los ficheros resguardo y esqueleto para el objeto remoto:

   ```
   rmic ImplEjemplo
   ```

   Los ficheros generados se pueden encontrar en el directorio como *ImplEjemplo_Skel.class* y *ImplEjemplo_Stub.class*. Los pasos 3 y 4 deben repetirse cada vez que se cambie la implementación de la interfaz.
5. Crear el programa del servidor de objeto *ServidorEjemplo.java*. Compilarlo y revisarlo hasta que no exista ningún error de sintaxis.
6. Si se desea descarga de resguardo, copiar el fichero resguardo en un directorio apropiado del servidor HTTP.
7. Si se utiliza el registro RMI y no ha sido ya activado, activarlo. Por ejemplo:

   ```
   rmiregistry <número de puerto, 1099 por defecto>
   ```

   Alternativamente, se puede codificar la activación en el programa del servidor de objeto.
8. Construir un fichero de políticas de seguridad para la aplicación denominado *java.policy* (u otro nombre), y colocarlo en un directorio apropiado o en el directorio actual.
9. Activar el servidor, especificando (1) el campo *codebase* si se utiliza descarga de resguardo, y (2) el fichero de políticas de seguridad.

   ```
   java –Djava.rmi.server.codebase=http://nodo.dom.edu/resguardos/
   -djava.security.policy=java.policy ServidorEjemplo
   ```

   Este mandato se ejecuta en una única línea, aunque se puede utilizar un carácter de continuación de línea ('\') en un sistema UNIX. Se recomienda poner el mandato en un fichero de texto ejecutable (tal como *ejecServidor.bat* en un sistema Windows o *ejecServidor* en un sistema UNIX) y ejecutar el fichero para arrancar el servidor.

   Para la aplicación *HolaMundo*, el fichero *ejecServidor.bat* contendría esta línea:

   ```
   java-Djava.security.policy=java.policy
   -Djava.rmi.server.codebase=
   http://www.csc.calpoly.edu/-mliu/resguardos/ HolaMundoServidor
   ```

   De nuevo, no debería haber saltos de líneas en el fichero.

Algoritmo para desarrollar el software de la parte cliente

1. Crear un directorio donde se almacenen todos los ficheros generados por la aplicación.
2. Obtener una copia del fichero *class* de la interfaz remota del servidor. Alternativamente, obtener una copia del fichero fuente para la interfaz remota y compilarlo utilizando *javac* para generar el fichero *class* de la interfaz *InterfazEjemplo*.
3. Crear el programa cliente *ClienteEjemplo.java* y compilarlo para generar la clase cliente.
4. Si se desea utilizar descarga de resguardo, obtener una copia del fichero *class* del resguardo (supóngase que se llama *ImplEjemplo_Stub.class*) y colocarlo en el directorio actual.
5. Construir un fichero de políticas de seguridad para la aplicación denominado *java.policy* (u otro nombre), y colocarlo en un directorio apropiado o en el directorio actual.
6. Activar el cliente, especificando el fichero de políticas de seguridad.

   ```
   java -Djava.security.policy=java.policy ClienteEjemplo
   ```

   Este mandato se ejecuta en una única línea, aunque se puede utilizar un carácter de continuación de línea ('\') en un sistema UNIX. Se recomienda poner el mandato en un fichero de texto ejecutable (tal como *ejecCliente.bat* en un sistema Windows o *ejecCliente* en un sistema UNIX) y ejecutar el fichero para arrancar el cliente.

   Para la aplicación *HolaMundo*, el fichero *ejecCliente.bat* contendría esta línea:

   ```
   java -Djava.security.policy=java.policy HolaMundoCliente
   ```

   De nuevo, no debería haber saltos de líneas en el fichero.

   Si se utiliza *callback* de cliente, deben insertarse en este algoritmo los pasos adicionales descritos en la sección anterior «Pasos para construir una aplicación RMI con *callback* de cliente».

RESUMEN

En este capítulo se han analizado algunas de las características avanzadas del API de Java RMI. Aunque estas características no son parte inherente del paradigma de objetos distribuidos, son interesantes y útiles para algunas aplicaciones.

Callback de cliente

- El *callback* de cliente es útil para las aplicaciones que deseen que el servidor les notifique la ocurrencia de algún evento.
- El *callback* de cliente permite que un servidor de objeto realice una invocación de método remoto de un cliente a través de la referencia a una interfaz remota de dicho cliente.
- Para dotar a la aplicación de *callback* de cliente, el software de la parte cliente debe proporcionar una interfaz remota, instanciar un objeto que implemente dicha interfaz y pasar la referencia del objeto al servidor. El servidor de objeto

guarda estas referencias del cliente en una estructura de datos. Cuando el evento esperado ocurre, el servidor de objeto invoca un método *callback* (definido en la interfaz remota del cliente), pasando los datos a los clientes apropiados.

- Se necesitan dos conjuntos de resguardo y esqueleto: uno para la interfaz remota del servidor y el otro para la interfaz remota del cliente.

Descarga de resguardo y gestor de seguridad

- La característica de descarga de resguardo permite que un cliente pueda cargar una clase resguardo en tiempo de ejecución.
- Esta técnica requiere la configuración de la propiedad *java.rmi.server.codebase* cuando se inicia el servidor: esta propiedad debe configurarse con el directorio de un servidor HTTP donde se encuentre almacenada la copia del fichero *class* del resguardo y accesible a todos los usuarios.
- Esta técnica también requiere la instalación de un gestor de seguridad RMI tanto en la parte cliente como en la servidora.
- Para llevar a cabo descarga de resguardo, es necesario el uso de un gestor de seguridad, ya que la ejecución de un objeto descargado de una máquina desconocida puede suponer una amenaza para el computador cliente.
- Un gestor de seguridad lleva a cabo las restricciones especificadas en un fichero de políticas de seguridad de Java, que puede ser el fichero de políticas del sistema o un fichero de políticas aplicado solamente a una aplicación individual.
- En este capítulo se mostró un ejemplo de fichero de políticas de seguridad para aplicaciones RMI.
- Por cuestiones de seguridad, el uso de los gestores de seguridad es recomendable en todas las aplicaciones RMI, independientemente de que utilicen descarga de resguardo o no.

EJERCICIOS

1. En el contexto de Java RMI, ¿qué es el *callback* de cliente? ¿Por qué es útil?
2. Pruebe el programa ejemplo ***Callback*** en una o más máquinas.
   a. Cree una carpeta (denominada *callback*) para este ejercicio. Cree dos subcarpetas (con los nombres ***Servidor*** y ***Cliente***, respectivamente) en el PC. Copie los ficheros fuente en las carpetas ***Servidor*** y ***Cliente*** respectivamente.
   b. Siga los algoritmos presentados en el capítulo para configurar y ejecutar el servidor de objeto y el cliente de objeto. Escriba un informe indicando las acciones realizadas y las salidas.
   c. Arranque varios clientes sucesivamente. Escriba en el informe las acciones y las salidas de los programas.
   d. Copie el contenido de la carpeta *callback* en una carpeta nueva. Modifique los ficheros fuente en la nueva carpeta, de forma que el servidor notifique solamente a un cliente cuando se hayan registrado para *callback exactamente tres* clientes. Muestre los cambios realizados en los ficheros fuente.
   e. Volviendo a la carpeta *callback*, arranque un servidor y un cliente, especificando un tiempo de duración del cliente de 600 segundos. Mientras el

cliente espera por *callback*, aborte el proceso cliente mediante el uso de la secuencia «control-c». Rápidamente arranque un nuevo proceso cliente de forma que el servidor intente realizar un *callback* a todos los clientes registrados. Apunte lo que se observa.

Realice cambios al código fuente de forma que los problemas observados no ocurran. Describa los cambios realizados a los programas. Entregue los listados fuente modificados.

3. En un ejercicio del capítulo anterior, se pedía utilizar RMI para escribir una aplicación que fuera un prototipo de un sistema de encuestas de opinión. Modifique la aplicación para que cada cliente proporcione una interfaz de usuario de forma que se pueda realizar un voto. Adicionalmente, debe mostrar en la pantalla del usuario el recuento actual siempre que se realice un nuevo voto (sea desde este cliente o desde cualquier otro cliente).

   Recopile todos los listados de los ficheros fuentes, incluyendo los ficheros de las interfaces. Se recomienda una demostración de los programas en el laboratorio.

4. En el contexto de Java RMI, ¿qué es descarga de resguardo? ¿Por qué es útil?

5. Experimente con descarga de resguardo utilizando el ejemplo presentado en la Sección 8.3 de este capítulo. Los ficheros deben encontrarse en la carpeta *stubDownloading* de los ejemplos de programas.

   a. Cree una carpeta (llamada *stubDownloading*) para este ejercicio. Cree dos subcarpetas en el PC y llámelas *Servidor* y *Cliente*, respectivamente. Copie los ficheros RMI del ejemplo *HolaMundo* en la carpeta correspondiente.

   b. Compile los ficheros. Utilice *rmic* para generar los ficheros resguardo y esqueleto en la carpeta *Servidor*. Copie el fichero class del resguardo en la carpeta *Cliente*.

   c. Arranque un servidor de la carpeta *Servidor* sin especificar descarga de resguardo (es decir, sin configurar la propiedad *codebase* al arrancar el intérprete Java). A continuación, arranque un cliente de la carpeta *Cliente*. Compruebe que funcionan de la forma esperada.

   d. En la carpeta *Cliente*, borre el fichero *HolaMundoImpl_Stub.class*. Arranque un cliente de nuevo. Debería obtenerse una notificación de excepciones Java en tiempo de ejecución debido a la ausencia de la clase resguardo.

   e. Volviendo a la carpeta *Servidor*, copie el fichero *HolaMundoImpl_Stub.class* a un servidor web desde el que haya acceso, en un directorio denominado *resguardos* (o con algún otro nombre).

      Donde sea aplicable, establezca las protecciones de acceso al directorio *resguardos* y al fichero del resguardo, de forma que se pueda leer por todo el mundo. Arranque un servidor desde la carpeta *Servidor*, esta vez especificando descarga de resguardo (es decir, configurando la propiedad *codebase* para permitir que se realice descarga de resguardo desde el directorio donde se encuentra el fichero class del resguardo).

   f. Volviendo a la carpeta *Cliente*, pruebe a ejecutar el cliente de nuevo. Si las funciones de descarga de resguardo funcionan como se espera, el cliente debe funcionar esta vez.

   g. Pruebe a duplicar la carpeta *Cliente* y arranque otros clientes en diferentes sistemas de la misma forma, utilizando descarga de resguardo.

Escriba un informe describiendo el experimento.

6. Repita el ejercicio en el ejemplo ***Callback***. Esta vez, el cliente debe poder realizar una descarga de resguardo del servidor dinámicamente, mientras que el servidor debe poder obtener el resguardo del cliente dinámicamente también.

   Se recomienda que la primera vez que se realice este experimento, se utilice una copia de los ficheros de resguardo. A continuación borre el resguardo de servidor de la carpeta cliente, y pruebe a ejecutar el cliente con descarga de resguardo. Consecutivamente, borre el resguardo del cliente de la carpeta servidor, y pruebe a ejecutar el servidor con descarga de resguardo. Escriba un informe describiendo el experimento.

REFERENCIAS

1. Descarga dinámica de código utilizando RMI, http://java.sun.com/products/jdk/1.2/docs/guide/rmi/codebase.html
2. Introducción a la computación distribuida con RMI, http://developer.java.sun.com/developer/onlineTraining/rmi/RMI.html
3. CERT® Coordination Center, CERT/CC Computer Virus Resources, http://www.cert.org/other_sources/viruses.html
4. Java Remote Method Invocation – Computación distribuida para Java, http://java.sun.com/marketing/collateral/javarmi.html

CAPÍTULO 9

Aplicaciones de Internet

Sin ningún tipo de duda la aplicación distribuida más conocida es la ***World Wide Web*** (WWW), o la Web. Desde el punto de vista tecnológico la Web es un sistema distribuido de servidores HTTP y clientes, también conocidos como servidores Web y clientes Web. Este capítulo examina diversos protocolos de Internet que son de interés no sólo por razones prácticas, sino porque proporcionan un caso de estudio de cómo los protocolos evolucionan a lo largo del tiempo en respuesta a su demanda de uso.

Antes de la aparición de la Web, la comunidad de usuarios de Internet estaba formada por investigadores y académicos que usaban servicios de red, tales como el correo y la transferencia de ficheros para el intercambio de datos.

La *World Wide Web* (Telaraña Mundial) nació gracias a Tim Berners-Lee [w3.org/People, 4] a finales de 1990 en el CERN, el Laboratorio Europeo de Física de Partículas en Ginebra, Suiza [public.Web.cern.ch, 6]. Tim Berners-Lee y Robert Cailliau realizaron la propuesta «*universal hypertext system*» en noviembre de 1990. Desde esta propuesta original, el crecimiento de la *World Wide Web* ha sido extraordinario (*véase* Figura 9.1); la Web se ha expandido más allá de la comunidad científica y académica, llegando a todos los sectores del mundo, tanto comerciales como domésticos. El *World Wide Web Consortium* (W3C) [w3.org, 11] es el encargado de coordinar el continuo desarrollo de las tecnologías Web.

En los años 60, Doub Engelbart, el inventor del ratón, realizó el prototipo de un sistema on-line (*oNLine System*) denominado NLS que tenía navegación y edición hipertexto. Sin embargo, fue Ted Nelson el que acuñó el término hipertexto [w3.org/History.html, 2].

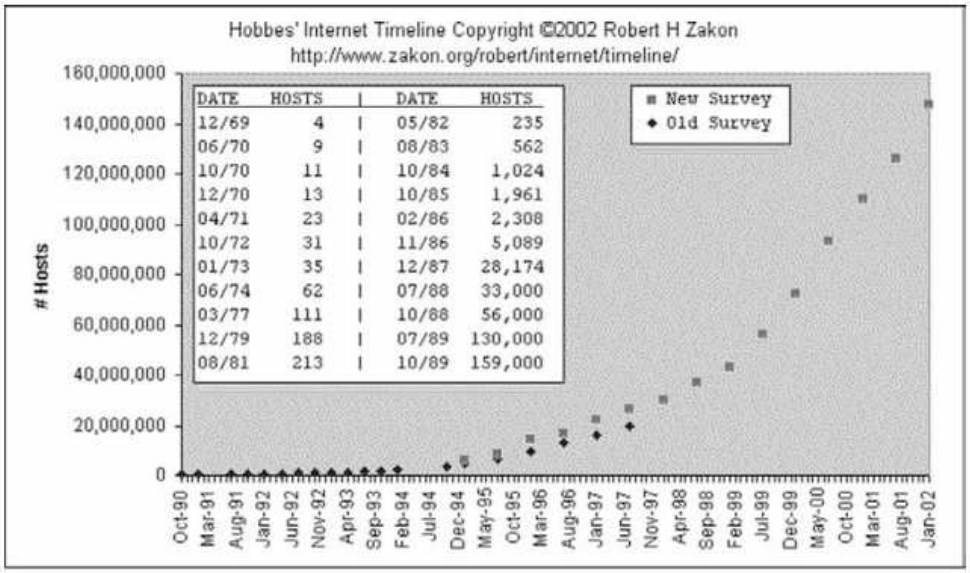


Figura 9.1. El crecimiento de la WWW [zakon.org, 1] (reimpresa con permiso).

La genialidad de la *World Wide Web* es que combina tres tecnologías de computación importantes y bien asentadas:

1. **Documentos hipertexto**. Los documentos **hipertexto** preceden a la WWW y son documentos en los que ciertas frases o palabras, normalmente destacadas, pueden ser usadas como enlaces a otros documentos. De esta forma, un usuario es capaz de acceder a documentos enlazados haciendo clic con el ratón en el texto destacado.
2. **Recuperación de información a través de una red**. Antes de la aparición de la WWW, el servicio FTP (*File Transfer Protocol*, Protocolo de Transferencia de Ficheros) [ietf.org, 12] era el servicio más utilizado para la recuperación de información.
3. **SGML (*Standard Generalized Marlup Language*, Lenguaje Estandarizado de Marcado General)**. En 1986 se creó un nuevo estándar ISO (ISO 8879) [iso.org, 3] que permitía marcar los documentos con etiquetas. Con este estándar los documentos se podían representar en un formato uniforme en cualquier plataforma e independiente del mecanismo de presentación.

Combinando estos tres conceptos, la *World Wide Web* permite marcar un documento con el lenguaje **HTML (*Hypertext Marlup Language*, Lenguaje de Marcado de Hipertexto)**, de forma que un documento enlazado puede ser automáticamente transferido de un computador remoto de Internet y presentado en un computador local.

Básicamente la *World Wide Web* es una aplicación cliente-servidor basada en el protocolo **HTTP (*Hypertext Transfer Protocol*, Protocolo de Transferencia de Hipertexto)** [faqs.org, 8], que ya ha sido descrito brevemente en capítulos anteriores. Un servidor web es un servidor orientado a conexión que implementa HTTP y que por defecto ejecuta en el puerto 80. Un usuario ejecuta un cliente *World Wide Web* (normalmente conocido como navegador) en una máquina local. El cliente interactúa con el servidor web de acuerdo al protocolo HTTP, especificando el documento a

ferir. Si el servidor localiza el documento en su directorio, devuelve su contenido al cliente, que lo presenta al usuario.

La popularidad de la Web fue tal que, según se dijo, la carga del primer servidor Web, info.cern.ch, se multiplicó por 10 cada año entre 1991 y 1994.

9.1. HTML

HTML (*Hypertext Markup Language*, lenguaje de marcado de hipertexto) [archive.ncsa.uiuc.edu, 14] es el lenguaje de etiquetado utilizado para crear documentos que pueden ser recuperados empleando la *World Wide Web*. HTML está basado en SGML, con semánticas apropiadas para representar la información de un amplio rango de tipos de documentos. HTML puede representar noticias, correos o documentación hipertexto; menús de opciones; resultados de acceso a bases de datos; documentos estructurados con imágenes embebidas; y vistas hipertexto de cuerpos de información existentes. La Figura 9.2 muestra una página web escrita en HTML.

Figura 9.2. Código HTML de ejemplo.

```
<HTML>
<HEAD>
<TITLE>Un ejemplo de pagina Web</TITLE>
</HEAD>
<HR>
<BODY>
<center>
<H1>Mi pagina principal</H1>
<IMG SRC="/imagenes/miFoto.gif">
<b>Bienvenido a la pagina de Kelly</b>
<p>
<! A continuacion una lista de hiperenlaces>
<a href="/doc/miCV.html">Mi curriculum vitae</a>
<p>
<a href="http://www.algunaUniversidad.edu/">Mi universidad<a>
</center>
<HR>
</BODY>
</HTML>
```

9.2. XML

Mientras que HTML es un lenguaje que permite etiquetar un documento para la posterior presentación de la información en él contenida, **XML (*Extensible Markup Language*, lenguaje extensible de etiquetado)** [w3.org/XML, 15] permite etiquetar un documento para estructurar su información. También basado en el SGML, XML utiliza etiquetas para describir la información contenida en el documento. La Figura 9.3 presenta un ejemplo muy simple de código XML [java.sun.com, 16] que describe la información estructurada que representa a un mensaje. Las etiquetas en este ejemplo

identifican el mensaje como una entidad, las direcciones de destino y de envío, el tema y el texto del mensaje.

Desde su introducción en 1998, XML ha sido ampliamente utilizado en la computación distribuida. Se utiliza en protocolos tales como SOAP (*Simple Object Access Protocol*, Protocolo Simple de Acceso a Objetos) para llamadas a procedimientos remotos basados en la Web y Jabber [jabber.org, 26] para mensajería instantánea.

Figura 9.3. Código XML de ejemplo.

```
<mensaje>
  <para>tu@tuDireccion.com</para>
  <de>yo@miDireccion.com</de>
  <tema>Esto es un mensaje</tema>
  <texto>
    ¡Hola mundo!
  </texto>
</mensaje>
```

9.3. HTTP

Originalmente concebido para el envío y la presentación de ficheros de texto, HTTP [w3.org/Protocols/HTTP/HTTP2.html, 7; ietf.org, 10; w3.org/Protocols/HTTP/AsImplemented.html, 17] se ha extendido para permitir la transferencia de contenidos web de tipos virtualmente ilimitados. Por eso se le conoce como el **protocolo de transporte** de la Web.

La primera versión de HTTP, **HTTP/0.9**, era un protocolo simple para la transferencia de datos crudos. La versión de HTTP más ampliamente utilizada es **HTTP/1.0**, cuyo borrador [w3.org/Protocols/HTTP/AsImplemented.html, 17] fue propuesto por Tim Berners-Lee en 1991. Aunque no tiene una especificación formal, su «uso común» se describe en el RFC 1945 [faqs.org, 8]. Desde ese momento, se ha desarrollado y a menudo adoptado un nuevo protocolo mejorado conocido como **HTTP/1.1**. HTTP/1.1 es un protocolo mucho más extenso que HTTP/1.0, pero las raíces básicas del protocolo están bien definidas en el más sencillo HTTP/1.0. En el resto de esta sección se presentará lo esencial del HTTP/1.0 y se indicarán las principales diferencias entre el HTTP/1.0 y el HTTP/1.1. En este capítulo, para mayor claridad, sólo se presentará un subconjunto del protocolo.

HTTP es un **protocolo orientado a conexión, sin estado y de petición-respuesta**. Un servidor HTTP, o servidor web, ejecuta por defecto en el puerto TCP 80. Los clientes HTTP, normalmente denominados navegadores web, son procesos que implementan el protocolo HTTP para poder conectarse a los servidores web y obtener documentos HTML. Estos documentos serán presentados de acuerdo a las etiquetas de dicho documento.

En HTTP/1.0 cada conexión sólo permite una ronda de petición-respuesta: un cliente obtiene una conexión y manda una petición; el servidor procesa la petición, emite una respuesta y finaliza cerrando la conexión. La Figura 9.4, que ya fue vista en el Capítulo 2, es un diagrama de eventos que describe una sesión HTTP/1.0.

HTTP es un protocolo de petición-respuesta basado en texto, ya que tanto la petición como la respuesta son cadenas de caracteres. Cada petición y cada respuesta están compuestas, en orden, por las siguientes partes:

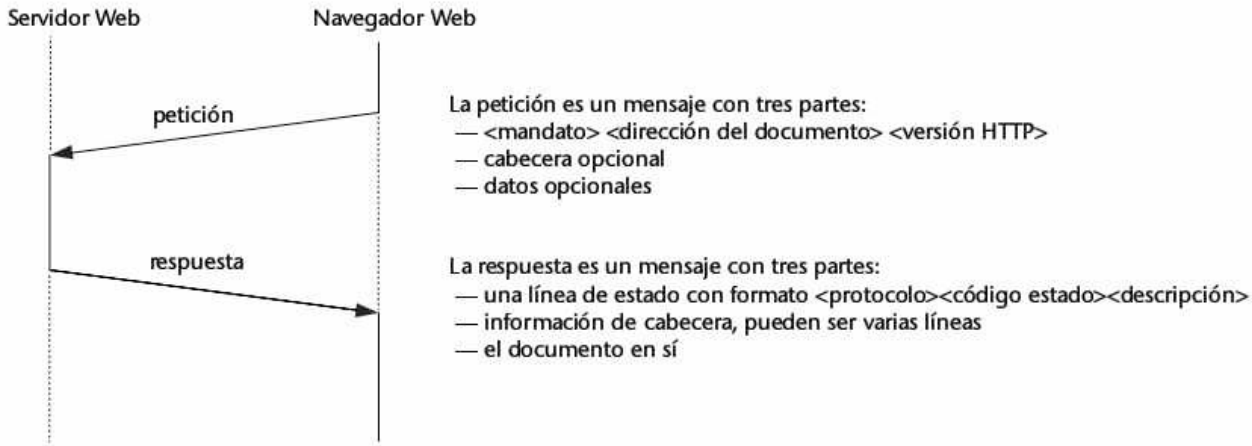


Figura 9.4. Diagrama de eventos del protocolo de transferencia de hipertexto.

1. La línea de **petición/respuesta**.
2. Una **sección de cabecera**.
3. Una línea en blanco.
4. El **cuerpo**.

A continuación se describen cada una de estas tres partes: primero la petición y después la respuesta.

La petición del cliente

La petición del cliente se envía al servidor después de establecer la conexión con el mismo.

La línea de petición

Una línea de petición tiene el siguiente formato:

```
<método HTTP><espacio><URI solicitado><espacio><especificación del
protocolo>\r\n
```

donde:

- *<método HTTP>* es el nombre de un método definido por el protocolo (que se describirán posteriormente),
- *<URI solicitado>* es el URI de un documento web o, de forma más genérica, un objeto web,
- *<especificación del protocolo>* es una especificación del protocolo del cliente,
- *<espacio>* es un carácter de espacio.

A continuación se muestra un ejemplo de línea de petición:

```
GET /index.html HTTP/1.0
```

El *<método HTTP>* en la petición de un cliente es una palabra reservada que especifica la operación que el cliente desea del servidor. Algunos de los principales métodos solicitados por los clientes son:

- *GET* – para solicitar el contenido de un objeto web referenciado por el URI especificado.
- *HEAD* – para solicitar sólo una cabecera del servidor, no el objeto completo.
- *POST* – usado para enviar datos a un proceso en el servidor.
- *PUT* – usado para solicitar al servidor que almacene el contenido adjunto a la petición en el archivo especificado por el URI.

El *<URI solicitado>* es un Identificador de Recurso Uniforme (*Uniform Resource Identifier*), ya introducido en el Capítulo 1, que tiene el siguiente formato.

```
/<nombre de directorio>.../<nombre de directorio>/<nombre de fichero>
```

La *<especificación del protocolo>* especifica el protocolo (nombre y versión) que asume el cliente. Por ejemplo, HTTP/1.0.

Cabecera de la petición

La línea de petición puede estar acompañada por una cabecera de petición. De acuerdo a [faqs.org, 8], «Los campos de la cabecera de la petición permiten al cliente pasar al servidor información adicional sobre la solicitud y sobre él mismo. Estos campos actúan como modificadores de la petición, con semánticas equivalentes a los parámetros en la invocación de métodos (procedimientos) en un lenguaje de programación».

Una cabecera está compuesta por una o más líneas, con el siguiente formato:

```
<clave>: <valor>\r\n
```

Algunas de las claves y valores que pueden aparecer en la cabecera de la petición son:

- **Accept** – especifica los tipos de contenido (*content-types*) aceptados por el cliente.
- **User-Agent** – especifica el tipo de navegador.
- **Connection** – se puede especificar «*Keep-Alive*» para solicitar que el servidor no cierre inmediatamente la conexión después de enviar la respuesta.
- **Host** – nombre del ordenador del servidor.

Un ejemplo de cabecera de la petición puede ser:

```
Accept: */*
Connection: Keep-Alive
Host: www.algunaU.edu
User-Agent: Generic
```

Cuerpo de la petición

Una petición puede finalizar con un cuerpo de petición, que contiene datos que necesitan ser enviados al servidor de forma conjunta con la petición. Por ejemplo, si se especifica el método POST en la línea de petición, el cuerpo deberá contener los datos que deben ser entregados al proceso destino. (Esta es una característica importante que se verá con mayor profundidad en las secciones de CGI y servlets).

A continuación se muestran algunos ejemplos de peticiones completas de cliente:

Ejemplo 1

```
GET / HTTP/1.1
<línea en blanco>
```

Ejemplo 2

```
HEAD / HTTP/1.1
Accept: */*
Connection: Keep-Alive
Host: algunaMaquina.com
User-Agent: Generic
<linea en blanco>
```

Ejemplo 3

```
POST /cgi/miServidor.cgi HTTP/1.0
Accept: */*
Connection: Keep-Alive
Host: algunaMaquina.com
User-Agent: Generic
Content-type: application/x-www-form-urlencoded
Content-length: 11
<línea en blanco>
Nombre=jorge&email=jorge@algunaU.edu
```

La respuesta del servidor

En respuesta a una petición de un cliente, el servidor HTTP debe enviar una respuesta. De forma similar a una petición, una respuesta HTTP está compuesta de las siguientes partes:

1. La respuesta o la línea de estado.
2. Una sección de cabecera.
3. Una línea en blanco.
4. El cuerpo.

La línea de estado

La línea de estado tiene el siguiente formato:

```
<protocolo><espacio><código de estado><espacio><descripción>\r\n
```

Los códigos de estado son los siguientes:

```
100-199 Informativo
200-299 Petición del cliente satisfactoria
300-399 Petición del cliente redirigida
400-499 Petición del cliente incompleta
500-599 Errores del servidor
```

Ejemplo 1

```
HTTP/1.0 200 OK
```

Es una línea de estado enviada por un servidor que ejecuta HTTP/1.0 y que indica que la petición se procesó de forma satisfactoria.

Ejemplo 2

```
HTTP/1.1 404 NOT FOUND
```

Es una línea de estado enviada por un servidor que ejecuta HTTP/1.1 y que indica que la respuesta no fue procesada de forma satisfactoria porque el documento especificado no se encontró.

Cabecera de la respuesta

La cabecera de la respuesta aparece a continuación de la línea de estado. La cabecera está compuesta de una o más líneas con el siguiente formato:

```
<clave>: <valor>\r\n
```

Existen dos tipos de líneas que pueden aparecer en la cabecera de la respuesta:

- **Líneas de respuesta**. Estas líneas de cabecera devuelven información sobre la respuesta, el servidor y detalles adicionales del acceso a los recursos solicitados, de la siguiente manera:

  ```
  Age: segundos
  Location: URI
  Retry-After: fecha|segundos
  Server: cadena
  WWW-Authenticate: método de autenticación
  ```

- **Líneas de entidad**. Estas líneas de cabecera contienen información sobre los contenidos de los objetos solicitados por el cliente, de la siguiente manera:

  ```
  Content-Encoding
  Content-Length
  Content-Type: tipo/subtipo (ver MIME)
  Expires: fecha
  Last-Modified: fecha
  ```

A continuación se muestra un ejemplo de cabecera de la respuesta:

```
Date: Mon, 30 Oct 2002 18:52:08 GMT
Server: Apache/1.3.9 (Unix) ApacheJServ/1.0
Last-modified: Mon, 17 June 2001 16:45:13 GMT
Content-Length: 1255
Connection: close
Content-Type: text/html
```

En [ietf.org, 10] se puede encontrar una lista completa de tipos de líneas de respuesta. Algunos de los tipos más importantes son:

- ***Content-Type*** que especifica el nombre del tipo de datos dentro del protocolo MIME, que será analizado más adelante en esta sección.

- ***Content-Encoding*** que especifica el esquema de codificado de los datos (tales como uuencode o base64), normalmente usados con propósito de compresión de datos.
- ***Content-length*** es el tamaño en número de bytes del contenido del cuerpo de la respuesta.
- ***Expiration date*** que da la fecha/hora (especificada en un formato definido por HTTP) después de la cual el objeto web se considera obsoleto.
- ***Last-Modified date*** que especifica la fecha en la que el objeto fue modificado por última vez.

uuencode significa codificación UNIX-a-UNIX (*UNIX-to-UNIX encode*) y en su origen fue un esquema de codificación utilizado para enviar contenido de texto entre sistemas UNIX. Sin embargo, ha sido extendido más allá de dichos sistemas.

Base64 es un esquema de codificación para contenidos de texto especificado en el protocolo MIME [faqs.org, 9].

Cuerpo de la respuesta

El cuerpo de la respuesta va a continuación de la cabecera y de una línea en blanco y contiene los contenidos del objeto web solicitado.

La Figura 9.5 contiene un ejemplo completo de respuesta HTTP.

Figura 9.5. Un ejemplo de respuesta HTTP.

```
HTTP/1.1 200 OK
Date: Sat, 15 Sep 2001 06:55:30 GMT
Server: Apache/1.3.9 (Unix) ApacheJServ/1.0
Last-Modified: Mon, 30 Apr 2001 23:02:36 GMT
ETag: "5b381-ec-3aedef0c"
Accept-Ranges: bytes
Content-Length: 236
Connection: close
Content-Type: text/html
<html>
<head>
<title>Mi página web</title>
</head>
<body>
¡Hola mundo!
</body></html>
```

Tipos de contenido y MIME

Una de las líneas de cabecera más importantes devueltas en la respuesta de un servidor es el tipo de contenido (*Content-Type*) del objeto solicitado. La especificación del tipo de contenido sigue un esquema establecido en el protocolo conocido como **MIME (*Multipurpose Internet Mail Extension*, extensiones multipropósito para el correo en Internet)**. Aunque originalmente se utilizó para correos electrónicos, hoy en día MIME es ampliamente utilizado para describir el contenido de los documentos mandados sobre una red.

MIME soporta un gran conjunto de tipos de contenido predefinidos, especificados con el formato **tipo/subtipo**. La Tabla 9.1 muestra un pequeño subconjunto de tipos y subtipos.

Tabla 9.1. Un subconjunto de Tipos de Contenido MIME.

Tipo	Subtipo
text	Plain, rich text, html, tab-separated values, xml
message	Email, news
application	Octet-stream (puede ser utilizado, por ejemplo, para enviar ficheros Java.class), Adobe-postscript, Mac-binhex40, xml
image	jpeg, gif
audio	basic,midi,mp3
video	mpeg, quicktime

Un cliente HTTP sencillo

Para afianzar los conocimientos obtenidos hasta este momento se va a realizar un cliente HTTP básico que ya se debería ser capaz de implementar. Indudablemente el lector ya habrá utilizado un navegador web comercial como Netscape o Internet Explorer para acceder y visualizar objetos web. Aunque la lógica de presentación en estos navegadores es compleja, la lógica del servicio no lo es, tal y como se puede apreciar en el cliente HTTP básico presentado en la Figura 9.6. En el código de ejemplo, el cliente utiliza un *socket* orientado a conexión para enviar al servidor web una petición formulada de acuerdo al HTTP. A continuación el cliente interpreta y presenta la respuesta devuelta por el servidor, línea por línea.

Figura 9.6. Un cliente HTTP sencillo escrito con la API de sockets.

```
1  // MiSocketStream es la clase Java presentada en el Capítulo 4
2  import MiSocketStream;
3  import java.net.*;
4  import java.io.*;
5
6  public class ClienteHTTP {
7
8  // Aplicación que se comunica con un servidor HTTP
9  // para recibir el contenido de texto de una página web
10 // Argumentos esperados, en orden:
11 // <nombre del servidor HTTP>
12 // <número de puerto del servidor HTTP>
13 // <ruta completa al documento web en la máquina servidora>
14
15  public static void main(String[] args) {
16   if (args.length != 3)
17     System.out.println
18        ("Este programa necesita 3 argumentos");
19   else {
20     try {
21        InetAddress maquina =
22           InetAddress.getByName(args[0]);
23        int puerto = Integer.parseInt(args[1]);
24        String nombreFichero = args[2].trim();
```

(continúa)

```
25          String peticion =
26             "GET " + nombreFichero + " HTTP/1.0\n\n";
27          MiSocketStream miSocket =
28             new MiSocketStream(maquina, puerto);
29          System.out.println("Conexión realizada");
30          miSocket.enviarMensaje(peticion);
31          // Ahora se recibe la respuesta del servidor HTTP
32          String respuesta;
33          respuesta = miSocket.recibirMensaje();
34          // Leer y mostrar una línea cada vez
35          while (respuesta != null) {
36             System.out.println(respuesta);
37             respuesta = miSocket.recibirMensaje();
38          }
39       }
40       catch (Exception ex) {
41          System.out.println("ERROR : " + ex) ;
42          ex.printStackTrace(System.out);
43       } // fin catch
44    } // fin else
45   } // fin main
46 } // fin class
```

El cliente HTTP abre una conexión con un servidor HTTP especificado por el usuario. A continuación el cliente formula una petición GET sencilla, envía la petición al servidor y visualiza la respuesta recibida del servidor. A diferencia de los navegadores comerciales, este navegador no interpreta la respuesta, simplemente muestra el texto tal y como lo recibe.

También es interesante saber que la API de Java proporciona una clase denominada URL, expresamente realizada para recibir datos de un objeto web identificado por su URI. La Tabla 9.2 describe dos constructores y un método clave dentro de esta clase, que son utilizados en el código de ejemplo de la Figura 9.7 para implementar este sencillo navegador web.

Tabla 9.2. Métodos del objeto Java URL.

Método/Constructor	Descripción
URL(String nombre)	Crea un objeto URL usando el nombre URL contenido en el String.
URL (String protocolo, String máquina, int puerto, String archivo)	Crea un objeto URL usando el protocolo, especificado, la maquina, el puerto y el archivo.
InputStream openStream( )	Abre una conexión a este URL y devuelve un InputStream con el cual leer de la conexión.

Figura 9.7. Un cliente HTTP sencillo escrito con la clase URL.

```
1  import java.net.*;
2  import java.io.*;
3
```

(continúa)

```
4  public class navegadorURL{
5
6  // Aplicación que utiliza el objeto URL para recibir
7  // el texto contenido en una página web.
8  // Argumentos esperados, en orden:
9  // <nombre del servidor HTTP>
10 // <número de puerto del servidor HTTP>
11 // <ruta completa al documento web en la máquina servidora>
12
13   public static void main(String[] args) {
14     if (args.length != 3)
15        System.out.println
16           ("Este programa necesita 3 argumentos");
17     else {
18        try {
19           String maquina = args[0];
20           String puerto = args[1].trim();
21           String nombreArchivo = args[2].trim();
22           String cadenaURL =
23              "http://" + maquina + ":" + puerto + nombreArchivo;
24           URL elURL = new URL("http", maquina, puerto, cadenaURL);
25
26           InputStream enStream = elURL.openStream( );
27           BufferedReader entrada =
28              new BufferedReader
29                 (new InputStreamReader(enStream));
30           String respuesta;
31           respuesta = entrada.readLine();
32            // Leer y mostrar una línea cada vez
33           while (respuesta != null) {
34              System.out.println(respuesta);
35              respuesta = entrada.readLine();
36           } //fin while
37        }
38        catch (Exception ex) {
39           System.out.println("ERROR : " + ex) ;
40           ex.printStackTrace(System.out);
41        } // fin catch
42     } // fin else
43   } // fin main
44 } //fin class
```

HTTP, un protocolo orientado a conexión y sin estado

Tal y como está especificado en [faqs.org, 8], HTTP es un protocolo orientado a conexión. Con HTTP/1.0 la conexión con el servidor se **cierra automáticamente** tan pronto como el servidor ha devuelto la respuesta. De esta forma, sólo se permite un turno de intercambio entre el cliente y el servidor web. Si un cliente necesita contactar con el mismo servidor más de una vez en una sesión, debe reconectarse al

servidor cada nueva petición. Este esquema es lógico con el propósito original de HTTP, es decir, recibir documentos simples por la red. Sin embargo, es ineficiente para recibir documentos que contienen un gran número de enlaces a objetos adicionales que deben ser pedidos al servidor, ya que la petición de cada uno de estos enlaces requiere reestablecer la conexión. También es un mecanismo insuficiente para aplicaciones web sofisticadas basadas en HTTP (como las que contienen carritos de compra). Como resultado HTTP/1.0 se extendió para permitir la cabecera de petición ***Connection: Keep-Alive***, que es enviada por los clientes que desean mantener una conexión persistente con el servidor; el servidor mantendrá la conexión abierta después de enviar la respuesta. En HTTP/1.1 las conexiones son persistentes por defecto. Una conexión persistente permite enviar múltiples respuestas sobre una misma conexión TCP.

HTTP/1.0 (al igual que la versión 1.1) es un protocolo sin estado: el servidor no mantiene ninguna información de estado sobre la sesión del cliente. Sin tener en cuenta si la conexión se mantiene abierta, cada petición se maneja por el servidor como una nueva. Tal y como sucedía con las conexiones no persistentes, un protocolo sin estado es adecuado para el propósito original del protocolo, pero no es apropiado para aplicaciones más complejas para las cuales HTTP ha sido extendido.

9.4. CONTENIDO WEB GENERADO DE FORMA DINÁMICA

Al principio HTTP se utilizaba para transferir contenido estático, es decir, contenido que existe de forma constante, como por ejemplo un fichero de texto plano o un fichero con una imagen. Según evolucionó la Web, las aplicaciones empezaron a utilizar HTTP con nuevos propósitos: como una aplicación que permite al navegador recibir datos basados en información dinámica introducida durante una sesión HTTP. Una aplicación web típica, tal como un carrito de la compra, requiere el envío de los datos introducidos por el cliente en tiempo de ejecución. Por ejemplo, una aplicación empresarial normalmente permite a un usuario introducir datos que más adelante son utilizados como parte de una consulta para recibir datos de una base de datos; a continuación, el resultado de esta consulta es formateado y enviado al usuario. Para implementar esta aplicación basada en la Web, es necesario permitir al cliente enviar datos durante una sesión web, para poder recibir datos de un servidor web, que sean representados por el navegador web (*véase* Figura 9.8). Con este propósito, el protocolo HTTP básico se extendió con protocolos adicionales que permitían la creación de contenidos web generados dinámicamente, es decir, contenidos que no existen de forma constante, sino que son generados dinámicamente de acuerdo a determinados parámetros en tiempo de ejecución.

En la Figura 9.8, las líneas punteadas con flechas representan el flujo de datos entre la fuente de datos (en este caso un sistema de base de datos) y el servidor HTTP. Las líneas de puntos indican que los datos no van directamente de la fuente de datos al servidor. La razón del flujo de los datos indirecto es que un servidor HTTP genérico no posee la lógica de aplicación para recoger los datos de la fuente de datos. En su lugar, un **proceso externo** que tiene la lógica de aplicación necesitará intervenir como intermediario. En este ejemplo, el proceso externo ejecuta en la máquina servidora, aceptando datos de entrada del servidor web, ejecutando su lógica de aplicación para obtener los datos de la fuente de datos y devolviendo el resultado al servidor web, que los transmitirá al cliente.

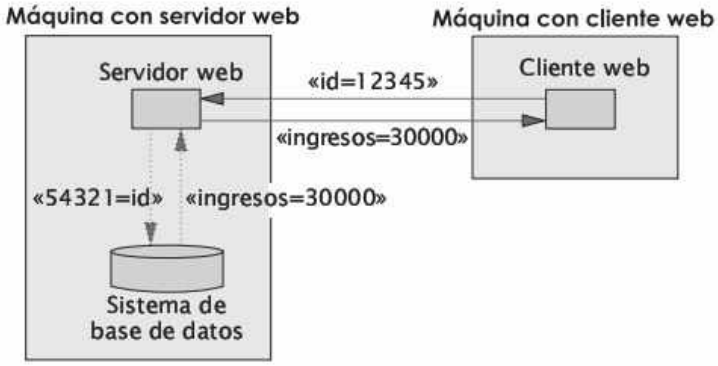


Figura 9.8. Contenido web dinámico.

El primer protocolo ampliamente adoptado para aumentar a HTTP en el soporte de contenido web generado en tiempo de ejecución es **CGI (*Common Gateway Interface*, interfaz estándar de pasarela)**, que se verá en el siguiente tema. Aunque es elemental en comparación con otros protocolos más sofisticados que han surgido posteriormente, CGI es el predecesor de estos sofisticados protocolos y facilidades (tales como Java servlet) que sirven para propósitos similares. Entender CGI y alguno de sus protocolos asociados es útil porque prepara para el estudio de protocolos y herramientas más avanzadas. Hoy en día los CGI siguen siendo extensamente utilizados en la Web.

9.5. CGI

CGI (*Common Gateway Interface*, interfaz estándar de pasarela) [hoohoo.ncsa.uiuc.edu/cgi/overview.html, 19; hoohoo.ncsa.uiuc.edu/cgi/interface.html, 20; comp.leeds.ac.uk, 21] es un protocolo que proporciona una interfaz, o pasarela, entre un servidor de información y un proceso externo (es decir, un proceso externo al servidor). Utilizando el protocolo CGI, un cliente web puede especificar un programa, conocido como ***script* CGI**, como objeto web de destino de una petición HTTP. El servidor web llama al *script* CGI y lo activa como un proceso, pasándole los datos de entrada transmitidos por el cliente web. El *script* web ejecuta y transmite su salida al servidor web, que devuelve los datos generados por dicho *script* como el cuerpo de la respuesta al cliente web.

El diagrama de secuencia de la Figura 9.9 muestra el funcionamiento de una aplicación CGI de ejemplo. Un navegador manda la petición de un documento estático *hola.html* (Figura 9.10). El servidor web busca el documento estático y devuelve su contenido al navegador dentro de cuerpo de la respuesta. El contenido (cuya sintaxis se verá en la próxima sección) contiene una llamada a un *script* web denominado *hola.cgi*. Cuando se presiona el botón Enviar, presentado en la pantalla del navegador, se manda una nueva petición al servidor web, especificando *hola.cgi* como objeto web a llamar. Suponiendo que el objeto pedido es un *script* CGI y, por tanto, ejecutable, el servidor comienza la ejecución del *script* pasándole los datos de entrada del navegador del usuario. En el transcurso de la ejecución del *script* CGI, se genera el contenido de una página web como salida del programa y se transmite al servidor. Por último, el servidor envía la página web generada dinámicamente al cliente web en el cuerpo de la respuesta.

