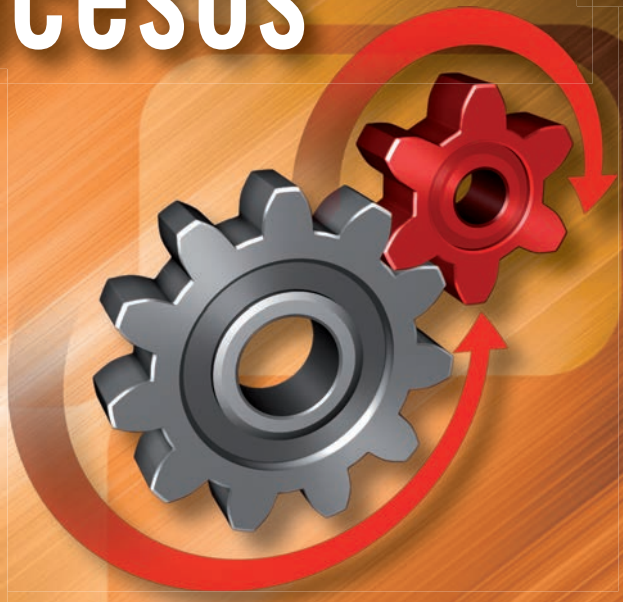
CF

GRADO SUPERIOR

CICLOS FORMATIVOS

R.D. 1538/2006

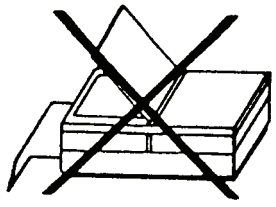
Programación de Servicios y Procesos



ALBERTO SÁNCHEZ CAMPOS
JESÚS MONTES SÁNCHEZ

Ra-Ma®

www.ra-ma.es/cf



La ley prohíbe
Copiar o Imprimir este libro

PROGRAMACIÓN DE SERVICIOS Y PROCESOS

© De la Edición Original en papel publicada por Editorial RA-MA
ISBN de Edición en Papel: 978-84-9964-240-6

Editado por:
RA-MA, S.A. Editorial y Publicaciones
Calle Jarama, 33, Polígono Industrial IGARSA
28860 PARACUELLOS DE JARAMA, Madrid
Teléfono: 91 658 42 80
Fax: 91 662 81 39
Correo electrónico: editorial@ra-ma.com
Internet: www.ra-ma.es y www.ra-ma.com

Maquetación: Gustavo San Román Borrueco
Diseño Portada: Antonio García Tomé

ISBN: 978-84-9964-391-5

E-Book desarrollado en España en Septiembre de 2014

Programación de Servicios y Procesos

ALBERTO SÁNCHEZ CAMPOS
JESÚS MONTES SÁNCHEZ

Ra-Ma®

Descarga de Material Adicional

Este E-book tiene disponible un material adicional que complementa el contenido del mismo.

Este material se encuentra disponible en nuestra página Web www.ra-ma.com.

Para descargarlo debe dirigirse a la ficha del libro de papel que se corresponde con el libro electrónico que Ud. ha adquirido. Para localizar la ficha del libro de papel puede utilizar el buscador de la Web.

Una vez en la ficha del libro encontrará un enlace con un texto similar a este:

"Descarga del material adicional del libro"

Pulsando sobre este enlace, el fichero comenzará a descargarse.

Una vez concluida la descarga dispondrá de un archivo comprimido. Debe utilizar un software descompresor adecuado para completar la operación. En el proceso de descompresión se le solicitará una contraseña, dicha contraseña coincide con los 13 dígitos del ISBN del libro de papel (incluidos los guiones).

Encontrará este dato en la misma ficha del libro donde descargó el material adicional.

Si tiene cualquier pregunta no dude en ponerse en contacto con nosotros en la siguiente dirección de correo: ebooks@ra-ma.com

A mi abuela Marina, por haber sido mi guía.
A mi chica, sin ella esto no hubiera sido posible.
A todos aquellos que me rodean, sin excepción:
hacen que sea quien soy.

Alberto Sánchez Campos

Índice

Introducción

Este libro surge con el propósito de acercar al lector a los aspectos más importantes que encierra la programación paralela para poder enfrentarse a los nuevos retos que supone la informática hoy en día. Como la informática se ha incorporado desde hace poco a nuestras vidas, aún no hemos tomado conciencia de cómo sacarle totalmente partido para aprovechar los nuevos avances en los ordenadores. Cuando ejecuta un ordenador, entran en juego muchas cosas que deben funcionar a la vez de forma cooperativa si queremos optimizar los tiempos de respuesta para el usuario. Por ejemplo, la inclusión de procesadores con varios núcleos (los cuales están copando el mercado actualmente) permite mejorar el rendimiento de los ordenadores, aunque sin el uso de paralelismo es imposible obtener mejoras. De la misma forma, la aparición de Internet ha hecho que los ordenadores estén conectados, y se pueda aprovechar no solo el poder computacional de un único ordenador, sino de varios de ellos en conjunto. Para saber cómo sacarle partido a todo ello salvando la problemática asociada, por ejemplo, en materia de seguridad, se ha escrito este libro.

Con tal propósito, sirve de apoyo para estudiantes del Módulo de **Programación de Servicios y Procesos** presente en el Ciclo Formativo de Grado Superior de **Desarrollo de Aplicaciones Multiplataforma**; así como para los Grados Universitarios de la rama Informática.

Para todo aquel que use este libro en el entorno de la enseñanza (Ciclos Formativos o Universidad), se ofrecen varias posibilidades: utilizar los conocimientos aquí expuestos para inculcar aspectos genéricos de la programación de procesos y servicios o simplemente centrarse en preparar a fondo alguno de ellos. La extensión de los contenidos aquí incluidos hace imposible su desarrollo completo en la mayoría de los casos.

Ra-Ma pone a disposición de los profesores una guía didáctica para el desarrollo del tema que incluye las soluciones a los ejercicios expuestos en el texto. Puede solicitarla a *editorial@ra-ma.com*, acreditándose como docente y siempre que el libro sea utilizado como texto base para impartir las clases.

Esperamos que el libro os sea útil o, al menos, entretenido.

1 Programación de procesos

OBJETIVOS DEL CAPÍTULO

- ✓ Comprender de los conceptos básicos del funcionamiento de los sistemas en lo relativo a la ejecución de diferentes programas.
- ✓ Comprender el concepto de concurrencia y cómo el sistema puede proporcionar multiprogramación al usuario.
- ✓ Entender las políticas de planificación del sistema para proporcionar multiprogramación y multitarea.
- ✓ Familiarizarse con la programación de procesos entendiendo sus principios y formas de aplicación.

1.1 CONCEPTOS BÁSICOS

Para poder empezar a entender cómo se ejecutan varios programas a la vez, es imprescindible adquirir ciertos conceptos.

- **Programa**: se puede considerar un programa a toda la información (tanto código como datos) almacenada en disco de una aplicación que resuelve una necesidad concreta para los usuarios.
- **Proceso**: cuando un programa se ejecuta, podremos decir de manera muy simplificada que es un proceso. En definitiva, puede definirse "proceso" como un programa en ejecución. Este concepto no se refiere únicamente al código y a los datos, sino que incluye todo lo necesario para su ejecución. Esto incluye tres cosas:
  - *Un contador del programa*: algo que indique por dónde se está ejecutando.
  - *Una imagen de memoria*: es el espacio de memoria que el proceso está utilizando.
  - *Estado del procesador*: se define como el valor de los registros del procesador sobre los cuales se está ejecutando.

¿SABÍAS QUE...?

La memoria principal almacena toda la información de un programa (instrucciones y datos) mientras se está procesando en la CPU. Todos los programas y datos antes de ser procesados por el procesador han de ser almacenados en esta memoria. Para que un proceso se pueda ejecutar tiene que estar en memoria toda la información que necesita. En este sentido, el tipo y cantidad de memoria, pueden influir decisivamente en la velocidad del ordenador. Si el tamaño de memoria es pequeño, se deberán traer de donde estén almacenados los programas (por ejemplo, el disco duro) más veces los datos del programa necesarios, ralentizando el ordenador.

Es importante destacar que los procesos son entidades independientes, aunque ejecuten el mismo programa. De tal forma, pueden coexistir dos procesos que ejecuten el mismo programa, pero con diferentes datos (es decir, con distintas imágenes de memoria) y en distintos momentos de su ejecución (con diferentes contadores de programa).

EJEMPLO 1.1

Se pueden tener, por ejemplo, dos instancias del programa Microsoft Word ejecutándose a la vez, modificando cada una un fichero diferente. Para que los datos de uno no interfieran con los del otro, cada proceso se ejecuta en su propio espacio de direcciones en memoria permitiendo independencia entre los procesos.

- **Ejecutable**: un fichero ejecutable contiene la información necesaria para crear un proceso a partir de los datos almacenados de un programa. Es decir, llamaremos "ejecutable" al fichero que permite poner el programa en ejecución como proceso.
- **Sistema operativo**: programa que hace de intermediario entre el usuario y las aplicaciones que utiliza y el hardware del ordenador. Entre sus objetivos, se pueden destacar:
  - *Ejecutar los programas del usuario*. Es el encargado de crear los procesos a partir de los ejecutables de los programas y de gestionar su ejecución para evitar errores y mejorar el uso del computador.
  - *Hacer que el computador sea cómodo de usar*. Hace de interfaz entre el usuario y los recursos del ordenador, permitiendo el acceso tanto a ficheros y memoria como a dispositivos hardware. Esta serie de abstracciones permiten al programador acceder a los recursos hardware de forma sencilla
  - *Utilizar los recursos del computador de forma eficiente*. Los recursos del ordenador son compartidos tanto por los programas como por los diferentes usuarios. El sistema operativo es el encargado de repartir los recursos en función de sus políticas a aplicar.

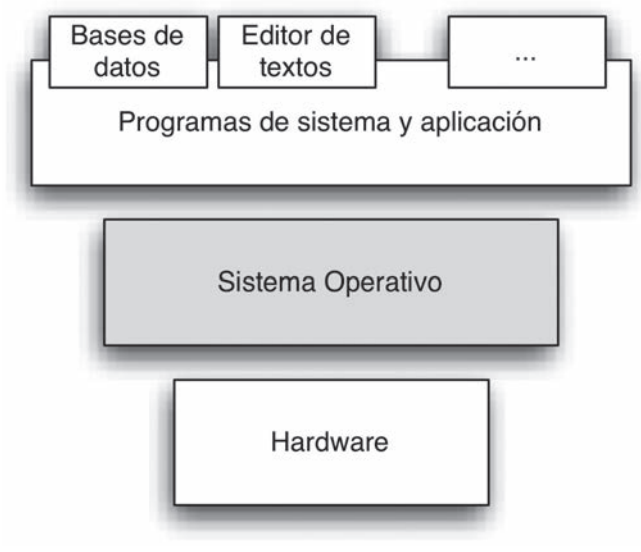


Figura 1.1. *Sistema operativo como intermediario entre aplicaciones y hardware*

- **Demonio**: proceso no interactivo que está ejecutándose continuamente en segundo plano, es decir, es un proceso controlado por el sistema sin ninguna intermediación del usuario. Suelen proporcionar un servicio básico para el resto de procesos.

¿SABÍAS QUE...?

La palabra "demonio" fue usada en 1963 por primera vez en el área de la informática, para denominar a un proceso que realizaba en segundo plano *backups* en unas cintas. El nombre proviene de Fernando J. Corbató, director del proyecto MAC del MIT basándose en el famoso demonio de James Maxwell. Este demonio era un elemento biológico que residía en medio de un recipiente dividido en dos, lleno de moléculas. El demonio se encargaba de permitir, dependiendo de la velocidad de la molécula, que estas pasaran de un lado al otro. Los demonios actúan de forma similar ya que están continuamente vigilando y realizando acciones en función de las necesidades del sistema. En Windows, los demonios se denominan "servicios", estableciendo una clara relación entre los procesos no interactivos, y los servicios que se estudiarán en el Capítulo 4.

EJEMPLO 1.2

El proceso nulo del sistema o hasta el propio recolector de basura (*garbage collector*) de Java pueden ser considerados como demonios.

Una vez llegados a este punto, ya han sido explicados los conceptos claves para poder avanzar con la programación de procesos.