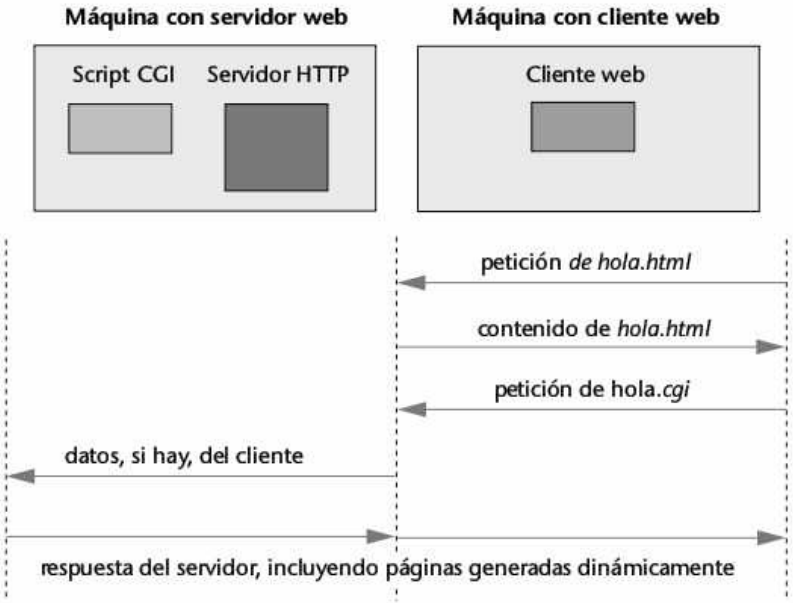


Figura 9.9. El protocolo CGI.

La Figura 9.10 contiene el código de la página web *hola.html*. Se verán los detalles de la sintaxis más adelante dentro de esta misma sección. Por el momento, es importante darse cuenta de que la etiqueta *ACTION* especifica el nombre del script CGI *hola.cgi*.

Figura 9.10. Hola.html.

```
<!Ejemplo de página web que invoca a un CGI>
<!Autor: M. Liu, 9/15/01>
<HTML>
<HEAD>
<TITLE>Página web que invoca a un script web</TITLE>
</HEAD>
<BODY>
<H1>Esta página muestra el uso de un script web</H1>
<P>
<FONT color = green>
Esta página contiene una etiqueta de acción que especifica un script
web. Después de recibir esta página del servidor web, el navegador
procesa su contenido y, cuando se presiona el botón ENVIAR, pide al
servidor que ejecute el script o programa especificado en la etiqueta
ACTION
</FONT>
<BR>
El script o programa puede estar escrito en un lenguaje de script, por
ejemplo Perl, o puede ser un ejecutable generado desde un código fuente
escrito en un lenguaje tal como C/C++
</P>
```

(continúa)

```
<HR>
<FORM METHOD="post" ACTION="hola.cgi">
<HR>
Presione <input type="submit" value="aquí"> para enviar su petición
</FORM>
<HR>
</BODY>
</HTML>
```

Java no se utiliza para implementar programas CGI. En el Capítulo 11 se verán los servlets de Java en profundidad; programas Java equivalentes a los *scripts* CGI.

Linefeed es un carácter de línea nueva (*new-line*), que tiene un valor ASCII de 10.

La Figura 9.11 muestra una implementación de *hola.cgi*. El ejemplo está escrito en C. Aunque pueda ser que el lector no esté familiarizado con el lenguaje C, su sintaxis es suficientemente parecida a Java como para poder entender los códigos de ejemplo de este capítulo. Un programa CGI puede estar escrito en cualquier lenguaje de programación, incluyendo los lenguajes interpretados (tales como Perl, TKL, Python, JavaScript, Visual Basic script), al igual que los lenguajes compilados (tales como C, C++, ADA). Sólo a modo de ejemplo, en la Figura 9.12 está una versión del programa escrita en el popular lenguaje de *script* Perl. Aunque sin estar familiarizado con Perl, es posible reconocer la sencilla sintaxis del programa, que simplemente produce el contenido de la página web línea por línea, comenzando con una línea de cabecera de respuesta que especifica el tipo de contenido, seguido por dos caracteres de nueva línea, y finalizado por las líneas HTML que especifican el mensaje «Hola Mundo» en color azul.

Figura 9.11. *Hola.cgi*, un programa en C con un *script* CGI.

```
/*
* Este programa C es un script CGI que genera
* la salida de una página web. Cuando es representado por
* el navegador, se muestra el mensaje "Hola Mundo" en azul
#include <stdio.h>

main(int argc, char *argv[]) {

   printf("Content-type: text/html%c%c",10,10);
   printf("<font color = blue>");
   printf("<H1>Hola Mundo</H1>");
   printf("</font>");
}
```

Las líneas de salida se transmiten al servidor web, que devuelve dichas líneas al cliente como respuesta. (Obsérvese que no hay línea de estado en la salida del script CGI: la ausencia de línea de estado se interpreta como estado correcto por el navegador.)

La Figura 9.12 presenta el mismo *script* web, escrito en el lenguaje interpretado Perl.

Figura 9.12. Hola.pl, un programa Perl con un script CGI.

```
#!/usr/local/bin/perl

# La línea superior debe ser la primera línea del programa.
# Especifica que se invoca al intérprete de Perl cuando
# se ejecuta el script. La ruta del directorio especificado
# debe ser la que contenga el intérprete de Perl
# en el servidor web.

# Hola.pl
# Un sencillo script CGI en Perl

print "Content-type: text/html\n\n";
print "<head>\n";
print "<title>Hola Mundo</title>\n";
print "</head>\n";
print "<body>\n";
print "<font color = blue>\n";
print "<h1> Hola Mundo</h1>\n";
print "</font>\n";
print "</body>\n";
```

El ejemplo que se ha presentado no hace uso de información suministrada por el usuario, y el contenido de la página web que se genera dinámicamente es predecible. Esto es debido a que el ejemplo proporcionado es un ejemplo básico del protocolo CGI. En la práctica, un *script* CGI se invoca desde una página web especial conocida como formulario web (que será descrito en la siguiente sección), que acepta entradas en tiempo de ejecución e invoca un *script* CGI que hace uso de esas entradas.

Un formulario web

Un formulario web [w3.org, 24] es un tipo especial de página web que (1) proporciona una interfaz gráfica de usuario que permite al usuario introducir datos y, (2) cuando el usuario pulsa el botón de Envío, invoca la ejecución de un programa externo en la máquina del servidor web. En la Figura 9.13 se puede ver la pantalla de un navegador mostrando un formulario web sencillo.

La Figura 9.14 presenta el código HTML que genera el formulario web. Se pueden encontrar más detalles sobre las etiquetas usadas en el código en [ftp.ics.uci.edu,13; archive.ncsa.uiuc.edu, 14].

Figura 9.14. Código de un formulario web sencillo [archive.ncsa.uiuc.edu, 14].

```
1 <HTML>
2 <HEAD>
3 <TITLE>Un ejemplo de formulario sencillo</TITLE>
4 </HEAD>
5 <BODY>
6 <H1>Esto es un formulario sencillo</H1>
7 <FORM METHOD="get" ACTION="formulario.cgi">
```

(continúa)

```
 8 <H2> Cuestionario: </H2>
 9 Introduce tu nombre: <INPUT NAME="nombre"><P>
10 ¿Cuál es tu pregunta?: <INPUT NAME="pregunta"><P>
11 ¿Cuál es tu color favorito?:
12 <SELECT NAME="color">
13 <OPTION SELECTED>verde amarillento
14 <OPTION>azul
15 <OPTION>pardo rojizo
16 <OPTION>aciano
17 <OPTION>gris oliva
18 <OPTION>bronce
19 <OPTION>añil
20 <OPTION>almendra blanca
21 <OPTION>verde
22 <OPTION>ocre
23 <OPTION>ópalo
24 <OPTION>ámbar
25 <OPTION>mostaza
26 </SELECT>
27 <P>
28 ¿Cuánto pesa una golondrina?:
29 <INPUT TYPE="radio" NAME="golondrina"
30 VALUE="african" checked> Golondrina africana o
31 <INPUT TYPE="radio" NAME="golondrina" VALUE="continental">
32 Golondrina continental
33 <P>
34 Algo más que desees añadir
35 <TEXTAREA NAME="texto" ROWS=5 COLS=60></TEXTAREA>
36 <P>
37 Presiona <INPUT TYPE="submit" NAME="sBoton" VALUE="aquí">
38 para enviar.
39 </FORM>
40 <HR>
41 </BODY>
42 </HTML>
```

HTML no es sensible a la diferencia entre letras mayúsculas y minúsculas: el uso de letras mayúsculas en las etiquetas es opcional.

El código que genera el formulario web se encierra entre las etiquetas *<FORM>* ... *</FORM>* (ver líneas 7 y 39 del ejemplo). Con la etiqueta *<FORM>* (ver línea 7), se pueden especificar atributos para suministrar información adicional al protocolo CGI, incluyendo:

- *ACTION=<URL>* es una cadena de caracteres que contiene el URL absoluto o relativo del programa externo que será iniciado por el servidor web al enviar el formulario; *véase* línea 7 de la Figura 9.14.
- *METHOD=<una palabra reservada>*, donde la palabra reservada es el nombre del método, *POST* o *GET*, que especifica la forma en que el programa externo espera recibir los datos enviados por el usuario, denominados datos de interro-

Un URL relativo está basado en el directorio de la página web en la que aparece dicho URL. Por ejemplo, si la página web es

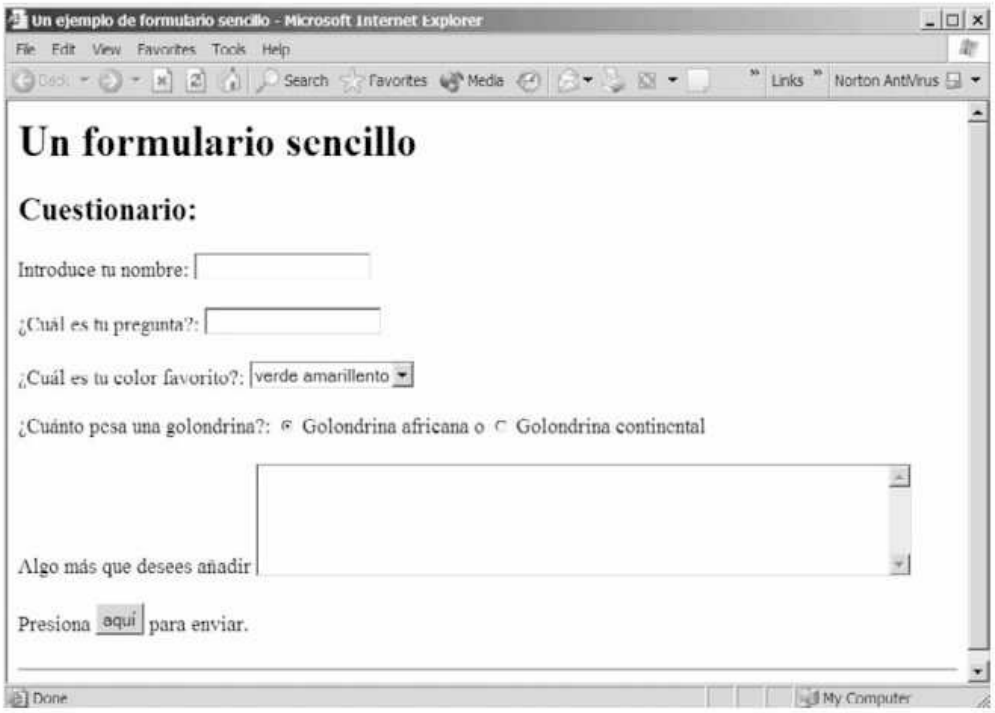


Figura 9.13. Un formulario web sencillo.

gación (*query data*). El significado de estos métodos se explicará más adelante; *véase* línea 7 de la Figura 9.14.

En el código del formulario, cada uno de los **elementos de entrada** tiene una etiqueta de *NOMBRE* (*véanse* líneas 9, 10, 12, 29, 31, 35 y 37). Para cada uno de estos elementos el usuario debe introducir o seleccionar un valor. La colección de todos los datos de los elementos de entrada es una cadena de caracteres, denominada **cadena de interrogación (*query string*)**, que está formada por pares *nombre=valor* separados por el carácter *&*. Cada par *nombre=valor* se codifica utilizando la codificación URL (*URL-encoding*) [w3.org, 22; blooberry.com, 23], de tal forma que los caracteres «no seguros» (tales como espacios, comillas, % y &) son representados de forma hexadecimal. Por ejemplo, la cadena «La respuesta es > 17%» se codifica como:

«La%20respuesta%20es%20%3E17%25».

A continuación se muestra un ejemplo de cadena de interrogación para el formulario de ejemplo:

```
nombre=Juan%20Nadie&pregunta=paz%20en%20mundo&color=azul&golondrina=continental
&texto=La%20respuesta%20es%20%3E17%25
```

El empaquetado de los datos en la cadena de caracteres, incluyendo la codificación de los valores, la realiza el navegador. Cuando el usuario envía el formulario, se le pasa la cadena de interrogación al servidor en la petición HTTP. La forma de envío de la cadena dependerá del atributo *METHOD* usado en el formulario. En la siguiente sección se explicará como se transmite la cadena de interrogación al servidor y más adelante al programa externo.

Procesamiento de la cadena de interrogación

Basándose en el formulario introducido, el navegador web construye la cadena de interrogación, tal y como se detalló previamente. La cadena se transmite al servidor web, que a su vez la pasa al programa externo (el script CGI especificado en el formulario). La forma en que se transmite la cadena depende de la especificación del atributo *METHOD* de la etiqueta *FORM* del formulario web.

El método *GET* en los formularios – envío de la cadena de interrogación al servidor

Si se especifica *GET* en la etiqueta *FORM METHOD*, la cadena de interrogación se transmite al servidor web en una petición HTTP con una línea de método *GET*, tal y como se describió en la sección 9.3. Se debe recordar que una petición HTTP *GET* especifica el URI del objeto web solicitado por el cliente. Para dar cabida a la cadena de interrogación, la sintaxis de la especificación del URI fue extendida para permitir que se adjuntara la cadena de interrogación al final del URI (para el *script* CGI), delimitándolo por el carácter '?', como por ejemplo,

```
GET /cgi/hola.cgi?nombre=Juan%20Nadie&pregunta=paz HTTP/1.0
```

Puesto que la longitud de la línea URI de la petición *GET* es limitada (debido al tamaño del *buffer* de entrada impuesto por los computadores), la longitud de la cadena de interrogación que puede ser agregada de esta manera tiene también un tamaño limitado. Por lo tanto, este método no es apropiado si el formulario necesita mandar una gran cantidad de datos, como pueden ser los datos introducidos en una caja de texto (*text box*).

El método *POST* en los formularios – envío de la cadena de interrogación al servidor

Si se especifica *POST* en la etiqueta *FORM METHOD*, la cadena de interrogación se transmite al servidor web en una cabecera HTTP con una línea de método *POST*, tal y como se describió en la sección 9.3. Se debe recordar que la petición HTTP *POST* tiene a continuación un cuerpo de petición, que contiene el texto a ser enviado al servidor. Utilizando el método *POST*, el URI del *script* CGI se especifica con la línea de petición *POST*, seguida por la cabecera de la petición, una línea en blanco, y por último la cadena de interrogación, como por ejemplo:

```
POST /cgi/hola.cgi HTTP/1.0
Accept: */*
Connection: Keep-Alive
Host: miHost.algo.edu
User-Agent: Generic

nombre=Juan%20Nadie&pregunta=paz%20en%20tierra&color=azul
```

Puesto que la longitud del cuerpo de la petición es ilimitada, la cadena de interrogación puede tener cualquier longitud. De esta forma, el método *POST* puede ser utilizado para enviar cualquier cantidad de datos al servidor.

El método *FORM GET* – envío de la cadena de interrogación al programa externo

Con el método *GET* el servidor invoca al *script* CGI y le pasa la cadena de interrogación que ha recibido del navegador, adjuntando el URI en la petición HTTP. El programa CGI, o en general el programa externo, recibirá el formulario codificado en una variable de entorno denominada *QUERY_STRING*. El programa CGI recupera la cadena de interrogación de la variable de entorno, decodifica la cadena de caracteres para obtener los pares nombre-valor, y hace uso de los valores de los parámetros durante la ejecución del programa para generar salida redactada en HTML.

Las variables de entorno son variables almacenadas por el sistema operativo de la máquina servidora.

El método *FORM POST* – envío de la cadena de interrogación al programa externo

El servidor invoca al *script* CGI y le pasa la cadena de interrogación que recibe del navegador en el cuerpo de la petición. El programa CGI, o en general el programa externo, recibirá el formulario codificado por la entrada estándar. El programa CGI lee la cadena de interrogación de la entrada estándar, decodifica la cadena de caracteres para obtener los pares nombre-valor, y hace uso de los valores de los parámetros durante la ejecución del programa para generar salida redactada en HTML.

En la mayor parte de los sistemas existe una entrada estándar para los procesos, normalmente el teclado, y una salida estándar, normalmente la pantalla.

Codificación y decodificación de la cadena de interrogación

Independientemente de que la cadena de interrogación sea obtenida de la variable de entorno *QUERY_STRING* o de la entrada estándar, el programa CGI debe decodificar la cadena y extraer los pares nombre-valor de la misma. De esta forma los parámetros podrán ser utilizados en la ejecución del programa. Gracias a la popularidad de los programas CGI, existen diversas bibliotecas o clases que proporcionan rutinas (funciones) y métodos para este propósito. Por ejemplo, Perl tiene unos procedimientos fáciles de usar, en una biblioteca denominada *CGI-lib*, para el decodificado y la extracción de los pares nombre-valor en una estructura de datos denominada *vector asociativo*; el NCSA (*The National Center for Supercomputing Applications*) proporciona una biblioteca de rutinas en C con el mismo propósito.

Un vector asociativo es una estructura de datos que mantiene un conjunto de pares (clave, valor).

La Figura 9.15 presenta un programa CGI, escrito en C, para cuando el formulario de ejemplo especifica el método *GET*. El programa utiliza unas rutinas denominadas *getword* y *unescape* (líneas 46-49) para la extracción y la decodificación de la cadena de interrogación. Los pares nombre-valor resultantes se almacenan en una estructura de datos declarada en las líneas 16-19 y 27, que se corresponde con un vector asociativo. Además, en la línea 39 el código hace uso de la rutina de C *getenv* para leer la cadena de interrogación de la variable de entorno *QUERY_STRING*. En este ejemplo, los nombres y valores recibidos son únicamente mostrados por pantalla (líneas 51-63).

Figura 9.15. *formularioGet.c* [hoohoo.ncsa.uiuc.edu, 19].

```
1 /* Autor: M. Liu, basado en un ejemplo del tutorial de CGIs de NCSA.
2 Este es el código fuente para formularioGet.cgi.
3 Se invoca desde formularioGet.html, y genera
4 dinámicamente una página web que muestra los pares
```

(continúa)

```
 5 nombre-valor obtenidos del formulario.
 6 Este programa utiliza la biblioteca CGI del NCSA para procesar
 7 la cadena de interrogación obtenida a través del método GET.
 8 */
 9 #include <stdio.h>
10 #ifndef NO_STDLIB_H
11 #include <stdlib.h>
12 #else
13 char *getenv();
14 #endif
15
16 typedef struct {
17 char nombre[128];
18 char valor[128];
19 } entrada;
20
21 void getword(char *word, char *line, char stop);
22 char x2c(char *what);
23 void unescape_url(char *url);
24 void plustospace(char *str);
25
26 main(int argc, char *argv[]) {
27   entrada entradas[10000];
28   register int x,m=0;
29   char *cl;
30
31   printf("Content-type: text/html%c%c",10,10);
32
33   if(strcmp(getenv("REQUEST_METHOD"),"GET")) {
34     printf("Este script debería ser referenciado
35     con el método GET.\n");
36     exit(1);
37   }
38
39   cl = getenv("QUERY_STRING");
40   if(cl == NULL) {
41     printf("No hay información que decodificar.\n");
42     exit(1);
43   }
44   for(x=0;cl[0] != '\0';x++) {
45     m=x;
46     getword(entradas[x].valor,cl,'&');
47     plustospace(entradas[x].valor);
48     unescape_url(entradas[x].valor);
49     getword(entradas[x].nombre,entradas[x].valor,'=');
50   }
51   printf("<BODY bgcolor=\"#CCFFCC\">");
52   printf("<H2>Esta página está generada
```

(continúa)

```
53      dinamicamente por formularioGet.cgi.</H2>");
54    printf("<H1>Resultados de la Cadena</H1>");
55    printf("Enviaste los siguientes pares nombre/valor",
56       "<p>%c",10);
57    printf("<ul>%c",10);
58
59    for(x=0; x <= m; x++)
60       printf("<li> <code>%s = %s</code>%c",
61          entradas[x].nombre, entradas[x].valor,10);
62    printf("</BODY>");
63    printf("</HTML>");
64 }
```

La Figura 9.16 presenta un programa CGI, escrito en C, para cuando el formulario de ejemplo especifica el método *POST*. El programa utiliza unas rutinas denominadas *makeword* y *unescape* (líneas 51-54) para la extracción y la decodificación de la cadena de interrogación. Los pares nombre-valor resultantes se almacenan en una estructura de datos declarada en las líneas 18-21 y 31, que corresponde a un vector asociativo. Obsérvese que el *for* que comienza en la línea 49 lee la cadena de interrogación de la entrada estándar, extrae una palabra cada vez, la decodifica, y escribe el nombre y el valor en la estructura de datos. En este ejemplo, los nombres y valores recibidos son únicamente mostrados por pantalla (líneas 56-66).

Figura 9.16. formularioPost.c [hoohoo.ncsa.uiuc.edu, 19].

```
 1 /* Autor: M. Liu, basado en un ejemplo del tutorial de CGIs de NCSA.
 2 Este es el código fuente para formularioPost.cgi.
 3 Se invoca desde formularioPost.html, y genera
 4 dinámicamente una página web que muestra los pares
 5 nombre-valor obtenidos del formulario.
 6 Este programa utiliza la biblioteca CGI del NCSA para procesar
 7 la cadena de interrogación obtenida a través del método POST.
 8 */
 9 #include <stdio.h>
10 #ifndef NO_STDLIB_H
11 #include <stdlib.h>
12 #else
13 char *getenv();
14 #endif
15
16 #define MAX_ENTRADAS 10000
17
18 typedef struct {
19 char *nombre;
20 char *valor;
21 } entrada;
22
23 char *makeword(char *line, char stop);
```

(continúa)

```
24 char *fmakeword(FILE *f, char stop, int *len);
25 char x2c(char *what);
26 void unescape_url(char *url);
27 void plustospace(char *str);
28
29
30 main(int argc, char *argv[]) {
31   entrada entradas[MAX_ENTRADAS];
32   register int x,m=0;
33   int cl;
34
35   printf("Content-type: text/html%c%c",10,10);
36   if(strcmp(getenv("REQUEST_METHOD"),"POST")) {
37     printf("Este script debería ser referenciado ",
38        "con el método POST.\n");
39     exit(1);
40   }
41   if(strcmp(getenv("CONTENT_TYPE"),
42        "application/x-www-form-urlencoded")) {
43     printf("Esta cadena de interrogación no contiene ",
44        "datos URLcodificados \n");
45     exit(1);
46   }
47   cl = atoi(getenv("CONTENT_LENGTH"));
48
49   for(x=0;cl && (!feof(stdin));x++) {
50     m=x;
51     entradas[x].valor = fmakeword(stdin,'&',&cl);
52     plustospace(entradas[x].valor);
53     unescape_url(entradas[x].valor);
54     entradas[x].nombre = makeword(entradas[x].valor,'=');
55   }
56   printf("<body bgcolor=\"#FFFF99\">");
57   printf("<H1>Resultados de la cadena</H1>");
58   printf("Enviaste los siguientes pares nombre-valor",
59     "<p>%c",10);
60   printf("<ul>%c",10);
61
62   for(x=0; x <= m; x++)
63     printf("<li> <code>%s = %s</code>\n",entradas[x].nombre,
64       entradas[x].valor);
65   printf("</body>");
66   printf("</html>");
67 }
```

Variables de entorno utilizadas en los CGI

Una **variable de entorno** es un parámetro del entorno de trabajo de un usuario en un computador, tal como la ruta por defecto para que el sistema localice los programas

invocados por un usuario o la versión del sistema operativo en uso. En un computador, las variables de entorno pueden ser empleadas en múltiples lenguajes y sistemas operativos para proporcionar información específica a una aplicación.

En sistemas UNIX, la configuración de variables de entorno se puede encontrar en ficheros ocultos (tales como .login o .cshrc) en el directorio raíz de las cuentas. En sistemas Windows se puede encontrar la configuración de las variables de entorno en el panel de control.

CGI utiliza variables de entorno que son escritas por el servidor HTTP para pasar información sobre las peticiones a los programas externos (script *CGI*).

A continuación se muestran algunas de las principales variables de entorno relacionadas con CGI:

- *REQUEST_METHOD*. El tipo de método con el que se ha realizado la petición. Para CGI puede ser *GET* o *POST*.
- *QUERY_STRING*. Si se ha especificado el método *GET* en el formulario, esta variable contiene una cadena de caracteres codificada, con los datos del formulario.
- *CONTENT_TYPE*. El tipo de contenido de los datos, que podría ser «*application/x-www-form-urlencoded*» para una cadena de interrogación.
- *CONTENT_LENGTH*. La longitud de la cadena de interrogación en número de bytes.

9.6. SESIONES WEB Y DATOS DE ESTADO DE LA SESIÓN

Durante una sesión de una aplicación web, tal como un carrito de la compra, se emiten diversas peticiones HTTP, cada una de las cuales puede invocar a un programa externo como puede ser un *script* CGI. La Figura 9.17 muestra una sesión simplificada de esta aplicación: el primer formulario web solicita un identificador (id) de cliente, que es validado por el *script* CGI *formulario1.cgi*. El *script* web genera dinámicamente un formulario web llamado *formulario2.html*. (Obsérvese que este fichero nunca se escribe a disco; sólo existe como salida del *script* web). El formulario pide al cliente que rellene una orden de compra. La orden de compra del cliente se manda a un segundo *script* web, *formulario2.cgi*, que genera dinámicamente otro formulario web transitorio (*formulario3.html*), que a su vez muestra los datos de la cuenta del cliente y el contenido de su carrito de la compra. La sesión puede continuar de esta forma invocando a *scripts* web adicionales y generando dinámicamente páginas web hasta que termina la sesión del usuario.

Es importante observar que en este ejemplo, para el segundo *script*, *formulario2.cgi*, es necesario saber el valor del elemento de datos *id* de la cadena de interrogación enviada por el primer *script* CGI, *formulario1.cgi*. Es decir, *id* es un dato de estado de sesión que necesita ser compartido entre los *scripts* web invocados a lo largo de la sesión. Sin embargo, debido a que los *scripts* web son programas separados ejecutados en contextos independientes, no comparten datos. Por otra parte, ni HTTP ni CGI tienen soporte para datos de estado de sesión, ya que ambos protocolos son sin estado y no tienen el concepto de sesión.

Debido a la popularidad de las aplicaciones de Internet han surgido diversos mecanismos que permiten compartir datos de sesión entre *scripts* CGI (y otros programas externos). Estos mecanismos pueden ser clasificados de la siguiente manera:

- **Mecanismos del lado del servidor**. Ya que los *scripts* CGI se ejecutan en el servidor HTTP, es lógico hacer uso de los mecanismos del mismo para mantener los datos de estado de la sesión.

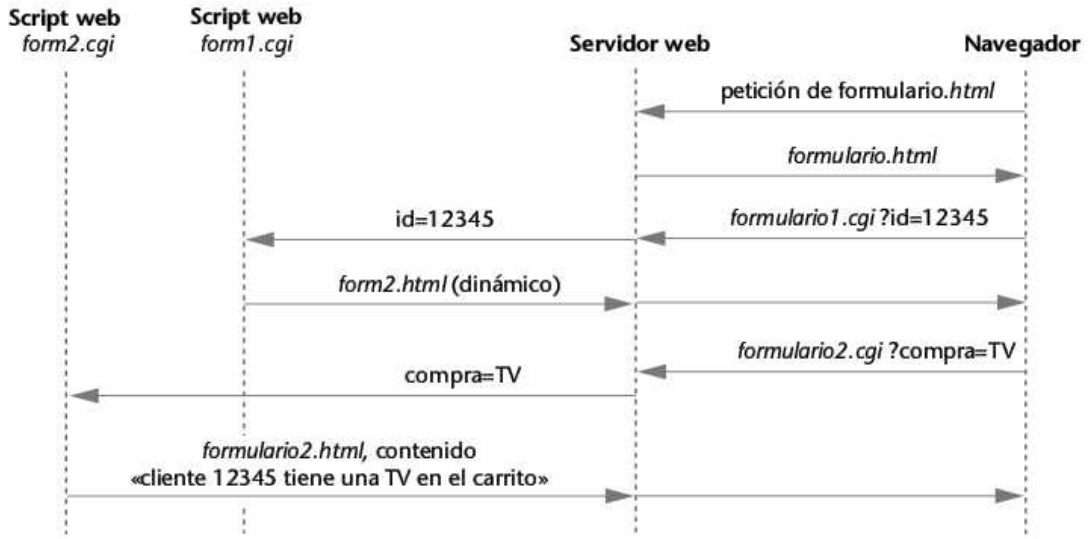


Figura 9.17. Una sesión web que utiliza múltiples programas externos.

Por ejemplo, se puede utilizar almacenamiento secundario (ficheros o bases de datos) en la máquina servidora como un repositorio para los datos de estado de la sesión: cada *script* web deposita sus datos de estado en el almacén, de tal forma que pueden ser accedidos por cualquier otro *script*. La desventaja de este esquema es la sobrecarga involucrada y la necesidad de administración del almacén para un gran número de sesiones concurrentes.

De forma alternativa, el servidor HTTP puede soportar objetos de datos persistentes que pueden ser empleados como repositorios de datos de estado. En el Capítulo 11 se verán algunos mecanismos basados en esta idea.

- **Mecanismos del lado del cliente**. Una ingeniosa idea para mantener los datos de estado de sesión es utilizar la ayuda de los clientes. Ya que cada sesión está asociada a un solo cliente, este esquema permite que los datos de estado se mantengan de forma descentralizada. Este esquema permite pasar los datos de estado del *script* web al cliente, que a su vez pasa estos datos al siguiente *script* web. Esta operación puede ser repetida durante toda la sesión web. La Figura 9.18 ilustra esa idea. El elemento de sesión *id* se pasa desde el primer *script* web, vía el servidor web, al navegador. Este elemento se envía del cliente web al siguiente *script* CGI cuando se invoca. A su vez, el *script* CGI 2 pasa los elementos de sesión *id* y *compra* al cliente web. En el siguiente paso, el navegador envía estos elementos al último *script* CGI, que aporta un elemento adicional, *cargo*, a los datos de estado de la sesión.

En el resto de este capítulo se verán dos esquemas que hacen uso de los mecanismos de mantenimiento de sesión del lado del cliente:

- **Campos ocultos de formulario**. Este esquema introduce los datos de estado de sesión en formularios web generados dinámicamente.
- ***Cookies***. Este mecanismo, de nombre tan gracioso[1] utiliza almacenamiento transitorio o persistente en la máquina del cliente para mantener los datos de estado. Estos datos se pasan en la cabecera de la petición HTTP al *script* web que los requiera.

[1] *N. del T.:* «cookies» significa en castellano «galletitas».

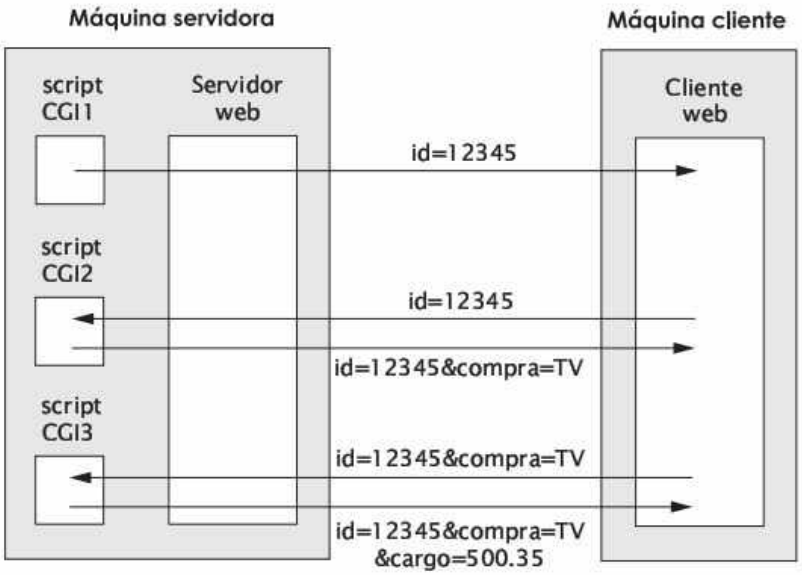


Figura 9.18. Datos de estado de sesión transmitidos vía cliente web.

Uso de campos ocultos de formulario para transferir datos de estado de sesión

Un campo oculto de formulario o **campo oculto** es un elemento de tipo *INPUT* en un formulario web que se especifica con *TYPE=HIDDEN*. Al contrario que otros elementos de tipo *INPUT*, el navegador no presenta un campo oculto y no requiere que el usuario introduzca datos. El valor de este elemento debe ser especificado en el atributo *VALUE* dentro del campo.

La Figura 9.19 muestra el uso de los campos ocultos para pasar datos entre *scripts* web. El primer *script* web *formulario1.cgi* genera el elemento *<input type=hidden name=«id» value=«l2345»>* dentro de *formulario2.html* dinámicamente generado. Cuando se representa *formulario2.html*, el navegador no mostrará este campo. El otro campo de entrada, que no está especificado como oculto, se presenta al usuario para que introduzca una compra. Cuando se envía *formulario2.html*, se envía la cadena de interrogación *«id=12345&compra=tv»* al segundo formulario web, *formulario2.cgi*. Cuando la cadena de interrogación se decodifica, el valor del elemento de estado *id* queda disponible para *formulario2.cgi*. La Figura 9.20 presenta al diagrama de secuencia.

El uso de campos ocultos es un esquema elemental para el mantenimiento de los datos de sesión. Tiene el mérito de la simplicidad, ya que sólo requiere la introducción de nuevos campos de formulario y no se necesitan recursos adicionales ni en el cliente ni en el servidor. En este esquema, el cliente HTTP se convierte en un repositorio temporal de la información de estado y los datos de sesión se envían utilizando el mecanismo normal de transmisión de cadenas de interrogación.

Como es lógico, existen contraprestaciones. La simplicidad del esquema conlleva el riesgo de privacidad y seguridad, en el sentido de que los datos de estado se transmiten como campos ocultos de formulario sin proteger. En el ejemplo mostrado en la Figura 9.20, el valor *id* de un cliente se transmite utilizando un elemento

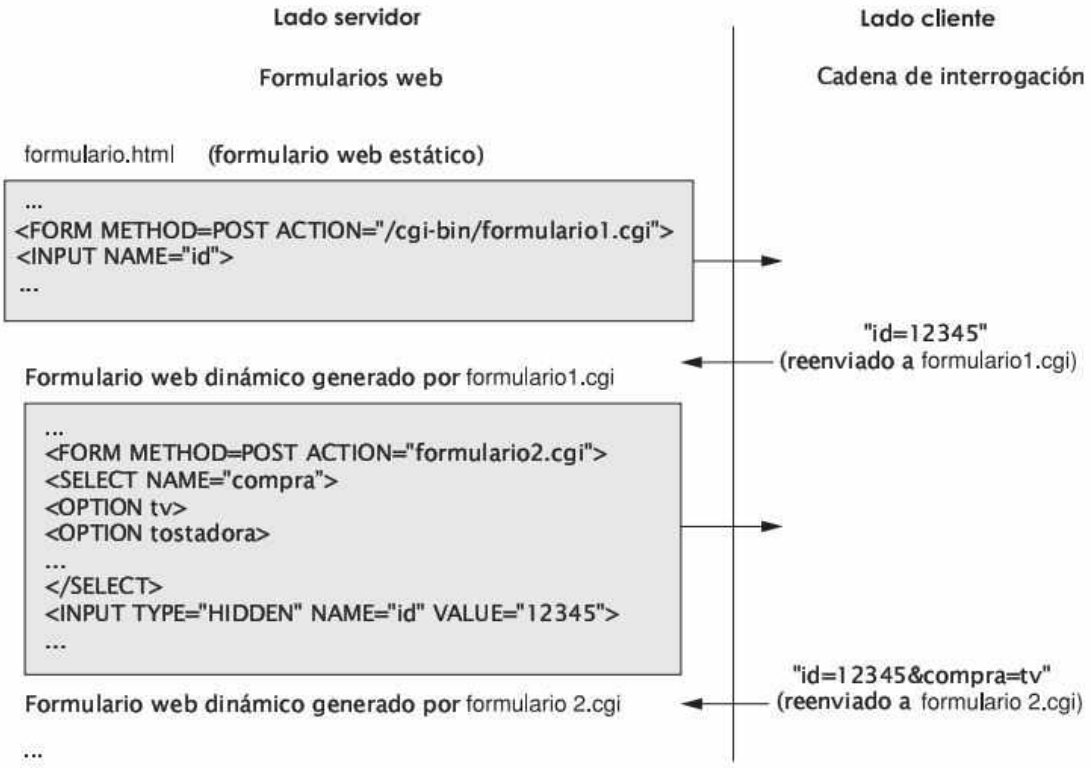


Figura 9.19. Uso de campos ocultos de formulario para enviar datos de estado; una visión general.

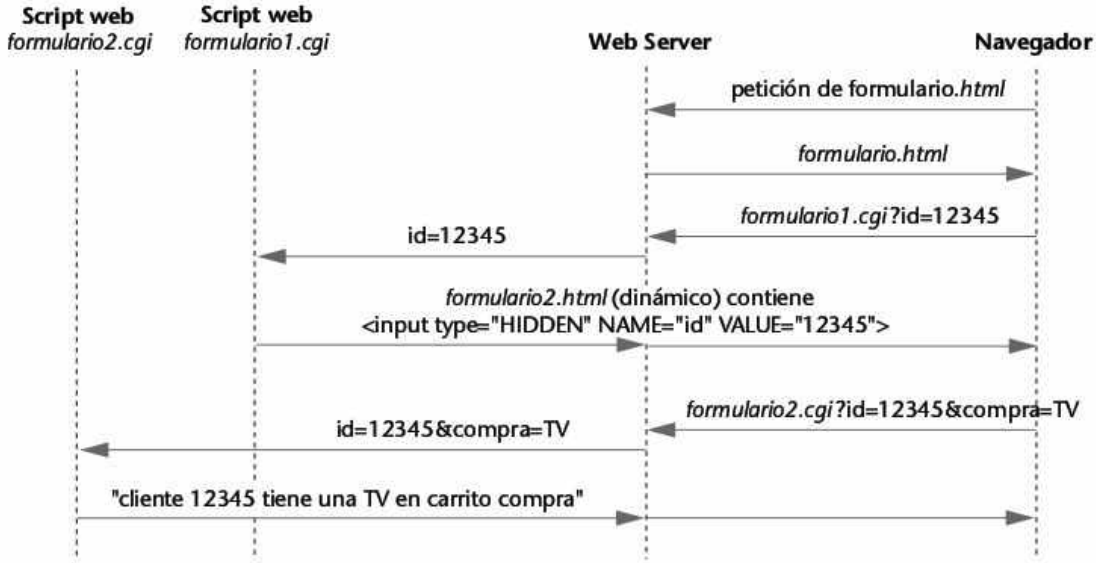


Figura 9.20. Uso de campos ocultos de formulario para pasar información de estado; vista detallada.

de entrada oculto. Aunque este elemento no es presentado por el navegador, va embebido en el código fuente de la página web generada dinámicamente, *formulario2.html*, cuyo código es visible por cualquier usuario del navegador. Por lo tanto, este esquema permite que los datos de estado sean expuestos y, por consiguiente, tiene el riesgo de privacidad y seguridad. Por esta razón, los campos ocultos no deben ser utilizados para transmitir datos sensibles tales como números de seguridad o balances de cuentas.

Las Figuras 9.21, 9.22 y 9.23 presentan ejemplos de código para el uso de campos ocultos de formulario para pasar datos de estado entre *scripts* CGI. El formulario web inicial (Figura 9.21) invoca al *script formularioOculto.cgi*, que extrae el par nombre-valor de la cadena de interrogación que recibe (líneas 43-48 de la Figura 9.22) y después incluye cada par en un campo oculto de formulario de la página web que genera (líneas 64-66 de la figura 9.22). Los campos de los datos ocultos se incluyen en la cadena de interrogación para el *script* CGI *formularioOculto2.cgi*, que extrae los pares nombre-valor (líneas 40-46) y simplemente los presenta (líneas 54-59).

Figura 9.21. Un ejemplo de campos ocultos de formulario: *formulario.html*.

```
<HTML>
<HEAD>
<TITLE>Ejemplo de campos ocultos de formulario</TITLE>
</HEAD>
<BODY>
<H1>Utilización de campos ocultos de formulario para información de
sesión</H1>
<P>
Este formulario invoca a un script CGI que utiliza campos
ocultos de formulario para pasar datos de estado a otro
script CGI que será ejecutado a continuación.
</P>
<HR>
<FORM method="post" action="formularioOculto.cgi">
<H2> Cuestionario: </H2>
Cuál es tu ID: <input name="id"><P>
<P>
Presiona <input type="submit" value="aquí"> para enviar tu solicitud.
</FORM>
<HR>
</BODY>
</HTML>
```

Figura 9.22. Un ejemplo de campos ocultos de formulario: formularioOculto.c.

```
1 /* formularioOculto.c – codigo fuente de formularioOculto.cgi
2 Author: M. L. Liu.
3 Este script genera dinámicamente una página web que
4 contiene campos ocultos de formulario con los pares
5 nombre-valor recibidos en la cadena de interrogación.
```

(continúa)

```
 6 */
 7
 8 #include <stdio.h>
 9 #ifndef NO_STDLIB_H
10 #include <stdlib.h>
11 #else
12 char *getenv();
13 #endif
14
15 #define MAX_ENTRADAS 10000
16
17 typedef struct {
18   char *nombre;
19   char *valor;
20 } entrada;
21
22 char *makeword(char *line, char stop);
23 char *fmakeword(FILE *f, char stop, int *len);
24 char x2c(char *what);
25 void unescape_url(char *url);
26 void plustospace(char *str);
27
28
29 main(int argc, char *argv[]) {
30   entrada entradas[MAX_ENTRADAS];
31   register int x,m=0;
32   int cl;
33
34   printf("Content-type: text/html%c%c",10,10);
35   if(strcmp(getenv("REQUEST_METHOD"),"POST")) {
36     printf
37        ("Este script debe ser utilizado con el método POST.\n");
38     exit(1);
39   }
40
41   cl = atoi(getenv("CONTENT_LENGTH"));
42
43   for(x=0;cl && (!feof(stdin));x++) {
44     m=x;
45     entradas[x].valor = fmakeword(stdin,'&',&cl);
46     plustospace(entradas[x].valor);
47     unescape_url(entradas[x].valor);
48     entradas[x].nombre = makeword(entradas[x].valor,'=');
49   }
50   /* Generar el formulario dinámico con los campos ocultos */
51   printf("<FORM method=\"post\" action=\"formularioOculto.cgi\">");
```

(continúa)

```
52   printf("<HR>Este formulario fue generado dinámicamente por ",
53      "formularioOculto.cgi</H2>");
54   printf("<H1>Resultados de la Petición</H1>");
55   printf("Has enviado los siguientes pares nombre/valor:<p>%c",10);
56   printf("<ul>%c",10);
57
58   for(x=0; x <= m; x++)
59      printf("<li> <code>%s = %s</code>%c",entradas[x].nombre,
60         entradas[x].valor,10);
61   printf("</ul>%c",10);
62
63   /* Se pone cada nombre-valor en un campo oculto de formulario */
64   for(x=0; x <= m; x++)
65      printf("<INPUT TYPE=\"HIDDEN\" NAME=%s VALUE=%s>\n",,
66         entradas[x].nombre, entradas[x].valor);
67
68   printf("Presiona <input type=\"submit\" value=\"aquí\"> para ",
69      "enviar tu petición.");
70   printf("<HR>");
71   printf("</FORM>");
72   printf("</BODY>");
73   printf("</HTML>");
74
75 }
```

Figura 9.23. Un ejemplo de campos ocultos de formulatio: formularioOculto2.c.

```
 1 */ formularioOculto2.c — código fuente para formularioOculto2.cgi
 2 Author: M. L. Liu.
 3 Este script presenta los pares nombre-valor que recibe
 4 en la cadena de interrogación. Deberían estar incluidos
 5 los pares nombre-valor que aparecen en el formulario oculto
 6 de la página web que invoca este script.
 7 */
 8 #include <stdio.h>
 9 #ifndef NO_STDLIB_H
10 #include <stdlib.h>
11 #else
12 char *getenv();
13 #endif
14
15 #define MAX_ENTRADAS 10000
16
17 typedef struct {
18 char *nombre;
19 char *valor;
20 } entrada;
```

(continúa)

```
21
22 char *makeword(char *line, char stop);
23 char *fmakeword(FILE *f, char stop, int *len);
24 char x2c(char *what);
25 void unescape_url(char *url);
26 void plustospace(char *str);
27
28 main(int argc, char *argv[]) {
29   entrada entradas[MAX_ENTRADAS];
30   register int x,m=0;
31   int cl;
32
33   printf("Content-type: text/html%c%c",10,10);
34   if(strcmp(getenv("REQUEST_METHOD"),"POST")) {
35     printf("Este script debería usar el método POST.\n");
36     exit(1);
37   }
38   cl = atoi(getenv("CONTENT_LENGTH"));
39
40   for(x=0;cl && (!feof(stdin));x++) {
41     m=x;
42     entradas[x].valor = fmakeword(stdin,'&',&cl);
43     plustospace(entradas[x].valor);
44     unescape_url(entradas[x].valor);
45     entradas[x].nombre = makeword(entradas[x].valor,'=');
46   }
47
48   printf("<H2>Este script fue generado dinámicamente por ",
49     "formularioOculto2.cgi</H2>");
50   printf("<H1>Resultados de la petición</H1>");
51   printf("Enviaste los siguientes pares nombre-valor:<p>%c",10);
52   printf("<UL>%c",10);
53
54   for(x=0; x <= m; x++)
55     printf("<li> <code>%s = %s</code>%c",entradas[x].nombre,
56       entradas[x].valor,10);
57   printf("</UL>%c",10);
58   printf("</HTML>");
59   printf("</BODY>");
60 }
```

Uso de *cookies* para el envío de datos de estado de sesión

Un esquema más complejo para repositorio de datos de estado de sesión en la parte cliente es el mecanismo conocido como ***cookies***, así denominado «por ninguna razón convincente» [ics.uci.edu, 5; home.netscape.com, 25]. Sin embargo, en http://www.webopedia.com se afirma que «el nombre de las *cookies* proviene de unos objetos de Unix denominados *magic cookies*. Son señales (*tokens*) que están ligados a un usuario o programa y que varían dependiendo de las áreas en que entra el usuario o el programa».