1.2 PROGRAMACIÓN CONCURRENTE

La **computación concurrente** permite la posibilidad de tener en ejecución al mismo tiempo múltiples tareas interactivas. Es decir, permite realizar varias cosas al mismo tiempo, como escuchar música, visualizar la pantalla del ordenador, imprimir documentos, etc. Pensad en todo el tiempo que perderíamos si todas esas tareas se tuvieran que realizar una tras otra. Dichas tareas se pueden ejecutar en:

- Un único procesador (multiprogramación). En este caso, aunque para el usuario parezca que varios procesos se ejecutan al mismo tiempo, si solamente existe un único procesador, solamente un proceso puede estar en un momento determinado en ejecución. Para poder ir cambiando entre los diferentes procesos, el sistema operativo se encarga de cambiar el proceso en ejecución después de un período corto de tiempo (del orden de milisegundos). Esto permite que en un segundo se ejecuten múltiples procesos, creando en el usuario la percepción de que múltiples programas se están ejecutando al mismo tiempo.

EJEMPLO 1.3

A la vez que se modifica un documento en Microsoft Word, se puede estar escuchando música en iTunes y navegando a través de la red con Google Chrome. Si los cambios entre los procesos se producen lo suficientemente rápido, parece que todo se ejecuta al mismo tiempo, y así la música se escucha sin cortes.

Este concepto se denomina **programación concurrente**. La programación concurrente no mejora el tiempo de ejecución global de los programas ya que se ejecutan intercambiando unos por otros en el procesador. Sin embargo, permite que varios programas parezca que se ejecuten al mismo tiempo.

- Varios núcleos en un mismo procesador (multitarea). La existencia de varios núcleos o *cores* en un ordenador es cada vez mayor, apareciendo en *Dual Cores, Quad Cores*, en muchos de los modelos *i3, i5* e *i7*, etc. Cada núcleo podría estar ejecutando una instrucción diferente al mismo tiempo. El sistema operativo, al igual que para un único procesador, se debe encargar de planificar los trabajos que se ejecutan en cada núcleo y cambiar unos por otros para generar multitarea. En este caso todos los *cores* comparten la misma memoria por lo que es posible utilizarlos de forma coordinada mediante lo que se conoce por **programación paralela**.
- La programación paralela permite mejorar el rendimiento de un programa si este se ejecuta de forma paralela en diferentes núcleos ya que permite que se ejecuten varias instrucciones a la vez. Cada ejecución en cada *core* será una tarea del mismo programa pudiendo cooperar entre sí. El concepto de "tarea" (o "hilo de ejecución") se explicará en más profundidad a lo largo del Capítulo 2. Y, por supuesto, se puede utilizar conjuntamente con la programación concurrente, permitiendo al mismo tiempo multiprogramación.

EJEMPLO 1.4

Mientras se está escribiendo un documento en Microsoft Word, al mismo tiempo se puede estar ejecutando una tarea del mismo programa que comprueba la ortografía.

- Varios ordenadores distribuidos en red. Cada uno de los ordenadores tendrá sus propios procesadores y su propia memoria. La gestión de los mismos forma parte de lo que se denomina **programación distribuida**.

  La programación distribuida posibilita la utilización de un gran número de dispositivos (ordenadores) de forma paralela, lo que permite alcanzar elevadas mejoras en el rendimiento de la ejecución de programas distribuidos. Sin embargo, como cada ordenador posee su propia memoria, imposibilita que los procesos puedan comunicarse compartiendo memoria, teniendo que utilizar otros esquemas de comunicación más complejos y costosos a través de la red que los interconecte. Se explicará en más profundidad en el Capítulo 3.

1.3 FUNCIONAMIENTO BÁSICO DEL SISTEMA OPERATIVO

La parte central que realiza la funcionalidad básica del sistema operativo se denomina *kernel*. Es una parte software pequeña del sistema operativo, si la comparamos con lo necesario para implementar su interfaz (y más hoy en día que es muy visual). A todo lo demás del sistema se le denomina **programas del sistema**. El *kernel* es el responsable de gestionar los recursos del ordenador, permitiendo su uso a través de llamadas al sistema.

En general, el *kernel* del sistema funciona en base a interrupciones. Una **interrupción** es una suspensión temporal de la ejecución de un proceso, para pasar a ejecutar una rutina que trate dicha interrupción. Esta rutina será dependiente del sistema operativo. Es importante destacar que mientras se está atendiendo una interrupción, se deshabilita la llegada de nuevas interrupciones. Cuando finaliza la rutina, se reanuda la ejecución del proceso en el mismo lugar donde se quedó cuando fue interrumpido.

Es decir, el sistema operativo no es un proceso demonio propiamente dicho que proporcione funcionalidad al resto de procesos, sino que él solo se ejecuta respondiendo a interrupciones. Cuando salta una interrupción se transfiere el control a la rutina de tratamiento de la interrupción. Así, las rutinas de tratamiento de interrupción pueden ser vistas como el código propiamente dicho del *kernel*.

Las **llamadas al sistema** son la interfaz que proporciona el *kernel* para que los programas de usuario puedan hacer uso de forma segura de determinadas partes del sistema. Los errores de un programa podrían afectar a otros programas o al propio sistema operativo, por lo que para asegurar su ejecución de la forma correcta, el sistema implementa una interfaz de llamadas para evitar que ciertas instrucciones peligrosas sean ejecutadas directamente por programas de usuario.

El **modo dual** es una característica del hardware que permite al sistema operativo protegerse. El procesador tiene dos modos de funcionamiento indicados mediante un bit:

- Modo usuario (1). Utilizado para la ejecución de programas de usuario.
- Modo *kernel* (0), también llamado "modo supervisor" o "modo privilegiado". Las instrucciones del procesador más delicadas solo se pueden ejecutar si el procesador está en modo *kernel*.

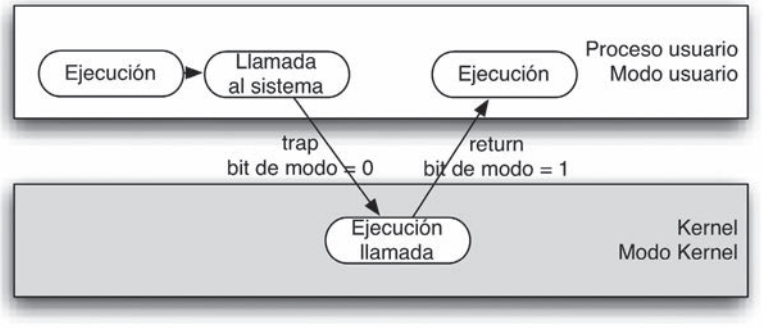


Figura 1.2. *Ejecución de llamadas al sistema*

Una llamada al sistema en un programa de usuario genera una interrupción (en este caso denominada *trap*). La interfaz de llamadas al sistema lanza la interrupción correspondiente que trata el sistema de la forma adecuada.

¿SABÍAS QUE...?

Las llamadas al sistema suelen estar escritas en un lenguaje de bajo nivel (C o C++), el cual es más cercano al funcionamiento de la arquitectura del ordenador. Aun así, el programador no suele utilizar las llamadas al sistema directamente, sino que utiliza una API de más alto nivel. Las API más comunes son Win32, API para sistemas Microsoft Windows, y la API POSIX, para sistemas tipo UNIX, incluyendo GNU Linux y Mac OS.

1.4 PROCESOS

Como se ha dicho anteriormente, el sistema operativo es el encargado de poner en ejecución y gestionar los procesos. Para su correcto funcionamiento, a lo largo de su ciclo de vida, los procesos pueden cambiar de estado. Es decir, a medida que se ejecuta un proceso, dicho proceso pasará por varios estados. El cambio de estado también se producirá por la intervención del sistema operativo.

¿SABÍAS QUE...?

Los primeros sistemas operativos como MS-DOS (anterior a Microsoft Windows) funcionaban con un único proceso (proceso residente). Los programas, o funcionaban en forma exclusiva en la máquina o no funcionaban. Con la creciente sofisticación de las aplicaciones y la creciente demanda de ordenadores personales, especialmente en lo relativo a aplicaciones gráficas y de red, los sistemas operativos multiproceso y multitarea se volvieron algo común.

1.4.1 ESTADO DE UN PROCESO

Los estados de un proceso son:

- **Nuevo**: el proceso está siendo creado a partir del fichero ejecutable.
- **Listo**: el proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse. El planificador del sistema operativo es el responsable de seleccionar que proceso está en ejecución, por lo que es el que indica cuando el proceso pasa a ejecución.
- **En ejecución**: el proceso se está ejecutando. El sistema operativo utiliza el mecanismo de interrupciones para controlar su ejecución. Si el proceso necesita un recurso, incluyendo la realización de operaciones de

entrada/salida (E/S), llamará a la llamada del sistema correspondiente. Si un proceso en ejecución se ejecuta durante el tiempo máximo permitido por la política del sistema, salta un temporizador que lanza una interrupción. En este último caso, si el sistema es de tiempo compartido, lo para y lo pasa al estado *Listo*, seleccionando otro proceso para que continúe su ejecución.

- **Bloqueado**: el proceso está bloqueado esperando que ocurra algún suceso (esperando por una operación de E/S, bloqueado para sincronizarse con otros procesos, etc.). Cuando ocurre el evento que lo desbloquea, el proceso no pasa directamente a ejecución sino que tiene que ser planificado de nuevo por el sistema.
- **Terminado**: el proceso ha finalizado su ejecución y libera su imagen de memoria. Para terminar un proceso, el mismo debe llamar al sistema para indicárselo o puede ser el propio sistema el que finalice el proceso mediante una excepción (una interrupción especial).

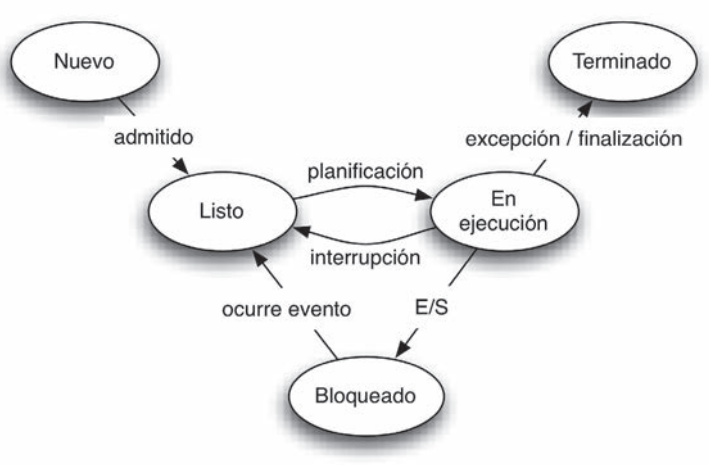


Figura 1.3. *Estados de un proceso*

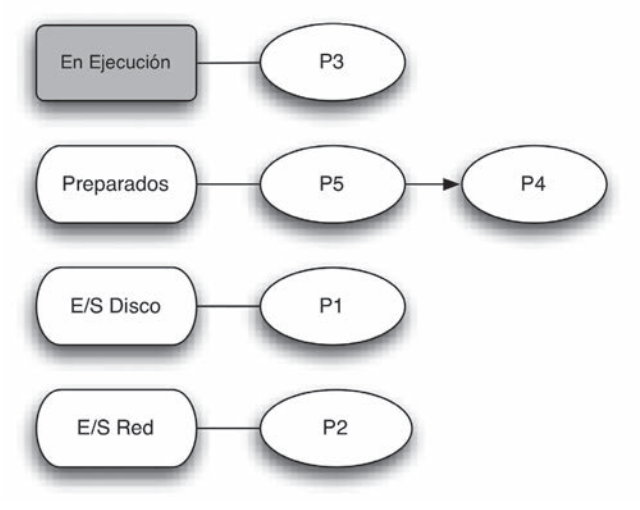
1.4.2 COLAS DE PROCESOS

Uno de los objetivos del sistema operativo es la multiprogramación, es decir, admitir varios procesos en memoria para maximizar el uso del procesador. Esto funciona ya que los procesos se irán intercambiando el uso del procesador para su ejecución de forma concurrente. Para ello, el sistema operativo organiza los procesos en varias colas, migrándolos de unas colas a otras:

- Una **cola de procesos** que contiene todos los procesos del sistema.
- Una **cola de procesos preparados** que contiene todos los procesos listos esperando para ejecutarse.
- Varias **colas de dispositivo** que contienen los procesos que están a la espera de alguna operación de E/S.

ACTIVIDADES 1.1

Supongamos que varios procesos (1, 2, 3, 4 y 5) se están ejecutando concurrentemente en un ordenador. El proceso 1 se está ejecutando cuando realiza una petición de E/S al disco. En ese momento, el proceso 2 pasa a ejecución, haciendo otra operación de E/S, en este caso por la tarjeta de red. El proceso 3 pasa a ejecutarse. En ese momento, las colas de procesos serán las siguientes:



En ese momento supongamos que el proceso 3 realiza una petición de E/S teniendo que esperar por el disco duro. Dibuja el estado de las colas correspondientes e identifica cuál sería el proceso que se ejecutará utilizando la filosofía FIFO para sacar los procesos de las colas.

¿SABÍAS QUE...?

Una pila es una estructura de datos en la que sus elementos se almacenan de forma que el último que entra es el primero que sale. Esta forma de almacenamiento se denomina LIFO (del inglés *Last In First Out*) y se basa en dos operaciones: *push*, que introduce un elemento en la pila, y *pop,* que desapila el último elemento introducido. En este sentido, una pila puede verse como un saco. El primer elemento que se introduce va al fondo del saco y, por tanto, es el último que sale.

En cambio, una cola es una estructura de datos que sigue un almacenamiento de tipo FIFO (del inglés *First In First Out*), donde el primer elemento que se almacena es el primero que sale. La operación *push* se realiza por un extremo de la cola mientras *pop* se realiza por el otro. Esto se puede comparar con la típica cola de espera en la caja de un supermercado. El primero que llega es al primero que atienden y mientras se le atiende, el resto que va llegando se coloca al final de la cola.

1.4.3 PLANIFICACIÓN DE PROCESOS

Para gestionar las colas de procesos, es necesario un planificador de procesos. El planificador es el encargado de seleccionar los movimientos de procesos entre las diferentes colas. Existen dos tipos de planificación:

- **A corto plazo**: selecciona qué proceso de la cola de procesos preparados pasará a ejecución. Se invoca muy frecuentemente (del orden de milisegundos, cuando se produce un cambio de estado del proceso en ejecución) por lo que debe ser muy rápido en la decisión. Esto implica que los algoritmos sean muy sencillos:
  - *Planificación sin desalojo o cooperativa*. Únicamente se cambia el proceso en ejecución si dicho proceso se bloquea o termina.
  - *Planificación apropiativa*. Además de los casos de la planificación cooperativa, se cambia el proceso en ejecución si en cualquier momento en que un proceso se está ejecutando, otro proceso con mayor prioridad se puede ejecutar. La aparición de un proceso más prioritario se puede deber tanto al desbloqueo del mismo como a la creación de un nuevo proceso.
  - *Tiempo compartido*: cada cierto tiempo (llamado *cuanto*) se desaloja el proceso que estaba en ejecución y se selecciona otro proceso para ejecutarse. En este caso, todas las prioridades de los procesos se consideran iguales.
- **A largo plazo**: selecciona qué procesos nuevos deben pasar a la cola de procesos preparados. Se invoca con poca frecuencia, por lo que puede tomarse más tiempo en tomar la decisión. Controla el grado de multiprogramación (número de procesos en memoria).

1.4.4 CAMBIO DE CONTEXTO

Cuando el procesador pasa a ejecutar otro proceso, lo cual ocurre muy frecuentemente, el sistema operativo debe guardar el contexto del proceso actual y restaurar el contexto del proceso que el planificador a corto plazo ha elegido ejecutar. La salvaguarda de la información del proceso en ejecución se produce cuando hay una interrupción.

Se conoce como **contexto** a:

- Estado del proceso.
- Estado del procesador: valores de los diferentes registros del procesador.
- Información de gestión de memoria: espacio de memoria reservada para el proceso.

El cambio de contexto es tiempo perdido, ya que el procesador no hace trabajo útil durante ese tiempo. Únicamente es tiempo necesario para permitir la multiprogramación y su duración depende de la arquitectura en concreto del procesador.

1.5 GESTIÓN DE PROCESOS

1.5.1 ÁRBOL DE PROCESOS

El sistema operativo es el encargado de crear y gestionar los nuevos procesos siguiendo las directrices del usuario. Así, cuando un usuario quiere abrir un programa, el sistema operativo es el responsable de crear y poner en ejecución el proceso correspondiente que lo ejecutará. Aunque el responsable del proceso de creación es el sistema operativo, ya que es el único que puede acceder a los recursos del ordenador, el nuevo proceso se crea siempre por petición de otro proceso. La puesta en ejecución de un nuevo proceso se produce debido a que hay un proceso en concreto que está pidiendo su creación en su nombre o en nombre del usuario.

EJEMPLO 1.5

Si un usuario pulsa en el icono de Adobe Photoshop para arrancar una instancia del mismo, la interfaz gráfica es la que hace la petición para crear el nuevo proceso de Photoshop.

En este sentido, cualquier proceso en ejecución siempre depende del proceso que lo creó, estableciéndose un vínculo entre ambos. A su vez, el nuevo proceso puede crear nuevos procesos, formándose lo que se denomina un **árbol de procesos**. Cuando se arranca el ordenador, y se carga en memoria el *kernel* del sistema a partir de su imagen en disco, se crea el proceso inicial del sistema. A partir de este proceso, se crea el resto de procesos de forma jerárquica, estableciendo padres, hijos, abuelos, etc.

Para identificar a los procesos, los sistemas operativos suelen utilizar un **identificador de proceso** (*process identifier* [PID]) unívoco para cada proceso. La utilización del PID es básica a la hora de gestionar procesos, ya que es la forma que tiene el sistema de referirse a los procesos que gestiona.

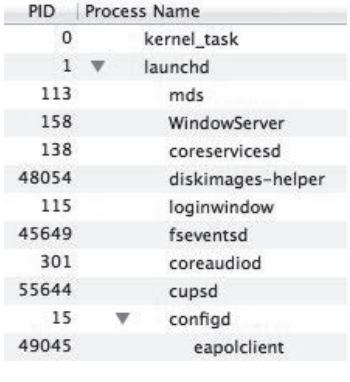


Figura 1.4. *Árbol de procesos de Mac OS X. Se puede observar cómo el proceso principal del cual cuelgan el resto de procesos es el proceso con PID 0*

ACTIVIDADES 1.2

- El Administrador de tareas en Microsoft Windows —llamado "Monitor de actividad" en Mac OS y "Monitor del sistema" en Ubuntu Linux— permite gestionar los procesos del sistema operativo. En él se puede ver qué procesos se están ejecutando con su uso de procesador y memoria.

  Utiliza el Administrador de tareas para obtener los procesos del sistema.

- Dibuja el árbol de procesos correspondiente para tu propio ordenador en ese momento.

1.5.2 OPERACIONES BÁSICAS CON PROCESOS

Siguiendo el vínculo entre procesos establecido en el árbol de procesos, el proceso creador se denomina **padre** y el proceso creado se denomina **hijo**. A su vez, los hijos pueden crear nuevos hijos. A la operación de creación de un nuevo proceso la denominaremos *create*.

Cuando se crea un nuevo proceso tenemos que saber que padre e hijo se ejecutan concurrentemente. Ambos procesos comparten la CPU y se irán intercambiando siguiendo la política de planificación del sistema operativo para proporcionar multiprogramación. Si el proceso padre necesita esperar hasta que el hijo termine su ejecución para poder continuar la suya con los resultados obtenidos por el hijo, puede hacerlo mediante la operación *wait*.

Como se ha visto al inicio del capítulo, los procesos son independientes y tienen su propio espacio de memoria asignado, llamado **imagen de memoria**. Padres e hijos son procesos y, aunque tengan un vínculo especial, mantienen esta restricción. Ambos usan espacios de memoria independientes. En general, parece que el hijo ejecuta un programa diferente al padre, pero en algunos sistemas operativos esto no tiene por qué ser así. Por ejemplo, mientras que en sistemas tipo Windows existe una función *createProcess()* que crea un nuevo proceso a partir de un programa distinto al que está en ejecución, en sistemas tipo UNIX, la operación a utilizar es *fork()*, que crea un proceso hijo con un duplicado del espacio de direcciones del padre, es decir, un duplicado del programa que se ejecuta desde la misma posición. Sin embargo, en ambos casos, los padres e hijos (aunque sean un duplicado en el momento de la creación en sistemas tipos UNIX) son independientes y las modificaciones que uno haga en su espacio de memoria, como escritura de variables, no afectarán al otro.

Como padre e hijo tienen espacios de memoria independientes, pueden compartir recursos para intercambiarse información. Estos recursos pueden ir desde ficheros abiertos hasta zonas de memoria compartida. La **memoria compartida** es una región de memoria a la que pueden acceder varios procesos cooperativos para compartir información. Los procesos se comunican escribiendo y leyendo datos en dicha región. El sistema operativo solamente interviene a la hora de crear y establecer los permisos de qué procesos pueden acceder a dicha zona. Los procesos son los responsables del formato de los datos compartidos y de su ubicación.

Al terminar la ejecución de un proceso, es necesario avisar al sistema operativo de su terminación para que de esta forma el sistema libere si es posible los recursos que tenga asignados. En general, es el propio proceso el que le indica al sistema operativo mediante una operación denominada *exit* que quiere terminar, pudiendo aprovechar para mandar información respecto a su finalización al proceso padre en ese momento.

El proceso hijo depende tanto del sistema operativo como del proceso padre que lo creó. Así, el padre puede terminar la ejecución de un proceso hijo cuando crea conveniente. Entre estos motivos podría darse que el hijo excediera el uso de algunos recursos o que la funcionalidad asignada al hijo ya no sea necesaria por algún motivo.

Para ello puede utilizar la operación *destroy*. Esta relación de dependencia entre padre e hijo, lleva a casos como que si el padre termina, en algunos sistemas operativos no se permita que sus hijos continúen la ejecución, produciéndose lo que se denomina "terminación en cascada".

En definitiva, cada sistema operativo tiene unas características únicas, y la gestión de los procesos es diferente en cada uno de ellos. Por simplificación y portabilidad, evitando así depender del sistema operativo sobre el cual se esté ejecutando hemos decidido explicar la gestión de procesos para la **máquina virtual de Java** (*Java Virtual Machine* [JVM]). JVM es un entorno de ejecución ligero y gratuito multiplataforma que permite la ejecución de binarios o *bytecode* del lenguaje de programación Java sobre cualquier sistema operativo, salvando las diferencias entre ellos. La idea que hay detrás de ello se conoce como *Write Once, Run Anywhere* ("escribe una vez y ejecuta en cualquier lugar", entendiendo "cualquier lugar" como plataforma o sistema operativo). Las próximas secciones del capítulo estarán basadas por tanto en la utilización de Java.

¿SABÍAS QUE...?

Java es un lenguaje de programación orientado a objetos concurrente, de propósito general y multiplataforma creado por Sun Microsystems. Es un lenguaje de alto nivel, es decir, sus instrucciones son cercanas al lenguaje natural, facilitando la comprensión del código al programador. Sin embargo, sus instrucciones son lejanas al hardware no pudiendo aprovecharse de la arquitectura física específica subyacente por lo que ofrece peor rendimiento y tiene menores capacidades que lenguajes de bajo nivel como C. Las aplicaciones de Java son traducidas a *bytecode*, el cual se puede ejecutar en cualquier JVM sin importar la arquitectura del ordenador donde está corriendo. Esto permite su ejecución sin tener que volver a compilar en diferentes plataformas. Se puede encontrar más información sobre Java en la página *http://www.java.com/es/about/*

Respecto a la utilización de Java, tenemos que saber que los procesos padre e hijo en la JVM no tienen por qué ejecutarse de forma concurrente. Además, no se produce terminación en cascada, pudiendo sobrevivir los hijos a su padre ejecutándose de forma asíncrona. Por último, hay que tener en cuenta que puede no funcionar bien para procesos especiales en ciertas plataformas nativas (por ejemplo, utilización de ventanas nativas en MS-DOS/Windows, *shell scripts* en GNU Linux/UNIX/Mac OS, etc.).

1.5.2.1 Creación de procesos (operación create)

La clase que representa un proceso en Java es la clase *Process*. Los métodos de *ProcessBuilder.start()* y *Runtime.exec()* crean un proceso nativo en el sistema operativo subyacente donde se está ejecutando la JVM y devuelven una objeto Java de la clase *Process* que puede ser utilizado para controlar dicho proceso.