Este esquema hace uso de una extensión del HTTP básico que permite que una respuesta del servidor pueda contener información de estado que el cliente deberá almacenar en un objeto. Según [home.netscape.com, 25], «En este objeto de estado debe ir incluida una descripción del rango de URLs para las cuales este estado es válido. Cualquier futura petición hecha por el cliente que caiga dentro de este rango incluirá la transmisión del valor actual del objeto de estado desde al cliente al servidor».

Cada *cookie* contiene un par nombre-valor con codificación URL, similar al par nombre-valor de la cadena de interrogación, para cada elemento de datos de estado (por ejemplo, id=12345). Un *script* CGI puede crear una *cookie* incluyendo la línea de cabecera *Set-Cookie* como parte de la respuesta HTTP que genera. Cuando el navegador recibe esta respuesta, crea un objeto, una *cookie*, que contiene el par nombre-valor especificado. La *cookie* se almacena en la máquina cliente, de forma temporal o de forma persistente. A continuación, el par nombre-valor de la *cookie* se recoge y se inserta en una línea de cabecera de tipo *Cookie* en cada petición enviada por el navegador al servidor web. Cuando el servidor web encuentra una línea de cabecera de tipo *Cookie*, recoge los pares nombre-valor en una cadena y la sitúa en una variable de entorno denominada *HTTP_COOKIE*. El formato de la cadena *HTTP_COOKIE* es el mismo que el descrito en la sección 9.5 para la cadena de interrogación.

La Figura 9.24 muestra el uso de las *cookies*. El *script formulario.cgi* introduce una *cookie* que contiene «id=12345», que se manda al navegador web en una línea de cabecera de respuesta generada dinámicamente. Como resultado, se almacena en la máquina cliente una *cookie* que contiene «id-12345». Cuando el navegador manda la siguiente petición HTTP para *formulario2.cgi*, se recoge el par nombre-valor (id=12345) de la *cookie* y se incluye en una línea de petición de cabecera. Según se recibe la petición, el servidor extrae la cadena nombre-valor («id=12345») de la línea de cabecera y la sitúa en la variable de entorno *HTTP_COOKIE*. El *script* web *formulario2.cgi* ya puede extraer los datos de estado (id=12345) de *HTTP_COOKIE*.

Obsérvese que *formulario2.cgi* puede generar una *cookie* adicional, que en este ejemplo, contiene el par nombre-valor «nombre=Juan». Si a continuación se invoca otro *script* CGI, los pares nombre-valor de todas las *cookies* generadas previamente se mandan en la cabecera de petición. En el ejemplo, *formulario3.cgi* recibirá en *HTTP_COOKIE* los datos de estado id=12345 y nombre=Juan de las dos *cookies* generadas por los *script* web previos.

Sintaxis de la línea de cabecera de respuesta HTTP *Set-Cookie*

La descripción previa presentaba un escenario simplificado del uso de las *cookies* para mantener los datos de estado. En realidad, la línea de cabecera *Set-Cookie*, utilizada por un *script* CGI para comenzar una *cookie*, tiene una sintaxis muy rica. La siguiente descripción está basada en [home.netscape.com, 25]:

La línea de cabecera *Set-Cookie* es una cadena con el siguiente formato (las palabras clave están en negrita):

```
Set-Cookie: NOMBRE=VALOR; expires=FECHA;
path=PATH; domain=NOMBRE_DOMINIO; secure
```

La línea comienza con la palabra clave «Set-Cookie» y el delimitador dos puntos (:), seguido por una lista de atributos separados por puntos y comas. Los atributos son los siguientes:

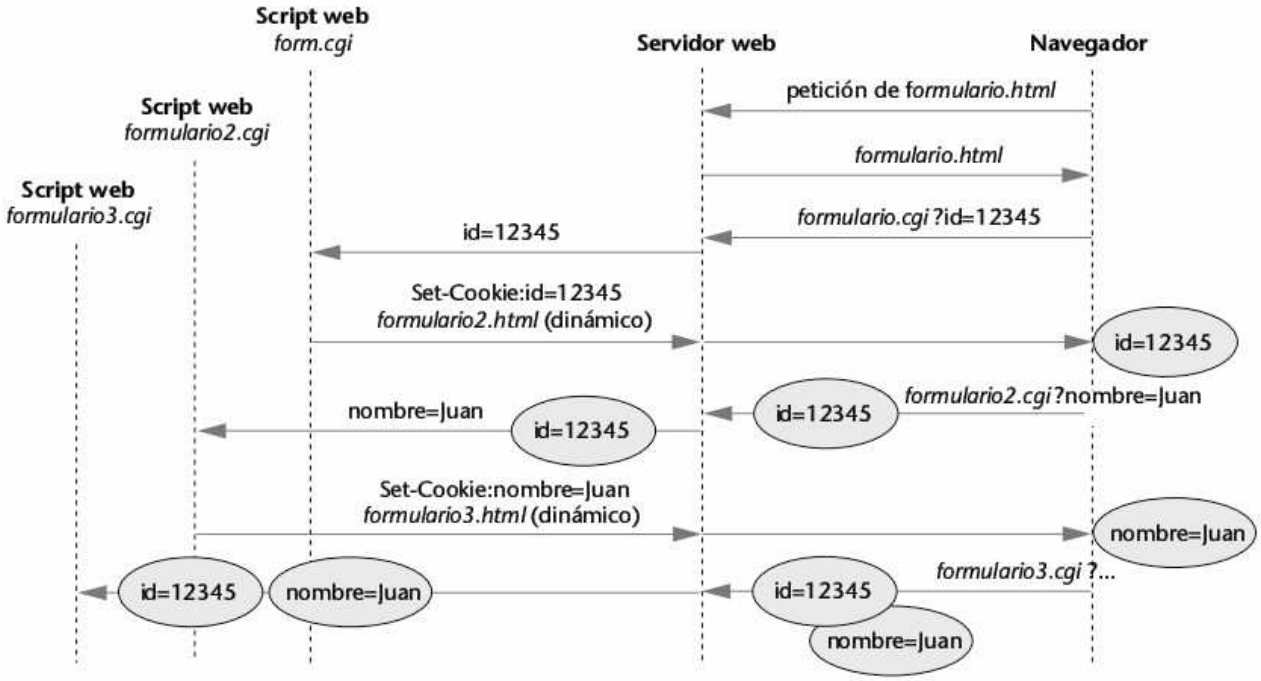


Figura 9.24. Uso de *cookies* para enviar estado de sesión.

- *NOMBRE=VALOR*. Similar a los datos de entrada de un formulario, es un par nombre-valor con codificación URL de los datos de estado a ser almacenados en la *cookie*. Este es el único atributo requerido en la línea de cabecera.
- *expires=<fecha>*. El atributo *expires* especifica una fecha que define el tiempo de validez de una *cookie*. Una vez que la fecha de expiración ha sido alcanzada, la máquina cliente tiene libertad de borrar la *cookie*, y no se puede asumir que los datos de estado contenidos en la *cookie* se enviarán al servidor en las siguientes peticiones HTTP.

  El formato de la fecha es el siguiente:

  ```
  día_semana, DD-Mes-AAAA HH:MM:SS GMT
  ```

  El formato está basado en RFC 822, RFC 850, RFC 1036 y RFC 1123, con la excepción de que la única zona de tiempo legal es GMT, y que los separadores entre los elementos deben ser puntos.

  expires es atributo opcional. Si no se especifica, la *cookie* expirará cuando finalice la sesión de usuario.
- *domain=<nombre de dominio>*. Este atributo establece el dominio para la *cookie* creada. De entre todas la *cookies* almacenadas en la máquina cliente, se supone que el navegador sólo envía aquellas cuyo atributo de dominio coincide con el nombre de dominio de Internet especificado en el URI del objeto de la petición HTTP (con la que la *cookie* es enviada). Si coincide el final de ambos nombres, se chequeará la coincidencia de la ruta para verificar si la *cookie* debe ser enviada. Por ejemplo, un atributo de dominio «acme.com» debería corresponderse con «yunque.acme.com» al igual que con «envios.acme.com». De esta forma, el par nombre-valor de la *cookie* asociada con el atributo de dominio

acme.com será enviada en la petición HTTP cuando el objeto solicitado tenga una URI que contenga el nombre de máquina yunque.acme.com (tal como yunque.acme.com/index.html) o envios.caja.acme.com (tal como envios.caja.acme.com/sales/shop.htm).

Sólo las máquinas dentro del dominio especificado pueden establecer una *cookie* para dicho dominio, y los dominios deben tener al menos dos o tres puntos para evitar dominios de la forma: .com, .edu y va.us.

El valor por defecto del dominio es el nombre la máquina servidora que generó la *cookie*. Por ejemplo, si el servidor es www.algunaU.edu, y no se ha especificado atributo de dominio, el dominio será www.algunaU.edu.

- *path=<cadena con ruta>*. El atributo *path* se utiliza para especificar el subconjunto de URIs dentro de un dominio, para los cuales la *cookie* es válida. Si una *cookie* tiene correspondencia con el nombre de dominio, entonces el componente de la ruta del URI se compara con el atributo *path*, y si hay correspondencia, la *cookie* se considera válida y se envía junto con la petición HTTP. Por ejemplo, la ruta «/foo» se corresponde con «/foobar» y con «/foo/bar.html». La ruta «/» es la más genérica.

  Si no se especifica el atributo *path*, se asume que la ruta es la misma que la del documento descrito por la cabecera que contiene la *cookie*. Los ejemplos de la Tabla 9.3 muestran el uso de los atributos *Domain* y *Path*. En los ejemplos, * representa cualquier cadena.
- *secure*. Si una *cookie* está marcada como segura, será transmitida sólo si el canal de comunicaciones es seguro, como lo que sucede con los servidores HTTPS (HTTP sobre SSL). Si no se especifica «secure», se considera que la cookie es suficientemente segura como para ser mandada en texto plano sobre un canal no seguro.

Tabla 9.3. Ejemplos de configuración de los atributos *Domain* y *Path* de las *cookies*.

Atributo *Domain*	Atributo *Path*	Los URI solicitados que harán que el par nombre-valor de la *cookie* se envíe en la cabecera de la petición
www.algunaU.edu	Ninguno	*.www.algunaU.edu/*
www.algunaU.edu	/	*.www.algunaU.edu/*
www.algunaU.edu	/área	*.www.algunaU.edu/área*
www.algunaU.edu	/área/área	*.www.algunaU.edu/área/área*

Sintaxis de la línea de cabecera de petición HTTP *cookie*

Cuando se solicita un URL a un servidor HTTP, el navegador comparará el URI contra todas las *cookies* almacenadas en la máquina cliente [home.netscape.com, 25]. Si se encuentra alguna *cookie* coincidente, se incluirá una línea con los pares nombre-valor de dichas cookies en la cabecera de petición HTTP. El formato de la línea es el siguiente:

```
Cookie: NAME1=VALUE1; NAME2=VALUE2; ...; NAMEn=VALUEn
```

Cuando el servidor HTTP encuentra esta línea en la cabecera de la petición, extrae la subcadena con los pares nombres-valor y la sitúa en una variable de entorno

HTTP_COOKIE. Cuando el *script* CGI se ejecuta, puede recibir los datos de estado como pares nombres-valor de la variable de entorno *HTTP_COOKIE*.

Por ejemplo, si se envía al servidor la siguiente petición

```
GET /cgi/hola.cgi?nombre=Juan&pregunta=paz HTTP/1.0
Cookie: edad=25
<línea en blanco>
```

El servidor colocará la cadena «nombre=Juan&pregunta=paz» en la variable de entorno *QUERY_STRING* y la cadena «edad=25» en *HTTP_COOKIE* para la invocación del *script* CGI.

Por otro lado, si la petición es

```
POST /cgi/hola.cgi HTTP/1.0
Cookie: edad=25
<línea en blanco>
nombre=Juan&pregunta=paz
```

entonces el servidor enviará la cadena «nombre=Juan&pregunta=paz» a la entrada estándar del *script* CGI, mientras que la cadena «edad=25» se colocará en la variable de entorno *HTTP_COOKIE*.

Los atributos *domain* y *path* de las *cookies* están diseñados para permitir compartir datos de estado entre *scripts* CGI predeterminados, tal y como se muestra en los siguientes ejemplos (tomados directamente de [home.netscape.com, 25]):

Ejemplo 1. Secuencia de transacciones

El cliente solicita un documento y recibe la siguiente respuesta:

```
Set-Cookie: CLIENTE=WILE_E_COYOTE; path=/; expires=Wednesday, 09-Nov-99
23:12:40 GMT
```

Cuando el cliente solicita un URL en la ruta «/» del servidor, envía

```
Cookie: CLIENTE=WILE_E_COYOTE
```

El cliente solicita un documento y recibe la siguiente respuesta:

```
Set-Cookie: NUMERO_PARTE=LANZACOHETES_0001; path=/
```

Cuando el cliente solicita un URL en la ruta «/» del servidor, envía

```
Cookie: CLIENTE=WILE_E_COYOTE; NUMERO_PARTE=LANZACOHETES_0001
```

El cliente recibe:

```
Set-Cookie: ENVIO=FEDEX; path=/area
```

Cuando el cliente solicita un URL en la ruta «/» del servidor, envía

```
Cookie: CLIENTE=WILE_E_COYOTE; NUMERO_PARTE=LANZACOHETES_0001
```

Cuando el cliente solicita un URL en la ruta «/área» del servidor, envía lo siguiente:

```
Cookie: CLIENTE=WILE_E_COYOTE; NUMERO_PARTE=LANZACOHETES_0001;
ENVIO=FEDEX
```

Ejemplo 2. Secuencia de transacciones

Se asume que todas las *cookies* anteriores han sido borradas.

El cliente recibe lo siguiente:

```
Set-Cookie: NUMERO_PARTE=LANZACOHETES_0001; path=/
```

Cuando el cliente solicita un URL en la ruta «/» del servidor, envía

```
Cookie: NUMERO_PARTE=LANZACOHETES_0001
```

El cliente recibe lo siguiente:

```
Set-Cookie: NUMERO_PARTE=COHETE_MONTABLE_0023; path=/municion
```

Cuando el cliente solicita un URL en la ruta «/munición» del servidor, envía lo siguiente:

```
Cookie: NUMERO_PARTE=COHETE_MONTABLE _0023;
NUMERO_PARTE=LANZACOHETES_0001
```

Nota: existen dos pares nombre-valor denominados NUMERO_PARTE, ya que hay dos *cookies* que coinciden con el atributo *path*: «/» y «/munición».

Código de ejemplo de la utilización de *cookies* para transmitir datos de estado

En esta sección se ponen en práctica diversos *scripts* CGI que hacen uso de la cookies. La Figura 9.25 muestra un formulario web estático, que una vez enviado, invoca el *script* CGI *cookie.cgi*.

El código fuente de *cookie.cgi* se muestra en la Figura 9.26. En las líneas 39-45, el *script* CGI obtiene la cadena de interrogación de la entrada estándar (ya que se utilizó el método *POST*), extrae los pares nombre-valor de la cadena de interrogación, y los coloca en un vector asociativo. En las líneas 50-54, cada uno de los pares nombre-valor del vector se coloca en una *cookie* de nueva creación. Por simplicidad, no se han definido más atributos de las *cookies* por lo que su vida será la de la actual sesión web. (Nota: en la aplicación real sólo los pares nombres-valor que son necesarios como datos de sesión deben ser situados en *cookies*).

Figura 9.25. *Cookie.html.*

```
<HEAD>
<TITLE>Un formulario sencillo que invoca un script CGI que crea cookies
</TITLE>
</HEAD>
<BODY>
<H1>Este es un formulario sencillo que invoca a un script CGI,
cookie.cgi
</H1>
<P>
<HR>
<FORM method="POST" action="cookie.cgi">
Cuál es tu nombre: <input name="nombre"><P>
Cuál es tu edad: <input name="edad"><P>
Cuál es tu id: <input name="id"><P>
Presiona <input type="submit" value="aquí"> para enviar tu consulta.
</FORM>
</BODY>
</HTML>
```

Figura 9.26. *cookie.c.*

```
/* cookie.c – código fuente de cookie.cgi
Author: M. L. Liu.
Este script, solicitado por cookie.html, crea una cookie
para cada par nombre-valor en la cadena de interrogación
*/

#include <stdio.h>
#ifndef NO_STDLIB_H
#include <stdlib.h>
#else
char *getenv();
#endif

#define MAX_ENTRADAS 10000

typedef struct {
char *nombre;
char *valor;
} entrada;

char *makeword(char *line, char stop);
char *fmakeword(FILE *f, char stop, int *len);
char x2c(char *what);
void unescape_url(char *url);
void plustospace(char *str);

main(int argc, char *argv[]) {
  entrada entradas[MAX_ENTRADAS];
  register int x,m=0;
  int cl;

  if(strcmp(getenv("REQUEST_METHOD"),"POST")) {
     printf("Este script debería ser referenciado con
        el método POST.\n");
     exit(1);
  }
  cl = atoi(getenv("CONTENT_LENGTH"));

  for(x=0;cl && (!feof(stdin));x++) {
     m=x;
     entradas[x].valor = fmakeword(stdin,'&',&cl);
     plustospace(entradas[x].valor);
     unescape_url(entradas[x].valor);
     entradas[x].nombre = makeword(entradas[x].valor,'=');
  }

   /* Se genera una línea de cabecera Set-cookie para cada par
   nombre-valor. Cada línea crea una cookie con los atributos
```

(continúa)

```
49    por defecto */
50    for (x=0; x<=m; x++)
51    {
52       printf("Set-cookie: %s=%s%c\n",
53       entradas[x].nombre, entradas[x].valor,'\0');
54    }
55    /*** IMPORTANTE: NINGÚN BLANCO ANTES O DESPUÉS DEL SIGNO IGUAL
***/
56    /*** Set-cookie debe comenzar la cabecera de la respuesta ***/
57    printf("Content-type: text/html%c%c",10,10);
58    printf("<form method=POST action=cookie2.cgi>\n");
59    printf("<body bgcolor=\"#CCFFCC\">");
60    printf("Presiona <input type=submit value=aqui> para
61       crear las cookies\n");
62    printf("</form>\n<hr></html>");
63 }
```

La Figura 9.27 muestra el código fuente del siguiente *script* CGI, *cookie2.cgi*, que es solicitado por el formulario web generado dinámicamente por *cookie.cgi* (ver línea 58 de Figura 9.26). El *script* CGI *cookie2.cgi* recoge todos los pares nombres-valor de la variable de entorno *HTTP_COOKIE* (líneas 41, 51-57), después solamente imprime sus valores. En una aplicación real, los datos de estado recogidos deberán ser usados en la lógica de la aplicación.

Figura 9.27. cookie2.c.

```
 1 /* cookie2.c — código fuente de cookie2.cgi
 2 Author: M. L. Liu.
 3 Este script CGI, invocado por la página web generada por
 4 cookie.cgi, recibe los datos de estado de las cookies
 5 creadas por cookie.cgi.
 6 */
 7
 8 #include <stdio.h>
 9 #ifndef NO_STDLIB_H
10 #include <stdlib.h>
11 #else
12 char *getenv();
13 #endif
14
15 #define MAX_ENTRADAS 10000
16
17 typedef struct {
18 char nombre[128];
19 char valor[128];
20 } entradaCookie;
21
22 char *makeword(char *line, char stop);
```

(continúa)

```
23 char *fmakeword(FILE *f, char stop, int *len);
24 char x2c(char *what);
25 void unescape_url(char *url);
26 void plustospace(char *str);
27
28 void getword(char *word, char *line, char stop);
29
30 main(int argc, char *argv[]) {
31   entradaCookie cEntradas[MAX_ENTRADAS];
32   /* almacenamiento para los pares nombre-valor recogidos
33   de HTTP_COOKIE */
34   register int x,m=0;
35   int cl;
36   char* cadenaCookie;
37
38   /* Código para la cadena de interrogación omitido por simplicidad */
39
40   /* recoger los datos de estado de las cookies */
41   cadenaCookie = getenv("HTTP_COOKIE");
42   if(cadenaCookie == NULL) {
43     printf("No hay información de cookies. \n");
44     exit(1);
45   }
46   printf("Content-type: text/html%c%c",10,10);
47
48   printf("<body bgcolor=\"#CCFFCC\">");
49   printf("<H1>Cookie recibida</H1>");
50   printf("La cadena de la cookie es: %s\n", cadenaCookie);
51   for(x=0; cadenaCookie[0] != '\0';x++) {
52     m=x;
53     getword(cEntradas[x].valor, cadenaCookie,';');
54     plustospace(cEntradas[x].valor);
55     unescape_url(cEntradas[x].valor);
56     getword(cEntradas[x].nombre,cEntradas[x].valor,'=');
57   }
58
59   printf("<p>Se han recogido los siguientes pares nombre/valor
60     de las cookies:<p>%c",10);
61   printf("<ul>%c",10);
62   for(x=0; x <= m; x++)
63     printf("<li> <code>%s = %s</code>%c",
64       cEntradas[x].nombre, cEntradas[x].valor,10);
65     printf("</form>\n<hr></html>");
66   }
```

Privacidad de los datos y consideraciones de seguridad

Las *cookies* han sido ampliamente utilizadas en aplicaciones web comerciales. Las *cookies* son un mecanismo más sofisticado que los campos ocultos de formulario para

el mantenimiento de los datos de sesión. Sin embargo, su uso ha creado muchas controversias.

En primer lugar, las *cookies* pueden ser escritas en ficheros de la máquina del usuario sin el conocimiento o aprobación explícita del usuario. (Es posible que un usuario configure al navegador para que no acepte *cookies*; desafortunadamente, esto puede causar problemas cuando el sitio web visitado espera ser capaz de utilizar *cookies*). Aunque los ficheros de las *cookies* son de tamaño pequeño, la acumulación de un gran número de estos ficheros ocupa recursos del sistema.

En segundo lugar, las *cookies* pueden ser utilizadas para recoger información personal de los usuarios web sin su conocimiento o aprobación. Los defensores de la privacidad han expresado su preocupación sobre el uso de las *cookies* para recoger los perfiles (patrones interesantes) de los usuarios de los navegadores. La sesión *State Management* de [research.att.com, 18] proporciona una explicación detallada de cómo las *cookies* pueden ser utilizadas para construir perfiles de usuario.

Por otra parte, las *cookies* persistentes, almacenadas en ficheros fácilmente accesibles, pueden revelar información personal de usuarios que comparten un computador, creando de esta forma una importante amenaza para la seguridad y la privacidad.

Por esta razón HTTP/1.1 realiza aportaciones específicas para solucionar estas preocupaciones. Desafortunadamente estas aportaciones no han sido ampliamente adoptadas, por lo que los problemas de privacidad derivados del uso de las *cookies* continúan.

Aquí concluye esta introducción a las aplicaciones de Internet. En el Capítulo 11 se estudiarán protocolos y herramientas más avanzadas, que incluyen los servlets y los servicios web.

RESUMEN

Este capítulo presenta una introducción a las aplicaciones de Internet y a los protocolos principales que las sustentan. A continuación se muestra un resumen de los temas tratados.

- HTML (*Hypertext Markup Language*, lenguaje de marcado de hipertexto) es un lenguaje de marcado utilizado para crear documentos que pueden ser recibidos utilizando la Web. Este lenguaje permite que un documento sea marcado a fin de poder presentar la información en él contenida.
- XML (*Extensible Markup Language*, lenguaje extensible de etiquetado) permite que un documento sea etiquetado para estructurar la información.
- HTTP (*Hypertext Transfer Protocol*, protocolo de transferencia de hipertexto) es el protocolo de transporte de la Web.
  - — HTTP es un protocolo que permite la transferencia de contenido web de cualquier tipo. HTTP es un protocolo orientado a conexión, sin estado y de petición-respuesta. Un servidor HTTP, un servidor web, ejecuta por defecto en el puerto TCP 80. Los clientes HTTP, coloquialmente denominados navegadores web, son procesos que implementan HTTP para poder interactuar con un servidor web y recibir documentos con formato HTML, cuyo contenido es presentado de acuerdo a las etiquetas del documento.

— En HTTP/1.0, cada conexión permite un único turno de petición/respuesta: un cliente obtiene una conexión, manda una petición; el servidor procesa la petición, genera una respuesta y finaliza cerrando la conexión.

— HTTP está basado en texto: la petición y la respuesta son cadenas de caracteres. Cada petición y respuesta están compuestas de cuatro partes: la línea de petición/respuesta; una sección de cabecera; una línea en blanco; el cuerpo.

Una línea de petición tiene el siguiente formato:

<método HTTP><espacio><URI solicitado><espacio><especificación de protocolo>\r\n

Los métodos principales de HTTP son GET y POST.

Una cabecera está compuesta por una o más líneas, cada una de ellas de la forma

<clave>: <valor>\r\n

El cuerpo de la petición contiene datos que necesitan ser enviados al servidor junto con la petición.

- Una respuesta HTTP está compuesta de la respuesta o la línea de estado, una cabecera de sección, una línea en blanco y el cuerpo.

  La línea de estado tiene el formato

  <protocolo><espacio><código estado><espacio><descripción>\r\n

  La cabecera de la respuesta está compuesta de una o más líneas, cada una de ellas con el siguiente formato

  <clave>: <valor>\r\n

  El cuerpo de la respuesta continúa a la cabecera y a una línea en blanco y contiene el contenido del objeto web solicitado.

- El protocolo CGI (***Common Gateway Interface***, Interfaz Estándar de Pasarela) es un protocolo que expande HTTP para la generación de contenido web en tiempo de ejecución. Utilizando el protocolo CGI, un cliente web puede especificar un programa externo, conocido como *script* CGI, como objeto web de la petición HTTP. Cuando es solicitado, el servidor web llama al *script* CGI y lo activa como un proceso. El *script* web ejecuta y transmite su salida al servidor web, que devuelve los datos generados por el script como respuesta HTTP al cliente.
- Un formulario web es un tipo especial de página web que (1) proporciona una interfaz gráfica de usuario que permite al usuario introducir datos y, (2) cuando el usuario pulsa el botón de Envío, invoca la ejecución de un programa externo en la máquina del servidor web.
- Los datos de entrada se recogen en la cadena de interrogación usando codificación URL. Esta cadena se envía al *script* web especificado en la etiqueta ***ACTION*** del formulario web.
- Han surgido diversos mecanismos para permitir que los datos de sesión sean compartidos entre los *script* web invocados durante una sesión web. Estos mecanismos pueden ser clasificados como mecanismos del lado del servidor y del lado del cliente. Los mecanismos del lado del cliente incluyen el uso de campos ocultos de formularios y de las denominadas *cookies*.
- El uso de campos ocultos de formularios y *cookies* crea amenazas de seguridad y privacidad.

EJERCICIOS

1. En el contexto de las aplicaciones basadas en la Web, ¿qué papel juega cada uno de los siguientes lenguajes/protocolos? HTML, MIME, XML, HTTP, CGI.
2. Cuando un servidor HTTP envía el contenido de un documento a un cliente en el cuerpo de la respuesta, utiliza la línea de cabecera ***Content-Lenght*** para especificar la longitud en bytes del cuerpo. Para los documentos estáticos la longitud en bytes la proporciona el sistema de ficheros. Pero para páginas web generadas dinámicamente, tales como las generadas por un script CGI, la longitud debe ser determinada en tiempo de ejecución.
   a. Para una página web generada dinámicamente, ¿cómo puede saber el servidor web la longitud del contenido especificado en la cabecera ***Content-Lenght***? Recuerde que las líneas de cabecera aparecen antes que el cuerpo en la respuesta HTTP.
   b. Mire cómo HTTP/1.1 soluciona este problema y resuma lo que encuentre (*véase* la Sección 4.4 de RFC 2068 [ietf.org, 10]).
3. Compile ***clienteHTTP.java***, el código de ejemplo mostrado en la Figura 9.6. Ejecútelo para contactar con un servidor web cuyo nombre conozca para (a) buscar un fichero que exista, y (b) buscar un fichero que sepa que no existe. Capture la salida de cada ejecución. Analice las líneas de la respuesta del servidor en cada caso e identifique las diferentes partes (línea de estado, línea(s) de cabecera y cuerpo).
4. Repita el ejercicio previo utilizando ***navegadorURL.java***, mostrado en la Figura 9.7. Para el resto de los ejercicios necesitará acceso a los siguientes recursos:
   - Un servidor web que soporte CGI (además necesitará preguntar al administrador del sistema en qué directorio de la máquina puede almacenar las páginas web y los *script* CGI), y
   - Un compilador C, tal como gcc, el compilador GNU de C (http://www.gnu.org/software/gcc/gcc.html), que genera código máquina para el servidor web.
5. Haga experimentos que el ejemplo ***CGI Hola*** presentado en las Figuras 9.10 y 9.11.
   a. Consiga los ficheros fuente del ***CGI Hola*** presentados en la Figura 9.10 y 9.11.
   b. Instale ***hola.html*** en el servidor web, asegurándose de que permite acceso de lectura a todo el mundo.
   c. Compile ***hola.c*** (con ***util.c*** que contiene las rutinas de procesado de la cadena de interrogación) para generar y ejecutar ***hola.cgi***. (El mandato es ***gcc hola.c util.c u hola.cgi***). Instale ***hola.cgi*** en el servidor web, asegurándose de que permite acceso de lectura/ejecución a todo el mundo.
   d. Abra un navegador y especifique el URL de ***hola.html***. Cuando se presente la página presione el botón de envío del formulario para invocar ***hola.cgi***.
   e. Resuma el experimento y analice los resultados.
6. Escriba un programa cliente que invoque a ***hola.cgi***.
   a. Escríbalo utilizando la API *socket* stream.

b. Escríbalo utilizando la clase *URL*.

c. Entregue el listado de los programas.

7. Realice los siguientes experimentos con el ejemplo de formulario web presentado en las Figuras 9.14, 9.15 y 9.16.

   a. Consiga los códigos de *formularioPOST.html*, *formularioPOST.c*, *formularioGET.html* y *formularioGET.c*.

   b. Instale *formularioPOST.html* en el servidor web.

   c. Compile *formularioPOST.c* con *util.c* para generar el ejecutable *formulario.cgi*. Instale *formulario.cgi* en el servidor web.

   d. Abra *formularioPOST.html*. Introduzca algunos datos que incluyan, al menos, un carácter en blanco. Presione el botón de envío.

   e. ¿Qué aparece en el campo URL del navegador (normalmente en la parte superior de la pantalla)? Dentro del URL identifique la parte que se corresponde con la cadena de interrogación. ¿Está la cadena codificada con codificación URL? ¿Cómo lo puede saber?

   f. Los datos que introdujo en el apartado d, ¿son presentados correctamente?

   g. Modifique *formulario.html* para especificar el método *POST* en la etiqueta *FORM* (en lugar de *GET*). Reinstale *formulario.html* en el servidor web.

   h. Compile *formularioPOST.c* con *util.c* para generar *formulario.cgi*. Instale *formulario.cgi* en el servidor web.

   i. Visualice, introduzca datos y envíe *formulario.html*.

   j. ¿Qué aparece en el campo URL del navegador (normalmente en la parte superior de la pantalla)?

   k. Los datos que introdujo en el apartado i, ¿son presentados correctamente?

   l. Resuma esta experimentación y sus observaciones. Incluya comentarios sobre las diferencias de comportamiento entre los métodos *GET* y *POST*.

   m. ¿Por qué no es buena idea utilizar el método *GET* para enviar datos tales como el DNI o el número de la tarjeta de crédito en una petición HTTP? (Sugerencia: observe el campo URL que muestre el navegador web cuando ejecute el ejemplo *formularioGET*).

8. Escriba un programa cliente que invoque directamente *formulario.cgi* con el método *GET* especificado, utilice la API *socket* stream o la clase *URL*. Entregue el programa. (Sugerencia: utilice una cadena de interrogación sencilla en su código, tal como «nombre=Pedro%20Rosales&pregunta=medalla%10oro». También puede servir de ayuda revisar el ejemplo 3 de la petición HTTP en la sección 9.3).

9. Escriba un programa cliente que invoque directamente *formulario.cgi* con el método *POST* especificado, utilice la API *socket* stream o la clase *URL*.

10. Experimente con el ejemplo de campos ocultos presentado en las Figuras 9.12, 9.22 y 9.23

    a. Obtenga el código fuente de *formulario.html*, *formularioOculto.c* y *formularioOculto2.c*.

    b. Instale *formulario.html* en el servidor.

    c. Compile *formularioOculto.c* con *util.c* para generar un ejecutable *formularioOculto.cgi*. Instale *formularioOculto.cgi* en el servidor web.

d. Compile *formularioOculto2.c* con *util.c* para generar un ejecutable *formularioOculto2.cgi*. Instale *formularioOculto2.cgi* en el servidor web.

e. Abra *formulario.html*. Introduzca un nombre y presione el botón Enviar. Cuando se presente la página web generada dinámicamente, utilice la opción *Ver código* de su navegador para ver el código fuente de la página web (tal y como fue generada por *formularioOculto.cgi*). ¿Ve los datos ocultos embebidos en el código fuente?

f. ¿Se pasaron los datos (el nombre introducido en el formulario estático) de sesión al último *script* web?

g. Resuma el experimento y sus observaciones.

11. Experimente con el ejemplo de *cookies* presentado en las Figuras 9.25, 9.26 y 9.27.

a. Obtenga el código fuente de *formulario.html*, *cookie.c* y *cookie2.c*.

b. Instale *formulario.html* en el servidor.

c. Compile *cookie.c* con *util.c* para generar un ejecutable *cookie.cgi*. Instale *cookie.cgi* en el servidor web.

d. Compile *cookie2.c* con *util.c* para generar un ejecutable *cookie2.cgi*. Instale *cookie2.cgi* en el servidor web.

e. Abra *formulario.html*. ¿Se pasaron los datos (el nombre introducido en el formulario estático) de sesión al último *script* web?

f. Resuma el experimento y sus observaciones.

12. En este ejercicio experimentará con el ejemplo *cookie* presentado en las Figuras 9.25, 9.26 y 9.27 para investigar el uso de los atributos *path* y *domain* en las *cookies*.

En el servidor, duplique el fichero *formulario.html* en dos directorios diferentes, *cookie1* y *cookie2*, respectivamente, de forma que el URL de estas páginas se diferencie en sus rutas, por ejemplo *www.alpha.org/cookie1/formulario.html* y *www.alpha.org/cookie2/formulario.html*.

a. Abra dos sesiones con el navegador de su computador. Acceda a *formulario.html* en el directorio *cookie1* en ambas sesiones, pero introduzca datos diferentes (por ejemplo, nombre="alfa" en uno y nombre="beta" en el otro) en cada uno de ellos. ¿Cuáles son los valores respectivos de los atributos *path* y *domain* de las *cookies* generadas en cada sesión? En cada sesión, ¿qué *cookie* o *cookies* enviará el navegador a *cookie2.cgi* a través del servidor? ¿Cuál es la salida esperada? Describa sus observaciones. Explique la salida.

b. Abra dos sesiones con el navegador de su computador. Acceda a *formulario.html* en el directorio *cookie1* en una sesión y *formulario.html* en el directorio *cookie2* de la otra sesión. Introduzca datos diferentes (por ejemplo, nombre="alfa" en uno y nombre="beta" en el otro) en cada uno de ellos. ¿Cuáles son los valores respectivos de los atributos *path* y *domain* de las *cookies* generadas en cada sesión? En cada sesión, ¿qué *cookie* o *cookies* enviará el navegador a *cookie2.cgi* a través del servidor? ¿Cuál es la salida esperada? Describa sus observaciones. Explique la salida.

c. Modifique el código del *script* CGI (en *cookie.c*) para que las *cookies* se generen con la ruta "/".

Abra dos sesiones con el navegador de su computadora. Acceda a *formulario.html* en el directorio *cookie1* en una sesión y *formulario.html* en el directorio *cookie2* de la otra sesión. Introduzca datos diferentes (por ejemplo, nombre="alfa" en uno y nombre="beta" en el otro) en cada uno de ellos. ¿Cuáles son los valores respectivos de los atributos *path* y *domain* de las *cookies* generadas en cada sesión? En cada sesión, ¿qué *cookie* o *cookies* enviará el navegador a *cookie2.cgi* a través del servidor? ¿Cuál es la salida esperada? Describa sus observaciones. Explique la salida.

d. Suponga que instala *formulario.html*, *cookie.c* y *cookie2.c* en otro servidor, por ejemplo *www.beta.org*, de forma que se tiene dos conjuntos de ficheros idénticos, uno en *www.alfa.org* y otro en *www.beta.org*. Abra dos sesiones de navegador diferente en el computador, acceda a *www.alfa.org/formulario.html* en la primera sesión y *www.beta.org/cookie2/formulario.html* en la segunda sesión. Introduzca "alfa" como nombre en la primera sesión y «beta» en la segunda. ¿Cuáles son los valores respectivos de los atributos *path* y *domain* de las *cookies* generadas en cada sesión? En cada sesión, ¿qué *cookie* o *cookies* enviará el navegador a *cookie2.cgi* a través del servidor? ¿Cuál es la salida esperada? Razone la respuesta.

13. Este ejercicio experimenta con el ejemplo *cookie* presentado en las Figuras 9.25, 9.26 y 9.27 para investigar el uso del atributo *expires* con las *cookies*.

    Modifique el código fuente del *script* CGI (en *cookie.c*) para añadir el atributo expires a la línea de cabecera *Set-Cookie*, de esta manera:

    ```
    Expires Mon, 09-Dec-2002 13:46:00 GMT
    ```

    para que las *cookies* se escriban a fichero (en lugar de existir de forma temporal en el navegador).

    Abra dos sesiones con el navegador de su computador. Acceda a *formulario.html* en el directorio *cookie* en ambas sesiones, pero introduzca datos diferentes (por ejemplo, nombre="alfa" en uno y nombre="beta" en el otro) en cada uno de ellos. ¿Cuáles son los valores respectivos de los atributos *path* y *domain* de las *cookies* generadas en cada sesión? En cada sesión, ¿qué *cookie* o *cookies* enviará el navegador a *cookie2.cgi* a través del servidor? ¿Cuál es la salida esperada? Describa sus observaciones. Explique la salida.

14. A la caza de la *cookie*:

    Cierre su navegador y reinícielo para comenzar una nueva sesión. Ejecute el ejemplo *cookie* que escribe las *cookies* en fichero (tal y como se describió en la anterior sección). Introduzca una cadena rara (por ejemplo, «dfgjhff») en el campo del formulario. Cuando se ejecute se debería crear una *cookie* con la cadena extraña que se introdujo.

    Utilice una herramienta de búsqueda de su sistema para encontrar el fichero que contiene la cadena extraña. (En un sistema UNIX se puede utilizar el comando *grep*. En un sistema Windows se puede utilizar la utilidad *encontrarfichero*). ¿Encontraste el fichero? En caso afirmativo, identifica sus elementos.

15. Lea la sección "*State Management*" ("Gestión de estado") en [research.att.com, 18].

    a. Resuma los problemas de seguridad sobre el uso de las *cookies*.

    b. Proporcione, con explicaciones, una sesión HTTP de ejemplo en la que se utilicen las *cookies* para construir un perfil consistente en (i) el número de

tarjeta de crédito del usuario y (ii) una traza de los URL que el usuario visitó en el sitio web durante su última sesión.

c. ¿Cuáles son soluciones específicas que se hicieron en el HTTP/1.1 para solucionar estos problemas? Resúmalas. (Puede ser útil consultar [ietf.org, 10] y [research.att.com, 18]).

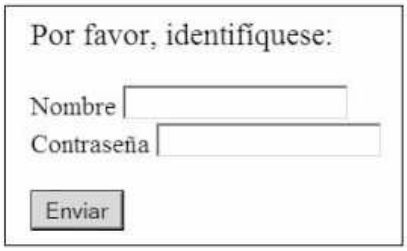
El resto de los ejercicios son para estudiantes que conocimientos avanzados de C u otros lenguajes de programación para implementar *scripts* CGI. Los estudiantes que no estén familiarizados con estos lenguajes, pueden saltarse estos ejercicios. Se tendrá la ocasión de practicar estos ejercicios utilizando el lenguaje Java en el Capítulo 11.

16. Modifique los ficheros de ejemplo *cookie* para implementar un conjunto de páginas web y *scripts* CGI con un carrito de la compra sencillo tal y como se muestra en la Figura 9.17. Los datos de estado que necesitan ser mantenidos son (a) ID del cliente y (b) nombre de la mercancía.
17. Escoja: (a) escriba un *script* CGI o (b) encuentre un *script* CGI gratuito (intente usar el programa *Login* de http://tectonicdesigns.com/freecgi) que le permita proporcionar seguridad a su página web. Cuando se solicite su página web, la salida debería ser parecida a la mostrada en la siguiente figura. Si la contraseña introducida es correcta, su página debería ser mostrada. Instale el *script* y verifíquelo. Describa el trabajo y describa cómo proporciona protección con contraseñas.

    Nota: el *script* CGI necesitará sacar una línea como la siguiente:

    ```
    <html><head><META HTTP-EQUIV="REFRESH" CONTENT="0;URL=<url de tu
    página web>"></head></html>\n
    ```

    para reenviar a su página web después de verificar la contraseña.



REFERENCIAS

1. Hobbes' Internet Timeline, *http://www.zakon.org/robert/internet/timeline/*
2. A Little History of the World Wide Web, *http://www.w3.org/History.html*
3. ISO 8879:1986, Information processing—Text and office systems—Standard Generalized Markup Language (SGML), International Organization for Standardization, *http://www.iso.org/*
4. People of the W3C, *http://www.w3.org/People/*
5. RFC 2109 HTTP State Management Mechanism, *http://www.ics.uci.edu/pub/ietf/http/rfc2109.txt*
6. Welcome to CERN, *http://public.Web.cern.ch/Public/*
7. HTTP: A protocol for networked information, *http://www.w3.org/Protocols/HTTP/HTTP2.html*
8. RFC 1945, Hypertext Transfer Protocol—HTTP/1.0, *http://www.faqs.org/rfcs/rfc1945.html*

9. RFC 1521, MIME (Multipurpose Internet Mail Extensions) Part One, *http://www.faqs.org/rfcs/rfc1521.html*
10. RFC 2068, Hypertext Transfer Protocol—HTTP/1.1, *http://www.ietf.org/rfc/rfc2068.txt*
11. The World Wide Web Consortium, *http://www.w3.org/*
12. RFC 959, File Transfer Protocol (FTP), *http://www.ietf.org/rfc/rfc0959.txt?number=959*
13. RFC 1866, Hypertext Markup Language 2.0, *http://ftp.ics.uci.edu/pub/ietf/html/rfc1866.txt*
14. NCSA—A Beginner's Guide to HTML, *http://archive.ncsa.uiuc.edu/General/Internet/WWW/HTMLPrimerAll.html*
15. Extensible Markup Language (XML), *http://www.w3.org/XML/*
16. A Quick Introduction to XML, *http://java.sun.com/xml/jaxp-1.1/docs/tutorial/overview/1_xml.html*
17. The Original HTTP as defined in 1991, Tim Berners-Lee, *http://www.w3.org/Protocols/HTTP/AsImplemented.html*
18. Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol. Key Differences between HTTP/1.0 and HTTP/1.1, *http://www.research.att.com/~bala/papers/h0vh1.html*
19. The Common Gateway Interface, *http://hoohoo.ncsa.uiuc.edu/cgi/overview.html*
20. The Common Gateway Interface Specification, *http://hoohoo.ncsa.uiuc.edu/cgi/interface.html*
21. CGI Tutorial, *http://www.comp.leeds.ac.uk/nik/Cgi/start.html*
22. RFC 1738, The Uniform Resource Locators, *http://www.w3.org/Addressing/rfc1738.txt*
23. URL Encoding, *http://www.blooberry.com/indexdot/html/topics/urlencoding.htm*
24. Hypertext Markup Language 2.0—Forms, *http://www.w3.org/MarkUp/html-spec/html-spec_8.html*
25. Client Side State—HTTP Cookies, Netscape Inc., *http://home.netscape.com/newsref/std/cookie_spec.htm*
26. *What is Jabber?*, Jabber Software Foundation, *http://www.jabber.org/about/overview.htm*

CAPÍTULO 10

CORBA - *Common Object Request Broker Architecture*

En el Capítulo 7 ya se presentó el paradigma de objetos distribuidos. Entre los temas cubiertos se encontraba la arquitectura para un sistema de objetos distribuidos típico, ilustrada mediante Java RMI.

En este capítulo, se presentará una arquitectura alternativa (una arquitectura estándar) para objetos distribuidos. A la arquitectura se la conoce como ***Common Object Request Broker Architecture* (CORBA)**. Las razones de este interés en CORBA son dos: en primer lugar, proporciona un caso de estudio que ilustra dos arquitecturas similares, pero en contraste, para un concepto determinado, los objetos distribuidos. En segundo lugar, CORBA proporciona un ejemplo de una arquitectura diseñada para alcanzar la máxima interoperabilidad.

CORBA es una arquitectura de objetos distribuidos diseñada para permitir que dichos objetos distribuidos interoperen sobre entornos heterogéneos, donde los objetos pueden estar implementados en diferentes lenguajes de programación y/o desplegados sobre diferentes plataformas. CORBA se diferencia de la arquitectura Java RMI en un aspecto significativo: RMI es una tecnología propietaria de Sun Microsystems, Inc. y sólo soporta objetos que se encuentren escritos en el lenguaje Java. Por el contrario, CORBA ha sido desarrollado por **OMG (*Object Management Group*)** [corba.org, 1], un consorcio de empresas, y

fue diseñado para maximizar el grado de interoperabilidad. Es importante saber que CORBA no es en sí mismo una herramienta para dar soporte a objetos distribuidos; se trata de un conjunto de protocolos. Una herramienta que dé soporte a dichos protocolos se denomina **compatible con CORBA** (*CORBA compliant*), y los objetos que se desarrollen sobre ella podrán interoperar con otros objetos desarrollados por otra herramienta compatible con CORBA.

CORBA proporciona un conjunto de protocolos muy rico [Siegel, 4], y el abordar todos ellos está más allá del ámbito de este libro. Sin embargo, se centrará en los conceptos clave de CORBA que estén relacionados con el paradigma de objetos distribuidos. También se estudiará una herramienta basada en CORBA: ***Interface Definition Language* (IDL) de Java**.

10.1. ARQUITECTURA BÁSICA

La Figura 10.1 muestra la arquitectura básica de CORBA [omg.org/gettingstarted, 2]. Como arquitectura de objetos distribuidos, guarda una gran semejanza con la arquitectura RMI. Desde el punto de vista lógico, un **cliente de objeto** realiza una llamada a un método de un objeto distribuido. El cliente interactúa con un *proxy*, un ***stub***, mientras que la implementación del objeto interactúa con un *proxy* de servidor, un ***skeleton***. A diferencia del caso de Java RMI, una capa adicional de software, conocida como **ORB (*Object Request Broker*)** es necesaria. En el lado del cliente, la capa del ORB actúa con intermediaria entre el *stub* y la red y sistema operativo local del cliente. En el lado del servidor, la capa del ORB sirve de intermediaria entre el *skeleton* y la red y sistema operativo del servidor. Utilizando un protocolo común, las capas ORB de ambos extremos son capaces de resolver las diferencias existentes en lo referente a lenguajes de programación, así como las relativas a las plataformas (red y sistema operativo), para de esta forma ayudar a la comunicación entre los dos extremos. El cliente utiliza un **servicio de nombrado** para localizar el objeto.

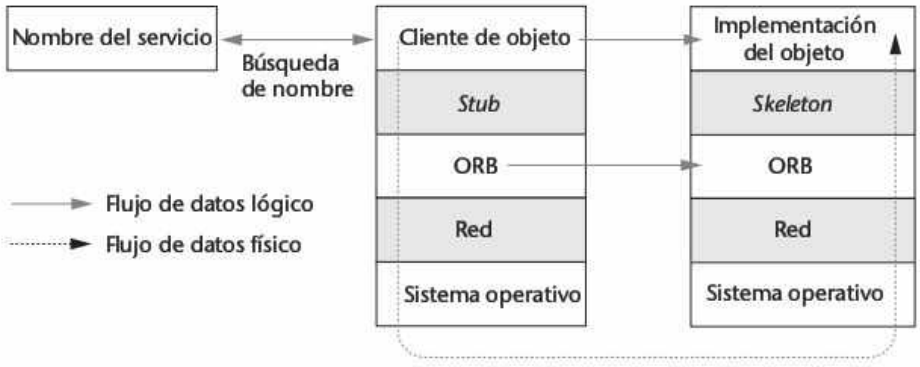


Figura 10.1. Arquitectura CORBA básica.

10.2. LA INTERFAZ DE OBJETOS DE CORBA

Un objeto distribuido se define usando un fichero similar al fichero que define la interfaz remota en Java RMI. Debido a que CORBA es independiente de lenguaje, la interfaz se define por medio de un lenguaje universal con una sintaxis específica, conocido como **CORBA *Interface Definition Language* (IDL-lenguaje de definición de interfaces)**. Encontrará que la sintaxis de CORBA IDL es muy similar a la de Java o C++, pero recuerde que un objeto definido en CORBA IDL puede implementarse en un gran número de lenguajes, incluyendo C, C++, Java, COBOL, Smalltalk, Ada, List, Python, e IDLScript [omg.org/gettingstarted, 2]. Para cada uno de estos lenguajes, OMG tiene una traducción (*mapping*) estándar de CORBA IDL a dicho lenguaje de programación, de forma que se pueda usar un compilador para procesar las interfaces CORBA y generar los *proxies* necesarios para servir de interfaz ante la implementación del objeto o ante el cliente, que pueden estar, por tanto, escritos en cualquier lenguaje soportado por CORBA.

La Figura 10.2 muestra una aplicación donde el cliente es un programa escrito en Java mientras que el objeto está implementado en C++. Cabe resaltar que el *stub* generado por la traducción de la interfaz del objeto CORBA es un *stub* en Java, mientras que el *skeleton* se ha generado traduciendo el interfaz a un *skeleton* C++. A pesar de estar implementados en diferentes lenguajes, los dos ORB pueden interactuar gracias a los protocolos comunes soportados por ambos.

Existe un gran número de ORB disponibles, tanto experimentales como comerciales [corba.org, 1; puder.org, 19].

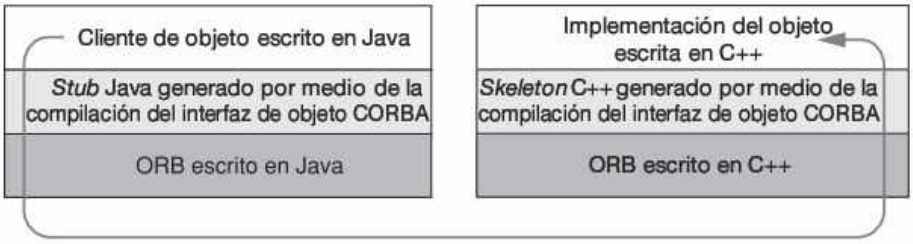


Figura 10.2. Independencia de lenguaje en CORBA.

10.3. PROTOCOLOS INTER-ORB

Para que dos ORB puedan interoperar, OMG ha especificado un protocolo conocido como ***General Inter-ORB Protocol* (GIOP)**, una especificación que proporciona un marco general para que se construyan protocolos interoperables por encima de un nivel de transporte específico. Un caso especial de este protocolo es el ***Internet Inter-ORB Protocol* (IIOP)**, que se corresponde con el GIOP aplicado sobre el nivel de transporte de TCP/IP.

La especificación de IIOP incluye los siguientes elementos:

1. **Requisitos de gestión de transporte**. Estos requisitos especifican qué se necesita para conectarse y desconectarse, y los papeles que el cliente y el objeto servidor interpretan en establecer y liberar conexiones.

2. **Definición de la representación común de datos**. Se necesita definir un esquema de codificación para empaquetar y desempaquetar los datos para cada tipo de datos del IDL.
3. **Formatos de los mensajes**. Se necesita definir los diferentes formatos de los distintos tipos de mensajes. Los mensajes permiten a los clientes enviar solicitudes al objeto servidor y recibir sus respuestas. Cada cliente usa un mensaje de petición para invocar a un método declarado en la interfaz CORBA de un objeto y recibe un mensaje de respuesta del servidor.

En arquitectura de computadores, un bus es equivalente a una autopista sobre la cual viajan los datos entre los componentes de un computador. En redes informáticas, un bus es un cable central que interconecta dispositivos.

Un ORB que soporte la especificación de IIOP puede interoperar con otro ORB compatible IIOP a través de Internet. De aquí surge la denominación **bus de objetos**: Internet se ve como un bus de objetos CORBA interconectados, como se muestra en la Figura 10.3.

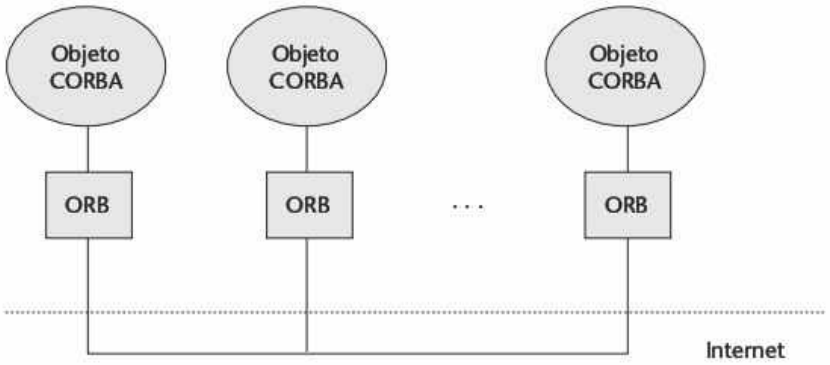


Figura 10.3. Internet como un bus de objetos.

10.4. SERVIDORES DE OBJETOS Y CLIENTES DE OBJETOS

Como en el caso de Java RMI, un **servidor de objetos** exporta cada objeto distribuido CORBA, similar al caso del servidor de objetos RMI. Un **cliente de objetos** obtiene la referencia a un objeto distribuido por medio de un servicio de nombres o de directorio (que se verá a continuación) y posteriormente invoca los métodos de dicho objeto distribuido.

10.5. REFERENCIAS A OBJETOS CORBA

También como en el caso de Java RMI, un objeto distribuido CORBA se localiza por medio de una **referencia a objeto**. Ya que CORBA es independiente de lenguaje, una referencia a un objeto CORBA es una entidad abstracta traducida a una referencia de objeto específica de cada lenguaje por medio del ORB, en una representación elegida por el desarrollador del propio ORB.

Por interoperabilidad, OMG especifica un protocolo para referencias abstractas de objetos, conocido como protocolo ***Interoperable Object Reference* (IOR)**. Un ORB que sea compatible con el protocolo IOR permitirá que una referencia a objeto se registre y se obtenga desde un servicio de directorio compatible con IOR. Las referen-

cias a objetos CORBA representadas en este protocolo se denominan también **IOR (*Interoperable Object References*)**.

Una IOR es una cadena de caracteres que contiene codificada la información siguiente:

- El tipo de objeto.
- El ordenador donde se encuentra el objeto.
- El número de puerto del servidor del objeto.
- Una clave del objeto, una cadena de *bytes* que identifica al objeto. La clave de objeto la utiliza el servidor de objetos para localizar el objeto internamente.

Una IOR es una cadena de caracteres parecida a esta:

```
IOR:000000000000000d49444c3a677269643a312e3000000
00000000001000000000000004c000100000000015756c74
72612e6475626c696e2e696f6e612e6965000009630000002
83a5c756c7472612e6475626c696e2e696f6e612e69653a67
7269643a303a3a49523a67726964003a
```

La representación consiste en un prefijo con los caracteres IOR: seguido por una secuencia hexadecimal de caracteres numéricos, cada carácter representa 4 bits de datos binarios en la IOR. Los detalles de la representación no son importantes para este estudio; a los lectores que estén interesados se les refiere a la bibliografía para más detalle.

10.6. SERVICIO DE NOMBRES Y SERVICIO DE NOMBRES INTEROPERABLE DE CORBA

En el Capítulo 7, cuando se estudió Java RMI, se presentó el registro RMI como un servicio de directorio distribuido para objetos RMI distribuidos. CORBA especifica un servicio de directorio con el mismo propósito. El **servicio de nombres (*Naming Service*)** [omg.org/technology, 5; java.sun.com/j2se, 16] sirve como un directorio de objetos CORBA, haciendo el papel de su análogo, el registro RMI, con la excepción de que el Servicio de Nombres de CORBA es independiente de plataforma y de lenguaje de programación.

Servicio de nombres de CORBA

El Servicio de Nombres permite que clientes basados en ORB obtengan las referencias a los objetos que desean usar. Permite asociar nombres a referencias objetos. Los clientes pueden consultar al Servicio de Nombres usando un nombre predeterminado para obtener la referencia asociada al objeto.

Para exportar un objeto distribuido, un servidor de objetos CORBA contacta con el Servicio de Nombres para **asociar (*bind*)** un nombre simbólico al objeto. El Servicio de Nombres mantiene una base de datos con los nombres y los objetos asociados a dichos nombres.

Para obtener una referencia a un objeto, un cliente de objetos solicita que el Servicio de Nombres busque el objeto asociado con dicho nombre (a este proceso se le denomina **resolución (*resolve*)** del nombre del objeto). El API para el Servicio de Nombres se encuentra especificada por medio de una interfaz IDL, que incluye métodos que permiten a servidores asociar nombres a objetos y a los clientes resolverlos. Para ser lo

más general posible, el esquema de nombrado de objetos en CORBA es necesariamente complejo. Ya que el espacio de nombrado es universal, se define una jerarquía de nombrado estándar de una manera similar a un directorio de ficheros, como se muestra en la Figura 10.4. En este esquema de nombrado, un **contexto de nombrado (*naming context*)** se corresponde con una carpeta o directorio en un árbol de ficheros, mientras que los nombres de objetos se corresponden con los ficheros. El nombre completo de un objeto, incluyendo todos los contextos de nombrado asociados, se denomina **nombre compuesto**. El primer componente de un nombre compuesto proporciona el nombre de contexto de nombrado donde se encuentra el segundo. Este proceso continúa hasta que se llega al último componente del nombre compuesto.

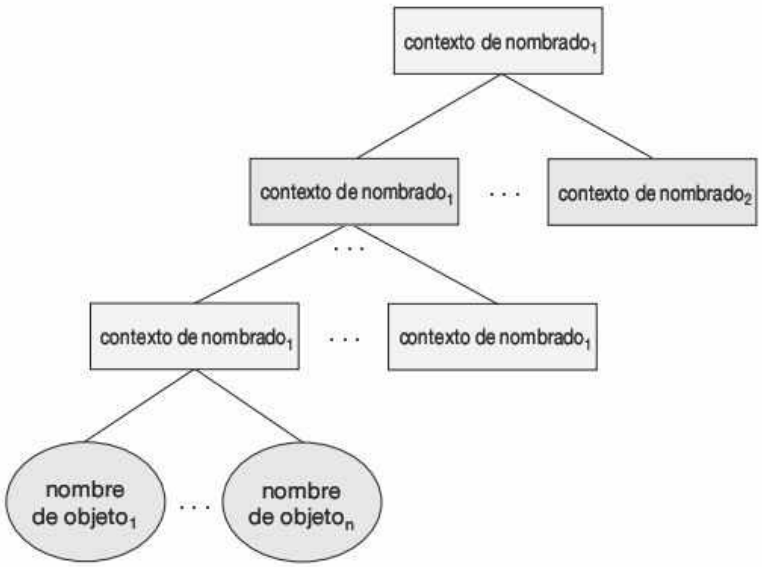


Figura 10.4. La jerarquía de nombrado de CORBA.

La sintaxis para un nombre de un objeto es la siguiente:

<contexto de nombrado>.<contexto de nombrado>. ... <contexto de nombrado>.<nombre del objeto>

Donde la secuencia de contextos de nombrado lleva hasta el nombre del objeto.

La Figura 10.5 muestra un ejemplo de una jerarquía de nombrado. Como se puede ver, un objeto que representa el departamento de ropa de hombres se denominaría *tienda.ropa.hombre*, donde *tienda* y *ropa* son contextos de nombrado, y *hombre* el nombre del objeto.

Los contextos de nombrado y los nombres se crean utilizando los métodos proporcionados por la interfaz del Servicio de Nombres.

Servicio de nombres interoperable de CORBA

El ***Interoperable Naming Service* (INS - Servicio de Nombres Interoperable)** [omg.org/technology, 5; java.sun.com/j2se, 16] es un sistema de nombrado basado en el formato URL para el Servicio de Nombres de CORBA. Permite que las aplicacio-

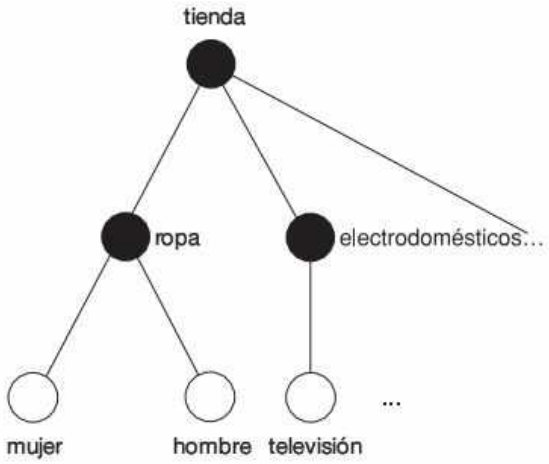


Figura 10.5. Un ejemplo de la jerarquía de nombrado de CORBA.

nes compartan un contexto inicial de nombrado y que proporcionen un URL para acceder a un objeto CORBA. Utilizando este sistema, un URL del tipo *corbaname::acme.com:2050#tienda/ropa/mujer* se podría usar para acceder al objeto *tienda/ropa/mujer* del servicio de nombres en el puerto 2050 del servidor con nombre *acme.com*.

10.7. SERVICIOS DE OBJETOS CORBA

Dentro de las especificaciones de CORBA se encuentran varias que proporcionan servicios usados habitualmente por los objetos distribuidos para construir aplicaciones. A continuación se recogen algunos de los servicios [corba.org, 1]:

- **Servicio de concurrencia (*Concurrency Service*)** – servicio que proporciona control de concurrencia.
- **Servicio de eventos (*Event Service*)** – para la sincronización de eventos.
- **Servicio de *log* (*Logging Service*)** – para registrar eventos.
- **Servicio de nombres (*Naming Service*)** – servicio de directorio, como se ha descrito en la sección anterior.
- **Servicio de planificación (*Scheduling Service*)** – para planificación de eventos.
- **Servicio de seguridad (*Security Service*)** – para gestión de seguridad.
- **Servicio de negociación (*Trading Service*)** – para localizar servicios por tipo (no por nombre).
- **Servicio de tiempo (*Time Service*)** – un servicio para eventos relativos al tiempo.
- **Servicio de notificación (*Notification Service*)** – para notificación de eventos.
- **Servicio de transacciones de objetos (*Object Transaction Service*)** – para procesamiento de transacciones.

Cada servicio se define por medio de un IDL estándar que puede ser implementado por los desarrolladores de objetos de servicio, y los métodos del objeto de servicio se pueden invocar desde cualquier cliente CORBA.

Como ejemplo, la especificación de CORBA del servicio de tiempo describe las siguientes funcionalidades:

- Permite al usuario obtener el tiempo en el instante actual junto con una estimación del error cometido.
- Hacer comprobaciones sobre el orden en el que se han producido varios «eventos».
- Generar eventos relativos al tiempo por medio de temporizadores y alarmas.
- Calcular el intervalo de tiempo entre dos eventos.

El servicio de tiempo consiste en dos servicios, cada uno de ellos descrito por medio de una interfaz de servicio:

1. El **Servicio de tiempo** maneja objetos de tipo tiempo universal (*Universal Time*) y objetos de tipo intervalo de tiempo (*Time Interval*) y está representado por la interfaz ***TimeService***. El servicio proporciona herramientas para registrar tiempos en aplicaciones CORBA.
2. El **Servicio de eventos de tiempo** gestiona objetos del tipo manejadores de eventos de tiempo (*Time Event Handlers*) y se encuentra recogido en la interfaz **TimeEventService**, que proporciona métodos para programar y cancelar temporizadores con tiempos absolutos, relativos o periódicos.

Por medio de estas especificaciones, se han desarrollado varios objetos para el servicio de tiempo de CORBA por parte de diferentes fabricantes, investigadores o individualmente.

10.8. ADAPTADORES DE OBJETOS

En la arquitectura básica de CORBA, la implementación de objetos distribuidos interactúa con el *skeleton* para conectarse con el *stub* en el lado del cliente. Cuando la arquitectura evolucionó, se añadió un componente software al *skeleton* en el lado del servidor: el **adaptador de objetos (*object adapter*)** (*véase* la Figura 10.6). Un adaptador de objetos simplifica las responsabilidades que tiene un ORB asistiéndole en hacer llegar las peticiones del cliente a la implementación del objeto. Cuando un ORB recibe una petición de un cliente, localiza el adaptador de objetos asociado a dicho objeto y redirige la petición a dicho adaptador. El adaptador interactúa con el *skeleton* de la implementación del objeto, el cual realiza el empaquetamiento e invoca el método apropiado del objeto.

Hay diferentes tipos de adaptadores de objetos CORBA. El ***Portable Object Adapter* (POA – adaptador de objetos portable)** [citeseer.nj.nec.com, 8] es un tipo particular de adaptador de objetos definido por una especificación de CORBA. Un adaptador de objetos de tipo POA permite que una implementación de objeto funcione en varios ORB, de forma que la implementación del objeto sea portable a través de varias plataformas.

10.9. IDL DE JAVA

Con las especificaciones de CORBA, un programador puede implementar cualquier componente especificado en el marco de CORBA. En realidad, un gran número de

Figura 10.6. Un adaptador de objetos.

herramientas CORBA se encuentran disponibles [corba.org, 1], muchas de las cuales puede obtenerse sin coste alguno. Algunas de estas herramientas están desarrolladas por empresas, otras por investigadores y algunas otras por grupos independientes.

Como ejemplo de una herramienta de este tipo, se va a ver el IDL de Java (o Java IDL). Java IDL es parte del la plataforma *Java 2, Standard Edition* (J2SE). Java IDL incluye un **ORB**, un **compilador *IDL-a-Java*** y un subconjunto de los **servicios estándar de CORBA**. Es necesario reseñar que además de Java IDL, Java proporciona varias herramientas compatibles con CORBA [java.sun.com/j2ee, 7], incluyendo RMI sobre IIOP [java.sun.com/j2se/1.3, 9], que permite que una aplicación CORBA se escriba con la sintaxis y la semántica de RMI. Con su conocimiento de RMI y CORBA, debería ser capaz de aprender RMI sobre IIOP por su cuenta. En el resto de este capítulo, se tratará Java IDL como un ejemplo de entorno CORBA.

Paquetes claves de Java IDL

Java IDL proporciona una serie de paquetes que contienen interfaces y clases para dar soporte a CORBA [java.sun.com/j2se/1.4, 17]:

- El paquete *org.omg.CORBA* contiene las interfaces y las clases que proporcionan la traducción (*mapping*) de las API de CORBA OMG al lenguaje de programación Java.
- El paquete *org.omg.CosNaming* contiene los interfaces y las clases que dan soporte al Servicio de Nombrado para Java IDL.
- *org.omg.CORBA* contiene interfaces y clases para soportar el API de acceso al ORB.

Herramientas de Java IDL

Java IDL dispone de un conjunto de herramientas para desarrollar aplicaciones CORBA [java.sun.com/products, 18]:

- ***idlj*** – el compilador IDL-a-Java (*Nota*: se desaconseja la utilización de la anterior versión de este compilador, *idl2java*.)
- ***orbd*** – proceso servidor que da soporte al Servicio de Nombrado, así como a otros servicios.
- ***servertool*** – proporciona una interfaz en línea de mandatos para que los programadores de aplicaciones puedan registrar/desregistrar objetos, y arrancar/parar servidores.
- ***tnameserv*** – una versión antigua del Servicio de Nombrado de Java IDL cuyo uso se desaconseja actualmente.

Una aplicación CORBA de ejemplo

El siguiente ejemplo ilustra el uso de las funcionalidades de soporte a IDL disponibles en el entorno *Java 2 Standard Edition* (J2SE), versión 1.4 [java.sun.com/j2se/1.4, 20]. Si usted está utilizando una versión anterior de Java, como J2SE 1.3, la sintaxis y algunos de los mecanismos que se describirán aquí serán diferentes. El ejemplo muestra un objeto CORBA que proporciona un método que devuelve la cadena de caracteres "*Hola mundo*". El ejemplo, aunque muy simple, permite ilustrar los fundamentos básicos.

En el ejemplo, el objeto distribuido se denomina *Hola*. Cuando desarrolle su aplicación, podrá substituirlos por un nombre diferente de objeto en cualquiera de los sitios en los que la palabra *Hola* aparezca como descripción.

Fichero de la interfaz CORBA

El punto de arranque de una aplicación CORBA es el desarrollo del fichero de interfaz CORBA escrito en el IDL de OMG, simplemente llamada fichero IDL. Recuerde que IDL es un lenguaje universal, de forma que la sintaxis de este fichero de interfaz es la misma independientemente de cuáles sean las herramientas CORBA utilizadas. La Figura 10.7 muestra un ejemplo de un posible fichero IDL que declara la interfaz llamada *Hola*. La interfaz define dos métodos: el método *decirHola()* no requiere ningún argumento y devuelve una cadena de caracteres (nótese que el tipo *string* se escribe en minúsculas; este tipo *string* es un tipo de datos del IDL de CORBA); el método *apagar()* desactiva el ORB y se recomienda incluirlo en todos los interfaces de servicio CORBA. Las palabras *module*, *interface*, *string*, y *oneway* son palabras reservadas en IDL. Un *module* (módulo), como indica la palabra, describe un módulo software. Se pueden declarar una o varias interfaces dentro de un mismo módulo, así como uno o varios métodos se pueden declarar dentro de cada interfaz. El modificador *oneway* denota que el método apagar requiere sólo una comunicación del cliente al servidor (y ninguna respuesta del servidor al cliente).

El fichero IDL puede colocarse en un directorio dedicado para esta aplicación. El fichero se compila usando el compilador *idlj* de la siguiente forma:

```
idlj –fall Hola.idl
```

Figura 10.7. *Hola.idl*.

```
1  module HolaApp
2  {
3     interface Hola
4     {
5        string decirHola();
6        oneway void apagar();
7     };
8  };
```

La opción *-fall* es necesaria para que el compilador genere todos los ficheros necesarios para el resto del ejemplo, que en este caso, los creará en el directorio *HolaApp*. En general, estos ficheros se generan en un subdirectorio denominado con el mismo nombre que el módulo compilado. Si la compilación tiene éxito se generan los siguientes ficheros:

1. *HolaOperations.java*
2. *Hola.java*
3. *HolaHelper.java*
4. *HolaHolder.java*
5. *_HolaStub.java*
6. *HolaPOA.java*

Estos ficheros se generan por el compilador idlj de forma automática como resultado de una compilación con éxito; estos ficheros no necesitan ninguna modificación por su parte. Se explicará brevemente cada uno de estos ficheros en los siguientes párrafos. Para este libro, resulta interesante de todas formas que conozca los contenidos de cada uno de estos ficheros.

HolaOperations.java, el interfaz de operaciones

El fichero *HolaOperations.java* (Figura 10.8), conocido de forma general como **interfaz de operaciones Java**, es un fichero de interfaz Java que traduce el fichero de interfaz IDL CORBA (*Hola.idl*).

El fichero contiene los métodos definidos en el fichero IDL original: en este caso los métodos *decirHola()* y *apagar()*.

Figura 10.8. *HolaApp/HolaOperations.java.*

```
1  package HolaApp;
2
3
4  /**
5  * HolaApp/HolaOperations.java
6  * Generated by the IDL-to-Java compiler
7  * version "3.1" from Hola.idl
8  */
9
10 public interface HolaOperations
11 {
12      String decirHola ();
13      void apagar ();
14 } // interface HolaOperations
```

Hola.java, el fichero de firma de interfaz

El fichero *Hola.java* (Figura 10.9), denominado **fichero de firma de interfaz**. Extiende las clases estándar de CORBA *org.omg.portable.IDLEntity*, *org.omg.CORBA.Object*, y la interfaz específica de la aplicación, *HolaOperations*, descrito en la sección anterior.

Figura 10.9. *HolaApp/Hola.java.*

```
1  package HolaApp;
2
3  /**
```

(continúa)

```
 4  * HolaApp/Hola.java
 5  * Generated by the IDL-to-Java compiler (portable),
 6  * version "3.1" from Hola.idl
 7  */
 8
 9  public interface Hola extends HolaOperations,
10       org.omg.CORBA.Object,
11       org.omg.CORBA.portable.IDLEntity
12  {
13  } // interface Hola
```

El fichero de firma de interfaz combina las características del interfaz de operaciones de Java (*HolaOperations.java*) con las características de las clases CORBA que extiende.

HolaHelper.java, la clase de ayuda

La clase Java *HolaHelper* (Figura 10.10), proporciona funcionalidades auxiliares necesarias para dar soporte a los objetos CORBA en el contexto del lenguaje de programación Java. En particular, un método, *narrow* (véase la línea 49) permite que una referencia a un objeto CORBA se pueda convertir a su tipo correspondiente en Java, de forma que el objeto CORBA pueda utilizarse con la sintaxis de un objeto Java. La descripción detallada de la sintaxis y la semántica del fichero está más allá del ámbito de este libro.

Figura 10.10. *HolaApp/HolaHelper.java.*

```
 1  package HolaApp;
 2
 3
 4  /**
 5  * HolaApp/HolaHelper.java
 6  * Generated by the IDL-to-Java compiler (portable),
 7  * version "3.1" from Hola.idl
 8  */
 9
10  abstract public class HolaHelper
11  {
12       private static String _id = "IDLHolaApp/Hola1.0";
13
14       public static void insert
15        (org.omg.CORBA.Any a, HolaApp.Hola that)
16       {
17        org.omg.CORBA.portable.OutputStream out =
18             a.create_output_stream ();
19        a.type (type ());
20        write (out, that);
21        a.read_value (out.create_input_stream (), type ());
```

(continúa)

```
22      }// end insert
23
24      public static HolaApp.Hola extract (org.omg.CORBA.Any a)
25      {
26       return read (a.create_input_stream ());
27      } //end extract
28
29      private static org.omg.CORBA.TypeCode __typeCode = null;
30      synchronized public static org.omg.CORBA.TypeCode type (
31      {
32       if (__typeCode == null)
33            {
34             __typeCode = org.omg.CORBA.ORB.init().
35                  create_interface_tc
36                  (HolaApp.HolaHelper.id (), "Hola");
37            }
38       return __typeCode;
39      } //end type
40
41      public static String id ()
42      {
43       return _id;
44      } // end id
45
46      public static HolaApp.Hola read
47       (org.omg.CORBA.portable.InputStream istream)
48      {
49       return narrow (istream.read_Object (_HolaStub.class));
50      } //end read
51
52      public static void write
53       (org.omg.CORBA.portable.OutputStream ostream,
54        HolaApp.Hola value)
55      {
56       ostream.write_Object ((org.omg.CORBA.Object) value);
57      } //end write
58
59      public static HolaApp.Hola narrow
60       (org.omg.CORBA.Object obj)
61      {
62       if (obj == null)
63            return null;
64       else if (obj instanceof HolaApp.Hola)
65            return (HolaApp.Hola)obj;
66       else if (!obj._is_a (id ()))
67            throw new org.omg.CORBA.BAD_PARAM ();
68       else
69            {
```

(continúa)

```
70              org.omg.CORBA.portable.Delegate delegate =
71                   ((org.omg.CORBA.portable.ObjectImpl)obj).
72                   _get_delegate ();
73              HolaApp._HolaStub stub = new HolaApp._HolaStub ();
74              stub._set_delegate(delegate);
75              return stub;
76             } // end else
77       }// end narrow
78
79  }// end class
```

HolaHolder.java, la clase contenedora

En IDL, un parámetro se puede declarar como *out* si es un parámetro de salida y *inout* si el parámetro contiene un valor de entrada y almacena a la vez un valor de salida.

La clase Java *HolaHolder* (Figura 10.11), contiene una referencia al objeto que implementa la interfaz *Hola*. La clase proyecta los parámetros de tipo *out* o *inout* del IDL a la sintaxis de Java.

Figura 10.11. *HolaApp/HolaHolder.java.*

```
 1  package HolaApp;
 2
 3  /**
 4  * HolaApp/HolaHolder.java
 5  * Generated by the IDL-to-Java compiler (portable), version "3.1"
 6  * from hello.idl
 7  * Sunday, December 29, 2002 34150 PM PST
 8  */
 9
10  public final class HolaHolder implements org.omg.CORBA.portable.
       Streamable
11  {
12       public HolaApp.Hola value = null;
13
14       public HolaHolder ()
15       {
16       }
17
18       public HolaHolder (HolaApp.Hola initialValue)
19       {
20        value = initialValue;
21       }
22
23       public void _read (org.omg.CORBA.portable.InputStream i)
24       {
25        value = HolaApp.HolaHelper.read (i);
26       }
27
28       public void _write (org.omg.CORBA.portable.OutputStream o)
```

(continúa)

```
29        {
30         HolaApp.HolaHelper.write (o, value);
31        }
32
33        public org.omg.CORBA.TypeCode _type ()
34        {
35         return HolaApp.HolaHelper.type ();
36        }
37
38    }
```

_HolaStub.java, el fichero de resguardo (*stub*)

La clase Java *HolaStub* (Figura 10.12), es el fichero de resguardo o *stub*, el *proxy* de cliente, que interactúa con el objeto cliente. Extiende *org.omg.CORBA.portable.ObjectImpl* e implementa la interfaz *Hello.java*.

Figura 10.12. *HolaApp/_HolaStub.java*.

```
 1  package HolaApp;
 2
 3
 4  /**
 5  * HolaApp/_HolaStub.java
 6  * Generated by the IDL-to-Java compiler (portable),
 7  * version "3.1" from Hola.idl
 8  */
 9
10  public class _HolaStub extends
11       org.omg.CORBA.portable.ObjectImpl
12       implements HolaApp.Hola
13  {
14
15       public String decirHola ()
16       {
17        org.omg.CORBA.portable.InputStream $in = null;
18        try {
19             org.omg.CORBA.portable.OutputStream $out =
20              _request ("decirHola", true);
21             $in = _invoke ($out);
22             String $result = $in.read_string ();
23             return $result;
24        } catch (org.omg.CORBA.portable.ApplicationException $ex) {
25             $in = $ex.getInputStream ();
26             String _id = $ex.getId ();
27             throw new org.omg.CORBA.MARSHAL (_id);
28        } catch (org.omg.CORBA.portable.RemarshalException $rm) {
29             return decirHola ();
```

(continúa)

```
30          } finally {
31               _releaseReply ($in);
32          }
33        } // decirHola
34
35        public void apagar ()
36        {
37         org.omg.CORBA.portable.InputStream $in = null;
38         try {
39              org.omg.CORBA.portable.OutputStream $out =
40               _request ("apagar", false);
41              $in = _invoke ($out);
42         } catch (org.omg.CORBA.portable.ApplicationException $ex) {
43              $in = $ex.getInputStream ();
44              String _id = $ex.getId ();
45              throw new org.omg.CORBA.MARSHAL (_id);
46         } catch (org.omg.CORBA.portable.RemarshalException $rm) {
47              apagar ();
48         } finally {
49              _releaseReply ($in);
50         }
51        } // apagar
52
53        // Type-specific CORBAObject operations
54        private static String[] __ids = {
55          "IDLHolaApp/Hola1.0"};
56
57        public String[] _ids ()
58        {
59         return (String[])__ids.clone ();
60        }
61
62        private void readObject (java.io.ObjectInputStream s)
63         throws java.io.IOException
64        {
65         String str = s.readUTF ();
66         String[] args = null;
67         java.util.Properties props = null;
68         org.omg.CORBA.Object obj =
69              org.omg.CORBA.ORB.init (args, props).
70              string_to_object (str);
71         org.omg.CORBA.portable.Delegate delegate =
72              ((org.omg.CORBA.portable.ObjectImpl) obj).
73              _get_delegate ();
74         _set_delegate (delegate);
75        }
76
77        private void writeObject (java.io.ObjectOutputStream s)
```

(continúa)

```
78          throws java.io.IOException
79        {
80          String[] args = null;
81          java.util.Properties props = null;
82          String str = org.omg.CORBA.ORB.init
83              (args, props).object_to_string (this);
84          s.writeUTF (str);
85        }
86    } // class _HolaStub
```

HolaPOA.java, el *skeleton* del servidor y el adaptador de objetos POA

La clase Java *HolaPOA* (Figura 10.13), es la combinación del *skeleton* (el *proxy* asociado al servidor) y el adaptador de objetos POA (*Portable Object Adapter*). Extiende la clase *org.omg.PosrtableServer.Servant*, e implementa las interfaces *InvokeHandler* y *HolaOperations*.

Figura 10.13. *HolaApp/HolaPOA.java.*

```
1   package HolaApp;
2
3
4   /**
5   * HolaApp/HolaPOA.java
6   * Generated by the IDL-to-Java compiler (portable),
7   * version "3.1" from Hola.idl
8   */
9
10  public abstract class HolaPOA extends
11        org.omg.PortableServer.Servant
12        implements HolaApp.HolaOperations,
13        org.omg.CORBA.portable.InvokeHandler
14  {
15
16        // Constructors
17
18        private static java.util.Hashtable _methods =
19          new java.util.Hashtable ();
20        static
21        {
22          _methods.put ("decirHola", new java.lang.Integer (0));
23          _methods.put ("apagar", new java.lang.Integer (1));
24        }
25
26        public org.omg.CORBA.portable.OutputStream _invoke
27          (String $method, org.omg.CORBA.portable.InputStream in,
28            org.omg.CORBA.portable.ResponseHandler $rh)
29        {
```

(continúa)

```
30        org.omg.CORBA.portable.OutputStream out = null;
31        java.lang.Integer __method =
32             (java.lang.Integer)_methods.get ($method);
33        if (__method == null)
34             throw new org.omg.CORBA.BAD_OPERATION
35              (0, org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE);
36
37        switch (__method.intValue ())
38             {
39             case 0 // HolaApp/Hola/decirHola
40              {
41                   String $result = null;
42                   $result = this.decirHola ();
43                   out = $rh.createReply();
44                   out.write_string ($result);
45                   break;
46              }
47
48             case 1 // HolaApp/Hola/apagarpackage HolaApp;
49              {
50                   this.apagar ();
51                   out = $rh.createReply();
52                   break;
53              }
54
55             default
56              throw new org.omg.CORBA.BAD_OPERATION
57                   (0,org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE);
58             }
59
60         return out;
61       } // _invoke
62
63       // Type-specific CORBAObject operations
64       private static String[] __ids = {
65         "IDLHolaApp/Hola1.0"};
66
67       public String[] _all_interfaces
68         (org.omg.PortableServer.POA poa, byte[] objectId)
69       {
70         return (String[])__ids.clone ();
71       }
72
73       public Hola _this()
74       {
75         return HolaHelper.narrow(
76                              super._this_object());
77       }
```

(continúa)

```
78
79        public Hola _this(org.omg.CORBA.ORB orb)
80        {
81         return HolaHelper.narrow(
82                                super._this_object(orb));
83        }
84
85    } // class HolaPOA
```

La aplicación

Aparte del fichero IDL (*Hola.idl*), los ficheros vistos hasta el momento, todos ellos se generan automáticamente cuando se compila el fichero IDL con el compilador *idlj* y no se necesitan modificar. A continuación se va a ver los ficheros fuente de la aplicación que sí debe desarrollar el programador.

Clases en el servidor

En el lado del servidor, se necesitan dos clases: el *servant* y el servidor. El *servant*, *HolaImpl*, es la implementación de la interfaz IDL *Hola*; cada objeto *Hola* es una instancia de esta clase.

El *servant* (N. del T.: *sirviente* en inglés), como se encuentra codificado en la Figura 10.14, es una subclase de *HolaPOA*. El *servant* contiene la definición de cada método declarado en la interfaz IDL: en este ejemplo, los métodos *decirHola* y *apagar*. Obsérvese que la sintaxis para escribir estos métodos es la misma que la de los métodos habituales en Java: la lógica para interactuar con el ORB, y para el empaquetamiento (*marshalling*) de datos se proporciona por medio del *skeleton*, cuyo código se incluye en *HolaPOA.java* (Figura 10.13).

Figura 10.14. *HolaApp/HolaImpl.java*.

```
 1  // El servant - la implementación del objeto - para el ejemplo
 2  // Hola. Obsérvese que se trata de una subclase de HolaPOA, cuyo
 3  // código fuente se genera de la compilación de Hola.idl usando
 4  // el compilador idlj.
 5
 6  import HolaApp.*;
 7  import org.omg.CosNaming.*;
 8  import org.omg.CosNaming.NamingContextPackage.*;
 9  import org.omg.CORBA.*;
10  import org.omg.PortableServer.*;
11  import org.omg.PortableServer.POA;
12
13  import java.util.Properties;
14
15  class HolaImpl extends HolaPOA {
16       private ORB orb;
```

(continúa)

```
17
18        public void setORB(ORB orb_val) {
19         orb = orb_val;
20        }
21
22        // implementación del método decirHola()
23        public String decirHola() {
24         return "\nHola mundo !!\n";
25        }
26
27        // implementación del método apagar()
28        public void apagar() {
29         orb.apagar(false);
30        }
31    } //fin clase
```

El servidor. La Figura 10.15 presenta el código de ejemplo para un servidor de objeto. El proceso servidor es el responsable de crear e inicializar el ORB (en este caso, el ORB de Java IDL), activar el gestor del adaptador de objetos POA (*Portable Object Adapter Manager*), crear una instancia de la implementación del objeto (un *servant*), y de registrar el objeto en el ORB. Obsérvese que en la implementación de Java, el Servicio de Nombrado se proporciona por el propio ORB. En el código, el Servicio de Nombrado Interoperable se utiliza para registrar el objeto con el nombre "*Hola*" (líneas 25-37). Después de anunciar que el servidor de objetos está listo, el servidor se queda a la espera de peticiones de los clientes redirigidas por el ORB (líneas 39-43).