- **Process ProcessBuilder.start**(): inicia un nuevo proceso utilizando los atributos indicados en el objeto. El nuevo proceso ejecuta el comando y los argumentos indicados en el método **command**(), ejecutándose en el directorio de trabajo especificado por **directory**(), utilizando las variables de entono definidas en **environment**().
- **Process Runtime.exec**(**String**[] **cmdarray**, **String**[] **envp**, **File dir**): ejecuta el comando especificado y argumentos en *cmdarray* en un proceso hijo independiente con el entorno *envp* y el directorio de trabajo especificado en *dir*.

Ambos métodos comprueban que el comando a ejecutar es un comando o ejecutable válido en el sistema operativo subyacente sobre el que ejecuta la JVM. El ejecutable se ha podido obtener mediante la compilación de código en cualquier lenguaje de programación. Al final, crear un nuevo proceso depende del sistema operativo en concreto que esté ejecutando por debajo de la JVM. En este sentido, pueden ocurrir múltiples problemas, como:

- ✓ No encuentra el ejecutable debido a la ruta indicada.
- ✓ No tener permisos de ejecución.
- ✓ No ser un ejecutable válido en el sistema.
- ✓ Etc.

En la mayoría de los casos, se lanza una excepción dependiente del sistema en concreto, pero siempre será una subclase de *IOException*.

EJEMPLO 1.6

Creación de un proceso utilizando *ProcessBuilder*:

```
import java.io.IOException;
import java.util.Arrays;

public class RunProcess {

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

      if (args.length <= 0) {
           System.err.println("Se necesita un programa a ejecutar");
                System.exit(-1);
      }

     ProcessBuilder pb = new ProcessBuilder(args);
     try {
           Process process = pb.start();
           int retorno = process.waitFor();
           System.out.println("La ejecución de " +
                      Arrays.toString(args) + " devuelve " +  retorno);
      }catch(IOException ex){
            System.err.println("Excepción de E/S!!");
                   System.exit(-1);
     }catch(InterruptedException ex){
           System.err.println("El proceso hijo finalizó
                      de forma incorrecta");
           System.exit(-1);
     }
  }
}
```

1.5.2.2 Terminación de procesos (operación destroy)

Un proceso puede terminar de forma abrupta un proceso hijo que creó. Para ello el proceso padre puede ejecutar la operación *destroy*. Esta operación elimina el proceso hijo indicado liberando sus recursos en el sistema operativo subyacente. En caso de Java, los recursos correspondientes los eliminará el *garbage collector* cuando considere.

¿SABÍAS QUE...?

El Administrador de tareas también permite eliminar procesos, siendo el propio sistema operativo en nombre del usuario (el cual puede convertirse en administrador) el encargado de ejecutar la operación *destroy*.

Si no se fuerza la finalización de la ejecución del proceso hijo de forma anómala, el proceso hijo realizará su ejecución completa terminando y liberando sus recursos al finalizar. Esto se produce cuando el hijo realiza la operación *exit* para finalizar su ejecución.

EJEMPLO 1.7

Creación de un proceso mediante *Runtime* para después destruirlo.

```
import java.io.IOException;

public class RuntimeProcess {

    public static void main(String[] args) {

        if (args.length <= 0) {
                System.err.println("Se necesita un programa a ejecutar");
                System.exit(-1);
        }

        Runtime runtime = Runtime.getRuntime();
        try {
             Process process = runtime.exec(args);
             process.destroy();
        }catch(IOException ex){
                System.err.println("Excepción de E/S!!");
                System.exit(-1);
        }
    }
}
```

1.6 COMUNICACIÓN DE PROCESOS

Es importante recordar que un proceso es un programa en ejecución y, como cualquier programa, recibe información, la transforma y produce resultados. Esta acción se gestiona a través de:

- La **entrada estándar** (*stdin*): lugar de donde el proceso lee los datos de entrada que requiere para su ejecución. No se refiere a los parámetros de ejecución del programa. Normalmente suele ser el teclado, pero podría recibirlos de un fichero, de la tarjeta de red o hasta de otro proceso, entre otros sitios. La lectura de datos a lo largo de un programa (por ejemplo mediante *scanf* en C) leerá los datos de su entrada estándar.
- La **salida estándar** (*stdout*): sitio donde el proceso escribe los resultados que obtiene. Normalmente es la pantalla, aunque podría ser, entre otros, la impresora o hasta otro proceso que necesite esos datos como entrada. La escritura de datos que se realice en un programa (por ejemplo mediante *printf* en C o *System.out.println* en Java) se produce por la salida estándar.
- La **salida de error** (*stderr*): sitio donde el proceso envía los mensajes de error. Habitualmente es el mismo que la salida estándar, pero puede especificarse que sea otro lugar, por ejemplo un fichero para identificar más fácilmente los errores que ocurren durante la ejecución.

La utilización de *System.out* y *System.err* en Java se puede ver como un ejemplo de utilización de estas salidas.

En la mayoría de los sistemas operativos, estas entradas y salidas en proceso hijo son una copia de las mismas entradas y salidas que tuviera su padre. De tal forma que si se llama a la operación *create* dentro de un proceso que lee de un fichero y muestra la salida estándar por pantalla, su hijo correspondiente leerá del mismo fichero y escribirá en pantalla. En Java, en cambio, el proceso hijo creado de la clase *Process* no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente. Todas sus salidas y entradas de información (*stdin*, *stdout* y *stderr*) se redirigen al proceso padre a través de los siguientes flujos de datos o *streams:*

- ***OutputStream***: flujo de salida del proceso hijo. El *stream* está conectado por un *pipe* a la entrada estándar (*stdin*) del proceso hijo.
- ***InputStream***: flujo de entrada del proceso hijo. El *stream* está conectado por un *pipe* a la salida estándar (*stdout*) del proceso hijo.
- ***ErrorStream***: flujo de error del proceso hijo. El *stream* está conectado por un *pipe* a la salida estándar (*stderr*) del proceso hijo. Sin embargo, hay que saber que, por defecto, para la JVM, *stderr* está conectado al mismo sitio que *stdout*.

  Si se desea tenerlos separados, lo que permite identificar errores de forma más sencilla, se puede utilizar el método *redirectErrorStream(boolean)* de la clase *ProcessBuilder*. Si se pasa un valor *true* como parámetro, los flujos de datos correspondientes a *stderr* y *stdout* en la JVM serán diferentes y representarán la salida estándar y la salida de error del proceso de forma correspondiente.

Utilizando estos *streams*, el proceso padre puede enviarle datos al proceso hijo y recibir los resultados de salida que este genere comprobando los errores.

Hay que tener en cuenta que en algunos sistemas operativos, el tamaño de los *buffers* de entrada y salida que corresponde a *stdin* y *stdout* está limitado. En este sentido, un fallo al leer o escribir en los flujos de entrada o salida del proceso hijo puede provocar que el proceso hijo se bloquee. Por eso, en Java se suele realizar la comunicación padre-hijo a través de un *buffer* utilizando los *streams* vistos.

EJEMPLO 1.8

Comunicación de procesos utilizando un *buffer*

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Arrays;

public class ComunicationBetweenProcess {

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

          Process process = new ProcessBuilder(args).start();
          InputStream is = process.getInputStream();
          InputStreamReader isr = new InputStreamReader(is);
          BufferedReader br = new BufferedReader(isr);
          String line;

          System.out.println("Salida del proceso " +
                    Arrays.toString(args) + ":");

          while ((line = br.readLine()) != null) {
               System.out.println(line);
          }

    }
}
```

Por último, hay que tener en cuenta cómo se está codificando la información que se envía y se recibe entre los diferentes procesos. La codificación de la información depende del sistema operativo subyacente donde se está ejecutando el proceso hijo.

¿SABÍAS QUE...?

GNU Linux, Mac OS, Android, UNIX, iOS, etc. utilizan el formato de codificación de caracteres UTF-8, que utiliza por debajo el estándar Unicode. Unicode es un estándar de codificación de caracteres diseñado para facilitar el tratamiento informático, transmisión y visualización de textos de múltiples lenguajes y disciplinas técnicas. El término "Unicode" proviene de los tres objetivos perseguidos: universalidad, uniformidad y unicidad.

Sin embargo, diferentes versiones de Microsoft Windows no utilizan UTF-8, sino sus propios formatos de codificación no compatibles con el resto, como Windows-Western.

Para mostrar los datos correctamente codificados en Java puede ser necesario especificar cómo se reciben los datos en el sistema operativo subyacente, ya que es en el que se está ejecutando los procesos hijos, con los que hay que comunicarse.

ACTIVIDADES 1.3

- Un proceso puede esperar recibir por su entrada estándar los datos con los que operar en un formato específico. Por ejemplo, si el proceso se crea a partir de un ejecutable en Unix, la comunicación de datos con el mismo debería producirse en UTF-8. Si los datos de entrada no contienen caracteres extraños (saltos de línea, tildes, *ñ*, etc.), esto no suele ser necesario, pero aun así veremos cómo puede hacerse.

```
//Importamos todos los paquetes necesarios
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Arrays;

public class UnixInteractor {

   public static void main(String[] command) {

        String line;

        //Preparamos el commando a ejecutar
        ProcessBuilder pb = new ProcessBuilder(command);
        pb.redirectErrorStream(true);
        try {
             //Se crea el Nuevo proceso hijo
             Process shell = pb.start();
             //Se obtiene stdout del proceso hijo
             InputStream is = shell.getInputStream();
             //Se convierte el formato de UTF-8 al de un String de Java
             BufferedReader br = new BufferedReader (new
             InputStreamReader(is,"UTF-8"));

             System.out.println("La salida del proceso
             hijo" + Arrays.toString(command) + ":" );
             //Se muestra la salida del hijo por pantalla
             while ((line = br.readLine()) != null) {
                System.out.println(line);
             }
             //Cuando finaliza se cierra el descriptor
             //de salida del hijo
             is.close();
                  } catch (IOException e) {
                System.out.println("Error ocurrió ejecutando el comando.
                Descripción:" + e.getMessage());
      }
   }
}
```

Modifica el ejemplo visto para que la comunicación se realice de forma efectiva con un ejecutable de Microsoft Windows. Justifica cualquier cambio realizado incluyendo los paquetes a utilizar.

Además de la posibilidad de comunicarse mediante flujos de datos, existen otras alternativas para la comunicación de procesos:

- Usando *sockets* para la comunicación entre procesos. Este concepto se estudiará en detalle en el Capítulo 3.
- Utilizando JNI (*Java Native Interface*). La utilización de Java permite abstraerse del comportamiento final de los procesos en los diferentes sistemas operativos. Sin embargo, al ser un lenguaje de alto nivel oculta funcionalidad de procesos de bajo nivel que puede depender del sistema subyacente. Por ejemplo, un *pipe* es un canal de comunicación unidireccional de bajo nivel que permite comunicarse a los procesos de forma sencilla. La comunicación de procesos en Java solo se puede hacer mediante los *streams* explicados, perdiendo parte de la funcionalidad ofertada por los *pipes* de bajo nivel. Sin embargo, se puede utilizar JNI para acceder desde Java a aplicaciones desarrolladas en otros lenguajes de programación de más bajo nivel, como C, que pueden sacar partido al sistema operativo subyacente.

¿SABÍAS QUE...?

La mayoría de los sistemas operativos actuales están desarrollados en C. C. es un lenguaje de bajo nivel cercano a la arquitectura que obtiene un elevado rendimiento. Al ser más cercano a la arquitectura del sistema, permite comunicar diferentes procesos utilizando métodos más cercanos al sistema operativo.

- Librerías de comunicación no estándares entre procesos en Java que permiten aumentar las capacidades del estándar Java para comunicarlos. Por ejemplo, CLIPC (*http://clipc.sourceforge.net/*) es una librería Java de código abierto que ofrece la posibilidad de utilizar los siguientes mecanismos que no están incluidos directamente en el lenguaje gracias a que utiliza por debajo llamadas a JNI para poder utilizar métodos más cercanos al sistema operativo:
  - *Memoria compartida*: se establece una región de memoria compartida entre varios procesos a la cual pueden acceder todos ellos. Los procesos se comunican escribiendo y leyendo datos en ese espacio de memoria compartida.
  - *Pipes*: permite a un proceso hijo comunicarse con su padre a través de un canal de comunicación sencillo.
  - *Semáforos*: mecanismo de bloqueo de un proceso hasta que ocurra un evento.

  Sin embargo, aunque dichas librerías pueden aumentar los métodos de comunicación entre procesos, hay que tener en cuenta que se encuentran actualmente en fase de investigación y desarrollo.

ACTIVIDADES 1.4

- Busca en Internet qué es JNI (*Java Native Interface*) y descubre cómo se puede programar una aplicación Java que llame a código desarrollado en otro lenguaje de programación. Gracias a JNI, se le puede sacar una mayor funcionalidad a los procesos al poder utilizar funcionalidad dependiente del sistema subyacente.

1.7 SINCRONIZACIÓN DE PROCESOS

Los métodos de comunicación de procesos se pueden considerar como métodos de sincronización ya que permiten al proceso padre llevar el ritmo de envío y recepción de mensajes.

Concretamente, los envíos y recepciones por los flujos de datos permiten a un proceso hijo comunicarse con su padre a través de un canal de comunicación unidireccional bloqueante.

EJEMPLO 1.9

Si el padre pide leer de la salida del hijo *stdout* a través de su *InputStream* se bloquea hasta que el hijo le devuelve los datos requeridos. En este sentido, padre e hijo pueden sincronizarse de la forma adecuada.

1.7.1 ESPERA DE PROCESOS (OPERACIÓN *WAIT*)

Además de la utilización de los flujos de datos se puede esperar por la finalización del proceso hijo y obtener su información de finalización mediante la operación *wait*. Dicha operación bloquea al proceso padre hasta que el hijo finaliza su ejecución mediante *exit.* Como resultado se recibe la información de finalización del proceso hijo. Dicho valor de retorno se especifica mediante un número entero. El valor de retorno significa cómo resulto la ejecución. No tiene nada que ver con los mensajes que se pasan padre e hijo a través de los *streams*. Por convención se utiliza 0 para indicar que el hijo ha acabado de forma correcta.

Mediante *waitFor*() de la clase *Process* el padre espera bloqueado hasta que el hijo finalice su ejecución, volviendo inmediatamente si el hijo ha finalizado con anterioridad o si alguien le interrumpe (en este caso se lanza la interrupción *InterruptedException*). Además se puede utilizar *exitValue*() para obtener el valor de retorno que devolvió un proceso hijo. El proceso hijo debe haber finalizado, si no, se lanza la excepción *IllegalThreadStateException.*

EJEMPLO 1.10

Implementación de sincronización de procesos:

```
import java.io.IOException;
import java.util.Arrays;

public class ProcessSincronization {

    public static void main(String[] args)
          throws IOException, InterruptedException {

    try{
          Process process = new ProcessBuilder(args).start();
          int retorno = process.waitFor();
          System.out.println("Comando " +  Arrays.toString(args)
               +  " devolvió: " + retorno);
    } catch (IOException e) {
          System.out.println("Error ocurrió ejecutando el
                              comando. Descripción: " + e.getMessage());
    } catch (InterruptedException e) {
          System.out.println("El comando fue interrumpido.
                              Descripción del error: " + e.getMessage());
    }
    }
}
```

1.8 PROGRAMACIÓN MULTIPROCESO

La programación concurrente es una forma eficaz de procesar la información al permitir que diferentes sucesos o procesos se vayan alternando en la CPU para proporcionar multiprogramación. La multiprogramación puede producirse entre procesos totalmente independientes, como podrían ser los correspondientes al procesador de textos, navegador, reproductor de música, etc., o entre procesos que pueden cooperar entre sí para realizar una tarea común.

El sistema operativo se encarga de proporcionar multiprogramación entre todos los procesos del sistema, ocultando esta complejidad tanto a los usuarios como a los desarrolladores. Sin embargo, si se pretende realizar procesos que cooperen entre sí, debe ser el propio desarrollador quien lo implemente utilizando la comunicación y sincronización de procesos vistas hasta ahora.

A la hora de realizar un programa multiproceso cooperativo, se deben seguir las siguientes fases:

1 **Descomposición funcional**. Es necesario identificar previamente las diferentes funciones que debe realizar la aplicación y las relaciones existentes entre ellas.

2 **Partición**. Distribución de las diferentes funciones en procesos estableciendo el esquema de comunicación entre los mismos. Al ser procesos cooperativos necesitarán información unos de otros por lo que deben comunicarse. Hay que tener en cuenta que la comunicación entre procesos requiere una pérdida de tiempo tanto de comunicación como de sincronización. Por tanto, el objetivo debe ser maximizar la independencia entre los procesos minimizando la comunicación entre los mismos.

3 **Implementación**. Una vez realizada la descomposición y la partición se realiza la implementación utilizando las herramientas disponibles por la plataforma a utilizar. En este caso, Java permite únicamente métodos sencillos de comunicación y sincronización de procesos para realizar la cooperación.

1.8.1 CLASE PROCESS

A la hora de realizar un algoritmo multiproceso en Java se utiliza la clase *Process*. Se presenta por simplicidad una tabla con los métodos recogidos más importantes explicados a lo largo del capítulo:

Método	Tipo de retorno	Descripción
getOutputStream()	*OutputStream*	Obtiene el flujo de salida del proceso hijo conectado al *stdin*
getInputStream()	*InputStream*	Obtiene el flujo de entrada del proceso hijo conectado al *stdout* del proceso hijo
getErrorStream()	*InputStream*	Obtiene el flujo de entrada del proceso hijo conectado al *stderr* del proceso hijo
destroy()	*void*	Implementa la operación *destroy*
waitFor()	*int*	Implementa la operación *wait*
exitValue()	*int*	Obtiene el valor de retorno del proceso hijo

1.9 CASO PRÁCTICO

En este caso práctico se va a desarrollar una solución multiproceso al problema de sincronizar y comunicar dos procesos hijos creados a partir de un proceso padre. La idea es escribir una clase Java que ejecute dos comandos (cada hijo creado ejecutará uno de ellos) con sus respectivos argumentos y redireccione la salida estándar del primero a la entrada estándar del segundo. Por sencillez, los comandos y sus argumentos irán directamente escritos en el código del programa para no complicar demasiado el problema.

El siguiente ejemplo muestra la ejecución de los comandos **ls –la** y **tr “d” “D”** en Unix (el resultado debería ser el mismo que el de ejecutar en la *shell* de Linux o Mac **ls –la | tr “d” “D”**):

```
total 6
Drwxr-xr-x 5 user users 4096 2011-02-22 10:59 .
Drwxr-xr-x 8 user users 4096 2011-02-07 09:26 ..
-rw-r--r-- 1 user users 30 2011-02-22 10:59 a.txt
-rw-r--r-- 1 user users 27 2011-02-22 10:59 b.txt
Drwxr-xr-x 2 user users 4096 2011-01-24 17:49 Dir3
Drwxr-xr-x 2 user users 4096 2011-01-24 11:48 Dir4
```

RESUMEN DEL CAPÍTULO

El capítulo ha mostrado el funcionamiento de los sistemas operativos en lo relativo a la ejecución de diferentes programas. Para ello, se ha definido el concepto de proceso, viéndolo como un programa en ejecución con toda la estructura necesaria que necesite para ello. El sistema operativo tiene que gestionar múltiples procesos. Aunque solo haya un único procesador, permitiendo la ejecución de un único proceso, el sistema operativo se encarga de aparentar que varios procesos se ejecutan a la vez (concepto conocido como "concurrencia" y en este caso en concreto, "multiprogramación") cambiando la ejecución del proceso lo suficientemente rápido. Estos cambios generan los estados de un proceso. Un proceso a lo largo de su ciclo de vida, puede pasar por diferentes estados —listo, en ejecución o bloqueado, entre otros—, saltando de una cola de espera a otra cuando no esté en ejecución. En función de cómo se gestionen las colas de espera de procesos, el funcionamiento del sistema variará. Esto se conoce como la "planificación del sistema", pudiendo ser apropiativa (desalojando los procesos más prioritarios a los menos prioritarios), cooperativa (sin desalojo, o de tiempo compartido [cambiando de proceso cada cierto tiempo]). Cuando un proceso pasa de estar en ejecución a otro estado se produce lo que se denomina un "cambio de contexto". En ese momento hay que salvar toda la información del proceso para poder ponerlo a ejecutarse desde el mismo punto donde lo dejó cuando fue desalojado.

El sistema operativo es el encargado de gestionar los procesos. Para ello se comunica con los procesos utilizando su identificador o PID, número unívoco que identifica a un proceso. Un proceso a su vez puede crear otros procesos. El proceso creador se denomina "padre", y el nuevo proceso, "hijo". Esto produce que se cree un árbol de procesos del sistema ya que al arrancar el ordenador solo existe un proceso, y el resto de procesos se van creando en forma arbórea partiendo de él. La creación de procesos se realiza mediante la operación *create,* permitiendo al proceso padre controlar a sus hijos. Puede destruirlos mediante la operación *destroy* o esperar por su finalización mediante *wait*. La comunicación de procesos se realiza a través de su interfaz con el mundo real. Dicha interfaz está formada por su entrada estándar (*stdin*) de donde recibe datos de entrada; la salida estándar (*stdout*), donde el proceso escribe los resultados; y la salida de error (*stderr*), donde envía los mensajes de error. La utilización de la interfaz, especialmente la recepción de datos por *stdin* es bloqueante, lo que permite junto a las otras operaciones sincronizar procesos para realizar operaciones multiproceso.

Cada sistema operativo tiene unas características únicas, y la gestión de los procesos es diferente en cada uno de ellos. Por interoperabilidad se ha visto cómo se gestionan procesos utilizando la JVM de Java, aunque su utilización oculta algunas de las posibilidades existentes para sincronizar y comunicar procesos a bajo nivel.

EJERCICIOS PROPUESTOS

1. Escribe una clase llamada *Ejecuta* que reciba como argumentos el comando y las opciones del comando que se quiere ejecutar. El programa debe crear un proceso hijo que ejecute el comando con las opciones correspondientes mostrando un mensaje de error en el caso de que no se realizase correctamente la ejecución. El padre debe esperar a que el hijo termine de informar si se produjo alguna anomalía en la ejecución del hijo.

2. Escribe un programa *Aleatorios* que haga lo siguiente:

- Cree un proceso hijo que está encargado de generar números aleatorios. Para su creación puede utilizarse cualquier lenguaje de programación generando el ejecutable correspondiente. Este proceso hijo escribirá en su salida estándar un número aleatorio del 0 al 10 cada vez que reciba una petición de ejecución por parte del padre. *Nota*: no es necesario utilizar JNI, solamente crear un ejecutable y llamar correctamente al mismo desde Java.
- El proceso padre lee líneas de la entrada estándar y por cada línea que lea solicitará al hijo que le envíe un número aleatorio, lo leerá y lo imprimirá en pantalla.
- Cuando el proceso padre reciba la palabra "fin", finalizará la ejecución del hijo y procederá a finalizar su ejecución.

Ejemplo de ejecución:

```
ab (enter)
7
abcdef (enter)
1
Pepe (enter)
6
fin (enter)
```

3. Escribe una clase llamada *Mayusculas* que haga lo siguiente:

- Cree un proceso hijo.
- El proceso padre y el proceso hijo se comunicarán de forma bidireccional utilizando *streams*.
- El proceso padre leerá líneas de su entrada estándar y las enviará a la entrada estándar del hijo (utilizando el *OutputStream* del hijo).
- El proceso hijo leerá el texto por su entrada estándar, lo transformará todo a letras mayúsculas y lo imprimirá por su salida estándar. Para realizar el programa hijo se puede utilizar cualquier lenguaje de programación generando un ejecutable.
- El padre imprimirá en pantalla lo que recibe del hijo a través del *InputStream* del mismo.

Ejemplo de ejecución:

```
hola (enter)
HOLA
mundo (enter)
MUNDO
```

TEST DE CONOCIMIENTOS

1 ¿Qué proporciona el modo dual?

a) Un mecanismo para ejecutar dos instrucciones en paralelo si se tienen varios núcleos.

b) Un mecanismo de protección que evita que determinadas instrucciones puedan ser ejecutadas directamente por código de usuario.

c) Un mecanismo que permite la ejecución del *kernel* del sistema operativo únicamente como respuesta a un *trap*.

d) Un mecanismo que permite la ejecución del *kernel* del sistema operativo únicamente como respuesta a una interrupción hardware.

2 Indica cuál de las siguientes respuestas es falsa:

a) Los procesos son independientes y tienen su propio espacio de memoria asignado.

b) El sistema operativo se refiere a los procesos que gestiona mediante su PID.

c) Dos procesos diferentes pueden tener el mismo PID.

d) La puesta en ejecución de un nuevo proceso se produce bajo la responsabilidad de otro proceso.

3 El *kernel* del sistema operativo se ejecuta:

a) Como un proceso más dentro del sistema atendiendo las peticiones que los procesos y el hardware le hacen.

b) En base a interrupciones.

c) En base a rutina de tratamiento de señales.

d) Utilizando *pipes* para comunicarse con los diferentes procesos.

4 Una interrupción software causada por una petición del usuario es:

a) Una excepción.

b) Una interrupción.

c) Un *trap*.

d) Un *fork*.

5 Si un proceso está en el estado de "En ejecución" y se produce una interrupción:

a) Pasará el estado "Listo".

b) Pasará al estado "Bloqueado".

c) Seguirá en ejecución.

d) Terminará.

6 Si un proceso está en el estado de "En ejecución" y solicita una operación de entrada/salida:

a) Pasará el estado "Listo".

b) Pasará al estado "Bloqueado".

c) Seguirá en ejecución.

d) Terminará.

7 Indica cuál de las siguientes afirmaciones es FALSA:

a) El cambio de contexto es tiempo perdido, durante el cual el procesador no ejecuta instrucciones de los procesos.

b) No importa que el tiempo de cambio de contexto sea grande, ya que es una operación que no se realiza muy frecuentemente.

c) El tiempo de cambio de contexto depende de la arquitectura del procesador.

d) El contexto de un proceso incluye el estado del proceso, el estado del procesador y la información de gestión de memoria.

8 Un planificador que solo toma decisiones cuando un proceso pasa de ejecución a espera, o cuando un proceso termina, es un planificador:

a) Apropiativo.
b) Cooperativo.
c) Perezoso.
d) Automático.

9 En la planificación por tiempo compartido:

a) De forma secuencial cada proceso preparado pasa a ejecución durante una cota de tiempo llamada "cuanto".
b) El proceso preparado que pasa a ejecución corresponde al de tiempo de ejecución restante más corto.
c) De acuerdo a su prioridad, cada proceso preparado pasa a ejecución durante una cota de tiempo llamada "cuanto".
d) El proceso preparado que pasa a ejecución corresponde al de mayor prioridad asignada.

10 Dado el siguiente fragmento de código:

```
Process p1 = new ProcessBuilder(args).start();
Process p2 = new ProcessBuilder(args1).start();
BufferedReader br1 = new BufferedReader(new
  InputStreamReader(p1.getInputStream()));
BufferedReader br2 = new BufferedReader(new
  InputStreamReader(p2.getInputStream()));
```

Indica el máximo número de procesos que pueden comunicarse entre sí utilizando br1 y br2.

a) 3
b) 2
c) 4
d) 0

2 Programación de hilos

OBJETIVOS DEL CAPÍTULO

- ✓ Comprender los conceptos básicos del funcionamiento de los sistemas en lo relativo a la ejecución de diferentes hilos.
- ✓ Comprender el concepto de paralelismo y cómo el sistema puede proporcionar multitarea al usuario.
- ✓ Saber utilizar los mecanismos de sincronización de hilos para construir aplicaciones paralelas.
- ✓ Familiarizarse con la programación de hilos entendiendo sus principios y formas de aplicación.

2.1 CONCEPTOS BÁSICOS

De igual manera que en el caso del capítulo anterior, antes de profundizar en la programación de hilos, es necesario tener claros algunos de los conceptos claves.

- **Hilo**: los hilos, o *threads*, son la unidad básica de utilización de la CPU, y más concretamente de un *core* del procesador. Así un *thread* se puede definir como la secuencia de código que está en ejecución, pero dentro del contexto de un proceso.
- **Hilo *vs*. Proceso**: hasta ahora hemos visto que el sistema operativo gestiona procesos, asignándoles la memoria y recursos que necesiten para su ejecución. En este sentido, el sistema operativo planifica únicamente procesos.

Los hilos se ejecutan dentro del contexto de un proceso, por lo que dependen de un proceso para ejecutarse. Mientras que los procesos son independientes y tienen espacios de memoria diferentes, dentro de un mismo proceso pueden coexistir varios hilos ejecutándose que compartirán la memoria de dicho proceso. Esto sirve para que el mismo programa en ejecución (proceso) pueda realizar diferentes tareas (hilos) al mismo tiempo. Un proceso siempre tendrá, por lo menos, un hilo de ejecución que es el encargado de la ejecución del proceso.

EJEMPLO 2.1

Gracias al uso de hilos podemos tener diferentes pestañas abiertas en el navegador Google Chrome, cada una cargando a la vez una página web diferente, o Microsoft Word puede tener un hilo comprobando automáticamente la gramática, a la vez que se está escribiendo un documento.

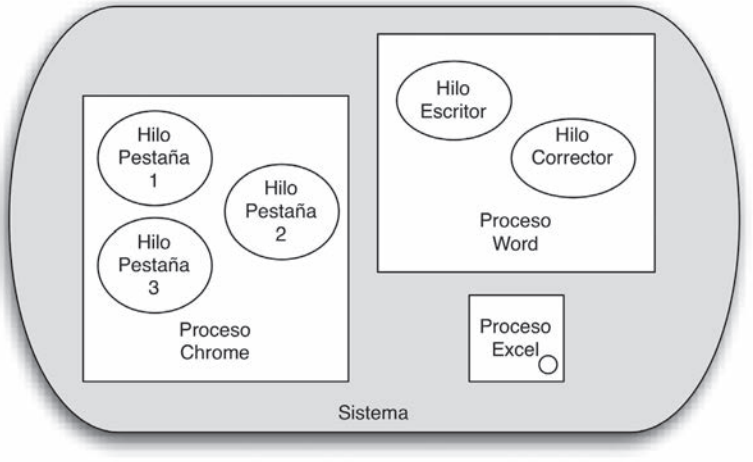


Figura 2.1. *Relación entre hilos y procesos*

Frente a la multiprogramación de procesos, la multitarea presenta bastantes ventajas:

- Capacidad de respuesta. Los hilos permiten a los procesos continuar atendiendo peticiones del usuario aunque alguna de las tareas que esté realizando el programa sea muy larga. Este modelo se utiliza mayoritariamente en la programación de un servicio en servidores. Un hilo se encarga de recibir todas las peticiones del usuario y por cada petición se lanza un nuevo hilo para responderla. De esta forma se están tratando varias peticiones al mismo tiempo.

- Compartición de recursos. Por defecto, los *threads* comparten la memoria y todos los recursos del proceso al que pertenecen. No necesitan ningún medio adicional para comunicarse información entre ellos ya que todos pueden ver la información que hay en la memoria del proceso. Sin embargo, debido a que todos los hilos pueden acceder y modificar los datos al mismo tiempo, se necesitan medios de sincronización adicionales para evitar problemas en el acceso.

- Como los hilos utilizan la misma memoria del proceso del cual dependen, la creación de nuevos hilos no supone ninguna reserva adicional de memoria por parte del sistema operativo. Es más barato en términos de uso de memoria y otros recursos crear nuevos *threads* que crear nuevos procesos.

- Paralelismo real. La utilización de *threads* permite aprovechar la existencia de más de un núcleo en el sistema en arquitecturas *multicore*. Sabemos que el sistema operativo planifica procesos de forma concurrente, intercambiando los procesos uno por otro mediante un cambio de contexto. En este sentido, solamente puede haber un proceso en ejecución en un momento dado independientemente del número de núcleos del procesador ya que solo existe una memoria (cuando se desea acceder a un dato se realiza mediante la traducción de las referencias a direcciones de memoria que se encuentran en el código del proceso a las direcciones efectivas en memoria principal). Es decir, la gestión de procesos no puede aprovecharse de la existencia de varios núcleos. Sin embargo, varios hilos pueden ejecutarse en el contexto de un mismo proceso. Cuando ese proceso está en ejecución, los hilos programados del mismo pueden utilizar todos los núcleos del procesador de forma paralela, permitiendo que se ejecuten varias instrucciones (una por cada *thread* y núcleo) a la vez.

¿SABÍAS QUE...?

La tendencia del mercado es añadir cada vez un número mayor de núcleos en el sistema en vez de aumentar las prestaciones de los mismos. La idea es que la suma de las potencias de los núcleos sea mayor que la prestación ofertada por un único procesador aunque los núcleos que se utilicen sean menos potentes de forma individual. La razón es que los procesadores menos potentes gastan menos energía por lo que se reduce el consumo. Así, se ha pasado de procesadores Intel Pentium 4 Prescott 570J a 3,8 GHz (3,8 millones de operaciones por segundo) en 2005 a Intel i7 Ivy Bridge de 2-3,8 GHz (8 núcleos de 2 hasta 3,8 millones de operaciones con *hyperthreading*, 2 *threads* en ejecución por cada núcleo) en 2012.

2.2 RECURSOS COMPARTIDOS POR HILOS

Un hilo es muy similar a un proceso pero con la diferencia de que un hilo siempre se ejecuta dentro del contexto de un proceso. Los procesos mantienen su propio espacio de direcciones y recursos de ejecución mientras que los hilos dependen del proceso, por lo que comparten con otros hilos la sección de código, datos y otros recursos. Únicamente cada hilo tiene su propio contador de programa, conjunto de registros de la CPU y pila para indicar por dónde se está ejecutando.

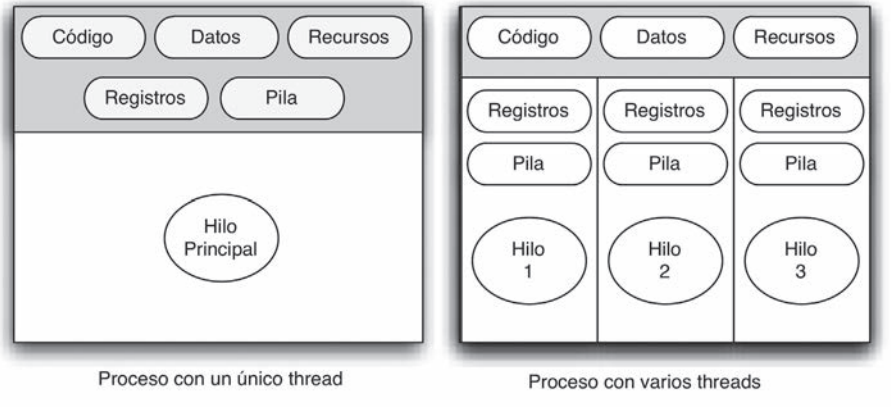


Figura 2.2. *Recursos compartidos por hilos*

2.3 ESTADOS DE UN HILO

Al igual que los procesos, los hilos pueden cambiar de estado a lo largo de su ejecución. Su comportamiento dependerá del estado en el que se encuentren:

- **Nuevo**: el hilo está preparado para su ejecución pero todavía no se ha realizado la llamada correspondiente en la ejecución del código del programa. Los hilos se inicializan en la creación del proceso correspondiente, ya que forman parte de su espacio de memoria pero no empiezan a ejecutar hasta que el proceso lo indica.
- **Listo**: el proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse. El planificador del sistema operativo es el encargado de elegir cuándo el proceso pasa a ejecutarse.
- **Pudiendo ejecutar** (*Runnable*): el hilo está preparado para ejecutarse y puede estar ejecutándose. A todos los efectos sería como si el *thread* estuviera en ejecución, pero debido al número limitado de núcleos no se puede saber si se está ejecutando o esperando debido a que no hay hardware suficiente para hacerlo. En general, por simplicidad se puede considerar que todos los *threads* del proceso se ejecutan al mismo tiempo (en paralelo) sabiendo que los hilos deben compartir los recursos del sistema.

- **Bloqueado**: el hilo está bloqueado por diversos motivos (esperando por una operación de E/S, sincronización con otros hilos, dormido, suspendido) esperando que el suceso suceda para volver al estado *Runnable*. Cuando ocurre el evento que lo desbloquea, el hilo pasaría directamente a ejecutarse.
- **Terminado**: el hilo ha finalizado su ejecución. Sin embargo, frente a los procesos que liberan los recursos que mantenían cuando finalizan, el hilo no libera ningún recurso ya que pertenecen al proceso y no a él mismo. Para terminar un hilo, él mismo puede indicarlo o puede ser el propio proceso el que lo finalice mediante la llamada correspondiente.

ACTIVIDADES 2.1

- Enumera en una lista las principales diferencias entre hilo y proceso.

2.4 GESTIÓN DE HILOS

2.4.1 OPERACIONES BÁSICAS

2.4.1.1 Creación y arranque de hilos (operación create)

Los *threads* comparten tanto el espacio de memoria del proceso como los recursos asociados (entorno de ejecución), siendo su creación más eficiente que la creación de procesos. Estos últimos tienen que crear estructuras especiales en el sistema además de hacer la reserva de memoria y recursos correspondiente. Cualquier programa a ejecutarse es un proceso que tiene un hilo de ejecución principal. Este hilo puede a su vez crear nuevos hilos que ejecutarán código diferente o tareas, es decir el camino de ejecución no tiene por qué ser el mismo.

¿SABÍAS QUE?

La GPU (*Graphics Processing Unit,* en inglés, por similitud con el nombre de CPU, *Central Processing Unit*) es el coprocesador dedicado al procesamiento gráfico que se encuentra en las tarjetas gráficas de última generación, para aligerar la carga de trabajo de la CPU al ejecutar videojuegos, animaciones 3D, vídeos de alta definición, etc. Frente a la CPU, que puede contar con unos pocos núcleos, las GPU presentan múltiples núcleos (del orden de más de 100), lo que permite ejecutar un elevado número de hilos en paralelo. Sin embargo, las operaciones que pueden realizar estos hilos están limitadas y en general deben realizar todos la misma tarea con distintos datos. Si el algoritmo paralelo se adapta a estas condiciones, su implementación en GPU mediante los lenguajes de programación CUDA (*Compute Unified Device Architecture*, en inglés) de NVidia o OpenCL (*Open Computing Language*, en inglés) puede obtener una mejora muy significativa frente a su implementación en CPU mediante los lenguajes de programación tradicionales.

A la hora de crear los nuevos hilos de ejecución dentro de un proceso, existen dos formas de hacerlo en Java: implementando la interfaz *Runnable*, o extendiendo de la clase *Thread* mediante la creación de una subclase.

La interfaz *Runnable* proporciona la capacidad de añadir la funcionalidad de un hilo a una clase simplemente implementando dicha interfaz. Java proporciona soporte para hilos a través esta interfaz. La interfaz de Java necesaria para cualquier clase que implemente hilos es *Runnable*.

Las clases que implementan la interfaz *Runnable* proporcionan una forma de realizar la operación *create*, encargada de crear nuevos hilos. La operación *create* inicia un *thread* de la clase correspondiente, pasándolo del estado "nuevo" a "pudiendo ejecutar".

En la utilización de la interfaz *Runnable*, el método *run()* implementa la operación *create* conteniendo el código a ejecutar por el hilo. Dicho método contendrá el hilo de ejecución, es decir viene a ser como el método *main*() en el hilo. Esto implica que el hilo finaliza su ejecución cuando finaliza el método *run*(). A menudo se denomina a este método "el cuerpo del hilo".

Además de la interfaz *Runnable*, otra clase principal para el uso de hilos es la clase *Thread*. La clase *Thread* es responsable de producir hilos funcionales para otras clases e implementa la interfaz *Runnable*. La interfaz *Runnable* debería ser utilizada si la clase solamente va a utilizar la funcionalidad *run* de los hilos. En otro caso se debería derivar de la clase *Thread* modificando los métodos que se consideren. Eso sí, hay que tener en cuenta que Java no soporta herencia múltiple de forma directa, es decir, no se puede derivar una clase de varias clases padre. Para poder añadir la funcionalidad de hilo a una clase que deriva de otra clase, siendo esta distinta de *Thread*, se debe utilizar la interfaz *Runnable*.

Para añadir la funcionalidad de hilo a una clase mediante la clase *Thread* simplemente se deriva de dicha clase ignorando el método *run()* (proveniente de la interfaz *Runnable*). La clase *Thread* define también un método para implementar la operación *create* para comenzar la ejecución del hilo. Este método es *start()*, que comienza la ejecución del hilo de la clase correspondiente. Al ejecutar *start*(), la JVM llama al método *run()* del hilo que contiene el código de la tarea.

Para crear un hilo utilizando la interfaz *Runnable* se debe crear una nueva clase que implemente la interfaz, teniendo que implementar únicamente el método *run()* con la tarea a realizar. Además se debe crear una instancia de la clase *Thread* dentro de la nueva clase, la cual representará el hilo a ejecutar. Como dicho hilo pertenece a la clase *Thread* se debe utilizar *start()* para ponerlo en ejecución o arrancarlo.