Figura 10.15. *HolaApp/ServidorHola.java*.

```
1   // Un servidor para el objeto Hola
2
3   public class ServidorHola {
4
5       public static void main(String args[]) {
6        try{
7             // crea e inicializa el ORB
8             ORB orb = ORB.init(args, null);
9
10            // obtiene la referencia al rootPOA y activa al POAManager
11            POA rootpoa =
12              (POA)orb.resolve_initial_references("RootPOA");
13            rootpoa.the_POAManager().activate();
14
15            // crea un servant y lo registra en el ORB
16            HolaImpl holaImpl = new HolaImpl();
17            holaImpl.setORB(orb);
18
19            // obtiene la referencia de objeto del servant
20            org.omg.CORBA.Object ref =
```

(continúa)

```
21              rootpoa.servant_to_reference(holaImpl);
22             // y la convierte a una referencia CORBA
23             Hola href = HolaHelper.narrow(ref);
24
25             // obtiene el contexto de nombrado raíz
26             // NameService invoca al servicio de nombres temporal
27             org.omg.CORBA.Object objRef =
28              orb.resolve_initial_references("NameService");
29             // Usa NamingContextExt, que es parte de la especificación
30             // Servicio de Nombrado Interoperable (INS).
31             NamingContextExt ncRef =
32              NamingContextExtHelper.narrow(objRef);
33
34             // enlza la referencia del objeto con el nombre
35             String name = "Hola";
36             NameComponent path[] = ncRef.to_name( name );
37             ncRef.rebind(path, href);
38
39             System.out.println
40              ("ServidorHola listo y esperando ...");
41
42             // espera las invocaciones de los clientes
43             orb.run();
44         }
45
46         catch (Exception e) {
47             System.err.println("ERROR " + e);
48             e.printStackTrace(System.out);
49         }
50
51         System.out.println("ServidorHola Saliendo ...");
52
53       } // fin main
54   } // fin clase
```

La aplicación cliente del objeto

La Figura 10.16 muestra un ejemplo de cliente para el objeto *Hola*. El ejemplo se encuentra escrito como una aplicación Java, aunque el programa cliente también puede tratarse de un *applet* o un *servlet*.

El código cliente se debe encargar de crear e inicializar el ORB (línea 14), buscar el objeto usando el Servicio de Nombrado Interoperable (líneas 16-22), invocar el método *narrow* del objeto *Helper* para convertir la referencia del objeto a una referencia de una implementación del objeto *Hola* (líneas 24-27), e invocar los métodos remotos usando dicha referencia (líneas29-33). El método *decirHola* del objeto se invoca para recibir una cadena de caracteres como valor de retorno, y el método *apagar* para desactivar el servicio.

Figura 10.16. HolaApp/ClienteHola.java.

```
1  // Un ejemplo de aplicación cliente del objeto
2  import HolaApp.*;
3  import org.omg.CosNaming.*;
4  import org.omg.CosNaming.NamingContextPackage.*;
5  import org.omg.CORBA.*;
6
7  public class ClienteHola
8  {
9      static Hola holaImpl;
10
11      public static void main(String args[]){
12       try{
13            // crea e inicializa el ORB
14            ORB orb = ORB.init(args, null);
15
16            // obtiene el contexto de nombrado raíz
17            org.omg.CORBA.Object objRef =
18             orb.resolve_initial_references("NameService");
19            // Usa NamingContextExt en lugar de NamingContext
20            // parte del Servicio de Nombrado Interoperable.
21            NamingContextExt ncRef =
22             NamingContextExtHelper.narrow(objRef);
23
24           // resuelve la Referencia al objeto en el Serv. de Nombrado
25            String nombre = "Hola";
26            holaImpl =
27             HolaHelper.narrow(ncRef.resolve_str(nombre));
28
29            System.out.println
30             ("Obtenido un manejador para el objeto servidor "
31              + holaImpl);
32            System.out.println(holaImpl.decirHola());
33            holaImpl.apagar();
34
35       }
36       catch (Exception e) {
37            System.out.println("ERROR   " + e) ;
38            e.printStackTrace(System.out);
39       }
40      } //fin main
41
42 } // fin clase
```

En el resto del capítulo se van a presentar los algoritmos para desarrollar aplicaciones usando Java IDL.

Compilación y ejecución de una aplicación Java IDL

Para los ejercicios finales del capítulo, será necesaria la compilación y ejecución de aplicaciones Java IDL. A continuación se muestra una descripción de dicho procedimiento.

Parte servidora

1. Colóquese en el directorio que contiene el fichero IDL *Hola.idl*.
2. Ejecute el compilador IDL-a-Java, *idlj*, con el fichero IDL. Este paso asume que usted ha incluido el directorio *java/bin* en su *path*.

   ```
   idlj –fall Hola.idl
   ```

   Debe utilizarse la opción *-fall* con el compilador *idlj* para generar todo el código de soporte cliente y servidor. Los ficheros que se generan proporcionan las funcionalidades estándares y no necesitan ser modificados.

   Con la opción *-fall* los ficheros que el compilador genera con el fichero *Hola.idl* son:

   - *HolaPOA.java*
   - *_HolaStub.java*
   - *Hola.java*
   - *HolaHelper.java*
   - *HolaHolder.java*
   - *HolaOperations.java*

   Estos ficheros se crean automáticamente en un subdirectorio, que en el caso de este ejemplo se denomina *HolaApp*.
3. Compile los ficheros *.java* del directorio *HolaApp*, incluyendo los *stubs* y los *skeletons*.

   ```
   javac *.java HolaApp/*.java
   ```
4. Arranque el Demonio del ORB Java, *orbd*, que incluye el servidor del Servicio de Nombrado.

   Para arrancar *orbd* en un sistema UNIX, introduzca sobre la línea de comando:

   ```
   orbd –ORBInitialPort <número de puerto>&
   ```

   En un sistema Windows, introduzca sobre la línea de comando:

   ```
   start orbd –ORBInitialPort <número de puerto>&
   ```

   Nótese que <número de puerto> es un puerto en el cual usted desea que el servidor de nombre se ejecute; debe ser un número de puerto por encima de 1024, por ejemplo 1234.

   En sistemas Unix, un «demonio» es una tarea que se ejecuta en *background* en respuesta a eventos.
5. Arranque el servidor *Hola*.

   Para arrancar el servidor *Hola* en un sistema UNIX, introduzca (en una sola línea) lo siguiente:

   ```
   java ServidorHola –ORBInitialPort 1234 –ORBInitialHost localhost
   ```

   En un sistema Windows, introduzca (en una sola línea) lo siguiente:

   ```
   start java ServidorHola –ORBInitialPort 1234 –ORBInitialHost localhost
   ```

La opción *–ORBInitialHost* especifica el ordenador en el cual el Servidor de Nombrado está. La opción *–ORBInitialHost* indica el puerto en el cual el Servidor de Nombrado (*orbd*) se encuentra, tal y como se describió en el paso 4.

Parte cliente

1. Obtenga y compile el fichero *Hola.idl* de la máquina servidora:

   ```
   idlj –fall Hola.idl
   ```

   Copie el directorio que contiene *Hola.idl* (incluyendo el subdirectorio generado por *idlj*) a la máquina cliente.

2. En el directorio *HolaApp* de la máquina cliente, cree el fichero *ClienteHola.java*. Compile los ficheros *.java*, incluyendo *stubs* y *skeletons* (que están en el directorio *HolaApp)*:

   ```
   javac *.java HolaApp/*.java
   ```

3. En la máquina cliente, ejecute la aplicación cliente *Hola* de la siguiente forma (todo en una línea):

   ```
   java ClienteHola –ORBInitialHost <ordenador del servidor de nombrado>
   –ORBInitialPort <puerto del servidor de nombrado>
   ```

   El `<ordenador del servidor de nombrado>` es la máquina donde está ejecutando Servidor de Nombrado. En este caso, puede ser el nombre de dominio o la dirección IP de la máquina servidora.

Callback de cliente

Recuerde que RMI proporciona la posibilidad de realizar *callbacks* hacia el cliente, que permite que un cliente se registre en un servidor de objetos de forma que el servidor pueda iniciar una llamada al cliente, posteriormente, debido al suceso de algún evento. La misma posibilidad existe en el caso de Java IDL. La referencia [java.sun.com/products, 18] proporciona un ejemplo de código de dicha aplicación.

10.10. COMPARATIVA

Una cuestión clave en este libro es que en informática distribuida (y en informática en general) hay típicamente muchas formas de acometer la misma tarea. Este capítulo presenta un ejemplo. Como se ha podido ver, y como experimentará cuando aborde algunos de los ejercicios del final de este capítulo, la misma aplicación (por ejemplo, un juego en red) se puede implementar usando las herramientas ofrecidas por Java RMI o por CORBA, tales como Java IDL. Se espera que en este punto, habiendo estudiado las dos herramientas, será capaz de hacer una comparativa inteligente de las dos y realizar las comparaciones, tal y como se solicita en algunos de los ejercicios.

RESUMEN

Este capítulo le ha presentado la arquitectura CORBA (*Common Object Request Broker Architecture*) así como una herramienta específica basada en esta arquitectura: Java IDL.

Los temas clave de CORBA que se han presentado son:

- La arquitectura básica de CORBA y su énfasis en la interoperabilidad de objetos y la independencia de plataforma.
- El ORB (***Object Request Broker***) y sus funcionalidades.
- El protocolo IOP (***Inter-ORB Protocol***) y su significado.
- Referencia a objetos CORBA y el protocolo IOR (***Interoperable Object Reference***).
- El Servicio de Nombrado de CORBA y el Servicio de Nombrado Interoperable (INS).
- Los servicios de objetos CORBA estándar y cómo se proporcionan.
- Adaptadores de objetos, POA (***Portable Object Adapter***), y su significado.

Los temas clave presentados junto a Java IDL son:

- Los paquetes Java disponibles que contienen interfaces y clases para dar soporte CORBA.
- Las herramientas para desarrollar una aplicación CORBA, incluido *idlj* (el compilador IDL) y *orbd* (el ORB y servidor de nombres).
- Se ha presentado una aplicación de ejemplo llamada ***Hola*** para ilustrar la sintaxis básica de Java IDL.
- Se han presentado los pasos para compilar y ejecutar una aplicación.

Las herramientas para CORBA y Java RMI son tecnologías compatibles y alternativas que proporcionan objetos distribuidos. Una aplicación se puede implementar usando cualquiera de las dos tecnologías, pero existen ventajas e inconvenientes en cada una de ellas.

EJERCICIOS

1. En el contexto de CORBA, ¿qué significan los siguientes acrónimos? Para cada acrónimo, indique el nombre completo y una breve descripción: CORBA, ORB, GIOP, IIOP, IOR, INS, POA.
2. Describa, por medio de un diagrama de bloques una descripción que ilustre la arquitectura CORBA. El diagrama debe incluir los siguientes componentes: un objeto distribuido, un servidor de objetos, un cliente de objetos, el ***skeleton***, el ***stub***, el ORB y el adaptador de objetos.

   Describa, por medio de otro diagrama de bloques que ilustre la arquitectura Java RMI, incluyendo los componentes equivalentes: un objeto distribuido, un servidor de objetos, un cliente de objetos, el ***skeleton*** y el ***stub***.

   Basándose en sus diagramas, escriba un párrafo que describa las diferencias principales entre las dos arquitecturas. Trate de explicar dichas diferencias.
3. Comparada con Java RMI, ¿cuáles son los principales puntos fuertes de una herramienta CORBA, si hay alguno? ¿Cuáles son los puntos débiles, si hay alguno?
4. Siga el algoritmo presentado en el capítulo para compilar y ejecutar el ejemplo ***Hola*** en una máquina. Escriba un informe describiendo su experiencia, incluyendo cualquier dificultad que haya encontrado y cómo la ha resuelto.

5. Siga el algoritmo presentado en el capítulo para compilar y ejecutar el ejemplo *Hola* en dos máquinas. Escriba un informe describiendo su experiencia, incluyendo cualquier dificultad que haya encontrado y cómo la ha resuelto. (Puede encontrar la referencia [java.sun.com/j2se/1.4, 13] útil.)
6. Use Java IDL para construir un servidor y un cliente que implemente el protocolo *Daytime*.
7. Usando Java IDL, escriba una aplicación para un prototipo de un sistema de consultas de opinión. Asúmase que sólo se va a encuestar un tema. Los entrevistados pueden responder *sí*, *no* o *ns/nc*. Escriba una aplicación servidora, que acepte los votos, guarde la cuenta (en memoria), y proporcione las cuentas actuales a aquellos que estén interesados.
   a. Escriba el fichero de interfaz primero. Debería proporcionar métodos remotos para aceptar una respuesta a la encuesta, proporcionando los recuentos actuales (ejemplo: 10 sí, 2 no, 5 ns/nc) sólo cuando el cliente lo requiera.
   b. Diseñe e implemente un servidor que (i) exporte los métodos remotos, y (ii) mantenga información de estado (las cuentas).
   c. Diseñe e implemente una aplicación cliente que proporcione una interfaz de usuario para aceptar una respuesta y/o una petición, y para interactuar con el servidor apropiadamente a través de la invocación de métodos remotos.
   d. Pruebe la aplicación ejecutando dos o más clientes en máquinas diferentes (preferiblemente en plataformas diferentes).
   e. Entregue los listados de los ficheros, que deben incluir los ficheros fuente (el fichero de interfaz, los ficheros del servidor y los ficheros del cliente), y un fichero LEEME que explique los contenidos y las interrelaciones de los ficheros fuente, así como el procedimiento para ejecutar el trabajo.
8. Cree una aplicación Java IDL para gestionar unas elecciones. El servidor exporta dos métodos:
   - *emitirVoto*, que acepta como parámetro una cadena de caracteres que contiene un nombre de candidato (Gore o Bush), y no devuelve nada, y
   - *obtenerResultado*, que devuelve, en un vector de enteros, el recuento actual para cada candidato.

   Pruebe la aplicación ejecutando todos los procesos en una máquina. A continuación pruebe la aplicación ejecutando el proceso cliente y servidor en máquinas *separadas*. Entregue el código fuente de la interfaz remota, el servidor y el cliente.
9. Construya un juego distribuido para dos jugadores de las tres en raya usando (a) Java RMI con *callbacks* de cliente y (b) Java IDL con *callbacks* de cliente.

   Diseñe sus aplicaciones de forma que el cliente sea lo más ligero posible; esto es, que los clientes deben almacenar el menor estado y tengan el menor código posible. El servidor mantendrá el estado del juego y sincronizará los turnos de los jugadores.

   Entregue (a) documentos de diseño, incluyendo un diagrama de clases UML, (b) listado de las fuentes, y (c) un informe comparando ventajas e inconvenientes entre las dos tecnologías, en términos de facilidad de implementación, independencia de lenguaje, independencia de plataforma, y sobrecarga en la ejecución.

REFERENCIAS

1. *Welcome To The OMG's CORBA Website*, *http://www.corba.org/*
2. CORBA FAQ, *http://www.omg.org/gettingstarted/corbafaq.htm*
3. CORBA for Beginners, *http://cgi.omg.org/corba/beginners.html*
4. Jon Siegel. CORBA 3 Fundamentals and Programming. New York, NY: John Wiley, 2000.
5. OMG Naming Service specification, *http://www.omg.org/technology/documents/formal/naming_service.htm*
6. CORBA Naming Service Evaluation, Sean Landis and William Shapiro, *http://www.cs.umd.edu/~billshap/papers/naming.doc*, 1999.
7. CORBA and Java™ technologies, *http://java.sun.com/j2ee/corba/*
8. Irfan Pyarali and D. C. Schmidt. "An Overview of the CORBA Portable Object Adapter." ACM StandardView 6, (March 1998). *http://citeseer.nj.nec.com/pyarali98overview.html*
9. JavaTM RMI-IIOP Documentation, *http://java.sun.com/j2se/1.3/docs/guide/rmi-iiop/index.html*
10. Cetus Links: 18,452 Links on Objects and Components/CORBA, *http://www.cetus-links.org/oo_corba.html*
11. The Java Tutorial Trail: IDL, *http://java.sun.com/docs/books/tutorial/idl/index.html*
12. Java™ IDL, *http://java.sun.com/products/jdk/idl/*
13. Java IDL: The "Hello World" Example on Two Machines, *http://java.sun.com/j2se/1.4/docs/guide/idl/tutorial/jidl2machines.html*
14. Java IDL Sample code, *http://java.sun.com/j2se/1.4/docs/guide/idl/jidlSampleCode.html*
15. Java IDL: Naming Service, *http://java.sun.com/products/jdk/1.2/docs/guide/idl/jidlNaming.html*
16. Naming Service, Sun MicroSystems, *http://java.sun.com/j2se/1.4/docs/guide/idl/jidlNaming. html*
17. Java IDL Technology Documentation, *http://java.sun.com/j2se/1.4/docs/guide/idl/index.html*
18. Java IDL: Example 3, Hello World with Callback Object, *http://java.sun.com/products/jdk/1.4/docs/guide/idl/jidlExample3.html*
19. CORBA Product Profiles, *http://www.puder.org/corba/matrix/*
20. Java 2 Platform, Standard Edition (J2SE), *http://java.sun.com/j2se/1.4/*

CAPÍTULO 11

Aplicaciones de Internet – Parte 2

En el Capítulo 9 se estudió el HTTP, los CGI y el mantenimiento de la información de estado para aplicaciones de Internet. Debido a la popularidad de las aplicaciones web, han surgido una gran cantidad de protocolos y conjuntos de herramientas. En este capítulo se explorarán algunos de los protocolos y mecanismos más recientes, incluyendo los applets, servlets y SOAP (*Simple Object Access Protocol*, Protocolo Simple de Acceso a Objetos).

11.1. APPLETS

Explicado brevemente, los applets [java.sun.com/docs, 1; java.sun.com/applets, 2; javaboutique.internet.com, 3; java.sun.com/sfaq, 17] son clases Java solicitadas por el navegador a un servidor web utilizando el protocolo HTTP y ejecutadas a continuación por la máquina virtual de Java en el entorno del navegador (véase la Figura 11.1).

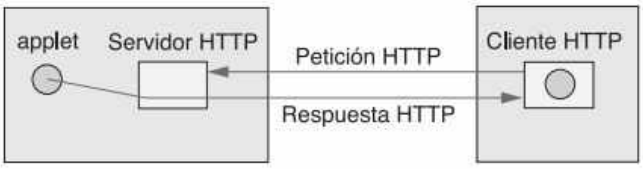


Figura 11.1. Un applet de Java se solicita utilizando HTTP.

Un applet se especifica en una página HTML usando la etiqueta APPLET, tal y como se muestra en la Figura 11.2.

Figura 11.2. Una página web que especifica un applet.

```
<Html>
<Head>
<Title>Ejemplo de applet</Title>
</Head>

<Body>
Esto es lo que muestra el applet después de su ejecución:<br>
<Applet Code="HolaMundo.class" width=200 Height=100>
</Applet>
```

Cuando el navegador analiza la etiqueta APPLET, se lanza una petición contra el servidor HTTP especificado en la etiqueta applet o, si no hay un servidor especificado, contra el servidor por defecto (el servidor del cual se ha descargado la página web). Un ejemplo de una petición HTTP sería el siguiente:

```
GET /applets/HolaMundo.class HTTP/1.1
<línea en blanco>
```

En la petición, HolaMundo.class es el nombre del fichero en donde está definida la clase del applet. El servidor HTTP localiza el fichero y envía su contenido al cliente en el cuerpo de la respuesta HTTP.

Una vez recibido el fichero de la clase del applet, el navegador lo ejecuta en su Máquina Virtual de Java (JVM) y muestra su resultado.

La Figura 11.3 muestra una sesión web durante la cual un cliente solicita una página web que contiene una etiqueta APPLET. A continuación la clase del applet se transmite desde el servidor al cliente (navegador), en cuya máquina se ejecuta la clase applet y se muestra el resultado.

Un visor de applets es un programa que se proporciona en el Java SDK para ejecutar un applet sin necesidad de utilizar un navegador.

Un applet es un tipo especial de programa Java que se carga en el navegador o en un visor de applets. Cada applet hereda de la clase Java *Applet*, la cual es una subclase de la clase Java *awt.Container*, que da soporte al pintado de gráficos. De esta forma, un applet puede ser fácilmente codificado para pintar gráficos. La Figura 11.4

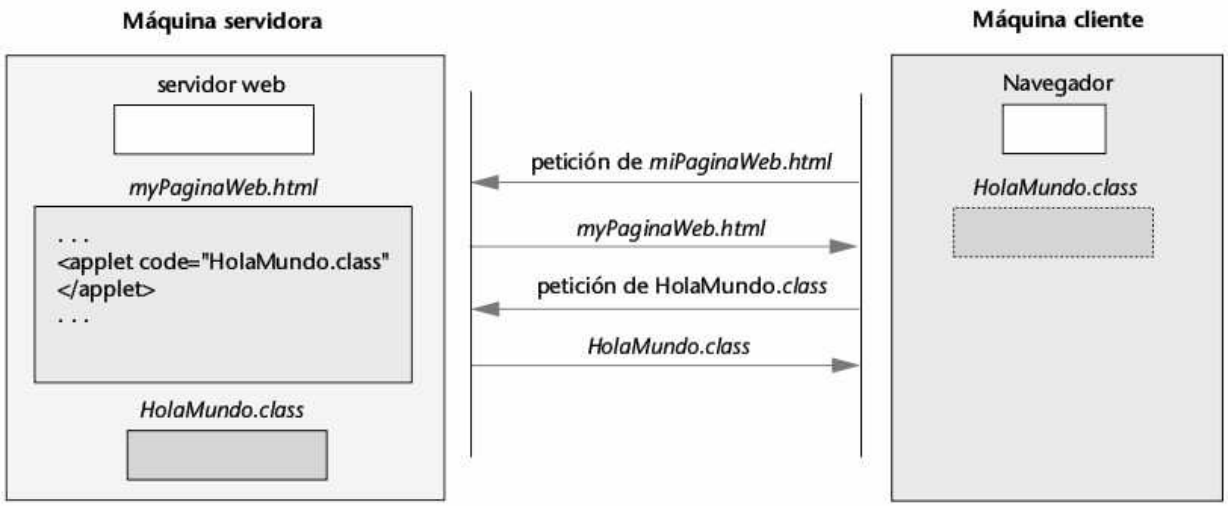


Figura 11.3. Una sesión web con un applet.

ilustra el código fuente de un applet que dibuja la cadena de texto «Hola mundo» cuando se ejecuta.

Figura 11.4. HelloWorld.java, the source code for an applet.

```
import java.applet.Applet;
import java.awt.Graphics;

public class HolaMundo extends Applet
{
public void pintar(Graphics g)
{
g.pintarCadena("Hola Mundo!", 50, 25);
} //fin pintar
} //fin clase
```

En la práctica, el código de un applet puede ser mucho más complejo, teniendo gráficos elaborados y manejo de eventos.

Es mejor probar los applets con un visor de applets antes de intentar ejecutarlos utilizando un navegador. Cuando se prueba un applet en un navegador, puede ser de ayuda mirar los mensajes que se muestran en la pantalla de la consola de Java. El Internet Explorer, por ejemplo, permite hacer esto si se selecciona la opción «consola Java activada» en el menú de herramientas de Internet.

Debido a que los applets se descargan de una máquina remota y se ejecutan en la máquina local, su ejecución está sometida a restricciones por razones de seguridad. (En el Capítulo 8 se vieron conceptos similares para la descarga del *stub* RMI.) Una de estas restricciones es que un applet no tiene permitido leer o escribir ficheros almacenados en el computador en el que está ejecutando. Otra restricción es que el applet no tiene permitido la realización de conexiones de red excepto a la máquina de la que proviene (*véase* la Figura 11.5). Existen otras restricciones [java.sun.com/sfaq, 17] impuestas para limitar el daño que se puede realizar a la máquina del sistema por un potencial objeto maligno descargado de un fuente de la que no se confía.

Los applets son programas interesantes. Sin embargo, en el contexto de la computación distribuida tienen una importancia limitada; se introducen aquí sobre todo por coherencia. Los lectores interesados pueden consultar otras fuentes [java.sun.com/docs, 1; java.sun.com/applets, 2; javaboutique.internet.com, 3; java.sun.com/sfaq, 17] para detalles adicionales. Se puede obtener código fuente de applets de ejemplo de numerosas fuentes, tales como [java.sun.com/sfaq, 17].

11.2. SERVLETS

Los servlets son otro tipo de programa Java. Mientras que los applets se mandan desde el servidor al cliente HTTP para ser ejecutados en la máquina cliente, los servlet son extensiones del servidor y son ejecutados en la máquina servidora. En el Capítulo 9 se estudiaron unos programas de extensión del servidor: scripts CGI. Como se recordará, los scripts CGI son programas externos que extienden las capacidades de un servidor HTTP para dar soporte al procesamiento de formularios web. De forma similar a los scripts CGI, un servlet HTTP (una forma especial de servlet) ejecuta en la máquina servidora como una acción desencadenada por la petición del cliente. A diferencia de un

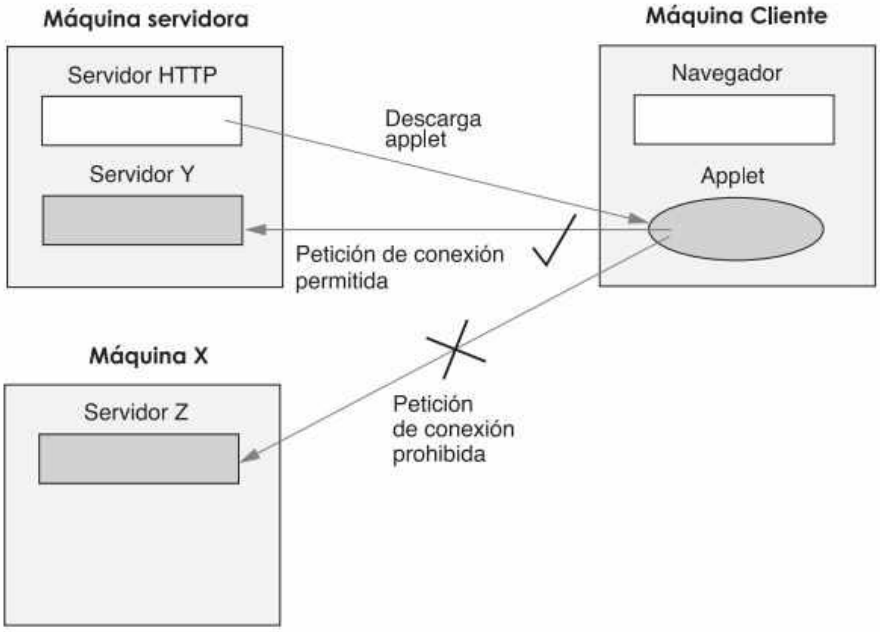


Figura 11.5. Un applet tiene prohibido realizar conexiones de red al exterior.

script CGI, sin embargo, un servlet puede ser utilizado para extender cualquier servidor que tenga un protocolo del tipo petición-respuesta. Los servlets se utilizan normalmente con servidores HTTP, en cuyo caso se denominan servlets HTTP.

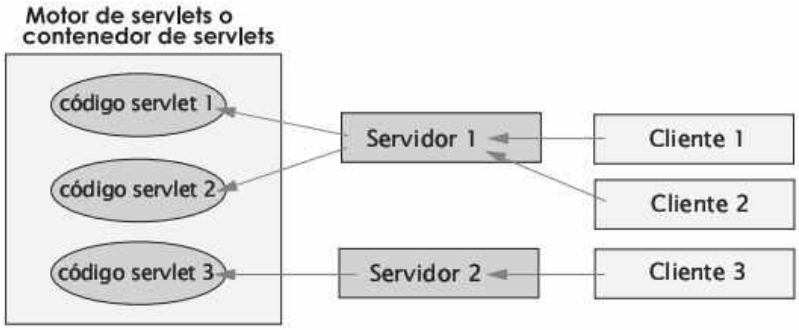
El resto de esta sección se centrará en los servlets HTTP, a los que se hará referencia simplemente como servlets; los lectores deberán tener en cuenta que se está abordando una clase especial de servlets.

Soporte arquitectónico

A diferencia de los scripts CGI (que ejecutan en la máquina servidora sin ningún sistema de soporte arquitectónico adicional) la ejecución de los servlets requiere la existencia de un módulo conocido como motor de servlets o contenedor de servlets (véase la Figura 11.6).

Cada servlet se ejecuta en el contexto proporcionado por el motor de servlets que ejecuta en la máquina servidora.

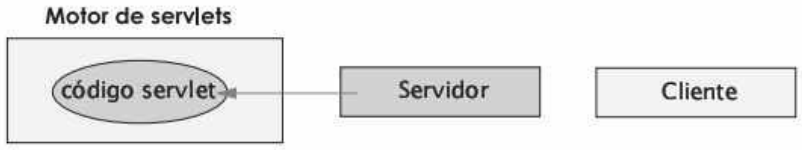
La Figura 11.7 muestra el ciclo de vida de un servlet. El código de un servlet, que es una clase Java, se carga en el motor de servlets. Posteriormente, el servidor actúa como un intermediario entre el cliente y el servlet: el cliente realiza peticiones al servlet a través del servidor, y el servlet manda la respuesta al cliente a través del servidor. Dependiendo de la implementación del servidor, un servlet puede persistir mientras siga teniendo peticiones, o de forma indefinida hasta que se apague el servidor. La persistencia es otra diferencia entre un servlet y un script CGI: un script CGI se recarga cada vez que un cliente lo solicita, mientras que una sola instancia de un servlet seguirá ejecutando mientras tenga peticiones. Debido a esta persistencia, un servlet puede mantener datos de estado de las sesiones de los clientes durante su tiempo de vida. Por ejemplo, se puede utilizar una variable para contar cuántas veces se ha solicitado un servlet desde su carga.



Se necesita un contenedor de servlets o motor de servlets. El motor de servlets puede formar parte del servidor o ser un módulo externo del mismo.

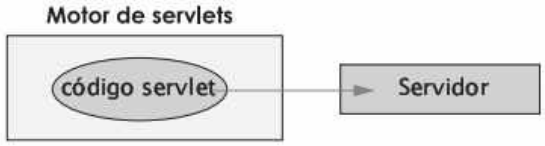
Figura 11.6. Soporte de la arquitectura de un servlet.

Un servlet es un objeto de la clase *javax.Servlet*, que es parte de una biblioteca de una clase de Java heredada denominada *Javax*; la biblioteca *Javax* no se incluye como parte del Java Development Kit (JDK) pero se puede descargar por separado [java.sun.com/products/servlet/download.html, 5].



El servidor carga el código fuente y lo inicializa, posiblemente como resultado de la petición de un cliente.

A través del servidor el servlet trata las peticiones de los clientes.



El servidor borra el servlet cuando no tiene más peticiones de clientes. (Algunos servidores realizan este paso sólo cuando se apagan).

Figura 11.7. El tiempo de vida de un servlet.

Existen varias implementaciones que proporcionan la arquitectura servlet. Las dos siguientes están fácilmente disponibles:

- El JSWDK (Java Server Web Development Kit) [java.sun.com/products/servlet/archive.html, 4] es un paquete gratuito proporcionado por Sun Microsystems desde que se introdujo por primera vez la tecnología servlet. JSWDK tiene la intención de ser una implementación de referencia, lo que significa que se proporciona por Sun Microsystems para demostrar la tecnología, pero no tiene la intención de ser utilizada en producción. Su simplicidad hace de este paquete un punto de inicio ideal para estudiantes. Desafortunadamente este paquete sólo está disponible en el sitio web de Sun, y no se puede asegurar su disponibilidad en el futuro.
- Apache Tomcat [jakarta.apache.org, 6] es una implementación gratuita de código abierto para las tecnologías Java Servlet y Java Server Pages desarrollado dentro del proyecto Yakarta en el Apache Software Foundation.

También está disponible el soporte a servlets en servidores de aplicación comerciales tales como WebLogic, iPlanet y WebSphere.

Entre los métodos especificados con cada objeto servlet están:

- *Init()* – invocado por el motor de servlets cuando se inicia un servlet.
- *Shutdown()* – invocado por el motor de servlets cuando un servlet ya no se necesita.
- *Service()* – invocado por el motor de servlets cuando la petición de un cliente se reenvía al servlet.

El diagrama de secuencia de la Figura 11.8 muestra la interacción entre el servidor HTTP, el contenedor de servlets, un servlet y los clientes concurrentes que mandan peticiones al servlet a través del servidor.

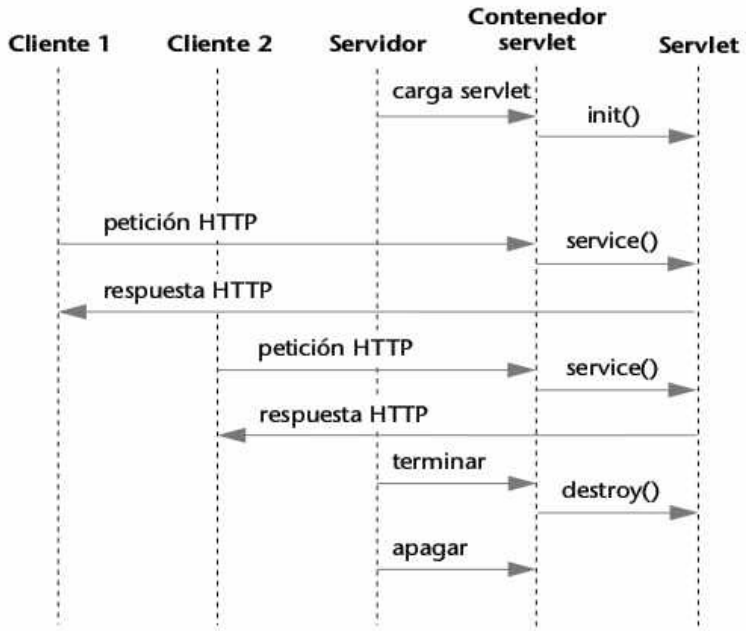


Figura 11.8. Interacción entre clientes, servidor, contenedor de servlets y servlets.

Programación de servlets

La programación de un servlet HTTP [Hunter and Crawford, 19] se simplifica gracias a la abstracción proporcionada por el API servlet, un conjunto de clases e interfaces que contienen los métodos *init()*, *destroy()* y *service()* que ya se han mencionado. La clase abstracta de Java *HTTPServlet*, que es una subclase de la interfaz servlet, proporciona abstracción adicional a los servlets HTTP tal y como se ilustra en la Figura 11.9 [jakarta.apache.org, 6].

Como una subclase de un servlet genérico, un servlet HTTP hereda los métodos *init()*, *destroy()* y *service()*. Dado el papel de los servlets como extensión del servidor HTTP, se definen métodos adicionales [java.sun.org/products/servlet/2.2, 7], dos de los cuales se muestran en la Tabla 11.1.

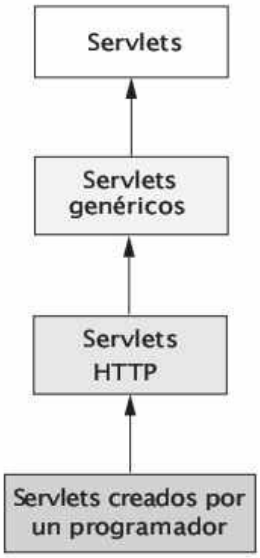


Figura 11.9. Jerarquía de clases de Java de un objeto servlet.

Tabla 11.1. Principales métodos de un servlet HTTP.

Método/Constructor	**Descripción**
void connect(InetAddress dirección, int puerto)	Crea una conexión lógica entre este *socket* y un *socket* en la dirección y puerto remotos.
void disconnect( )	Termina la conexión actual, si existe, de este *socket*.
Método	**Descripción**
protected void *doGet(HttpServletRequest req, HttpServletResponse resp)*	Llamado por el servidor (a través del método *service*) para permitir a los servlets manejar la petición *GET*.
protected void *doPost(HttpServletRequest req, HttpServletResponse resp)*	Llamado por el servidor (a través del método *service*) para permitir a los servlets manejar la petición *POST*.

Como se recordará del Capítulo 9, una petición HTTP puede invocar a un programa externo (tal como un *script* CGI) y pasarle parámetros utilizando los métodos *GET* o *POST*. De forma similar, un servlet se puede invocar a través de una petición *HTTP*, pasándole los parámetros en la cadena de interrogación tal y como fue descrito en el Capítulo 9. Revisión: si una petición especifica el método *GET*, la cadena de interrogación se concatena al URI y el servidor HTTP la sitúa en la variable de entorno *QUERY_STRING*; si la petición especifica el método *POST*, la cadena de interrogación se sitúa en el cuerpo de la petición y el servidor HTTP escribe la cadena en la entrada estándar del programa externo (un *script* CGI o un servlet).

Con los servlets, los parámetros pasados se encapsulan en la clase *HTTPServletRequest*. Cuando se solicita un servlet con *GET*, el servidor HTTP llama al método *doGet* del servlet; el servidor llamará al método *doPost* del servlet si éste se solicita con *POST*. Algunos métodos de la clase *HTTPServletRequest* facilitan la extracción de los parámetros de la cadena de interrogación, independientemente de si la petición se hizo especificando el método *GET* o *POST*.

La Figura 11.10 muestra el código fuente de un formulario web de ejemplo que invoca a un servlet. En este ejemplo, el formulario web especifica el método *POST*. Cuando se envía el formulario web, el servidor HTTP carga el código del servlet (almacenado con el nombre *formularioServlet*) en el contenedor de servlets. El contenedor de servlets inicia el servlet invocando su método *init()* y a continuación llama a su método *service()*. Como se muestra en la Figura 11.11, el método *service()* a su vez invoca al método *doPost()* en el servlet, ejecutando su código. Si el código genera una respuesta, se envía al navegador desde el servlet a través del servidor HTTP. Si el formulario especifica el método GET, se invocará al método *doGet()* en su lugar.

Figura 11.10. Código fuente de un formulario web de ejemplo que invoca a un servlet.

```
<html>
<head>
<title>Un formulario web que invoca a un servlet</title>
</head>
<body>
<H1>Este es un formulario sencillo que invoca a un servlet</H1>
<P>
Este es un ejemplo que muestra el uso de un formulario web procesado
con un servlet de Java.
<P>
<font color = red>
<HR>
<form method="post"
action="http://localhost:8080/ejemplos/servlet/formularioServlet">
<H2> Cuestionario: </H2>
Cuál es tu peso: <input name="peso"><P>
Cuál es tu pregunta: <input name="pregunta"><P>
Cuál es tu color favorito:
```

(continúa)

```
<select name="color">
<option selected>verde amarillento
</select>
<P>
Cuál es el peso de una golondrina: <input type="radio"
name="golondrina" value="africana" checked> Golondrina africana o
<input type="radio" name="golondrina" value="continental"> Golondrina
continental
<P>
Tienes algo que añadir
<textarea name="texto" rows=5 cols=60></textarea>
<P>
Presiona <input type="submit" value="aquí"> para enviar tu petición.
</form>
<hr>
</body>
</html>
```

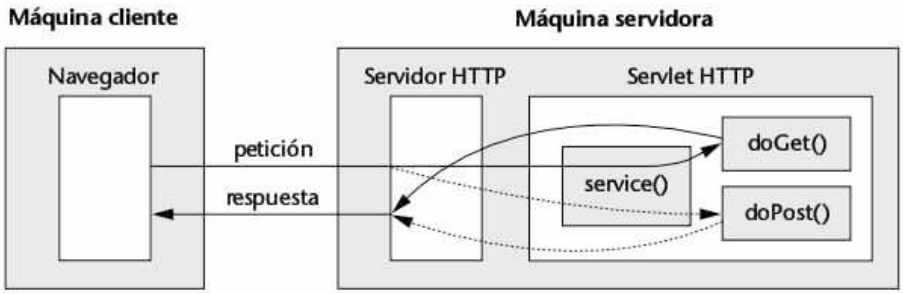


Figura 11.11. Los métodos *doPost* and *doGet* en un servlet HTTP.

Como se muestra en la Tabla 11.1, los métodos *doPost* y *doGet* reciben como parámetros referencias a dos objetos: primero, un objeto *HttpServletRequest* y, segundo, un objeto *HttpServletResponse*. Un objeto del primer tipo encapsula una petición HTTP, tal y como se vio en el Capítulo 9. Un objeto del segundo tipo encapsula una respuesta HTTP, como también se vio en el Capítulo 9. Dentro del servlet, la información de la petición se extrae utilizando los métodos apropiados del objeto *HttpServletRequest*, algunos de los cuales se muestran en la Tabla 11.2. Se procesa la información y se genera una respuesta utilizando los métodos apropiados del objeto *HttpServletResponse*. En la Tabla 11.3 se muestran algunos de los métodos principales de la clase *HttpServletResponse*.

La Figura 11.12 muestra el código de ejemplo de un servlet. El código hace uso de los métodos introducidos en las Tablas 11.2 y 11.3 para procesar la petición HTTP generada cuando se envía la página web de la Figura 11.10. Si se compara este código con el código del *script* CGI *formularioPost.c* (Figura 9.16) del Capítulo 9, se verá la misma lógica de aplicación y que la abstracción proporcionada por Java hace al código mucho más legible. Observe que este ejemplo sobrescribe el método *doPost*, ya que el formulario web que invoca al servlet especifica el método *POST*.

Tabla 11.2. Métodos seleccionados del objeto *HttpServletRequest*.

Método	Descripción
public *String getHeader(String name)*	Devuelve el valor de la cabecera de la petición especificada como una cadena.
public *String getMethod()*	Devuelve el nombre del método HTTP con el que fue realizada la petición. Por ejemplo, *GET*, *POST* o *PUT*.
public *String getQueryString()*	Devuelve la cadena de interrogación enviada con la petición.
public *String getParameter(String name)*	Devuelve el valor de un parámetro de la petición como una cadena, o *null* si el parámetro no existe.
public *Enumeration getParameterNames()*	Devuelve una enumeración de objetos cadena con el nombre de los parámetros contenidos en la petición. Si la petición no tiene parámetros el método devuelve una enumeración vacía.
public *String[ ]getParameterValues(String name)*	Devuelve un vector de objetos cadena con todos los valores que tiene el parámetro solicitado, o *null* si el parámetro no existe.

Tabla 11.3. Métodos seleccionados del objeto *HttpServletResponse*.

Método	Descripción
public void *setContentType(String type)*	Establece el tipo de contenido de la respuesta a ser enviada al cliente. Este método debe ser llamado antes de que se obtenga el objeto *PrintWriter* de la respuesta.
void *addHeader(String name, String value)*	Añade una línea de cabecera de respuesta con el nombre y valor dados.
void *sendError(int sc, String msg)*	Envía una respuesta de error al cliente utilizando el código de estado especificado y el mensaje de descripción.
void *setHeader(String name, String value)*	Establece una cabecera de respuesta con el nombre y valor dados.
void *setStatus(int sc)*	Establece el código de estado de esta respuesta.
public *ServletOutputStream getOutputStream()* throws *java.io.IOException*	Devuelve un objeto ServletOutputStream adecuado para escribir datos binarios en la respuesta. Para escribir el cuerpo se puede llamar o bien a este método o bien al método *getWriter()*, pero no a ambos.
public *PrintWriter getWriter()* throws *java.io.IOException*	Devuelve un objeto *PrintWriter* que puede enviar caracteres de texto al cliente. Para escribir el cuerpo se puede llamar o bien a este método o bien al método *getOutputStream()*, pero no a ambos.
public void *sendRedirect (String location)* throws *java.io.IOException*	Invoca la ejecución del servlet en el URL especificado en la cadena *location*. La redirección debe ser invocada antes de que la respuesta (en caso de existir) del servlet actual se escriba al objeto HTTPServletResponse. La ejecución del servlet actual continuará después de la redirección.

Figura 11.12. Código de ejemplo de un servlet que procesa un formulario.

```
1 // Un ejemplo de servlet que procesa un formulario web
2 // Author: M. Liu
```

(continúa)

```
 3
 4 import java.io.*;
 5 import javax.servlet.*;
 6 import javax.servlet.http.*;
 7
 8 public class servletFormulario extends HttpServlet {
 9
10   public void doPost (HttpServletRequest request,
11      HttpServletResponse res)
12      throws ServletException, IOException (
13
14      // Se debe establecer primero el tipo de
15      // contenido del cuerpo de la respuesta
16      res.setContentType("text/html");
17
18      // Stream de salida para el cuerpo de la respuesta
19      ServletOutputStream salida = res.getOutputStream();
20
21      salida.println("<html>");
22      salida.println("<head><title>Respueta del servlet" +
23         "</title></head>");
24      salida.println("<body>");
25      salida.println("<body bgcolor=\"beige\">");
26      salida.println("<P>Hola, </FONT><FONT FACE=" +
27         "\"Arial\" SIZE=5 COLOR=\"#ff0000\">");
28      // Recoger el valor del parámetro "nombre"
29      salida.println(request.getParameter("nombre") +
30         "</P></FONT>");
31      salida.println("<br>");
32      salida.println("<hr><br>");
33      // Recoger el valor del parámetro "pregunta"
34      salida.println("<UL><LI>Pregunta: " +
35      request.getParameter("pregunta") + "</LI>");
36      salida.println("<LI>Color: " +
37      request.getParameter("color") + "</LI>");
38      salida.println("<LI>Tipo de golondrina: " +
39      request.getParameter("golondrina") + "</LI>");
40      salida.println("<LI>Y dijiste: " +
41      request.getParameter("texto"));
42      salida.println("</UL><BR>");
43      salida.println("<HR>");
44      salida.println("<P><h3>Petición procesada por: " +
45         "<font color =\"#00AAFF\">");
46      salida.println(request.getRemoteHost() +
47         "</FONT></h3></P>");
48      salida.println("</body></html>");
49   }
50 } //fin class
```

Mantenimiento de la información de estado en la programación de servlets

De vuelta al Capítulo 9, cuando se introdujeron los *scripts* CGI, se estudiaron los métodos a través de los cuales la información de estado se puede pasar entre *scripts*. Ya que los scripts CGI son programas independientes, se requieren mecanismos especiales para permitir la compartición de información entre ellos. Se debe recordar que alguno de los mecanismos descritos en el Capítulo 9 eran campos ocultos de formulario y *cookies*.

Los servlets tienen una selección más amplia de mecanismos, algunos de los cuales se presentan a continuación, incluyendo el uso de (1) variables servlet, (2) campos ocultos de formulario, (3) *cookies* y (4) objetos de sesión.

Variables servlet

Como ya se mencionó, los servlets son persistentes: una sola instancia de un servlet, una vez cargado en el motor de servlets, se ejecuta hasta que el servlet se destruye. Por lo tanto, es posible almacenar información de estado en las variables del servlet. Sin embargo, no es común que los programadores utilicen este método para mantener la información de estado, por las siguientes razones:

Un programador no tiene control sobre el ciclo de vida de un servlet. Dependiendo de la implementación del servidor que interactúa con el motor de servlets, un servlet puede persistir durante un determinado tiempo, o puede persistir de forma indefinida hasta que se apaga el motor.

Debido a que en un determinado momento sólo está ejecutando una copia única del servlet, la información de estado almacenada en las variables del servlet será global a todos los clientes, haciendo difícil separar los datos de estado de las sesiones concurrentes.

La Figura 11.13 presenta el código fuente de un servlet que utiliza una variable para mantener un contador a lo largo del tiempo de vida del servlet. El contador se incrementa cada vez que se ejecuta el servlet. Se puede experimentar con este ejemplo ejecutando el servlet varias veces, de forma secuencial o concurrente, tal y como se pide hacer en uno de los ejercicios al final de este capítulo.

Figura 11.13. *Contador.java.*

```
1 // Un ejemplo que muestra el uso de variables
2 // servlet para almacenar información de estado.
3 // M. Liu
4
5 import java.io.*;
6 import javax.servlet.*;
7 import javax.servlet.http.*;
8
9 public class Contador extends HttpServlet {
10
11   int contador = 0;
12
13   public void doGet(HttpServletRequest request,
```

(continúa)

```
14      HttpServletResponse response)
15      throws ServletException, IOException {
16      response.setContentType("text/plain");
17      PrintWriter salida = response.getWriter();
18
19      contador++;
20
21      salida.println("Este servlet ha sido" +
22         " accedido " + contador+ " veces.");
23    } //fin doGet
24 } //fin class
```

Contador.java no es seguro con los hilos (*thread safe*) ya que la operación de incremento de la variable contador (línea 19) se puede interrumpir, por lo que una operación de incremento puede sobreescribir a otra. La Figura 11.14 es una versión segura que utiliza un método de sincronización para asegurar que sólo una invocación del servlet puede acceder e incrementar el contador.

Figura 11.14. *Contador2.java.*

```
 1 // Un servlet que mantiene un contador para
 2 // el número de veces que ha sido accedido
 3 // desde su carga.
 4 // M. Liu
 5
 6 import java.io.*;
 7 import javax.servlet.*;
 8 import javax.servlet.http.*;
 9
10 public class Contador2 extends HttpServlet {
11
12   public int contador = 0;
13
14   public void doGet(HttpServletRequest request,
15      HttpServletResponse response)
16      throws ServletException, IOException {
17       response.setContentType("text/plain");
18       PrintWriter salida = response.getWriter();
19       incremento(salida);
20   } //fin doGet
21
22   private synchronized void incremento(PrintWriter output){
23      output.println("Este servlet ha sido" +
24         " accedido " + counter + " veces.");
25      counter++;
26   } //fin incremento
27
28 } //fin class
```

Campos ocultos de formulario

El uso de campos ocultos de formulario, como se describió en el Capítulo 9, puede ser aplicado exactamente de la misma forma con servlets y con *scripts* CGI para pasar información de estado. Las sentencias HTML que contienen campos ocultos de formulario se pueden escribir a un *stream* de salida de un *PrintWriter* asociado con el *HTTPServletResponse* para almacenar los datos de estado a ser enviados al siguiente servlet en una cadena de interrogación. El siguiente fragmento de código muestra la salida de una línea en la respuesta HTTP que contiene un campo oculto de formulario cuyo nombre es ID y cuyo valor es el recibido en la variable *algúnValor*:

```
response.setContentType("text/plain");
PrintWriter salida = response.getWriter( );
salida.println("<INPUT TYPE=\"HIDDEN\" NAME=ID VALUE=" + algunValor);
```

Cookies

Las *cookies* también pueden aplicarse de la misma manera que en los *scripts* CGI. Java proporciona clases y métodos para facilitar su uso.

La clase, *HttpCookie*, representa las *cookies* que se vieron en el Capítulo 9. En la Tabla 11.4 se describen los principales métodos de esta clase.

Tabla 11.4. Principales métodos en la clase *Cookie*.

Método	Descripción
public *Cookie(String name, String value)*	Construye una *cookie* con el nombre y valor especificado.
public *String getDomain( )*	Devuelve el nombre de dominio de esta *cookie*.
public *int getMaxAge( )*	Devuelve la edad máxima de la *cookie*, especificada en segundos. Por defecto, -1 indica que la *cookie* será persistente hasta que se cierre el navegador.
public *String getName( )*	Devuelve el nombre de la *cookie*.
public *String getPath( )*	Devuelve la ruta en el servidor a la que el navegador devuelve la *cookie*.
public *String getValue( )*	Devuelve el valor de la *cookie*.
public void *setDomain(String pattern)*	Establece el atributo dominio de esta *cookie*.
public void *setMaxAge(int expiry)*	Establece el atributo expiración de esta *cookie* a un periodo de tiempo especificado en segundos.
public void *setPath(String uri)*	Establece el atributo ruta de esta *cookie*.
public void *setValue(String newValue)*	Asigna un valor a la *cookie*.

La clase *HttpRequest* proporciona un método, *getCookies*, para recoger las *cookies* enviadas con la petición HTTP, como se muestra en la Tabla 11.5.

Tabla 11.5. El método *getCookie* de la clase *HttpRequest*.

Método	Descripción
public *Cookie[ ] getCookies( )*	Devuelve un vector con todos los objetos *Cookie* enviados por el cliente en la petición.

Las Figuras 11.5 a 11.18 presentan una serie de códigos que muestran la implementación básica de un carrito de la compra utilizando programación con servlets.

Figura 11.15. El formulario web *carrito.html*.

```
<HTML>
<HEAD>
<TITLE>Frutas Online</TITLE>
</HEAD>
<BODY bgcolor=#CCFFCC>
<CENTER><H1>Tenemos los siguientes artículos</H1></CENTER>
<HR>
<FORM ACTION="http://localhost:8080/ejemplos/servlet/Carrito" METHOD="POST">
<TABLE CELLSPACING="5" CELLPADDING="5">
<TR>
  <TD ALIGN="center"><B>Añadir al carrito</B></TD>
  <TD ALIGN="center"></TD>
  <TD ALIGN="center"></TD>
</TR>
<TR>
  <TD ALIGN="center"><INPUT TYPE="Checkbox"
    NAME="objeto_a" VALUE="manzana $1"></TD>
  <TD ALIGN="left">manzana</TD>
</TR>
<TR>
  <TD ALIGN="center"><INPUT TYPE="Checkbox"
    NAME="objeto_b" VALUE="naranja $2"></TD>
  <TD ALIGN="left">naranja</TD>
</TR>
<TR>
  <TD ALIGN="center"><INPUT TYPE="Checkbox"
    NAME="objeto_c" VALUE="pera $3"></TD>
  <TD ALIGN="left">pera</TD>
</TR>
</TABLE>
<HR><BR>
<CENTER>
Presiona
<INPUT TYPE="Submit" NAME="Cart1_submit" VALUE="Enviar">
para enviar tu pedido.
</CENTER>
</FORM>
</BODY>
</HTML>
```

Figura 11.16. El servlet *Carrito*.

```
1 import javax.servlet.*;
2 // Código fuente del servlet Carrito, invocado cuando
```

(continúa)

```
// se envía el formulario web carrito.html
// M. Liu

import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class Carrito extends HttpServlet
{
  public void doPost(HttpServletRequest request,
     HttpServletResponse response)
     throws ServletException, IOException
  {
     response.setContentType("text/html");
     ServletOutputStream salida = response.getOutputStream();
     salida.println("<html>");
     salida.println("<head><title>Respuesta del servlet" +
        "</title></head>");
     salida.println("<body>");
     Cookie c;

     /* Recoger datos del formulario */
     Enumeration claves;
     String name, value, prefix;
     claves = request.getParameterNames();
     while (claves.hasMoreElements())
     {
        nombre = (String)claves.nextElement();
        prefijo = nombre.substring(0,4);

        if (prefijo.equals("objeto"))
        // Este chequeo es necesario para eliminar
        // campos de entrada que no son objetos.
        {
           /* Recoger el valor del parámetro */
           valor = request.getParameter(nombre);
           /* Crear una cookie */
           salida.println("<H4>Establecer cookie: " + nombre +
              " " + valor + "</H4>");
           c = new Cookie(nombre, valor);

           /* Establacer expiración en un día */
           /* c.setMaxAge(1*24*60*60); */
           response.addCookie(c);
        }//fin if
     } //fin while
     salida.println("</body></html>");

```

(continúa)

```
51      /* Realizar una redirección para enviar las cookies e
52      invocar otro servlet que muestre los objetos
53      del carrito de la compra */
54      response.sendRedirect("Carrito2");
55
56
57    } //fin doPost
58  } //fin class
```

Figura 11.17. El servlet *Carrito2*.

```
 1 // Servlet que muestra el contenido del carrito de la compra
 2 // (datos almacenados por el servlet Carrito)
 3 // M. Liu, basado en varias fuentes
 4
 5 import javax.servlet.*;
 6 import javax.servlet.http.*;
 7 import java.io.*;
 8 import java.util.*;
 9
10 public class Carrito2 extends HttpServlet
11 {
12   /* Visualizar objetos del carrito */
13   public void doGet(HttpServletRequest request,
14      HttpServletResponse response)
15      throws ServletException, IOException {
16
17      response.setContentType("text/html");
18      ServletOutputStream salida = response.getOutputStream();
19      salida.println("<html>");
20      salida.println("<head><title>Respuesta del servlet" +
21         "</title></head>");
22      salida.println("<body>");
23      salida.println("<body bgcolor=\"beige\">");
24      salida.println("Contenido de tu carrito de la compra<UL>");
25
26      /* Recoger las cookies */
27      Cookie cookies[];
28
29      cookies = request.getCookies();
30      if (cookies != null)
31      {
32        for (int i = 0; i < cookies.length; i++)
33        {
34          /* Nota: es importante identificar las cookies
35          por su nombre, ya que puede haber otras cookies
36          en uso de este sitio */
37          if (cookies[i].getName().startsWith("objeto"))
```

(continúa)

```
38          {
39             salida.println("<LI>" + cookies[i].getName() +
40                "\t" + cookies[i].getValue());
41          }
42        } // fin for
43     } // fin if
44
45     salida.println("</UL>");
46     salida.println("<HR>");
47     salida.println("</body></html>");
48
49   } // fin doGet
50
51 } // fin Carrito
```

Figura 11.18. *Carrito2.html.*

```
<HTML>
<CENTER>
<FORM ACTION="Carrito2" METHOD="GET">
<h1>Visualizar los objetos actuales en el carrito</h1>
<p>
Presiona
<INPUT TYPE="Submit" NAME="Carrito2_enviar" VALUE="Enviar"> para ver
el contenido de tu carrito de la compra.
</CENTER>
</FORM>
</BODY>
</HTML>
```

La Figura 11.15 es el código fuente del formulario web *Carrito.html*. Cuando se muestra el formulario presenta tres objetos disponibles para su selección.

Cuando se envía el formulario, se inicia el servlet *Carrito* (Figura 11.6). En el ejemplo, se asume que el servlet ejecuta en la máquina local. Sin embargo, el nombre del dominio y el número de puerto se pueden reemplazar con los de un servidor alternativo. El nombre de los parámetros de la petición pasados por el formulario web se obtienen utilizando el método *getParameterNames* (línea 27), para a continuación obtener el valor de los parámetros con el método *getParameter* (línea 38). Para cada parámetro cuyo nombre comience con el prefijo adecuado ("objeto", en nuestro caso), el nombre y el valor se utilizan para crear una nueva *cookie* (línea 42), que se añade a la respuesta utilizando el método *addCookie* (línea 46). Si se selecciona naranja, por ejemplo, la *cookie* generada tendrá el nombre "objeto_b", y el valor "naranja $2". Hay sentencias en el código (líneas 44-45) que ajustan las *cookies* para durar un día, aunque estas sentencias están comentadas.

Se llama al método *sendRedirect* del objeto *HttpServletResponse* para invocar al siguiente servlet, *Carrito2*. El servlet *Carrito2* genera dinámicamente una página web que muestra el contenido del carrito de la compra.

En *Carrito2.java* (Figura 11.17), las *cookies* se recogen una por una (líneas 29-43) utilizando el método *getCookies* del objeto *HttpServletRequest*. Sólo se procesan las *cookies* con el prefijo adecuado (objeto). Se supone que el valor de cada *cookie* contiene el nombre y el precio del objeto seleccionado. En la página web generada dinámicamente se incluye una descripción de cada objeto.

La Figura 11.18 muestra el formulario web *Carrito2.html* que, cuando se envía, invoca al servlet *Carrito2* directamente para permitir a un usuario visualizar los contenidos del carrito de la compra durante una sesión.

Debido a que un servlet es persistente, uno se puede preguntar, ¿el uso de las *cookies*, tal y como se muestra en el ejemplo del carrito de la compra, puede presentar compartición de información de sesión, causando que los contenidos del carrito de la compra de un usuario se le muestren a otro? Se considera que el cliente A y el cliente B, que están usando de forma concurrente *carrito.html* desde computadores diferentes. Cada petición HTTP de los clientes desencadenará la invocación de la misma instancia del servlet *Carrito*, que genera las *cookies* para las selecciones de los usuarios. La petición del cliente A hace que las *cookies* se generen basándose en los parámetros de la petición que envía A, mientras que la petición del cliente B hace que las *cookies* se generen basándose en los parámetros de la petición que envía B. Las *cookies* de A se almacenan en el navegador de la computadora A. De forma análoga, las *cookies* de B se almacenan en el navegador de la computadora B. Las *cookies* son enviadas por cada navegador en las siguientes peticiones HTTP (a la máquina con el servlet *Carrito*) que realiza su usuario, por lo que no hay confusión en los datos de los diferentes carritos de la compra.

Sin embargo, la situación es diferente si las sesiones paralelas se realizan desde el mismo computador. Hay un ejercicio al final del capítulo en el cual se puede investigar y experimentar con este escenario.

Objeto *Session*

El API de Servlets proporciona un mecanismo especial para mantener información de estado específica de una sesión particular de un cliente HTTP. El mecanismo se conoce como objetos *session*. Un objeto *session* implementa la interfaz *HttpSession*. Un servlet puede crear este objeto y después utilizarlo como un repositorio de datos de estado a lo largo de la sesión del cliente. Para diferenciar sesiones diferentes, cada objeto de sesión debe tener un identificador único. El identificador se asigna de forma automática por el contenedor de servlets y es transparente al usuario. A lo largo de la sesión del cliente, el identificador de sesión se pasa entre el servidor y el cliente utilizando una *cookie* o algún otro mecanismo. Una vez que se ha creado el objeto sesión, un servlet puede depositar en él uno o más objetos que contienen información de estado. Cada objeto añadido al objeto sesión se especifica con un nombre. Más adelante, el objeto puede ser accedido por otro servlet, o incluso por el mismo servlet, invocado en la misma sesión de usuario.

El objeto *session* persistirá durante un intervalo de inactividad que puede ser establecido por código, o por un valor por defecto que depende de la implementación. Cuando el intervalo finalice, el contenedor de servlets invalidará el objeto *session* por lo que su contenido ya no podrá ser accedido. La invalidación de un objeto *session* también se puede iniciar por programa; es una buena práctica hacerlo en el código al final de una sesión.

La Tabla 11.6 muestra los principales métodos que se pueden invocar con *HTTPSession*. El método *setAttribute* se utiliza para añadir datos de sesión a un objeto *session*, mientras que el método *getAttribute* u opcionalmente el método *getAttributeNames* se puede utilizar para recoger datos de sesión de un objeto *session* existente. El método *setMaxInactiveInterval* permite establecer el intervalo de tiempo de inactividad para un objeto *session*, mientras que el método *invalidate* invalidará el objeto *session*.

Tabla 11.6. Principales métodos de la interfaz HTTPSession.

Método	Descripción
public *Object getAttribute(String name)* throws *java.lang.IllegalStateException*	Devuelve el objeto de esta sesión con el nombre especificado, o *null* en caso de no existir dicho nombre.
public *Enumeration getAttributeNames( )* throws *java.lang.IllegalStateException*	Devuelve una enumeración de objetos cadena con el nombre de todos los objetos de esta sesión.
public void *setAttribute (String name, Object value)* throws *java.lang.IllegalStateException*	Enlaza un objeto con esta sesión, utilizando el nombre especificado. Si ya existe un objeto con el mismo nombre en la sesión, se reemplaza dicho objeto.
public void *invalidate( )* throws *java.lang.IllegalStateException*	Invalida esta sesión y a continuación elimina el enlace con los objetos de la misma.
public *int getMaxInactiveInterval( )*	Devuelve el máximo tiempo del intervalo, un entero que especifica el número de segundos que el contenedor de servlets mantendrá esta sesión abierta entre accesos del cliente. Después de este intervalo, el contenedor de servlets invalidará la sesión. El tiempo máximo del intervalo se puede establecer con el método *setMaxInactiveInterval*. Un tiempo negativo indica que la sesión nunca finaliza.
public void *setMaxInactiveInterval (int interval)*	Especifica el tiempo, en segundos, entre llamadas de clientes antes de que el contenedor de servlets invalide la sesión. Un tiempo negativo indica que la sesión no debe finalizar.
public *String getId( )*	Devuelve una cadena que contiene el identificador único de esta sesión. El identificador lo asigna el contenedor de servlets y es dependiente de la implementación.

Un objeto *session* se puede crear y posteriormente recuperar utilizando el método *getSession* de la clase *HttpRequest*, como se muestra en la Tabla 11.7.

Las Figuras 11.19 y 11.20 revisan el código fuente de los servlets *Carrito* y *Carrito2*, previamente presentados en las Figuras 11.16 y 11.17. En este conjunto de códigos de ejemplo se utiliza un objeto *session* (en lugar de *cookies*) para mantener los objetos del carrito de la compra.

Tabla 11.7. El método *getSession* de la clase *HttpRequest*.

Método	Descripción
public *HttpSession getSession(boolean create)*	Devuelve el *HttpSession* actual asociado con esta petición o si no hay sesión actual y el parámetro *create* contiene *true*, devuelve una nueva sesión. Si *create* vale *false* y la petición no tiene un *HttpSession* válido, el método devuelve *null*.

Figura 11.19. El servlet *Carrito* utilizando el objeto *session*.

```
1  import javax.servlet.*;
2  // Código fuente del servlet Carrito invocado cuando
3  // se envía el formulario web carrito.html
4  // M. Liu
5
6  import javax.servlet.http.*;
7  import java.io.*;
8  import java.util.*;
9
10 public class Carrito extends HttpServlet
11 {
12   public void doPost(HttpServletRequest request,
13      HttpServletResponse response)
14      throws ServletException, IOException {
15
16
17      /* Recupera el objeto session o crea
18          uno nuevo */
19      HttpSession session = request.getSession(true);
20
21      Integer contadorObjetos =
22         (Integer) session.getAttribute("contadorObjetos");
23      Vector objetos =
24         (Vector) session.getValue("objetos");
25      /* Si todavía no se ha seleccionado ningún objeto,
26          poner el contador a cero y crear un vector. */
27      if (contadorObjetos == null) {
28         contadorObjetos = new Integer(0);
29         objetos = new Vector();
30      }
31
32      // Se recomienda obtener el objeto session
33      // antes de escribir cualquier salida.
34      PrintWriter salida = response.getWriter();
35      response.setContentType("text/html");
36
37      /* Recoger los parámetros enviados */
38      Enumeration claves;
39      String nombre, valor, prefijo;
40      int contador = itemCount.intValue( );
41      claves = request.getParameterNames();
42      while (claves.hasMoreElements())
43      {
44         nombre = (String)claves.nextElement();
45         prefijo = name.substring(0,4);
46         salida.println("name=" + nombre + " prefix=" +
47            prefijo);
48         if (prefijo.equals("objeto"))
```

(continúa)

```
49        {
50          // añadir el objeto a la lista de objetos
51          valor = request.getParameter(nombre);
52          salida.println("añadiendo el objeto:" +
53            valor + " contador=" + contador);
54          objetos.add(valor);
55          contador++;
56        }//fin if
57      } //fin while
58      contadorObjetos = new Integer(contador);
59      session.putValue("contadorObjetos", contadorObjetos);
60      if (objetos != null)
61        session.setAttribute("objetos", objetos);
62
63      /* Realizar una redirección para invocar otro
64          servlet que muestre los objetos del carrito
65          de la compra */
66
67      response.sendRedirect
68        ("http://localhost:8080/ejemplos/servlet/Carrito2");
69    } //fin doPost
70
71 } //fin class
```

Figura 11.20. El servlet *Carrito2* utilizando el objeto *session*.

```
 1 // Servlet para visualizar qué hay en el carrito de la compra
 2 // (datos almacenados por el servlet Carrito)
 3 // M. Liu, basado en varias fuentes
 4
 5 import javax.servlet.*;
 6 import javax.servlet.http.*;
 7 import java.io.*;
 8 import java.util.*;
 9
10 public class Carrito2 extends HttpServlet {
11
12   /* Visualizar objetos en el carrito de la compra */
13   public void doGet(HttpServletRequest request,
14     HttpServletResponse response)
15     throws ServletException, IOException {
16
17
18     // Recoger el objeto session, si existe
19     HttpSession session = request.getSession(false);
20     Integer contadorObjetos;
```

(continúa)

```
21      Vector objetos = null;
22      if (session == null)
23      {
24        // no se ha creado ningún objeto session
25        contadorObjetos = new Integer(0);
26      }
27      else
28      {
29        contadorObjetos =
30          (Integer) session.getValue("contadorObjetos");
31        objetos =
32          (Vector) session.getValue("objetos");
33      }
34      // Se recomienda obtener el objeto session
35      // antes de escribir cualquier salida.
36      PrintWriter salida = response.getWriter( );
37      response.setContentType("text/html");
38
39      salida.println("<html>");
40      salida.println("<head><title>Respuesta del servlet " +
41        "</title></head>");
42      salida.println("<body>");
43      salida.println("<body bgcolor=\"beige\">");
44      salida.println
45        ("Contenido de tu carrito de la compra " +
46        " utilizando el objeto session<UL>");
47
48      int contador = contadorObjetos.intValue( );
49      /* Recoger los objetos del objeto session */
50      for (int i = 0; i < contador; i++)
51        salida.println("<LI>" + objetos.get(i));
52
53      salida.println("</UL>");
54      salida.println("<HR>");
55      salida.println("</body></html>");
56
57    } // fin doGet
58
59  } // fin Carrito2
```

Aquí concluye la introducción a los servlets. El material que se ha presentado proporciona una visión general de la tecnología. Al final del capítulo encontrarás ejercicios que te permitirán practicar con los servlets.

Una tecnología muy cercana a los servlets es JSP (*Java Server Pages*, Páginas Java de Servidor) [java.sun.com, 25], que permiten crear páginas web con código servlet embebido. El uso de JSP puede simplificar significativamente el código para generar páginas web de forma dinámica. Los lectores interesados pueden consultar las referencias [java.sun.com, 25] y [javaboutique.internet.com, 26].

11.3. SERVICIOS WEB

En el capítulo 3 se vio el paradigma de servicios de red para computación distribuida. Utilizando este paradigma, una aplicación ejecuta alguna de sus tareas haciendo uso de servicios ya implementados disponibles en la red. En estas aplicaciones los servicios de red se pueden integrar dinámicamente, según se necesite.

Este paradigma ha sido extendido a **servicios web**, una tecnología que ha surgido recientemente. Los servios web proporcionan servicios de red transportados por HTTP, y están siendo propuestos como una nueva forma de construir aplicaciones de red desde componentes distribuidos (servicios) independientes del lenguaje y de la plataforma. Los protocolos y las API de la tecnología están evolucionando todavía. En esta sección se verá uno de estos protocolos, el **SOAP (*Simple Object Access Protocol*, protocolo simple de acceso a objetos)** [w3.org, 9], y una API de ejemplo, el **API SOAP de Apache**.

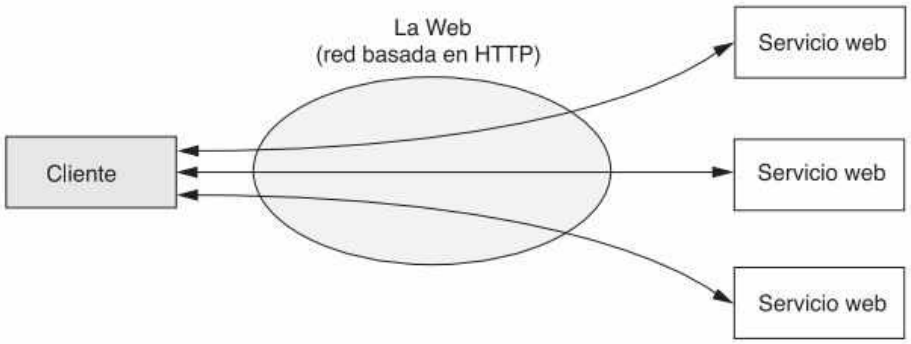


Figura 11.21. El modelo conceptual de servicios web.

La Figura 11.21 muestra el modelo conceptual de servicios web. Un servicio web se proporciona por un objeto servidor y se accede por un cliente. El servidor y el cliente intercambian mensajes de acuerdo a protocolos estándares desarrollados para los servicios web. La Figura 11.22 representa la jerarquía del protocolo. Lógicamente, el servidor y el cliente intercambian mensajes en la **capa de aplicación**. Físicamente, se requiere una serie de protocolos para dar soporte al intercambio de mensajes. Un **protocolo de descubrimiento de servicios** permite al servicio ser registrado y localizado. Las funcionalidades proporcionadas en la **capa de descripción del servicio** permiten que un servicio sea descrito en el directorio. La **capa de mensajería** proporciona los mecanismos para la intercomunicación de procesos, incluyendo funcionalidades de empaquetamiento (*marshaling*) de datos. La **capa de transporte** envía los mensajes. Finalmente, la **capa de red** representa la jerarquía del protocolo de red para la transmisión física y el encaminamiento de paquetes.

La Figura 11.23 muestra los protocolos predominantes utilizados para servicios web. Para el descubrimiento de servicios, el protocolo estándar se denomina **UDDI (*Universal Description, Discovery and Integration*, descripción, descubrimiento e integración universales)** [uddi.org, 21]. La sintaxis y semántica para describir servicios se especifica utilizando **WSDL (*Web Service Description Language*, lenguaje de descripción de servicios web)** [w3.org, 22]. En la capa de mensajería, se intercambian mensajes codificados en **XML** siguiendo el protocolo **SOAP**

Jabber es un protocolo abierto, basado en XML y utilizado para mensajería instantánea y asistencia [jabber.org, 23].

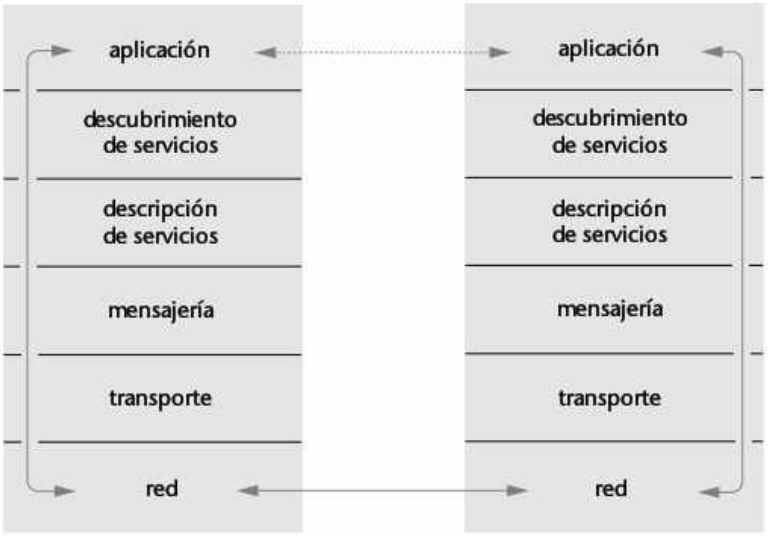


Figura 11.22. Jerarquía de protocolos de servicios web.

[mole.informatik.uni-stuttgart.de, 9; sun.com, 10; Edwards, 11]. En la capa de transporte, **HTTP** sirve para transmitir peticiones y respuestas, se utilizan **SMTP** o **Jabber** para transmitir mensajes y se utiliza **TCP** para transmitir los datos. Finalmente, el protocolo del nivel de red es **IP**.

La Figura 11.24 muestra la arquitectura software de un servicio web. El escuchador del servicio en la máquina servidora recoge las peticiones de servicios transmitidas sobre la red. Cuando se recibe una petición, se reenvía al *proxy* del servicio. El *proxy* invoca a la lógica de aplicación del objeto servicio y transmite el valor devuelto al llamante.

Aunque todos los protocolos mencionados en la Figura 11.23 son de interés, SOAP es particularmente importante para este estudio. El resto de este capítulo se centrará en el protocolo SOAP y sus aplicaciones.

aplicación	
descubrimiento de servicios	UDDI (Universal Description, Discovery, and Integration, Descripción, Descubrimiento e Integración Universales).
descripción de servicios	WSDL (Web Service Description Language, Lenguaje de Descripción de Servicios Web).
mensajería	XML, SOAP (Simple Object Access Protocol, Protocolo Simple de Acceso a Objetos).
transporte	TCP, HTTP, SMTP, Jabber.
red	IP.

Figura 11.23. Protocolos de servicios web.

Figura 11.24. Arquitectura software de servicios web.

11.4. SOAP

En los pasados capítulos se estudió el paradigma de objetos distribuidos, incluyendo dos protocolos/arquitecturas que soportan el paradigma: Java RMI y CORBA. SOAP es un protocolo que incorpora el paradigma de los objetos distribuidos y los protocolos de Internet. En concreto, es un protocolo que extiende HTTP para permitir acceso a objetos distribuidos que representan servicios web.

La Figura 11.25 muestra el modelo para el protocolo simple de acceso a objetos. Un cliente web manda una petición HTTP, cuyo cuerpo contiene un mensaje con formato SOAP que representa la llamada a un método de un objeto de servicio. La petición se transmite a un servidor web, que la reenvía, junto con los parámetros de la llamada al método. A continuación se invoca el método. Una vez completado, el valor devuelto por el método se envía al servidor web y a continuación se transmite al cliente web en el cuerpo de la respuesta HTTP.

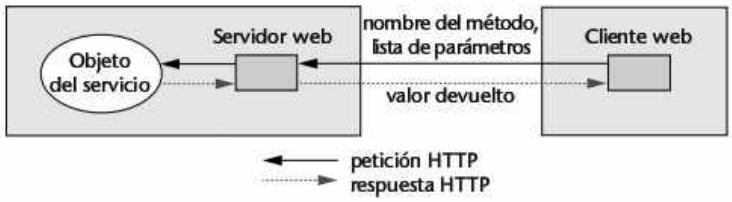


Figura 11.25. El modelo SOAP.

Por razones de interoperabilidad, el mensaje SOAP se codifica en XML. Cada mensaje SOAP tiene un formato sencillo, como se representa en la Figura 11.26.

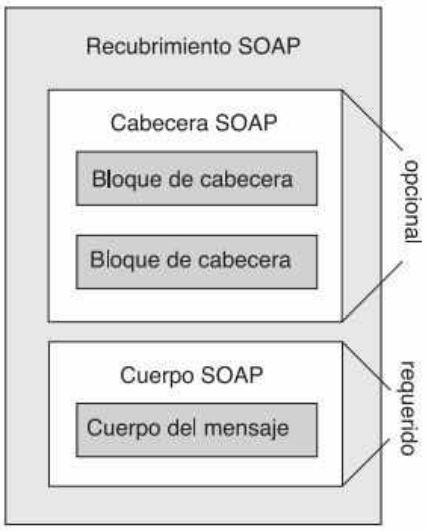


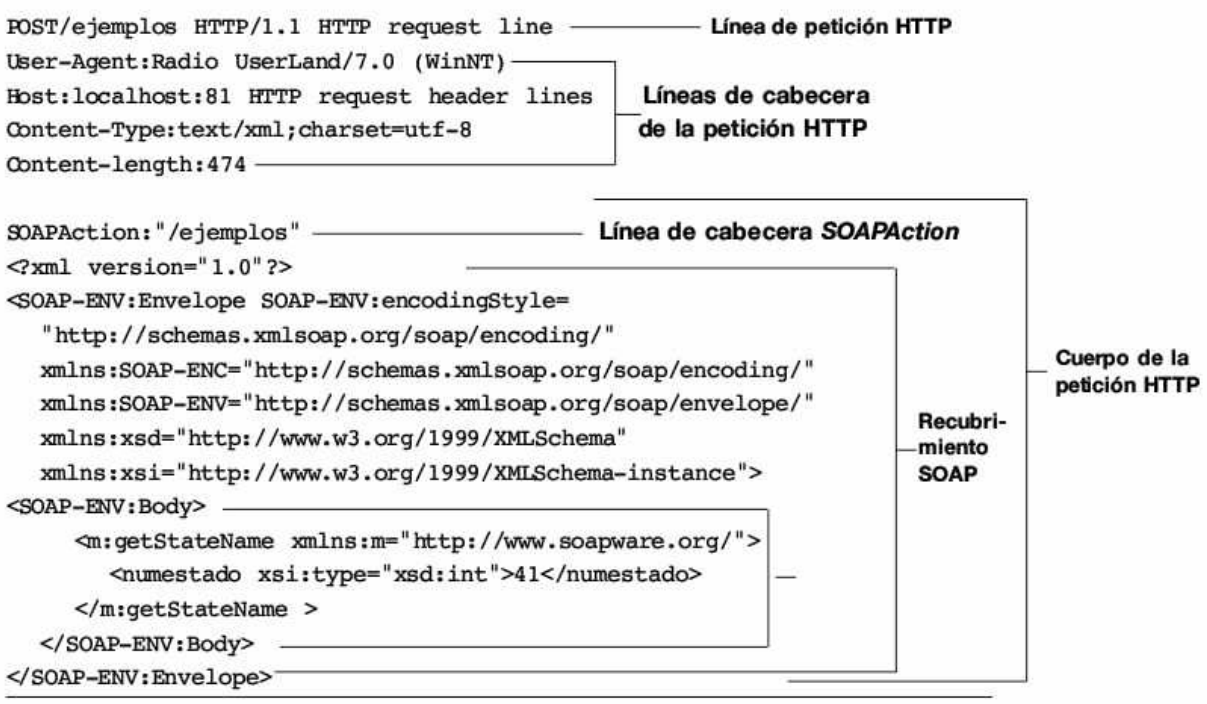
Figura 11.26. Esquema de un mensaje de petición SOAP.

Cada mensaje SOAP se transporta en una petición o respuesta HTTP, tal como se explicará en las siguientes secciones.

Una petición SOAP

La Figura 11.27 muestra la sintaxis de una petición HTTP con una petición SOAP. Los elementos de la petición HTTP se describen en los siguientes párrafos.

Figura 11.27. Una petición HTTP con una petición SOAP (Fuente: [soapware.org, 11]).



Líneas de cabecera de la petición HTTP

El URI en la primera línea de la cabecera de petición HTTP debe especificar el objeto al que se dirige la llamada a método remoto. En el ejemplo 11.27, el objeto remoto es /ejemplos. Las líneas de cabecera *User-Agent* y *Host* se deben especificar.

El *Content Type* debe especificarse como *text/xml*. El *charset* es una especificación de la representación de caracteres aceptada; por defecto es *US-ASCII*. Otras especificaciones *charset* que se aceptan son *UTF-8* y *UTF-16*, que son esquemas de codificación Unicode.

El *Content Length*, si está especificado, debe ser la longitud en bytes del cuerpo de la petición.

La línea de cabecera *SOAPAction* especifica el objeto remoto al que se dirige la petición. La interpretación de este elemento de cabecera depende del programa. En la mayor parte de los casos, el URI (especificado en la primera línea de cabecera) y el *SOAPAction* tendrán el mismo valor.

Figura 11.28. Una petición SOAP
(Fuente: [soapware.org, 11]).

```
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle=
   "http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
   xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
   xmlns:xsd="http://www.w3.org/1999/XMLSchema"
   xmlns:xsi ="http://www.w3.org/1999/XMLSchema-instance">
   <SOAP-ENV:Body>
                          nombre del método                    nombre del servidor
      <m:obtenerNombreEstado xmlns:m="http://www.soapware.org/">
         <numestado xsi:type="xsd:int">41</numestado>
      </m:obtenerNombreEstado>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Parámetro *numestado* de tipo *int* y valor 41

El cuerpo de la petición

La Figura 11.28 destaca la sintaxis del cuerpo de la petición, que está codificada en XML. Hay dos partes en el cuerpo: el recubrimiento SOAP y el cuerpo SOAP.

El recubrimiento SOAP. Este recubrimiento SOAP se define con el elemento <SOAP-ENV:Envelope>. El elemento tiene un conjunto de atributos requeridos que especifican el esquema de codificación y el estilo del recubrimiento.

El cuerpo SOAP. El cuerpo SOAP está definido con la etiqueta <SOAP-ENV:Body>. El cuerpo SOAP contiene un solo elemento, que representa la llamada al método. Junto con el elemento se especifican el nombre del método (*obtenerNombreEstado* en el ejemplo), el nombre de cada parámetro (*numestados* en el ejemplo), el valor de cada parámetro (41 en el ejemplo) y el tipo de datos de cada parámetro.

Tipos de datos

SOAP tiene un rico conjunto de tipos de datos independientes del lenguaje, que están basados en los tipos de datos de los esquemas XML. La descripción completa de los tipos de datos está fuera del alcance de este libro; los lectores interesados pueden ver [w3.org, 24].

La Tabla 11.8 resume un subconjunto de los principales tipos de datos escalares soportados por un subconjunto de SOAP 1.1.

Los tipos de datos no escalares, incluyendo objetos, estructuras, matrices, vectores y enumerados, también están soportados en SOAP. Alguno de estos tipos se comentan en las siguientes secciones.

Estructuras. Un valor puede ser una estructura, que se especifica con un elemento XML que contiene subelementos. Las estructuras pueden estar anidadas y pueden contener cualquier otro tipo de datos, incluyendo una matriz.

A continuación se muestra un ejemplo de una estructura de dos elementos:

Una estructura es un tipo de estructura de datos en C y su derivado C++. También se denomina registro.

```
<param>
   <limiteInferior xsi:type="xsd:int">18</limiteInferior>
   <limiteSuperior xsi:type="xsd:int">139</limiteSuperior>
</param>
```

Tabla 11.8. Tipos de datos escalares del esquema XML (Fuente: [soapware.org, 11]).

Valor del atributo	Tipo	Ejemplo
xsd:int	Entero con signo de 32-bit	-12
xsd:boolean	Valor booleano, 1 ó 0	1
xsd:string	Cadena de caracteres	Hola Mundo
xsd:float o xsd:double	Número de coma flotante con signo	–12,214
xsd:timeInstant	Fecha/Hora	2001-03-27T00:00:01-08:00
SOAP-ENC:base64	Binario codificado en Base64	eW91IGNhbid0IHJlYWQgdGhpcyE=

Los nombres de la estructura son significativos; el orden de los elementos no.

Matrices. Un valor puede ser una matriz, que se especifica como un elemento XML con un atributo *SOAP-ENC:arrayType* cuyos valores comienzan con *ur-type*[<número de elementos de la matriz>].

El siguiente es un ejemplo de una matriz de cuatro elementos:

```
<param SOAP-ENC:arrayType="xsd:ur-type[4]"
xsi:type="SOAP-ENC:Array">
<item xsi:type="xsd:int">12</item>
<item xsi:type="xsd:string">Egipto</item>
<item xsi:type="xsd:boolean">0</item>
<item xsi:type="xsd:int">-31</item>
<param>
```

El orden de los elementos de la matriz es significativo; el nombre de los elementos no.

Objetos. Se puede transmitir un objeto en la petición/respuesta SOAP si el proveedor del servicio define y registra el tipo del objeto como un subtipo, y las dos partes proporcionan un serializador y deserializador apropiado. A continuación el nombre del subtipo se declara como el atributo *xsi:type* del parámetro.

Una respuesta SOAP

La Figura 11.29 muestra una respuesta HTTP que contiene una respuesta SOAP exitosa. La cabecera HTTP tiene el formato habitual. Obsérvese que el *content type* es *text/xml*.

Figura 11.29. Una respuesta HTTP que contiene una respuesta SOAP con éxito (Fuente: [soapware.org, 11]).

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 499
Content-Type: text/xml; charset=utf-8
Date: Wed, 28 Mar 2001 05:05:04 GMT
Server: UserLand Frontier/7.0-WinNT

<?xml version="1.0"?>
```

(continúa)

```
<SOAP-ENV:Envelope SOAPENV:
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAPENC="
http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
  <SOAP-ENV:Body>
    <m:obtenerNombreEstadoResponse xmlns:m="http://www.soapware.org/">
      <Result xsi:type="xsd:string">Dakota del Sur</Result>
    </m:obtenerNombreEstadoResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

La Figura 11.30 destaca la respuesta SOAP contenida en la respuesta HTTP. Como con la petición SOAP, la respuesta está compuesta de dos partes: el recubrimiento y el cuerpo.

La sintaxis del recubrimiento es la misma que con la petición. La sintaxis del cuerpo también es análoga a la de la petición. El único elemento contenido en *<SOAP-ENV:Body>* tiene un nombre que coincide con el nombre del método que ha sido llamado, con la palabra *Response* añadida al final del nombre del método (*obtenerNombre Estado* en el ejemplo). El tipo de datos (string) y el valor (Dakota del Sur) del valor devuelto está contenido en el subelemento *Result*.

Figura 11.30. Una respuesta SOAP (Fuente: [soapware.org, 11]).

```
<?xml version="1.0"?>
<SOAP-ENV:Envelope SOAP-ENV:
  encodingStyle ="http://schemas.xmlsoap.org /soap/encoding/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org /soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org /soap/envelope/"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">

  <SOAP-ENV:Body>     nombre del método          nombre del servidor
    <m:obtenerNombreEstadoResponse xmlns:m="http://www.soapware.org/">
      <Result xsi:type="xsd:string">Dakota del Sur</Result>   valor
    </m:obtenerNombreEstadoResponse >                          devuelto
  </SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

Una llamada a método SOAP puede fallar, quizá debido a errores en la especificación del nombre del método o en los parámetros. Cuando una llamada a método no puede ser completada de forma satisfactoria, la respuesta HTTP (*véase* Figura 11.31) contiene un cuerpo SOAP que define un código y una cadena de error. El código de error (*SOAP-ENV:Client* en el ejemplo) identifica el error, mientras que la cadena de error proporciona una descripción del mismo.

Figura 11.31. Una respuesta HTTP que contiene una llamada a método SOAP fallida (Fuente: [soapware.org, 11]).

```
HTTP/1.1 500 Server Error
Connection: close
Content-Length: 511
Content-Type: text/xml; charset=utf-8
Date: Wed, 28 Mar 2001 05:06:32 GMT
Server: UserLand Frontier/7.0-WinNT

<?xml version="1.0"?>
<SOAP-ENV:Envelope SOAPENV:
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAPENV="
http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
  <SOAP-ENV:Body>
    <SOAP-ENV:Fault>
      <faultcode>SOAP-ENV:Client</faultcode>
      <faultstring>No se puede llamar a obtenerNombreEstado
       porque hay demasiados parámetros.</faultstring>
    </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Apache SOAP

Como se puede observar en las descripciones precedentes, escribir código para generar directamente la sintaxis XML para las peticiones y respuestas SOAP puede ser tedioso y propenso a errores. Por consiguiente, han aparecido numerosas API para SOAP que proporcionan la abstracción necesaria para facilitar la programación que compete a las peticiones/respuestas SOAP. Entre los conjuntos de herramientas disponibles está *Apache SOAP* y *Apache Axis* para la programación en Java; y *SOAP::Lite* para programar en Perl. *Microsoft.NET* también soporta SOAP.

Apache Axis es el paso siguiente de Apache SOAP [xml.apache.org, 20]. Es una reimplementación de Apache SOAP, presumiblemente con tiempos de ejecución mejorados.

El resto de esta sección proporcionará una visión general de la API Apache SOAP [xmethods.com, 12; 106.ibm.com, 13; xml.apache.org, 14; xml.apache.org, 15] para la construcción de clientes y de objetos de servicios. Primero se verán algunas clases importantes de Apache SOAP y luego se verán algunos códigos de ejemplo.

La clase *RPCMessage*

El API Apache SOAP proporciona una clase denominada *RPCMessage* que encapsula una petición SOAP. Un objeto de esta clase contiene las siguientes instancias de campos: *targetObjectURI*, *methodName*, *params* y *header*, que representan los diversos campos de una petición SOAP. La clase tiene métodos para establecer y recuperar los valores de cada una de estos campos.

La clase *Call*

La clase *Call* es una subclase de la clase *RPCMessage* y representa una llamada a método remoto. Se puede crear un objeto de esta clase en un programa cliente SOAP,

que puede llamar al método *invoke* para realizar la llamada a método remoto. La Tabla 11.9 presenta la especificación del método *invoke*.

Tabla 11.9. El método invoke de la clase Call.

Método	Descripción
public *Response invoke* *(URL url, String SOAPActionURI)* throws *SOAPException*	Invoca esta llamada en el URL especificado.

La clase *Parameter*

Un objeto de la clase *Parameter* representa tanto parámetros como los valores devueltos por una llamada a método. En el programa cliente, se crea un objeto de esta clase por cada parámetro de método remoto, invocando al constructor de esta clase, cuya especificación se muestra en la Tabla 11.10. En el servidor, se construye un objeto de esta clase para el valor devuelto.

Tabla 11.10. El constructor de la clase *Parameter*.

Constructor	Descripción
public *Parameter(String name,* *Class type,* *Object value,* *String encodingStyleURI)*	Crea un objeto *Parameter* con el nombre, tipo de datos, valor y estilo de codificación dados.

La clase *Response*

Un objeto de la clase *Response* representa la respuesta de una llamada a método. Tanto el cliente como el servidor utilizan objetos *Response* para representar el resultado de una invocación a método. El servidor crea la respuesta. El cliente extrae información de la respuesta. En la Tabla 11.11 se presentan los principales métodos de la clase *Response*.

Tabla 11.11. Métodos principales de la clase *Response*.

Método	Descripción
public *Parameter getReturnValue( )*	Este método se invoca desde los clientes para recibir el valor devuelto de una llamada a método.
public boolean *generatedFault( )*	Este método se invoca desde los clientes para ver si una llamada a método ha generado un error.
public *Fault getFault()*	Este método puede ser invocado por un cliente para analizar el error que causó al fallo de la llamada a método.

La clase *Fault*

Un objeto de la clase *Fault* representa el contenido y la semántica del elemento *<SOAP-ENV:Fault>*, y lo devuelve el método *getFault* ejecutado por el cliente. En

el código de ejemplo, el método *getFaultString* (Tabla 11.12) se invoca desde un cliente para recibir la descripción del error que causó el fallo de la llamada a método.

Tabla 11.12. El método *getFaultString* de la clase *Fault*.

Método	Descripción
public *String getFaultString( )*	Devuelve una cadena que contiene una descripción breve del error que causó el fallo de la llamada a método.

Servicios web ya implementados

La idea de los servicios web es permitir a los desarrolladores de software hacer uso de servicios ya implementados. Como se puede imaginar, estos servicios irán desde servicios lucrativos proporcionados por proveedores comerciales (tales como validadores de tarjetas de crédito) hasta servicios gratuitos proporcionados por la comunidad de usuarios (tales como juegos en red o traducciones).

Actualmente hay disponibles un cierto número de servicios web para aquellos que estén interesados en experimentar con la tecnología. En [xmethods.net, 16] se proporciona una lista de estos servicios, muchos de los cuales están accesibles utilizando el API Apache SOAP.

En la Figura 11.32 se muestra un ejemplo de servicio descrito en [xmethods.net, 16].

Figura 11.32. Descripción de un ejemplo de servicio web ya implementado.

```
XMethods ID 8
Service Owner:xmethods.net
Contact Email:support@xmethods.net
Service Home Page:
Description:Current temperature in a given U.S. zipcode region.
SOAP Implementation:Apache SOAP
```

Con cada servicio listado en [xmethods.net, 16] se proporciona un perfil como en mostrado en la Figura 11.33.

Figura 11.33. Descripción de un ejemplo de servicio web ya implementado.

Method Name	`getTemp`
Endpoint URL	`http://services.xmethods.net:80/soap/servlet/rpcrouter`
SOAPAction	
Method Namespace	`URI urn:xmethods-Temperature`
Input Parameters	`zipcode string`
Output Parameters	`return float`

El perfil contiene información para invocar al servicio, incluyendo el URL del objeto de servicio (http://services.xmethods.net:80/soap/servlet/rpcrouter, en el ejemplo), el nombre del método (*getTemp*, en este caso) o métodos proporcionados por el servicio y los parámetros (un *string*) y valor devuelto (un *float*) del método.

Invocación de un servicio web utilizando Apache SOAP

La figura 11.34 muestra el código de ejemplo de un programa que invoca a un servicio SOAP.

En las líneas 1.6, el programa cliente importa los diversos paquetes requeridos por Apache/SOAP. Para prepararse para la invocación de un método proporcionado por el servicio web, el programa instancia el objeto *Call* y asigna valores a sus campos utilizando los métodos del objeto (líneas 26-34):

```
Call call = new Call(); // preparar la invocación del servicio
call.setEncodingStyleURI( Constants.NS_URI_SOAP_ENC );
call.setTargetObjectURI("urn:xmethods-Temperature" );
call.setMethodName( "getTemp" );
```

El método *setTargetObjectURI* se especifica con un URI que identifica el objeto de servicio SOAP en la máquina remota. En el ejemplo, el URL es el listado en *Method Namespace* de la Figura 11.33.

Para preparar los argumentos para la invocación, se instancia un objeto de la clase *Parameter* para cada parámetro. Cada objeto *Parameter* se inicializa con el nombre, el tipo de datos, el valor y el estilo de codificación del argumento (que por defecto es *null*), como en las líneas 37-38:

```
Parameter aParam = new Parameter("zipcode", String.class, zipcode, null);
```

El parámetro nombre (zipcode) y el tipo de datos (String.class) se lista en la descripción del servicio mostrada en la Figura 11.33. Por simplicidad, el valor del parámetro de entrada se escribe directamente en la línea 14 del código de ejemplo, pero podría ser obtenido en tiempo de ejecución.

La lista de parámetros se recoge en un vector (líneas 39-40):

```
Vector params = new Vector ();
params.addElement (aParam);
```

Y a continuación se asocia el vector al objeto *Call* (línea 41):

```
call.setParams(params);
```

Para realizar la llamada a método del servicio web, se especifica el método *invoke()* del objeto *Call* en el URL (línea 44):

```
Response response = call.invoke(url, "");
```

donde *url* se refiere a un objeto de la clase Java URL, instanciado con el servicio web URL (líneas 12-13):

```
URL url = new URL("http://localhost:8080/soap/servlet/rpcrouter");
```

El URL se puede encontrar en la descripción del servicio web (listado como *Endpoint URL* en la Figura 11.33). La invocación del método debe ser chequeada en busca de errores (líneas 47-53), y, si no hay ninguno, el valor devuelto puede ser utilizado en el procesado del resto del programa (líneas 57-59).

Figura 11.34. Un ejemplo de cliente de servicio web.

```
1 import java.io.*;
2 import java.net.*;
3 import java.util.*;
```

(continúa)

```
4 import org.apache.soap.util.xml.*;
5 import org.apache.soap.*;
6 import org.apache.soap.rpc.*;
7
8 public class TempClient{
9
10  public static void main(String[] args) {
11   try {
12     URL url= new URL(
13        http://services.xmethods.com:80/soap/servlet/rpcrouter;
14     String zipcode= "93420";
15     float temp = getTemp(url, zipcode);
16     System.out.println("La temperature es " + temp);
17   }
18   catch (Exception e) {
19     e.printStackTrace();
20   }
21  } //fin main
22
23  public static float getTemp (URL url, String zipcode)
24   throws Exception {
25
26   Call call = new Call ();
27
28   // Especificación de la codificación SOAP
29   String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
30   call.setEncodingStyleURI(encodingStyleURI);
31
32   // Establecer los parámetros de localización de servicios
33   call.setTargetObjectURI ("urn:xmethods-Temperature");
34   call.setMethodName ("getTemp");
35
36   // Crear vector de parámetros
37   Parameter aParam =
38     new Parameter("zipcode", String.class, zipcode, null);
39   Vector params = new Vector ();
40   params.addElement (aParam);
41   call.setParams (params);
42
43   // Invocar al servicio
44   Response resp = call.invoke (url,"");
45
46   // Procesar la respuesta
47   if (resp.generatedFault ()) {
48     // La llamada no fue satisfactoria
49     Fault f = resp.getFault(); // hubo un error
50     System.err.println( "Fault= " + f.getFaultCode() +
51        ", " + f.getFaultString() );
```

(continúa)

```
52      throw new Exception(f.getFaultString());
53    }
54    else {
55      // La llamada fue satisfactoria
56      // Extraer el valor devuelto y devolver el resultado
57      Parameter result = resp.getReturnValue ();
58      Float readOut=(Float) result.getValue();
59      return readOut.floatValue();
60    }
61   } //fin getTemp
62
63 } //fin class
```

Implementación de un servicio web utilizando Apache SOAP

Un servicio web se define utilizando una interfaz de Java, que contiene las declaraciones de los métodos proporcionados por el servicio. La Figura 11.35 muestra la interfaz *ITemp* para un servicio web de ejemplo, *Temp*, que proporciona el método *getTemp* llamado por nuestro cliente en ejemplo en la Figura 11.34.

Figura 11.35. Una interfaz Java para un servicio web sencillo.

```
1 // Un ejemplo de interfaz de objeto de servicio SOAP.
2 // Este método acepta una cadena con el código postal
3 // y devuelve la temperatura del área.
4
5 public interface ITemp
6 {
7  float getTemp(String zipCode);
8
9 } //fin interface
```

La interfaz del servicio web se implementa como una clase de Java. La Figura 11.36 muestra una definición de ejemplo de la clase *Temp*, que implementa la interfaz *ITemp*. Por simplicidad se devuelve un valor fijo.

Figura 11.36. Implementación de un servicio web sencillo.

```
1 // Un ejemplo de implementación de objeto de servicio SOAP
2
3 public class Temp implements ITemp
4 {
5  public float getTemp(String zipCode)
6  {
7     System.out.println("Temperatura para el
8       código postal " + zipCode + " solicitado.");
9     return 74.5F; // devuelve una constante por simplicidad
```

(continúa)

```
10
11    } //fin getTemp
12
13 } //fin class
```

Un servicio SOAP necesita ser instalado y configurado en la máquina servidora. Este procedimiento depende de la implementación. Para Apache SOAP, se puede encontrar información en las referencias [xmethods.com, 12] y [xml.apache.org,14].

RESUMEN

Este capítulo ha presentado tres protocolos y mecanismos de aplicaciones de Internet.

Applets

- Un applet es una clase Java cuyo código se descarga desde el servidor web y ejecuta en el entorno del navegador en la máquina cliente.
- Un navegador solicita un applet cuando escanea una página web y encuentra una clase especificada en la etiqueta *APPLET*.
- Por razones de seguridad, la ejecución de un applet está expuesta a restricciones: por definición, un applet no puede acceder a ficheros del sistema de ficheros de la máquina cliente o hacer conexiones de red a otra máquina que no sea de la que proviene.

Servlets

- Un servlet es una extensión de un servidor petición-respuesta. Un servlet HTTP es, como un *script* CGI, una extensión de un servidor HTTP.
- Un servlet HTTP es una clase Java cuyo código se carga en un contenedor de servlets en la máquina servidora y que se inicia por el servidor HTTP en respuesta a una petición HTTP del servlet.
- Al contrario que los *script* CGI, un servlet HTTP es persistente: un *script* CGI se recarga cada vez que un cliente lo solicita, mientras que una sola instancia del servlet ejecutará al menos mientras haya peticiones que lo soliciten.
- Para la programación de servlets: la clase *HTTPServletRequest* encapsula una petición HTTP, mientras que la clase *HTTPServletResponse* encapsula la respuesta.
- Para el mantenimiento de la información de estado, un servlet puede utilizar los mecanismos disponibles para los *scripts* CGI, tales como campos ocultos o *cookies*. Además, la información de estado se puede mantener con los siguientes mecanismos:
  - — Las variables servlets pueden almacenar datos globales.
  - — Se puede crear y mantener, en la máquina donde está ejecutando el servlet, un objeto *session* que contenga objetos de datos de sesión.

Protocolo simple de acceso a objeto

- SOAP es un protocolo que hace uso de las peticiones y respuestas HTTP para efectuar llamadas a métodos remotos en servicios web.
- Una llamada a método SOAP se embebe en una petición HTTP y se codifica en XML; el valor devuelto se embebe en una respuesta HTTP y se codifica con XML.
- Existen varias API de SOAP disponibles para programación de servicios web y llamadas a métodos. En este capítulo se ha introducido el API Apache.

EJERCICIOS

Nota: alguno de estos ejercicios requiere un servidor web que soporte applets y servlets. Para los ejercicios de programación de servlets, es posible descargar e instalar el Apache Tomcat Server [apache.org, 18] o el servidor Java JSWDK.

Ejercicios de Applets

1. Instale *HolaMundo.html* y *HolaMundo.class* (compilado de *HolaMundo.java*) en un servidor web en el que tenga acceso. Si el servidor es una máquina UNIX, asegúrese de poner los permisos de lectura y ejecución universales a estos ficheros. Utilice un navegador para acceder a *HolaMundo.html*. Describa la salida y explique los eventos que suceden para llevar a este resultado.
2. Modifique *NetConnectApplet.java* para reemplazar el nombre de la máquina *www.alpha.edu* por el servidor web que está utilizando para estos ejercicios. Instale *miApplet.html* y *NetConnectApplet.class* (compilado de *NetConnectApplet.java*) en el servidor web. Utilice su navegador para abrir *miApplet.html*. ¿Realizó el applet la conexión *socket* de forma satisfactoria?
3. Vuelva a modificar *NetConnectApplet.java* para reemplazar el nombre de la máquina *www.beta.edu* por otro servidor web. Compile y guarde la nueva versión en el servidor web que utilizó en el ejercicio 2. Utilice su navegador para abrir *miApplet.html*. ¿Realizó el applet ambas conexiones *socket* de forma satisfactoria? Describa y explique los resultados.
4. ¿Cuáles son las dos restricciones de seguridad de los applets comentadas en este capítulo? Para cada de una de las restricciones, explique por qué es necesaria.

Ejercicios genéricos de servlets

1. Rellene la siguiente tabla para comparar los servlets con los applets.

	Applet	Servlet
Lenguaje de programación. ¿Cuál es la clase base de la que debe heredar la clase applet/servlet? Soporte software necesario para ejecutar el programa (en el servidor).		

(continúa)

	Applet	Servlet
Soporte software necesario para ejecutar el programa (en el cliente).		
¿Dónde se ejecuta el programa, en el cliente o en el servidor?		
Esboza un código de la petición HTTP para invocar al programa.		
¿Cómo se carga el programa para la ejecución?		
Muestra un buen uso de este tipo de programa en una aplicación web.		
Restricciones, si hay, en este tipo de programa (por razones de seguridad).		
Liste otras diferencias.		

2. Rellene la siguiente tabla para comparar y contrastar los servlets con los *scripts* CGI.

	Servlets	Scripts CGI
Lenguaje(s) de programación.		
Soporte software requerido para ejecutar el programa.		
Esboza un código de la petición HTTP para invocar al programa.		
¿Cómo se carga el programa para la ejecución?		
¿Es persistente el programa? (Es decir, ¿se ejecuta múltiples veces la misma instancia?)		
Nombre los mecanismos que pueden ser utilizados para mantener los datos de sesión.		
Liste otras diferencias.		

Ejercicios de programación con servlets

Los ficheros de programa de estos ejercicios se pueden encontrar en el directorio *servlets\simple* de los ejemplos de este capítulo.

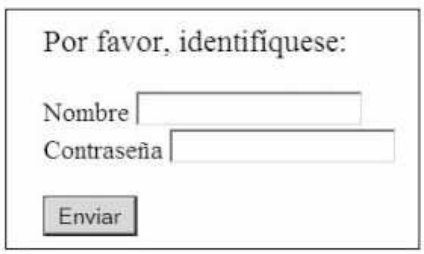
1. Inicie el servidor Apache Tomcat [jakarta.apache.org, 6] si no está iniciado en su computador. Pruebe el servlet de ejemplo que viene con el servidor, introduciendo la siguiente dirección:

   ```
   http://<nombre servidor web>:8080
   ```

   Elija el enlace *Servlet Examples*. Se verá un conjunto de servlets de ejemplo; para cada uno de ellos se puede ver el código o se puede ejecutar. Mire el código fuente de cada uno de ellos y ejecútelos. Fíjese en la ruta URL que se muestra en el navegador cuando se ejecuta el servlet: esta es la ruta URL que deberá especificar en sus servlets. En el servidor Apache Tomcat, la ruta por defecto es http://localhost:8080/examples/servlet/. Por ejemplo, abrir http://localhost:8080/examples/servlet/HelloWord ejecutará el servlet *HelloWorldExample*.

2. Cree un directorio en su PC y copie los ficheros del directorio *servlets\simple*. Compile *HelloWorld.java* y *Counter2.java*. Después cópielos en el directorio de las clases servlet. (En el servidor Apache Tomcat, el directorio por defecto de las clases servlet es TOMCAT_HOME\webapps\examples\WEB-INF\classes).
   a. Utilice el navegador para ejecutar el servlet *HelloWorld*.
   b. Para verificar que los servlet se pueden ejecutar en una máquina servidora que no es el *localhost*, utilice el navegador de su sistema para ejecutar el servlet *HelloWorld* en el sistema de su profesor o de sus compañeros.
   c. Compile el fichero *Counter1.java*, e instale el fichero con la clase resultante *Counter1.class* en el directorio de servlets. A continuación acceda a él utilizando la ruta URL del servlet. Refresque el navegador repetidamente para ejecutar el servlet *Counter1* varias veces. Describa y explique el valor del contador mostrado por el navegador en las ejecuciones.
   d. Abra otro navegador y acceda al servlet *Counter1*. Describa y explique el valor del contador mostrado por el navegador.
   e. Cierre las ventanas del navegador. A continuación abra un nuevo navegador y acceda al servlet *Counter1*. Describa y explique el valor del contador mostrado por el navegador.
   f. Pare el servidor y reinícielo. A continuación refresque la pantalla del navegador de forma que el servlet *Counter1* se reejecute. Describa y explique el valor del contador mostrado por el navegador.
   g. Basándose en los experimentos con el *Counter1*, describa el tiempo de vida de un servlet en el entorno del servidor Jakarta Tomcat.
   h. Modifique *Counter1.java* para que el valor del contador se incremente en 2 unidades cada vez. Recompile e instale el fichero clase. (Nota: en algunos servidores, tales como el JSWDK, se debe apagar el servidor y volverlo a iniciar antes de que el nuevo servlet surta efecto). Demuestre el cambio realizado.
   j. Compile *GetForm.java* y *PostForm.java*. Instale los ficheros de clase resultantes en el directorio de los ficheros de clase del servidor.
   k. Abra las páginas *GetForm.html* y *PostForm.html*. ¿Ejecutan los servlet correctamente? Compare las salidas, incluyendo el URL mostrado en el navegador, con las generadas utilizando *scripts* CGI.
3. Escriba un formulario web HTML y su servlet para realizar un control de acceso sencillo en su página web.

   La página mostrada debería tener el siguiente aspecto:



Si los datos de la cuenta (por ejemplo: nombre Pepe y contraseña «12345») se introducen correctamente, tu página web debe ser mostrada; en caso contrario, se mostrará un mensaje de «Datos inválidos». Instale el servidor y pruébelo.

(Nota: para acceder a su página web después de que la contraseña haya sido verificada, el servlet necesitará escribir la siguiente línea:

```
<html><head><META HTTP-EQUIV="REFRESH" CONTENT="0;URL=<url de su
página web>"></head></html>
```

No se olvide de que un carácter con comillas dobles en una cadena necesita ser precedido de una barra inversa.

Realice el código fuente de su servlet.

Utilización de *cookies* con los servlets

Nota: asegúrese de abrir una sesión nueva del navegador cuando ejecute o reejecute cualquiera de los siguientes experimentos, ya que las *cookies* generadas en una sesión persistirán a lo largo de la misma.

1. Copie los ficheros del directorio *servlets\cookies* a un directorio.
2. Comente la sentencia que redirige el servlet *Cart2* al final de *Cart.java*.
3. Compile e instale el servlet *Cart*. Abra *Cart.html* y seleccione «naranja». Verifique la salida mostrada al enviar el formulario.
4. Deshaga los cambios del paso 2 para que el servlet *Cart* se redirija a *Cart2*. Compile e instale el servlet *Cart2*.

   Abra *Cart.html*. Seleccione un elemento. Verifique que el elemento está en el carrito de la compra. Abra *Cart2.html* y verifique que el carrito de la compra se muestra correctamente.

   Utilice el botón de regreso de su navegador para volver a *Cart.html*. Seleccione otro objeto. Verifique de nuevo su carrito de la compra.

   Utilice de nuevo el botón de regreso de su navegador para volver a *Cart.html*. Seleccione el último elemento. Verifique su carrito de la compra de nuevo. Describa y explique sus comentarios.
5. Descomente la llamada al método *setMaxAge* en *Cart.java* para que las *cookies* generadas no sean transitorias. Compile y reinstale el servlet *Cart*. Abra dos sesiones de navegador diferentes y acceda a *Cart.html* en cada una de ellas. Seleccione un objeto en la primera sesión, envíelo, y seleccione un objeto diferente en la segunda sesión. Envíe la segunda página.

   Describa y explique sus observaciones.

   Para librarse de las *cookies* no transitorias: modifique el método *setMaxAge* y ponga su argumento a 0. Compile y reinstale el servlet *Cart*. Abra una nueva sesión del navegador y acceda a *Cart.html*. Seleccione los tres objetos. Cuando envíe el formulario debería ver el carrito vacío.
6. Quite el comentario del método *setMaxAge* en *Cart.java* para que las *cookies* generadas no sean transitorias. Modifique el formulario web y servlets para (i) permitir al usuario elegir la cantidad de cada objeto a comprar (por ejemplo, 4 naranjas), y (ii) incluya en la salida el precio total de los objetos del carrito de la compra.

Forma de actuar sugerida:

a. Modifique *Cart.html* para añadir un campo de entrada para la cantidad de cada objeto, como sigue:

```
<TR>
<TD ALIGN="center"><INPUT TYPE="Checkbox"
NAME="item_a" VALUE="apple $1"></TD>
<TD ALIGN="left">apple</TD>
<TD ALIGN="left">How many?
<input name="quantity_item_a"></TD>
</TR>
```

Acceda a la página modificada para asegurarse de que la salida es correcta.

b. Segundo, modifique *Cart.java* de forma que la cantidad introducida en cada objeto seleccionado sea adjuntada en el comienzo del valor de la *cookie*. Por ejemplo: 10 apple $1, si se han seleccionado 10 apples.

Descomente de forma temporal la última sentencia (la redirección) en *Cart.java* de forma que pueda verificar la cadena generada para cada valor de la *cookie*.

c. Modifique *Cart2.java*. Para cada valor de la *cookie*, obtenga la cantidad y el precio. Mantenga una suma acumulativa del precio total. Imprima la salida cuando se hayan procesado todas las *cookies*.

El código recomendado para recoger el valor de cada *cookie* es el siguiente:

```
try {
   value = cookies[i].getValue();
   out.println("<LI>" + value);
   st = new StringTokenizer(value);
   quantity = Integer.parseInt(st.nextToken( ));
   st.nextToken("$\n");
   price = Integer.parseInt(st.nextToken());
   // añadir código para imprimir
   // valor de la cookie y procesar
   // la cantidad de elementos y el
   // valor de los elementos
}
catch (Exception ex)
// excepción formato número
{ } // sencillamente se salta este elemento
```

Utilización de un objeto *session* con servlets

Nota: asegúrese de abrir una sesión nueva del navegador cuando ejecute o reejecute cualquiera de los siguientes experimentos, ya que las objeto *session* generados en una sesión persistirán a lo largo de la misma.

1. Copie los ficheros del directorio *servlets\session* a un directorio.
2. Comente la sentencia que redirige al servlet *Cart2* al final de *Cart.java*.
3. Compile e instale el servlet *Cart*. Abra *Cart.html* y seleccione naranja. Verifique la salida mostrada al enviar el formulario. Averigüe si se puede verificar la salida mirando el código de *Cart.java*.

4. Deshaga los cambios del paso 2 para que el servlet *Cart* se redirija a *Cart2*. Compile e instale el servlet *Cart2*.

   Abra *Cart.html*. Seleccione un elemento. Verifique que el elemento está en el carrito de la compra. Abra *Cart2.html* y verifique que el carrito de la compra se muestra correctamente.

   Utilice el botón de regreso de su navegador para volver a *Cart.html*. Seleccione otro objeto. Verifique de nuevo su carrito de la compra.

   Utilice de nuevo el botón de regreso de su navegador para volver a *Cart.html*. Seleccione el último elemento. Verifique su carrito de la compra de nuevo. Describa y explique sus comentarios. ¿Qué encuentra diferente respecto al apartado 4 del anterior conjunto de problemas, cuando se utilizaban las *cookies* para almacenar el contenido del carrito de la compra?

   Abra dos sesiones de navegador diferentes y acceda a *Cart.html* en cada una de ellas. Seleccione un objeto en la primera sesión, envíelo, y seleccione un objeto diferente en la segunda sesión y envíelo.

5. Describa y explique sus observaciones. ¿Qué encuentra diferente respecto al apartado 5 del anterior conjunto de problemas, cuando se utilizaban las *cookies* para almacenar el contenido del carrito de la compra?

6. Modifique el formulario web y servlets para (i) permitir al usuario elegir la cantidad de cada objeto a comprar (por ejemplo, 4 naranjas), y (ii) incluya en la salida el precio total de los objetos del carrito de la compra.

   Forma de actuar sugerida:

   a. Modifique *Cart.html* para añadir un campo de entrada para la cantidad de cada objeto, como sigue:

```
<TR>
<TD ALIGN="center"><INPUT TYPE="Checkbox"
NAME="item_a" VALUE="apple $1"></TD>
<TD ALIGN="left">apple</TD>
<TD ALIGN="left">How many?
<input name="quantity_item_a"></TD>
</TR>
```

   Acceda a la página modificada para asegurarse de que la salida es correcta.

   b. Cree una clase denominada *Item*, que tiene los siguientes datos de instancia: nombre del elemento, precio y cantidad. *Item.Java* se proporciona en el mismo directorio. Compile y mueva *Item.class* al directorio donde residen los ficheros con las clases servlet.

   c. Modifique *Cart.java* para que cada selección se recoja en una referencia a un objeto *Item* que sea añadido al vector de elementos. Comente temporalmente la última sentencia (la redirección) en *Cart.java* para que pueda ver los valores de los objetos que se están añadiendo al objeto *session*.

   Aquí está el código sugerido:

```
while (keys.hasMoreElements())
{
  name = (String)keys.nextElement();
  prefix = name.substring(0,4);
  out.println("name=" + name + "prefix=" + prefix);
```

(continúa)

```
if (prefix.equals("item"))
{
  quantityName = "quantity_" + name;
  out.println("quantityName = " + quantityName);
  quantity = request.getParameter(quantityName);
  // añadir elemento a la lista de elementos
  value = request.getParameter(name);
  st = new StringTokenizer(value);
  name = st.nextToken("$\n");
  price = st.nextToken();
  out.println("adding name=" + name + " price=" + price +
    "quantity=" + quantity);
  items.add(new Item(name,Integer.parseInt(price),
    Integer.parseInt(quantity)));
  count++;
 }//fin if
}//fin while
```

Compile e instale el servlet *Cart*.

Cuando esté satisfecho con la salida, descomente el método de redirección al final de *Cart.java*.

d. Modifique *Cart2.java*. Para cada objeto *Item* recibido del objeto de sesión, extraiga el precio de la unidad y la cantidad. Mantenga una suma del precio total hasta el momento. Imprima la suma total cuando todos los objetos *Item* hayan sido procesados.

Aquí está el código sugerido:

```
for (int i = 0; i < count; i++) {
nextItem = (Item)items.get(i);
out.println("<LI>" + i + " " + nextItem.toString( ));
total += nextItem.getPrice( ) * nextItem.getQuantity( );
}
```

Compile e instale el servlet *Cart2*. Abra *Cart.html* y verifique la salida.

7. Considere la utilización de objetos *session* para almacenar datos de sesión. ¿Por qué no es necesario utilizar exclusión mutua para proteger la recuperación y actualización del objeto *session*?

Ejercicios SOAP

1. Acceda a la página http://www.xmethods.net/ [xmethods.net, 16]. Elija uno de los servicios con estilo RPC y escriba un cliente Java que utilice el API Apache SOAP para llamar al servicio y mostrar el resultado devuelto. Entregue el código fuente.
2. Considere la siguiente petición HTTP [soapware.org, 11]:

```
POST /examples HTTP/1.1
User-Agent: Radio UserLand/7.0 (WinNT)
Host: localhost:81
Content-Type: text/xml; charset=utf-8
Content-length: 474
```

(continúa)

```
SOAPAction: "/examples"
<?xml version="1.0"?>
<SOAP-ENV:Envelope SOAPENV:
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAPENV="
http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
  <SOAP-ENV:Body>
    <m:getStateName xmlns:m="http://www.soapware.org/">
      <statenum xsi:type="xsd:int">41</statenum>
    </m:getStateName>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Identifique los componentes SOAP en la petición.

3. Considere la siguiente respuesta HTTP:

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 499
Content-Type: text/xml; charset=utf-8
Date: Wed, 28 Mar 2001 05:05:04 GMT
Server: UserLand Frontier/7.0-WinNT
<?xml version="1.0"?>
<SOAP-ENV:Envelope SOAPENV:
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:SOAPENV="
http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
<SOAP-ENV:Body>
  <m:getStateNameResponse xmlns:m="http://www.soapware.org/">
    <Result xsi:type="xsd:string">South Dakota</Result>
  </m:getStateNameResponse>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Identifique los componentes SOAP en la petición.

7. Escriba un servicio SOAP (interfaz y clase) que proporcione dos métodos: (i) *add*, que acepta dos enteros y devuelve la suma, y (ii) *subtract*, que reciba dos enteros y devuelve la diferencias.

   Escriba un programa cliente que invoque a los dos métodos y procese la salida.

   Si es posible, instale y configure el servicio creado y ejecute su programa cliente para testear el servicio.

REFERENCIAS

1. Overview of Applets, *http://java.sun.com/docs/books/tutorial/applet/overview/index.html*
2. Applets, *http://java.sun.com/applets/?frontpage-spotlight*
3. Java(TM) Boutique, Free Java Applets, Games, Programming Tutorials, and Downloads—Applet Categories, *http://javaboutique.internet.com/*
4. Java Servlet Technology Implementations & Specifications Archive, *http://java.sun.com/products/servlet/archive.html*
5. Java(TM) Servlet Technology—Implementations & Specifications, *http://java.sun.com/products/servlet/download.html*
6. The Apache Tomcat Server, *http://jakarta.apache.org/tomcat/index.html*
7. *javax.servlet.http.HttpServlet* class specification, *http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/http/HttpServlet.html*
8. Neil Gunton. "SOAP: Simplifying Distributed Development." *Dr. Dobb's Journal*, September 2001.
9. Simple Object Access Protocol (SOAP) 1.1, *http://www.w3.org/TR/SOAP/*
10. SOAP Tutorial, *http://www.w3schools.com/soap/default.asp*
11. Dave Winer and Jake Savin. SoapWare.Org: A Busy Developer's Guide to SOAP 1.1, *http://www.soapware.org/bdg*. UserLand Software, April 2, 2001.
12. A Quick-Start Guide for Installing Apache SOAP, *http://www.xmethods.com/gettingstarted/apache.html*
13. developerWorks: Web services: The Web services (r)evolution: Part 2, Hello world, Web service-style, *http://www-106.ibm.com/developerworks/library/ws-peer2/*
14. Apache SOAP Documentation: User's Guide, *http://xml.apache.org/soap/docs/index.html*
15. Apache SOAP V2.2 Documentation, *http://xml.apache.org/soap/docs/index.html*
16. XMethods—Web Service Listings, *http://www.xmethods.net/*
17. *java.sun.com*. Frequently Asked Questions—Applet Security, *http://java.sun.com/sfaq/*
18. Welcome!–The Apache Software Foundation, *http://apache.org/*
19. Jason Hunter and William Crawford. *Java Servlet Programming*. Sebastopol, CA: O'Reilly, 1998.
20. Apache Axis, *http://xml.apache.org/axis/*
21. *uddi.org*, *http://www.uddi.org/*
22. World Wide Web Consortium. Web Service Definition Language (WSDL), *http://www.w3.org/TR/wsdl*
23. Jabber Software Foundation, *http://www.jabber.org/*
24. World Wide Web Consortium. XML Schema Part 2: Datatypes, *http://www.w3.org/TR/xmlschema-2/*
25. Java Server Page Tutorial, *http://java.sun.com/products/jsp/pdf/jsptut.pdf*
26. Tutorial: Servlets and JSP, *http://javaboutique.internet.com/tutorials/jsp.html*

CAPÍTULO 12

Paradigmas avanzados de computación distribuida

En los capítulos anteriores se han explorado un diverso número de paradigmas inicialmente presentados en el Capítulo 3, donde se mostraba una jerarquía de paradigmas repetida aquí en la Figura 12.1.

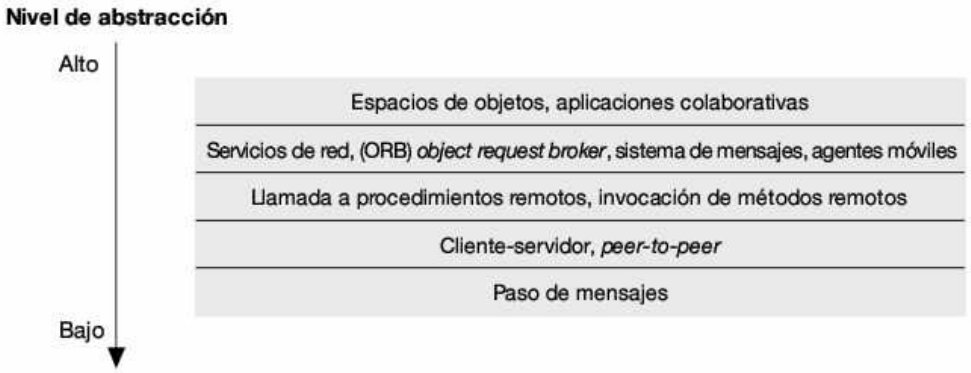


Figura 12.1. Los paradigmas de computación distribuida y su nivel de abstracción.

Este capítulo presenta alguno de los paradigmas más avanzados. Algunos de ellos están siendo aún investigados, mientras que otros son usados ampliamente en la práctica. Especialmente, este capítulo dará una visión general de los siguientes paradigmas de computación distribuida: **sistemas de colas de mensajes**, **agentes móviles**, **servicios de red**, **espacios de objetos** y **computación colaborativa**.

12.1. PARADIGMA DE SISTEMAS DE COLAS DE MENSAJES

El paradigma de sistemas de colas de mensajes, también denominado **middleware orientado a mensajes (MOM, *message-oriented middleware*)**, es una elaboración del paradigma de paso de mensajes. En este paradigma, un sistema de mensajes actúa como intermediario entre procesos separados e independientes.

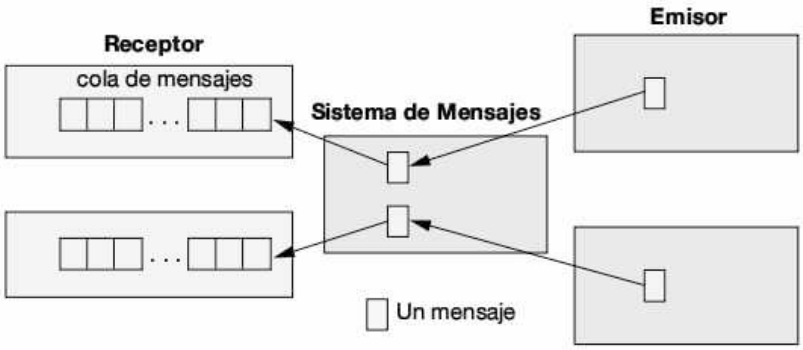


Figura 12.2. Paradigma de sistemas de colas de mensajes.

La Figura 12.2 ilustra el paradigma. Los mensajes se envían al sistema de mensajes, que actúa de conmutador de los mismos, encaminándolos a los receptores apropiados.

A través de un sistema de mensajes, los procesos intercambian mensajes de forma asíncrona, de una manera desacoplada. Un emisor deposita un mensaje en el sistema, el cual lo reenvía a una cola de mensajes asociada con cada receptor. Una vez que el mensaje se ha enviado, el emisor se libera para poder realizar otras tareas. El mensaje se reenvía por parte del sistema de mensajes hasta los clientes. Cada cliente puede extraer de su cola bajo demanda.

Los modelos de sistemas de colas de mensajes se pueden clasificar en dos subtipos, **punto-a-punto** y **publicación/suscripción**, que se describen a continuación:

Modelo de mensajes punto-a-punto

El modelo mostrado en la Figura 12.2 se corresponde con el modelo de mensajes punto-a-punto. En este modelo, un sistema de mensajes redirige un mensaje desde el emisor hasta la cola de mensajes del receptor. A diferencia del modelo básico de paso de mensajes, el modelo de mensaje punto-a-punto proporciona un depósito de los mensajes que permite que el envío y la recepción estén desacoplados.

Comparado con el modelo básico de paso de mensajes, este paradigma proporciona una abstracción adicional para **operaciones asíncronas**: no existe un bloqueo entre emisor y receptor.

Modelo de mensajes publicación/suscripción

En este modelo, mostrado en la Figura 12.3, cada mensaje se asocia con un determinado tema o evento. Las aplicaciones interesadas en el suceso de un evento específi-

co se pueden suscribir a los mensajes de dicho evento. Cuando ocurre el evento que se aguarda, el proceso publica un mensaje anunciando el evento o asunto. El sistema de cola de mensajes distribuye el mensaje a todos los suscriptores.

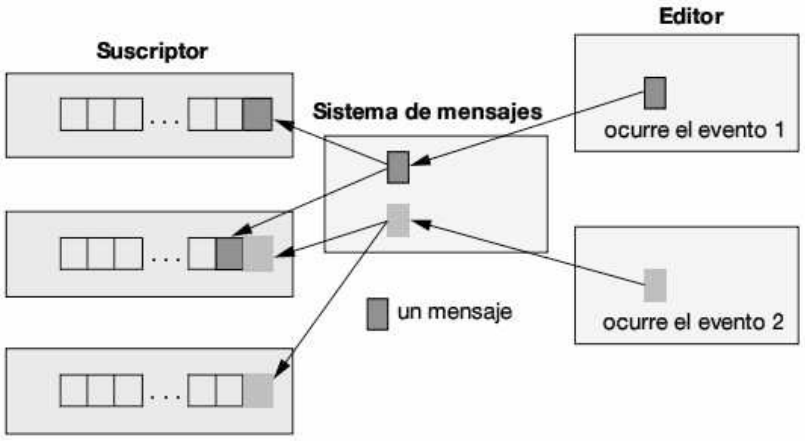


Figura 12.3. El modelo de sistema de mensajes publicación/suscripción.

El modelo de mensajes publicación/suscripción ofrece una potente abstracción para **multidifusión** o **comunicación en grupo**. La operación **publicar** permite al proceso difundir a un grupo de procesos, y la operación **suscribir** permite a un proceso escuchar dicha difusión de mensajes.

El paradigma de sistema de mensajes se emplea comercialmentre de forma muy amplia en aplicaciones distribuidas. Existe un gran número de herramientas que dan soporte a este paradigma. Algunas de ellas son:

- *MQ*Series* de IBM (rebautizado como WebSphere MQ) [4.ibm.com, 3],
- *Microsoft's Message Queue* (MSMQ) [microsoft.com, 2],
- *Java Message Service* (JMS) [java.sun.com, 1] disponible con el J2EE SDK [java.sun.com, 4] versiones 1.3 o superiores.

Para aquellos estudiantes interesados en una API que dé soporte a un modelo de sistema de paso de mensajes, JMS es un buen punto de arranque.

La Figura 12.4 y la Figura 12.5 muestran un ejemplo de programación con JMS. Un *EmisorMensaje* envía un mensaje a una cola JMS, mientras que *ReceptorMensaje* lo recibe de la cola. Para poder apreciar el asíncronismo que el sistema de cola de mensajes proporciona, deberá experimentar arrancando una o varias copias de *ReceptorMensaje*, y posteriormente el *EmisorMensaje*. Cada uno de los clientes debe recibir un mensaje ("¡Hola Mundo!") una vez que se ha enviado. (Nota: los programas de ejemplo sólo compilarán y ejecutarán en un sistema que tenga instalado JMS; consulte [java.sun.com, 1] para obtener información sobre cómo descargar e instalar JMS.)

Figura 12.4. Un ejemplo de un emisor de mensajes punto-a-punto.

```
1 /**
2  * La clase EmisorMensaje manda un mensaje "¡Hola Mundo!" a
3  * una cola de mensajes de Java Message System (JMS). Este
```

(continúa)

```
 4  * programa está preparado para ejecutarse con la clase
 5  * ReceptorMensaje, que recibe el mensaje vía el servicio
 6  * JMS.
 7  * Para ejecutar el programa: El proveedor JMS debe estar
 8  * arrancado y se debe haber creado una cola. El nombre de
 9  * la cola se debe especificar como una rgumento en la línea
10  * de mandatos al arrancar el programa.
11  * M. Liu, basado en los ejemplos del
12  * tutorial http://java.sun.com/products/jms/tutorial/
13  */
14 import javax.jms.*; // para las clases JMS
15 import javax.naming.*; // para las clases JNDI
16
17 public class EmisorMensaje {
18
19     public static void main(String[] args) {
20       String nombreCola = null;
21       Context contextoJNDI = null;
22       QueueConnectionFactory fabricaConexiones
23            = null;
24       QueueConnection conexionCola = null;
25       QueueSession sesionCola = null;
26       Queue cola = null;
27       QueueSender emisorCola = null;
28       TextMessage mensaje = null;
29       final int NUM_MSGS;
30
31       if ( (args.length != 1) ) {
32           System.out.println("Uso: java " +
33                                  "EmisorMensaje <nombre-cola>");
34           System.exit(1);
35       }
36       nombreCola = new String(args[0]);
37       System.out.println("Nombre de la cola es " + nombreCola);
38
39       /* Crea un objeto InitialContext de JNDI si
40          no existe ningunao. */
41
42       try {
43            contextoJNDI = new InitialContext();
44       } catch (NamingException e) {
45            System.out.println("No se ha podido crear el contexto " +
46                                       "JNDI: " + e.toString());
47            System.exit(1);
48       }
49
50       /* Búsqueda de la fábrica de conexiones y la cola. Si
51          alguna de las dos no existe. */
```

(continúa)

```
52
53        try {
54             fabricaConexiones =
55                  (QueueConnectionFactory)
56                  contextoJNDI.lookup("QueueConnectionFactory")
57                  cola = (Queue) contextoJNDI.lookup(nombreCola);
58        } catch (NamingException e) {
59             System.out.println("Búsqueda JNDI fallida: " +
60                                       e.toString());
61             System.exit(1);
62        }
63
64        try {
65             /* crear conexión */
66             conexionCola =
67                  fabricaConexiones.createQueueConnection();
68
69             /* crear sesión */
70             sesionCola =
71                  conexionCola.createQueueSession(false,
72                                            Session.AUTO_ACKNOWLEDGE);
73
74             /* crear emisor y objeto mensaje. */
75             emisorCola = sesionCola.createSender(cola);
76             mensaje = sesionCola.createTextMessage();
77
78             /* definir mensaje */
79             mensaje.setText("¡Hola Mundo!");
80             System.out.println("Enviando mensaje: " +
81                                       mensaje.getText());
82
83             /* enviar a JMS */
84             emisorCola.send(mensaje);
85
86        } catch (JMSException e) {
87             System.out.println("Ha ocurrido una excepción: " +
88                                       e.toString());
89        } finally {
90             /* cerrar la cola de conexión */
91             if (conexionCola != null) {
92                  try {
93                       conexionCola.close();
94                  } catch (JMSException e) {}
95             }
96        } //fin finally
97      } //fin main
98 } //fin class
```

Figura 12.5. Un ejemplo de un receptor de mensajes punto-a-punto.

```
/**
 * La clase ReceptorMensaje se debe usar junto con
 * la clase EmisorMensaje, la cual manda un mensaje
 * via the JMS.
 * Se debe dar el mismo nombre de cola que en EmisorMensaje
 * por medio de un argumento en la línea de mandatos
 * M. Liu, basado en los ejemplos del
 * Tutorial,http://java.sun.com/products/jms/tutorial/
 */

import javax.jms.*;
import javax.naming.*;

public class ReceptorMensaje {

    public static void main(String[] args) {
      String nombreCola = null;
      Context contextoJNDI = null;
      QueueConnectionFactory fabricaConexiones
          = null;
      QueueConnection conexionCola = null;
      QueueSession sesionCola = null;
      Queue cola = null;
      QueueReceiver receptorCola = null;
      TextMessage mensaje = null;

      if (args.length != 1) {
          System.out.println("Uso: java " +
                                  "ReceptoMensaje <nombre-cola>");
          System.exit(1);
      }
      nombreCola = new String(args[0]);
      System.out.println("Nombre de la cola es " + nombreCola);

      /* Crea un objeto InitialContext de JNDI si
         no existe ningunao. */

      try {
          contextoJNDI = new InitialContext();
      } catch (NamingException e) {
          System.out.println("No se ha podido crear el contexto " +
                                  "JNDI: " + e.toString());
          System.exit(1);
      }

      /* Búsqueda de la fábrica de conexiones y la cola. Si
         alguna de las dos no existe. */
```

(continúa)

```
48        try {
49             fabricaConexiones =
50                  (QueueConnectionFactory)
51                  contextoJNDI.lookup("QueueConnectionFactory");
52             cola = (Queue) contextoJNDI.lookup(nombreCola);
53        } catch (NamingException e) {
54             System.out.println("Búsqueda JNDI fallida: " +
55                                       e.toString());
56             System.exit(1);
57        }
58
59        try {
60             /* crear conexión */
61             conexionCola =
62                  fabricaConexiones.createQueueConnection();
63             /* crear sesión de la conexión */
64             sesionCola =
65                  conexionCola.createQueueSession(false,
66                                            Session.AUTO_ACKNOWLEDGE);
67             /* crear un receptor */
68             receptorCola = sesionCola.createReceiver(cola);
69             conexionCola.start();
70             /* recibe el mensaje */
71             Message m = receptorCola.receive(1);
72             mensaje = (TextMessage) m;
73             System.out.println("Leyendo mensaje: " +
74                                       mensaje.getText());
75        } catch (JMSException e) {
76             System.out.println("Ha ocurrido una excepción: " +
77                                       e.toString());
78        } finally {
79             if (conexionCola != null) {
80                  try {
81                       conexionCola.close();
82                  } catch (JMSException e) {}
83             }
84        }//fin finally
85      }//fin main
86 }//fin class
```

Se recomienda a los lectores interesados en más detalles sobre el API de JMS la referencia [java.sun.com,1] que muestra más ejemplos, incluyendo los relativos al modelo **publicación/suscripción**.

12.2. AGENTES MÓVILES

Un **agente móvil** es un paradigma de computación distribuida que ha interesado a los investigadores desde los años 80. Los agentes móviles se han convertido en tecnoló-

gicamente viables gracias a las últimas tecnologías y hoy por hoy tienen el potencial de revolucionar las aplicaciones en red.

En el contexto de la informática, un **agente** es un programa software independiente que se ejecuta en representación del usuario.

Un agente móvil es un programa que, una vez lanzado por el usuario, puede viajar de ordenador a ordenador **autónomamente** y puede continuar con sus función incluso si el usuario se desconecta. La Figura 12.6 ilustra el concepto de un agente móvil.

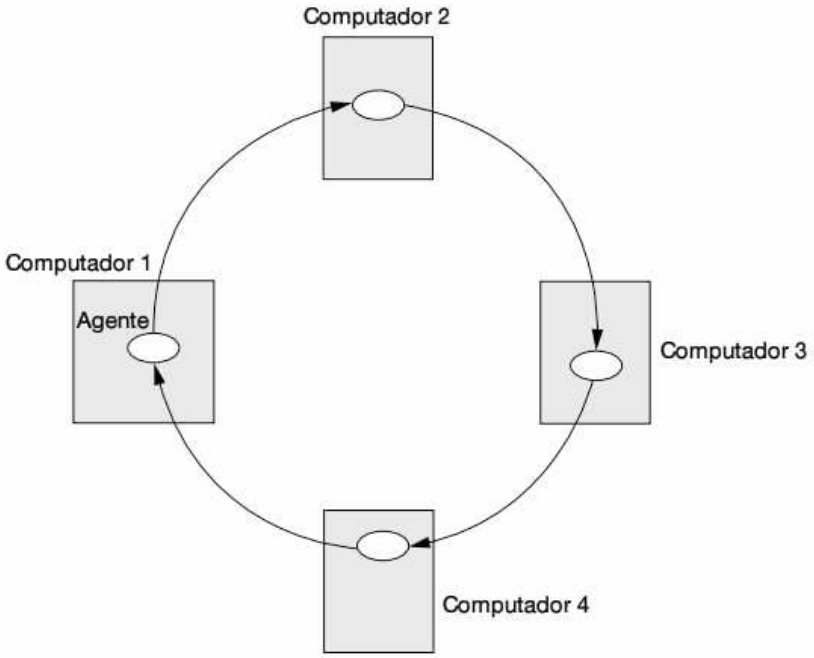


Figura 12.6. Un agente móvil viaja de un computador a otro.

Arquitectura básica

Un agente móvil es un **objeto serializable**: un objeto cuyos datos al igual que el estado se pueden empaquetar para ser transmitidos por la red. El lector recordará el término **empaquetado de datos (*data marshaling*)**, presentado por primera vez en el Capítulo 1, como el aplanamiento y codificación de estructuras de datos con el fin de transmitirlas de un computador a otro. Un objeto se puede serializar y transmitir entre dos ordenadores de la misma forma. Una vez llegado, el objeto se reconstruye y se deserializa, con su estado restaurado como al momento de serializarse, y entonces el objeto puede retomar su ejecución en el sistema donde acaba de llegar.

La arquitectura básica para dar soporte a agentes móviles se muestra en la Figura 12.7. Un objeto serializable, representando al agente móvil, se lanza en un computador. El agente contiene los siguientes datos:

- **Información de identidad:** Información que permite identificar al agente.
- **Itinerario:** Una lista de direcciones de ordenadores que el agente debe visitar.

- **Datos de la tarea:** Los datos que el agente requiere para realizar sus tareas, o bien datos recogidos por el agente.

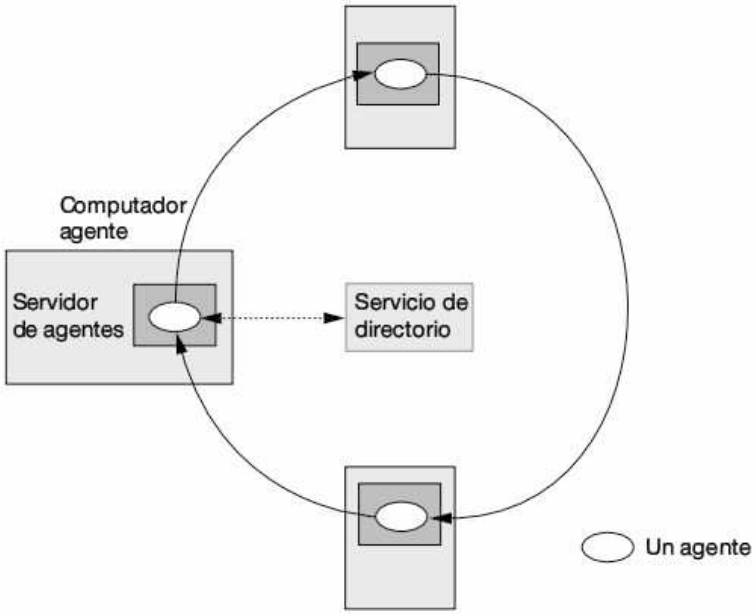


Figura 12.7. Un agente móvil viaja por varios ordenadores.

Un agente también lleva consigo la lógica (código) para realizar las tareas.

En cada parada, el agente llega a un servidor. A través del servidor de agentes, el agente móvil usa los recursos locales para realizar sus tareas.

Como parte de la arquitectura, se necesita un servicio de directorio que permita al agente buscar el servidor en cada parada. Cuando el agente ha concluido su tarea en un sitio, el objeto agente se serializa, y, con la ayuda de un servidor de agentes, el objeto se transporta al siguiente ordenador del itinerario.

Hay varios entornos que se han desarrollado para proporcionar arquitecturas de soporte a agentes móviles, por ejemplo Aglet [Lange and Oshima,5; aglets.sourceforge.net, 6], Concordia [merl.com, 7], y Grasshopper [grasshopper.de, 8]. Por simplicidad, sin embargo, se mostrará el concepto de agente móvil usando objetos serializables y Java RMI. (*Nota*: la implementación presentada vale para demostrar el concepto de agente móvil. Para verdaderos desarrollos, existen consideraciones, a discutir, que la demostración no trata).

La Figura 12.8 presenta la interfaz Java de un agente; la implementación de la interfaz se ve en la Figura 12.9.

Figura 12.8. *InterfazAgente.java*.

```
// Una interfaz para un objeto transportable
// que representa un agente móvil
// M. Liu
```

(continúa)

```
import java.io.Serializable;
public interface InterfazAgente extends Serializable {
     void ejecuta();
}
```

Figura 12.9. *Agente.java.*

```
 1 // Una implementación de un agente móvil
 2
 3 import java.io.*;
 4 import java.util.*;
 5 import java.rmi.*;
 6 import java.rmi.registry.Registry;
 7 import java.rmi.registry.LocateRegistry;
 8
 9 public class Agente implements InterfazAgente {
10
11      int indiceNodo; // cuál es el siguiente ordenador a visitar
12      String nombre;
13      Vector listaNodos; // el itinerario
14      int puertoRMI = 12345;
15
16      public Agente(String miNombre, Vector laListaComputadoras,
17                    int elPuertoRMI ) {
18           nombre = miNombre;
19           listaNodos = laListaNodos;
20           indiceNodo = 0;
21           puertoRMI = elPuertoRMI;
22      }
23
24      // Este método define las tareas que realiza el agente
25      // móvil una vez que llega a un servidor.
26      public void ejecuta() {
27           String actual, siguiente;
28           dormir (2); // pausa para poder visualizarlo
29           System.out.println("¡Aquí el agente 007!");
30           actual = (String) listaNodos.elementAt(indiceNodo);
31           indiceNodo++;
32           if (indiceNodo < listaNodos.size()) {
33                // si hay más computadoras que visitar
34                siguiente = (String) listaNodos.elementAt(indiceNodo);
35                dormir (5); // pausa para poder visualizarlo
36                try {
37                     // Localiza el registro RMI en el siguiente nodo
38                     Registry registro = LocateRegistry.getRegistry
39                          ("localhost", puertoRMI);
40                     ServerInterface h = (ServerInterface)
41                          registro.lookup(siguiente);
```

(continúa)

```
42                    System.out.println("Buscando " + siguiente +
43                                  " en " + actual + " completado ");
44                    dormir (5); // pausa para poder visualizarlo
45                   // Pide al servidor del siguiente nodo que reciba a
46                    // este agente.
47                    h.recibe(this);
48               } // fin try
49               catch (Exception e) {
50                    System.out.println
51                        ("Excepción en el ejecuta del Agente: " + e);
52               }
53          } // fin if
54          else { //si se han hecho todas las paradas
55               dormir (5); // pausa para poder visualizarlo
56              System.out.println("El Agente 007 ha regresado a casa");
57               dormir (5); // pausa para poder visualizarlo
58          }
59      }
60
61      // El método dormir suspende la ejecución de este objeto
62      // un número determinado de segundos.
63      static void dormir (double time ){
64          try {
65               Thread.sleep( (long) (time * 1000.0));
66          }
67          catch (InterruptedException e) {
68               System.out.println ("excepción en dormir);
69          }
70      } // fin dormir
71
72  } // fin class Agente
```

El núcleo de la implementación del agente (Figura 12.9) está en su constructor (líneas 16 –22) y en su método *ejecuta* (líneas 26 –59). El constructor inicializa los datos de estado del agente, incluyendo el itinerario y el índice al siguiente computador a visitar en dicho itinerario. El método *ejecuta* contiene la lógica de las tareas que se espera que el agente realice en cada ordenador. En esta demo, las tareas se resumen en la salida de un simple mensaje (línea 56) y en la búsqueda del servidor de agentes de la siguiente parada (líneas 32 –53).

La implementación de servidor de agentes se muestra en las Figuras 12.10 y 12.11.

Figura 12.10. *InterfazServidor.java.*

```
// Fichero de interfaz del servidor de agentes - M. Liu
import java.rmi.*;
public interface InterazServidor extends Remote {
     public void recibe(Agente h)
   throws java.rmi.RemoteException;
}
```

El servidor, por su parte, se registra en el registro RMI y proporciona un método, *recibe*, que el agente móvil invoca cuando está preparado para transportarse al servidor. El método imprime un mensaje que anuncia la llegada del agente, y posteriormente llama al método *ejecuta* del agente. Nótese que el método *recibe* acepta como argumento el objeto agente serializable.

Figura 12.11. *Servidor.java.*

```
// Una implementación de un servidor agentes
import java.rmi.*;
import java.rmi.server.*;
import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.net.*;
import java.io.*;

public class Servidor extends UnicastRemoteObject
     implements InterfazServidor{
     static int puertoRMI = 12345;
     public Servidor() throws RemoteException {
          super();
     }

     public void recibe(Agent h) throws RemoteException {
          dormir (3); // pausa para poder visualizarlo
          System.out.println
               ("*****El Agente " + h.name + " ha llegado." );
          h.ejecuta();
     }

     public static void main(String args[]) {
          InputStreamReader is = new InputStreamReader(System.in);
          BufferedReader br = new BufferedReader(is);
          String s;
          String miNombre = "servidor" + args[0];
             //Nota: Los servidores se espera que se arranquen con los
            //argumentos 1, 2 y 3 de la línea de mandatos respectivamente,
             try{
                  System.setSecurityManager(new RMISecurityManager());
                  Servidor h = new Servidor();
                  Registry registro =
                       LocateRegistry.getRegistry(puertoRMI);

                  registro.rebind( miNombre, h);

                 System.out.println("*****************************");
                 System.out.println(" Agente " + miNombre + " listo.");
                 System.out.println("*****************************");
```

(continúa)

```
41              }// fin try
42              catch (RemoteException re) {
43                  System.out.println("Excepción en el main del Servidor: "
44                                                  + re);
45              } // fin catch
46          } // fin main
47
48          // El método dormir suspende la ejecución de este objeto
49          // un número determinado de segundos.
50          static void dormir (double time ){
51              try {
52                  Thread.sleep( (long) (time * 1000.0));
53              }
54              catch (InterruptedException e){
55                  System.out.println ("excepción en dormir");
56              }
57          } // fin dormir
58      } // fin class
```

Para finalizar, se necesita un programa cliente que cree una instancia y lance al agente hacia la primera parada del itinerario. La implementación se muestra en la Figura 12.12.

Figura 12.12. *Cliente.java.*

```
1  //Cliente.java - programa cliente que lanza al agente móvil
2  // M. Liu
3
4  import java.io.*;
5  import java.util.*;
6  import java.rmi.*;
7  import java.rmi.registry.Registry;
8  import java.rmi.registry.LocateRegistry;
9
10 public class Cliente {
11      static int puertoRMI = 12345;
12      public static void main(String args[]) {
13
14          System.setSecurityManager(new RMISecurityManager());
15
16          try {
17              Registry registro = LocateRegistry.getRegistry
18                  ("localhost", puertoRMI);
19              ServerInterface h = (ServerInterface)
20                  registro.lookup("servidor1");
21              System.out.println
22                  ("Lookup for servidor1 completed " );
23              System.out.println("***Buen viaje, " +
```

(continúa)

```
24                                          " agent 007.");
25
26              Vector listaNodos = new Vector();
27              listaNodos.addElement("servidor1");
28              listaNodos.addElement("servidor2");
29              listaNodos.addElement("servidor3");
30              Agent a = new Agent("007", listaNodos, puertoRMI);
31              h.receive(a);
32              System.out.println("***Buen trabajo, 007");
33          }
34          catch (Exception e) {
35              System.out.println("Excepción en main: " + e);
36          }
37      } //fin main
38 } //fin class
```

Como paradigma de computación distribuida, los agentes móviles son radicalmente diferentes al resto de paradigmas que se han visto. En otros paradigmas, procesos independientes colaboran intercambiando datos sobre sus enlaces de red. Con los agentes móviles, un proceso se transmite, llevando consigo los datos a compartir y visitando la lista de procesos de su itinerario. La Figura 12.13 muestra esta característica discriminatoria.

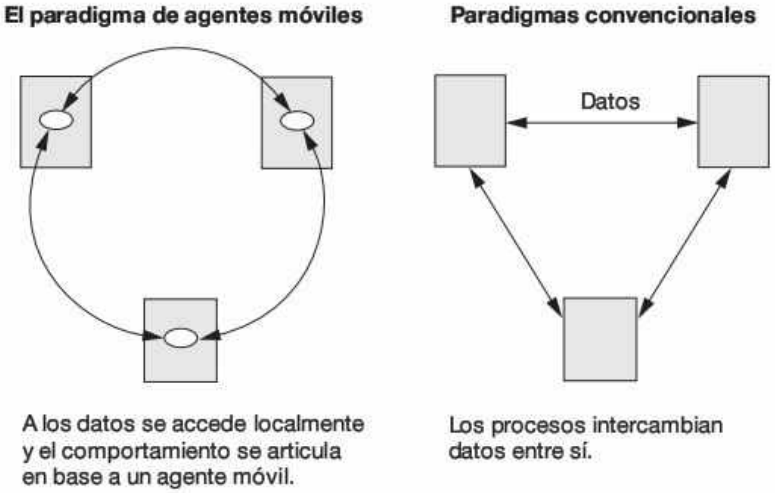


Figura 12.13. Contraste entre el paradigma de agentes móviles y los paradigmas convencionales.

Ventajas de los agentes móviles

El paradigma de agentes móviles proporciona las siguientes ventajas:

- Los agentes móviles permiten un uso eficiente y económico de los canales de comunicación, que pueden ser de ancho de banda reducido, latencia elevada, y pudiendo ser proclives a errores. En contra, los paradigmas convencionales re-

quieren el intercambio repetido de paquetes de datos sobre la red. Si esto afecta a grandes volúmenes de datos, el consumo de ancho de banda puede ser considerable, con el consecuente retardo en el tiempo de respuesta o la latencia. Por medio de agentes móviles, un único objeto se serializa y se transmite por la red, potencialmente reduciendo el consumo de ancho de banda. Además, ya que un agente móvil sólo se necesita transportar entre dos nodos una vez, la probabilidad de fallo debido a un error de comunicación se reduce. Por estas dos razones, los agentes móviles son especialmente interesantes para enlaces de red inalámbricos.

- Los agentes móviles permiten el uso de dispositivos de comunicación portátiles y de bajo coste para realizar tareas complejas incluso cuando el dispositivo está desconectado de la red. Un agente se puede lanzar desde cualquier dispositivo que soporte la arquitectura necesaria. Una vez lanzado, el agente se desacopla de su origen y es capaz de realizar las tareas de forma independiente de su dispositivo inicial. Con una implementación apropiada, un agente móvil puede tener la inteligencia de saltarse nodos que fallen o de buscar refugio momentáneo en nodos fiables.
- Los agentes móviles permiten operaciones asíncronas y una verdadera descentralización. La ejecución de un agente móvil está desacoplada de su servidor original y de los servidores participantes. Las tareas se realizan en cada uno de los nodos individuales de una manera asíncrona, y no resulta necesario que ninguno de los participantes asuma el papel de coordinador.

A pesar de las ventajas ofrecidas por los agentes móviles, el paradigma no se ha desarrollado de forma amplia en aplicaciones industriales o comerciales. Una de las principales pegas del paradigma es que puede suponer un riesgo de seguridad para los participantes.

Existen consideraciones para ambos, servidores y agentes móviles. Desde el punto de vista de los servidores que reciben un agente, agentes maliciosos o no autorizados pueden hacer un uso fraudulento y destruir recursos locales. Sin restricciones, un agente malicioso puede causar un caos en el servidor, de igual manera que lo hacen los virus. Desde el punto de vista de un agente, servidores maliciosos pueden destruir o alterar los datos del agente o su lógica. Por ejemplo, el itinerario de un agente móvil se podría alterar o destruir por un servidor malicioso, de forma que el agente no pudiese continuar con su viaje. O, los datos almacenados por el agente se podrían alterar de forma que el agente se llevase información errónea.

La seguridad en sistemas de agentes es un área de investigación muy activa [mole.informatik.uni-stuttgart.de, 9]. Algunas de las contramedidas que se han propuesto son:

- **Autenticación**. Un agente debe autentificarse ante el servidor, y un servidor de agentes debe autentificarse ante el agente.
- **Cifrado**. Un agente debe cifrar sus datos sensibles.
- **Acceso a recursos**. Un servidor forzará un estricto control de acceso a sus recursos.

Sistemas basados en entornos para agentes móviles

Hay disponible un gran número de sistemas basados en entornos de desarrollo para agentes móviles. Además de proporcionar la arquitectura para transportar agentes mó-

viles, estos sistemas también tratan cuestiones relativas a la seguridad, el soporte transaccional, y la gestión de agentes. Los lectores que estén interesados en este tema pueden buscar en referencias como [trl.ibm.com, 6], [concordiaagents.com, 7], y [grasshopper.de, 8].

12.3. SERVICIOS DE RED

En el paradigma de servicios de red [Edwards, 11], la red se ve como una infraestructura para servicios, que puede incluir aplicaciones, ficheros, bases de datos, servidores, sistemas de información, y dispositivos, tales como electrodomésticos móviles, almacenamiento o impresoras. La Figura 12.14 ilustra este paradigma. Una aplicación cliente puede utilizar uno o varios de estos servicios. Los servicios se pueden añadir y eliminar de la red de forma autónoma, y los clientes pueden localizar los servicios disponibles, a través de un servicio de directorio.

El protocolo SOAP (*Simple Object Access Protocol*), que se vio en el Capítulo 11, se basa en este paradigma. Jini, que es anterior a SOAP, es un conjunto de herramientas basado en este paradigma.

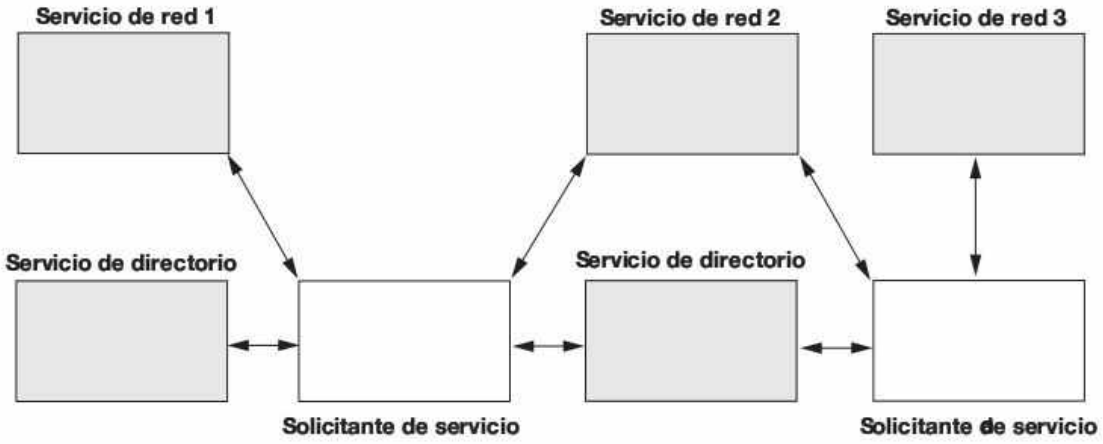


Figura 12.14. El paradigma de servicios de red.

Las Figuras 12.15 y 12.16 muestran la sintaxis de la implementación de un proveedor de servicios en Jini, que proporciona el método *decirHola()*. La Figura 12.17 muestra la implementación de un cliente de dicho servicio. Nótese que Jini hace uso de RMI, que ya se vio en los Capítulos 7 y 8.

Figura 12.15. *InterfazServidorHola.java.*

```
1 // Interfaz para el ServidorHola
2
3 import java.rmi.*;
4
5 public interface InterfazServidorHola extends Remote {
6      public String decirHola() throws RemoteException;
7 }
```

Figura 12.16. *ServidorHola.java.*

```
1 // Un ejemplo de servidor Jini sencillo
 2 // M. Liu, basado en el ejemplo "Noel's Nuggets" [cc.gatech.edu, 14]
 3 import net.jini.core.entry.*;
 4 import net.jini.core.lookup.*;
 5 import net.jini.core.discovery.*;
 6 import net.jini.lookup.entry.*;
 7 import com.sun.jini.lookup.*;
 8 import com.sun.jini.lease.*;
 9 import java.io.*;
10 import java.rmi.*;
11 import java.rmi.server.*;
12
13 public class ServidorHola extends UnicastRemoteObject
14      implements InterfazServidorHola, ServiceIDListener {
15
16      public ServidorHola () throws RemoteException {
17           super();
18      }
19
20      public String decirHola () throws RemoteException {
21           return ("¡Hola Mundo!");
22      }
23
24      // Este método está a la escucha de ServiceID de parte del
25      // servicio de directorio, cuando el ID esté disponible.
26      public void serviceIDNotify (ServiceID id) {
27           System.out.println (" ServiceID recibido: " + id);
28      }
29
30      public static void main (String[] args){
31           ServidorHola servidor;
32           Entry[] atributos;
33           JoinManager manager;
34
35           try {
36                System.setSecurityManager
37                     (new RMISecurityManager ());
38               // Crea los atributos como un vector de objetos
39               // Entry que describen este servicio, y
40               // se registra en el servicio de búsqueda vía
41               // el JoinManager. El ServiceID, cuando esté disponible,
42            // le será indicado por medio del listerner del ServiceID.
43
44           atributos = new Entry[1];
45           atributos[0] = new Name("ServidorHola");
46           servidor = new ServidorHola();
47           manager = new JoinManager (
48               servidor, atributos, servidor,
```

(continúa)

```
49              new LeaseRenewalManager()
50              );
51      }
52
53          catch (Exception ex) {
54              ex.printStackTrace( );
55          }
56      } //fin main
57 } //fin class
```

Figura 12.17. Cliente Jini.

```
 1 // Un ejemplo de un cliente Jini
 2 // M. Liu, basado en el ejemplo "Noel's Nuggets" [cc.gatech.edu, 14]
 3
 4 import net.jini.core.entry.*;
 5 import net.jini.core.lookup.*;
 6 import net.jini.core.discovery.*;
 7 import net.jini.lookup.entry.*;
 8 import com.sun.jini.lookup.*;
 9 import java.rmi.*;
10
11 class ClienteHola{
12      public static void main (String[] args){
13
14          Entry[] atributos;
15          LookupLocator buscador;
16          ServiceID id;
17          ServiceRegistrar registrar;
18          ServiceTemplate plantilla;
19          InterfazServidorHola servidorHola;
20
21          try {
22              System.setSecurityManager
23                  (new RMISecurityManager ());
24
25              // Localizar el servcio de búsqueda de Jini
26              buscador = new LookupLocator("jini://localhost");
27              // Buscar el ServiceRegistrar
28              registrar = buscador.getRegistrar();
29              // Búsqueda del servicio
30              atributos = new Entry[1];
31              atributos[0] = new Name ("ServidorHola");
32              plantilla = new ServiceTemplate
33                  (null, null, atributos);
34              servidorHola = (InterfazServidorHola)
35                  registrar.buscador(plantilla);
36              // Invocar el método del servicio
```

(continúa)

```
37                System.out.println(servidorHola.decirHola());
38           }
39
40           catch (Exception ex) {
41                ex.printStaceTrace( );
42           }
43      }
44 }
```

La introducción que se ha presentado apenas toca los aspectos fundamentales de Jini y no hace justicia a las sofisticadas capacidades proporcionadas por este conjunto de herramientas. Un concepto interesante en Jini es el uso de alquileres (*leases*) (línea 49 en la Figura 12.16). En un sistema Jini, cada servicio se asocia a **un gestor de renovación de alquileres**. Cuando un cliente localiza un servicio Jini a través de un servicio de búsqueda, el propio servicio de búsqueda actúa como elemento que concede periodos de alquiler para el servicio. El cliente puede realizar peticiones mientras que su alquiler no haya expirado. El uso de alquileres permite al sistema proporcionar tolerancia a fallos de una forma más eficiente. (Los fallos en sistemas distribuidos y la tolerancia a fallos se presentaron en el Capítulo 1).

12.4. ESPACIOS DE OBJETOS

Esta sección muestra el paradigma que proporciona el mayor nivel de abstracción: los **espacios de objetos**. El paradigma de espacios de objetos tiene su origen en el Espacio de Tuplas de Linda (*Linda Tuplespace*) desarrollado por David Gelernter [Ahuja, Carrier, y Gelernter, 17] y es la base de los sistemas de Pizarras en el campo de la Inteligencia Artificial [Corkill, 15]. En el paradigma de espacios de objetos, un **espacio** es un repositorio para objetos, compartido y accesible por red. En lugar de comunicarse entre sí, los procesos se coordinan **intercambiando objetos** a través de uno o varios espacios. La Figura 12.18 ilustra este concepto. Los procesos interactúan compartiendo los datos de estado en uno o más objetos o actualizando el estado de los objetos según sus necesidades.

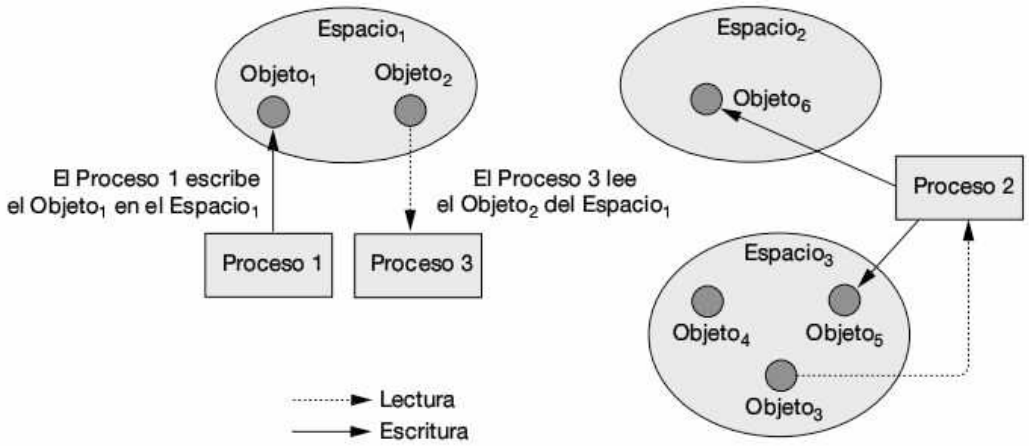


Figura 12.18. El paradigma de espacio de objetos.

Un proceso puede depositar un objeto en un espacio. A diferencia de los modelos de mensajes, los procesos no modifican los objetos en el espacio ni invocan los métodos de los objetos directamente. Un proceso que desea acceder a un objeto utilizará un servicio de directorio para localizar el objeto en el espacio. Para modificar el objeto, un proceso debe *retirarlo* explícitamente, actualizarlo y *reinsertarlo* en el espacio.

El paradigma de espacios de objetos es un área activa de investigación. Para poder experimentar con este paradigma existe un conjunto de herramientas basado en Java, JavaSpaces, que proporciona un entorno de trabajo para espacios de objetos.

En el API de JavaSpaces, se proporciona una interfaz llamada *Entry*. Un objeto *Entry* puede escribirse o leerse del espacio. De la Figura 12.19 a la 12.21 se muestra un programa de ejemplo sencillo que utiliza un objeto *Entry* que contiene un mensaje ¡*Hola Mundo*! y un contador. El servidor crea el objeto *Entry* y lo escribe en el espacio de objetos por defecto. El cliente lee (toma) el objeto del espacio e incrementa el contador, antes de escribirlo de nuevo en el espacio. Nótese que JavaSpaces utiliza el API de Jini presentada en la sección 12.3.

Figura 12.19. Un ejemplo de una clase *Entry* de *JavaSpaces*.

```
1 // Una clase Entry de ejemplo para un objeto de espacio JavaSpaces
2 // M. Liu, basado en el ejemplo proporcionado en:
3 // JAVASPACES PRINCIPLES, PATTERNS, AND PRACTICE [18]
4
5
6
7 import net.jini.core.entry.Entry;
8
9 public class ObjetoDeEspacio implements Entry {
10      public String mensaje;
11      public Integer contador; //Es necesario que sea un objeto
12
13      public ObjetoDeEspacio( ) {
14      }
15
16      public ObjetoDeEspacio(String mensaje) {
17           this.mensaje = mensaje;
18           contador = 0;
19      }
20
21      public String toString( ) {
22           return mensaje + " leido " + contador + " veces.";
23      }
24
25      public void incremento( ) {
26           contador = new Integer(contador.intValue() + 1);
27      }
28 } //fin class
```

Figura 12.20. Un programa de ejemplo que inicializa un objeto *JavaSpaces*.

```
// Un programa que inicializa un Entry de JavaSpaces.
// M. Liu, basado en el ejemplo proporcionado en:
// JAVASPACES PRINCIPLES, PATTERNS, AND PRACTICE [18]



import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

public class HolaMundo {
     public static void main(String[] args) {
          try {
               SpaceObject mens = new SpaceObject("¡Hola Mundo!");

               JavaSpace espacio;
               espacio = (JavaSpace)space();
               espacio.write(mens, null, Lease.FOREVER);

               SpaceObject plantilla = new SpaceObject();
               while (true) {
                    SpaceObject resultado = (SpaceObject)
                        espacio.read(plantilla, null, Long.MAX_VALUE);
                    System.out.println(resultado);
                    Thread.sleep(1000);
               } //fin while
          }
          catch (Exception ex) {
               ex.printStackTrace();
          }
     }
} //fin class
```

Figura 12.21. Un ejemplo de un cliente *JavaSpaces*.

```
// Un programa cliente que accede a un Entry de JavaSpaces
// M. Liu, basado en el ejemplo proporcionado en:
// JAVASPACES PRINCIPLES, PATTERNS, AND PRACTICE [18]



import net.jini.core.lease.Lease;
import net.jini.space.JavaSpace;

public class ClienteHolaMundo {
     public static void main(String[] args) {
          try {
               JavaSpace espacio = (JavaSpace)space();
               SpaceObject plantilla = new SpaceObject();
```

(continúa)

```
15                // Retira el objeto del espacio repetidas veces,
16                // lo modifica, y lo deposita de nuevo en el espacio
17                while (true) {
18                     // Lee el objeto del espacio
19                     SpaceObject result = (SpaceObject)
20                         espacio.take(plantilla, null, Long.MAX_VALUE);
21                     result.incremento();
22                     // Escribe el objeto en el espacio
23                     espacio.write(result, null, Lease.FOREVER);
24                     Thread.sleep(1000);
25                }//fin while
26           }
27           catch (Exception ex) {
28                ex.printStackTrace();
29           }
30      }
31 } //fin class
```

Es especialmente interesante que la exclusión mutua es una característica interna de diseño del paradigma de espacios de objetos, ya que un objeto compartido sólo puede ser accedido por un participante a la vez. La referencia [Ahuja, Carrier, y Gelernter, 17] proporciona excelentes ejemplos que ilustran las aplicaciones que se pueden construir usando este paradigma.

A los lectores interesados en JavaSpaces se les recomienda consultar las referencias [java.sun.com, 18], [jiniworld.net, 19], [onjava.com, 20], y [Alter, 23].

Tendencias futuras

Este capítulo ha presentado un gran número de paradigmas avanzados de computación distribuida, muchos de los cuales se encuadran dentro del área de investigación. Existen verdaderamente más paradigmas y herramientas que irán surgiendo. Por ejemplo, *peer-to-peer* es un paradigma que recientemente ha recibido mucha atención; se han desarrollado varios sistemas y API para dar soporte a este paradigma, incluyendo el proyecto **JXTA** [jxta.org, 16]. Otro paradigma, **computación colaborativa**, usa el concepto de canales para permitir que procesos independientes colaboren en tiempo real; el entorno JSDT (*Java Shared Data Toolkit*) [java.sun.com, 21] se puede utilizar para crear aplicaciones de red, como pizarras compartidas o entornos de *chat*. También se puede utilizar para hacer presentaciones remotas, simulaciones compartidas, y distribuir fácilmente datos para trabajos en grupo.

Se espera que usted, como lector del libro, haya adquirido los conceptos básicos y comprensión que le permitirá explorar paradigmas nuevos con los que no esté familiarizado así como nuevas API de computación distribuida.

RESUMEN

Este capítulo ha proporcionado una visión general de los siguientes paradigmas avanzados de computación distribuida:

- El paradigma de Sistemas de Colas de Mensajes, que proporciona emisión y recepción de mensajes de forma asíncrona a través del uso de colas de mensajes. Un entorno que da soporte a este paradigma es *Java Message System* (JMS).
- El paradigma de agentes móviles, que da soporte al uso de programas transportables. A pesar de que los agentes móviles se pueden implementar directamente usando RMI, se recomienda el uso de entornos como Aglet, Concordia, y Grasshopper.
- El paradigma de servicios de red, el cual muestra la red como una federación de proveedores de servicios y de consumidores de los mismos. Una aplicación puede consumir uno o más recursos, según lo requiera. El protocolo SOAP (*Simple Object Access Protocol*) se utiliza para este paradigma. Jini de Java es un conjunto de herramientas muy sofisticado que hace uso de un concepto interesante que son los alquileres de servicios.
- El paradigma de espacios de objetos, el cual proporciona espacios lógicos donde los objetos pueden depositarse y retirarse por elementos colaboradores. JavaSpaces es un conjunto de herramientas que soportan este paradigma.
- Hay otros paradigmas y otras tecnologías que no se han cubierto, incluyendo *peer-to-peer* y computación colaborativa.

EJERCICIOS

1. Siga las instrucciones de [java.sun.com, 1] para instalar JMS (*Java Message System*) en su ordenador.
   a. Experimente con el ejemplo HolaMundo mostrado en las Figuras 12.4 y 12.5.
      — Compile los ficheros fuente.
      — Arranque un receptor, luego un emisor. Describa y explique el comportamiento.
      — Arranque dos receptores, luego un emisor. Describa y explique el comportamiento.
   b. Repita parte del experimento con el ejemplo punto-a-punto proporcionado en [java.sun.com, 1].
   c. Repita parte del experimento con el ejemplo publicación/suscripción proporcionado en [java.sun.com, 1].
2. Describa una aplicación que use una o más colas de mensajes punto-a-punto. Explique cómo la aplicación saca partido del uso de cola(s) de mensajes en términos de facilidad de implementación.
3. Describa una aplicación que use una o más colas de mensajes publicación/suscripción. Explique cómo la aplicación saca partido del uso de cola(s) de mensajes en términos de facilidad de implementación.
4. Siga las instrucciones del fichero README de la carpeta de ejemplos de agentes móviles para ejecutar la demo de agentes móviles mostrada en las Figuras de la 12.8 a las 12.12. Describa y explique el comportamiento.
5. Investigue el entorno de trabajo Aglets [trl.ibm.com, 6], un entorno para agentes móviles.
   a. ¿Cómo se proporciona transporte de código dentro del entorno Aglet?

b. ¿Qué otros servicios proporciona el entorno Aglet además del de transporte de código?

c. Descargue e instale el paquete, experimente con los ejemplos proporcionados.

6. Considere las siguientes aplicaciones listadas a continuación. Describa cómo cada uno de estos paradigmas (sistemas de colas de mensajes, agentes móviles y espacios de objetos) pueden aplicarse para desarrollar cada aplicación. Nótese que no se le pide que realice la implementación. Su descripción debe enmarcarse en términos del paradigma genérico, no de una API específica:

   Las aplicaciones a considerar son:

   a. Monitorización de la carga de mercancías a transportar por un vehículo en camino entre almacenes y tiendas.

   b. Una sala de *chat*.

   c. Una subasta *on-line*.

   d. Un servicio que permita a su universidad proporcionar información de los cursos a los estudiantes. La información se actualiza frecuentemente.

7. Investigue sobre JXTA [jxta.org, 16] y escriba un informe del soporte para el paradigma *peer-to-peer*. Incluya en su informe una descripción de la arquitectura, del API, y programas de ejemplo. ¿Dónde pondría este paradigma dentro de la jerarquía de paradigmas distribuidos?

8. Investigue sobre JSDT (*Java Shared Data Toolkit*) [pandonia.canberra.edu.au, 22] y escriba un informe sobre el paradigma al que da soporte. Incluya en su informe una descripción de la arquitectura, del API, y programas de ejemplo. ¿Dónde pondría este paradigma dentro de la jerarquía de paradigmas distribuidos?

REFERENCIAS

1. Java Message Service Tutorial, *http://java.sun.com/products/jms/tutorial/*
2. Microsoft MSMQ Home Page, *http://www.microsoft.com/msmq/default.htm*
3. IBM's MQ-Series, *http://www-4.ibm.com/software/ts/mqseries/*
4. Java™ 2 Platform, Enterprise Edition, *http://java.sun.com/j2ee/*
5. D. Lange y M. Oshima. "Seven Good Reasons for Mobile Agents." *Communications of the ACM*, March 1999.
6. IBM Aglet. *http://www.trl.ibm.com/aglets/*
7. Concordia. *http://www.concordiaagents.com/*
8. Grasshopper 2, *http://www.grasshopper.de/index.html*
9. Security in mobile agent systems, *http://mole.informatik.uni-stuttgart.de/security.html*
10. Jini™ Network Technology, *http://www.sun.com/jini/*
11. Keith Edwards. *Core Jini*. Upper Saddle River, NJ: Prentice Hall PTR, 2000.
12. Jan Newmarch's Guide to JINI Technologies, *http://jan.netcomp.monash.edu.au/java/jini/tutorial/Jini.xml*
13. Jini Planet, *http://www.kedwards.com/jini/*
14. Noel's Nuggets, *http://www.cc.gatech.edu/~kris/cs7470/nuggets/:* Jini tutorial, with examples and instructions.
15. Daniel D. Corkill. "Blackboard Systems." *AI Expert 6*, no. 9 (September 1991): 40–47.

16. Project JXTA, *http://www.jxta.org/*
17. Sudhir Ahuja, Nicholas Carriero, and David Gelernter. "Linda and Friends." *Computer*, August 1986: 26–34.
18. Eric Freeman, Susanne Jupfer, y Ken Arnold. *JavaSpaces: Principles, Patterns, and Practice*. Reading, MA: Addison-Wesley, 1999. *http://java.sun.com/docs/books/jini/javaspaces/*
19. The Nuts and Bolts of Compiling and Running JavaSpaces Programs, *http://www.jiniworld.net/document/javaspace/The Nuts and Bolts of Compiling and Running JavaSpaces(TM).htm*

Epílogo

Espero que hayas encontrado los materiales presentados en este libro tan interesantes como yo los encontré cuando, como estudiante, me introduje en el campo de la computación distribuida. Esto no significa, sin embargo, que este libro haga justicia al amplio rango de temas que existen dentro del espectro de la computación distribuida.

La computación distribuida es un apasionante campo, con nuevas ideas y tecnologías en constante desarrollo, y viejas ideas y tecnologías siendo redescubiertas. Es cierto que cualquier persona involucrada en este campo estará siempre retada con nuevos modelos, teorías y herramientas. Tengo la esperanza de que este libro haya contribuido a que comprendas los principios de la computación distribuida y que te haya preparado para una larga vida de aprendizaje dentro de este campo.

Existen muchas áreas que necesitan más atención. La tolerancia a fallos es un tema de gran importancia para cualquier investigador o practicante de la computación distribuida. Asimismo, no se puede desestimar la importancia del área de la seguridad. También está el tema de los algoritmos distribuidos, que yo misma enseño a los estudiantes más avanzados, muchos de los cuales lo encuentran particularmente estimulante. Te invito a que estudies estos campos en mayor profundidad a fin de que amplíes tus conocimientos en el campo de la computación distribuida.

M. L. Liu

Índice alfabético

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

Computación Distribuida
Fundamentos y Aplicaciones

Liu

El objetivo principal de este manual es dar a conocer los conceptos fundamentales de la intercomunicación entre procesos.

Características importantes:

- Está diseñado para introducir a los estudiantes universitarios en los fundamentos de la computación distribuida.
- Se centra en las capas más altas de la arquitectura de la computación basada en red, y más especificamente en los paradigmas y abstracciones de dicha computación.
- Incorpora temas conceptuales y prácticos, utilizando programas de ejemplo y ejercicios para ilustrar y reforzar los conceptos presentados.
- Combina la teoría y la práctica de la computación distribuida.
- Está diseñado para el aprendizaje con la experimentación: se presentan ejemplos de programación sobre los temas tratados y se incorporan actividades de laboratorio y ejercicios al final de cada capítulo.

PEARSON
Educación

www.pearsoneducacion.com

ISBN 978-84-782-9066-6