```
public class NuevoThread implements Runnable {

   Thread hilo;
   public void run() {
      // Código a ejecutar por el hilo
   }
}
```

El siguiente ejemplo muestra el proceso:

EJEMPLO 2.2

Creación de un hilo implementando la interfaz *Runnable*:

```
class HelloThread implements Runnable {

    Thread t;
    HelloThread () {
        t = new Thread(this, "Nuevo Thread");
        System.out.println("Creado hilo: " + t);
        t.start(); // Arranca el nuevo hilo de ejecución. Ejecuta run
    }

    public void run() {
        System.out.println("Hola desde el hilo creado!");
        System.out.println("Hilo finalizando.");
    }
}

class RunThread {
    public static void main(String args[]) {

        new HelloThread(); // Crea un nuevo hilo de ejecución
        System.out.println("Hola desde el hilo principal!");
        System.out.println("Proceso acabando.");
    }
}
```

El otro mecanismo de creación de hilos, como ya hemos dicho, consistiría en la creación previa de una subclase de la clase *Thread*, la cual podríamos instanciar después. Es necesario sobrecargar el método *run()* con la implementación de lo que se desea que el hilo ejecute. Como hemos visto, nunca se ejecuta de forma directa este método, sino que se llama al método *start()* de dicha clase para arrancar el hilo. En este caso se heredan los métodos y variables de la clase padre.

```
public class NuevoThread extends Thread {

   public void run() {
      // Código a ejecutar por el hilo
   }
}
```

El siguiente ejemplo muestra el proceso:

EJEMPLO 2.3

Creación de un hilo extendiendo la clase *Thread*:

```
class HelloThread extends Thread {

     public void run() {
          System.out.println("Hola desde el hilo creado!");
     }
}

public class RunThread {
     public static void main(String args[]) {

     new HelloThread().start();// Crea y arranca un nuevo hilo de ejecución
     System.out.println("Hola desde el hilo principal!");
     System.out.println("Proceso acabando.");
     }
}
```

A la hora de decantarse por una u otra opción hay que saber que la utilización de la interfaz *Runnable* es más general, ya que el objeto puede ser una subclase de una clase distinta de *Thread*, pero no tiene ninguna otra funcionalidad además de *run()* que la incluida por el programador. La segunda opción es más fácil de utilizar, ya que está definida una serie de métodos útiles para la administración de hilos, pero está limitada porque los clases creadas como hilos deben ser descendientes únicamente de la clase *Thread*. Con la implementación de la interfaz *Runnable*, se podría extender la clase hilo creada, si fuese necesario. Hay que recordar que siempre se arrancan los hilos ejecutando *start*(), la cual llamará al método *run()* correspondiente.

ACTIVIDADES 2.2

- Crea un hilo que realice el cálculo de los primeros N números de la sucesión de Fibonacci. La sucesión de Fibonacci comienza con los números 1 y 1 y el siguiente elemento es la suma de los dos elementos anteriores. Así la sucesión de Fibonacci sería 1, 1, 2, 3, 5, 8, 11, 19, 30, 49… El parámetro N será indicado cuando se llamé al constructor de la clase *Thread* correspondiente.

2.4.1.2 Espera de hilos (operaciones join y sleep)

Si todo fue bien en la operación *create*, el objeto de la clase debería contener un hilo cuya ejecución depende del método *run*() especificado para ese objeto.

Como varios hilos comparten el mismo procesador, si alguno no tiene trabajo que hacer, es bueno suspender su ejecución para que haya un mayor tiempo de procesador disponible para el resto de hilos del sistema. La ejecución del hilo se puede suspender mediante la operación *join* esperando hasta que el hilo correspondiente finalice su ejecución.

Además se puede dormir un hilo mediante la operación *sleep* por un período especificado, teniendo en cuenta que el tiempo especificado puede no ser preciso, ya que depende de los recursos proporcionados por el sistema operativo subyacente.

2..4.1.2.1 interrupción

Ambas operaciones de espera pueden ser interrumpidas, si otro hilo interrumpe al hilo actual mientras está suspendido por dichas llamadas.

Una **interrupción** es una indicación a un hilo que debería dejar de hacer lo que esté haciendo para hacer otra cosa.

Un hilo envía una interrupción mediante la invocación del método *interrupt()* en el objeto del hilo que se quiere interrumpir. El hilo interrumpido recibe la excepción *InterruptedException* si están ejecutando un método que la lance (por ejemplo, los que implementan las operaciones *join* y *sleep*), pero podrían haber sido interrumpidos por *interrupt()* mientras tanto. En este sentido se debe invocar periódicamente el método *interrupted()* para saber si se ha recibido una interrupción.

Una vez comprobado si el hilo ha sido interrumpido se puede o bien finalizar su ejecución, o lanzar *InterruptedException* para manejarla en una sentencia *catch* centralizada en aplicaciones complejas.

EJEMPLO 2.4

Gestión de interrupciones

```
public void run() {
    for (int i = 0; i < NDatos; i++) {
        try {
            System.out.println("Esperando a recibir dato!");
            Thread.sleep(500);
        } catch (InterruptedException e) {
            System.out.println("Hilo interrumpido.");
            return;
        }
        // Gestiona dato i
    }
    System.out.println("Hilo finalizando correctamente.");
}
```

Queda a criterio del programador decidir exactamente cómo un hilo responde a una interrupción, pero en muchos de los casos lo que se hace es finalizar su ejecución.

Por último, como no se puede controlar cuándo un *thread* finaliza su ejecución, al depender de su código en el método *run*(), se puede utilizar el método *isAlive*() para comprobar si el método no ha finalizado su ejecución antes de trabajar con él.

¿SABÍAS QUE...?

Los métodos *stop*(), *suspend*() y *resume*() de la clase *Thread* que se usaban para controlar hilos están deprecados desde la versión Java 1.4 porque pueden tener un comportamiento imprevisto. La parada de un *thread* causa que todos los cerrojos que había bloqueados se desbloqueen, pudiendo provocar un comportamiento imprevisto en el resto de *threads* que los estén utilizando. En este sentido debe evitarse su utilización utilizando **variables globales** para indicar la finalización del hilo.

Por ejemplo:

```
private Thread ejemplo;

public void start() {
   ejemplo = new Thread(this);
   ejemplo.start();
}

public void stop() {
   ejemplo.stop(); // INSEGURO!
}

public void run() {
   while (true) {
        try {
              Thread.sleep(intervalo);
        } catch (InterruptedException e){
    }
    …
   }
}
```

Podría modificarse de la siguiente forma:

```
private volatile Thread ejemplo;

public void stop() {
     ejemplo = null;
}

public void run() {
      Thread thisThread = Thread.currentThread();
      while (ejemplo == thisThread) {
           try {
               Thread.sleep(intervalo);
           } catch (InterruptedException e){
       }
      …
    }
}
```

2.4.1.3 Clase Thread

A la hora de utilizar hilos en Java se utiliza la clase *Thread*. Se presenta, por simplicidad, una tabla con los métodos recogidos más importantes explicados a lo largo del capítulo:

Método	Tipo de retorno	Descripción
start()	*void*	Implementa la operación *create*. Comienza la ejecución del hilo de la clase correspondiente. Llama al método *run()*
run()	*void*	Si el hilo se construyó implementando la interfaz *Runnable*, entonces se ejecuta el método *run()* de ese objeto. En caso contrario, no hace nada
currentThread()	*static Thread*	Devuelve una referencia al objeto hilo que se está ejecutando actualmente
join()	*void*	Implementa la operación *join* para hilos
sleep(long milis)	*static void*	El hilo que se está ejecutando se suspende durante el número de milisegundos especificado
interrupt()	*void*	Interrumpe el hilo del objeto
interrupted()	*static boolean*	Comprueba si el hilo ha sido interrumpido
isAlive()	*boolean*	Devuelve *true* en caso de que el hilo esté vivo, es decir, no haya terminado el método *run()* en su ejecución

2.4.2 PLANIFICACIÓN DE HILOS

Cuando se trabaja con varios hilos, a veces es necesario pensar en la planificación de *threads*, para asegurarse de que cada hilo tiene una oportunidad justa de ejecutarse.

El planificador del sistema operativo determina qué proceso es el que se ejecuta en un determinado momento en el procesador (uno y solamente uno). Dentro de ese proceso, el hilo que se ejecutará estará en función del número de núcleos disponibles y del algoritmo de planificación que se esté utilizando. Por ejemplo, si se utiliza uno basado en prioridades, el valor de prioridad de un hilo indica que en caso de no disponibilidad de núcleos para la ejecución de todos los hilos del proceso en ejecución a la vez, se ejecute antes el que tenga mayor prioridad sobre uno que tenga una valor de prioridad menor.

Como algoritmos de planificación de hilos se pueden entender los mismos algoritmos vistos para procesos (ver *Planificación de procesos*, Capítulo 1). Esto implica que, en función del algoritmo de planificación, por ejemplo en planificación cooperativa, se pueda ejecutar completamente un hilo de prioridad superior, hasta que cambie su estado a *Bloqueado* o *Terminado*, antes que un hilo de prioridad inferior en estado *Runnable* pueda pasar a ejecutarse.

Java, por defecto, utiliza un planificador apropiativo. Si en un momento dado un hilo que tiene una prioridad mayor a cualquier otro hilo que se está ejecutando, pasa al estado *Runnable*, entonces el sistema elige a este nuevo hilo para su ejecución. Si los hilos tienen la misma prioridad, será el planificador el que asigne a uno u otro el núcleo correspondiente para su ejecución utilizando tiempo compartido, en caso de que el sistema operativo subyacente lo permita.

La prioridad de los hilos se puede establecer utilizando el método *setPriority()* de la clase *Thread*. Las prioridades de un hilo varían en un rango de enteros comprendido entre MIN_PRIORITY y MAX_PRIORITY (definidas en la misma clase y habitualmente 1 y 10, donde 1 significa mínima y 10 significa máxima prioridad). Cuando se crea un hilo, este hereda la prioridad del proceso que lo creó, pero puede modificarse dicha prioridad en cualquier momento. Los procesos en Java no pueden cambiar de prioridad, ya que Java le deja esa planificación al sistema operativo, pero en cambio sí permite que cambie la prioridad de los *threads* que se ejecutan.

EJEMPLO 2.5

Utilización de prioridades para gestionar hilos:

```
class CounterThread extends Thread {

      String name;

      public CounterThread(String name) {
           super();
           this.name = name;
      }

       public void run() {
             int count = 0;
             while (true) {
                   try {
                         sleep(10);
                    } catch (InterruptedException e) {
                            e.printStackTrace();
                    }
                      if (count == 1000)
                           count = 0;
                      System.out.println(name+":" + count++);
                }
           }
}

public class Prioridad{
             public static void main(String[] args) {
                      CounterThread thread1 = new CounterThread("thread1");
                      thread1.setPriority(10);
                      CounterThread thread2 = new CounterThread("thread2");
                      thread2.setPriority(1);
                      thread2.start();
                      thread1.start();
            }
}
```

Eso sí, los sistemas operativos no están obligados a tener en cuenta la prioridad de hilo ya que trabajan a nivel de procesos en sus algoritmos de planificación.

ACTIVIDADES 2.3

- Comprobad el funcionamiento del ejemplo anterior en vuestro propio ordenador cambiando la prioridad de los hilos. ¿El cambio de prioridades está afectando significativamente al resultado? ¿A qué se debe ese comportamiento?

2.5 SINCRONIZACIÓN DE HILOS

Los *threads* se comunican principalmente mediante el intercambio de información a través de variables y objetos en memoria. Como todos los *threads* pertenecen al mismo proceso, pueden acceder a toda la memoria asignada a dicho proceso y utilizar las variables y objetos del mismo para compartir información, siendo este el método de comunicación más eficiente. Sin embargo, cuando varios hilos manipulan concurrentemente objetos conjuntos, puede llevar a resultados erróneos o a la paralización de la ejecución. La solución es la sincronización.

2.5.1 PROBLEMAS DE SINCRONIZACIÓN

2.5.1.1 Condición de carrera

Se dice que existe una **condición de carrera** si el resultado de la ejecución de un programa depende del orden concreto en que se realicen los accesos a memoria. Veamos un ejemplo:

EJEMPLO 2.6

Dos hilos, *sumador* y *restador*, se ejecutan al mismo tiempo sobre la misma variable:

- *Sumador*: cuenta++;
- *Restador*: cuenta--;

En código maquina eso se representa por:

- registroX = cuenta
- registroX = registroX (operación: suma o resta) 1
- cuenta = registroX

Supongamos que *cuenta* vale 10, y los dos hilos están ejecutándose. Puede suceder que ambos lleguen a la instrucción que modifica *cuenta* a la vez y se ejecute lo siguiente:

- T0: *sumador* registro1 = cuenta {registro1 = 10}
- T1: *sumador* registro1 = registro1 + 1 {registro1 = 11}
- T2: *restador* registro2 = cuenta {registro2 = 10}
- T3: *restador* registro2 = registro2 - 1 {registro2 = 9}
- T4: *sumador* cuenta = registro 1 {cuenta = 11}
- T5: *restador* cuenta = registro2 {cuenta = 9}

Se ha llegado al valor de cuenta = 9, el cual es incorrecto. En función del orden en que se ejecuten cada una de las sentencias del código máquina el resultado podría ser tanto 9 como 10 u 11. Si el resultado depende del orden en la ejecución en concreto realizada, existirá una condición de carrera. Debido a que el resultado es impredecible y podría devolver el resultado esperado (en este caso 10) las condiciones de carrera son complicadas de detectar y de solucionar.

2.5.1.2 Inconsistencia de memoria

Una **inconsistencia de memoria** se produce cuando diferentes hilos tienen una visión diferente de lo que debería ser el mismo dato. Las causas de los errores de coherencia son complejas y van desde la no liberación de datos obsoletos hasta el desbordamiento del *buffer* (*buffer overflow,* en inglés), entre otros muchos.

ACTIVIDADES 2.4

- El problema conocido como *buffer overflow* es una vulnerabilidad muy conocida por los *hackers* en Internet. Se aprovechan del fallo de coherencia de memoria para tomar el control de un ordenador.

  Busca información acerca del problema y comprende en qué se basa.

Hay que tener especial cuidado al programar para evitar posibles inconsistencias en memoria. Veamos un ejemplo:

EJEMPLO 2.7

Dos hilos, *sumador* e *impresor*, realizan el siguiente código partiendo de cuenta=0

- *Sumador*: cuenta++;
- *Impresor*: System.out.println(cuenta).

Si las dos sentencias se ejecutaran por el mismo hilo, se podría asumir que el valor impreso sería 1. Pero si se ejecutan en hilos separados, el valor impreso podría ser 0, ya que no hay garantía para el *impresor* de que el *sumador* haya realizado ya su operación a menos que el programador haya establecido una relación de ocurrencia entre estas dos sentencias.

2.5.1.3 Inanición

Se conoce como *inanición* al fenómeno que tiene lugar cuando a un proceso o hilo se le deniega continuamente el acceso a un recurso compartido. Este problema se produce cuando un proceso nunca llega a tomar el control de un recurso debido a que el resto de procesos o hilos siempre toman el control antes que él por diferentes motivos. Por ejemplo, esto ocurriría si un proceso tiene baja prioridad y existen procesos más prioritarios que él en el sistema. La inanición puede ocurrir tanto por prioridad como por la propia estructuración del código. Es un problema muy complicado de encontrar y diagnosticar ya que no tiene por qué ocurrir en todas las ejecuciones que se realicen, por lo que hay que tener especial cuidado para evitarlo.

2.5.1.4 Interbloqueo

A pesar de las ventajas que proporcionan la multiprogramación y la multitarea, hay que tener especial cuidado para evitar un problema muy grave conocido como "interbloqueo". Un *interbloqueo* se produce cuando dos o más procesos o hilos están esperando indefinidamente por un evento que solo puede generar un proceso o hilo bloqueado. Dicho error no tiene por qué aparecer en todas las ejecuciones realizadas, sino que puede ocurrir únicamente en casos muy concretos que dependen del orden específico en que se ejecuten los hilos.

EJEMPLO 2.8

Un ejemplo de un interbloqueo sería:

```
H0                      H1
bloquear (S);           bloquear (Q);
bloquear (Q);           bloquear (S);
...                     ...
desbloquear (S);        desbloquear (Q);
desbloquear (Q);        desbloquear (S);
```

2.5.1.5 Bloqueo activo

Un *bloqueo activo* es similar a un interbloqueo, excepto que el estado de los dos procesos envueltos en el bloqueo activo cambia constantemente con respecto al otro. Frente al interbloqueo, donde los procesos se encuentran bloqueados, en un bloqueo activo no lo están, sino que es una forma de inanición debido a que un proceso no deja avanzar al otro.

EJEMPLO 2.9

En un ejemplo del mundo real, un bloqueo activo ocurre, por ejemplo, cuando dos personas se encuentran en un pasillo avanzando en sentidos opuestos y cada una trata de ser amable moviéndose a un lado para dejar a la otra persona pasar. Si se mueven ambas de lado a lado, encontrándose siempre en el mismo lado, se están moviendo, pero ninguna podrá avanzar.

2.5.2 MECANISMOS DE SINCRONIZACIÓN

Las condiciones de carrera y las inconsistencias de memoria se producen porque se ejecutan varios hilos concurrentemente pudiendo ser ordenados de forma diferente a la esperada. La solución pasa por provocar que cuando los hilos accedan a datos compartidos, los accesos se produzcan de forma ordenada o **síncrona**. Cuando estén ejecutando código que no afecte a datos compartidos, podrán ejecutarse libremente en paralelo, proceso también denominado "ejecución **asíncrona**".

2.5.2.1 Condiciones de Bernstein

Las condiciones de Bernstein describen que dos segmentos de código *i* y *j* son independientes y pueden ejecutarse en paralelo de forma asíncrona en diferentes hilos sin problemas si cumplen:

1. **Dependencia de flujo**: todas las variables de entrada del segmento *j* tienen que ser diferentes de las variables de salida del segmento *i*. Si no fuera así, el segmento *j* dependería de la ejecución de *i*.

2. **Antidependencia**: todas las variables de entrada del segmento *i* tienen que ser diferentes de las variables de salida del segmento *j*. Es el caso contrario a la primera condición, ya que si no fuera así, el segmento *i* tendría una dependencia del otro segmento.
3. **Dependencia de salida**: todas las variables de salida del segmento *i* tienen que ser diferentes de las variables de salida del segmento *j*. En caso contrario, si dos segmentos de código escriben en el mismo lugar, el resultado será dependiente del último segmento que ejecutó.

El resto del código que no cumpla las dependencias de Bernstein debería ejecutarse de forma síncrona. La utilización de operaciones atómicas y secciones críticas permite separar la ejecución de los segmentos de código que pueden ejecutarse de forma asíncrona de aquellos que deben ejecutarse de forma ordenada.

2.5.2.2 Operación atómica

Una *operación atómica* es una operación que sucede toda al mismo tiempo. Es decir, siempre se ejecutará de forma continuada sin ser interrumpida, por lo que ningún otro proceso o hilo puede leer o modificar datos relacionados mientras se esté realizando la operación. Es como si se realizará en un único paso.

EJEMPLO 2.10

Las operaciones de sacar dinero a través de un cajero automático suelen ser atómicas. Una vez realizada la operación, siempre que se obtenga el dinero quedará reflejado en la cuenta correspondiente. En caso de fallo, no se obtendrá el dinero y no se modificará la cuenta corriente. Una operación (sacar dinero) no puede realizarse sin que se realice en el mismo paso la otra (apuntar en la cuenta).

En un sistema con un único procesador, si una operación se ejecuta en una sola instrucción de la CPU, será atómica. Si una operación requiere de múltiples instrucciones de la CPU, entonces puede ser interrumpida, produciéndose el correspondiente cambio de contexto. Esto indica que no hay atomicidad. Las escrituras en memoria u operaciones matemáticas como *a*++ no son atómicas.

En sistemas multiprocesador, con múltiples hilos de ejecución sobre el mismo proceso, asegurar atomicidad es mucho más complicado. Los hilos podrían modificar datos a la vez sobre los que se esté operando. Una forma de asegurar atomicidad es declarando las variables como *volatile*. Dicha declaración indica al sistema que la actualización de la variable no se realiza en un registro, sino directamente en memoria, teniendo que realizarla en un único paso. Sin embargo, aunque esto permite solucionar el problema específico de accesos a variables, existen muchos casos en los cuales se desea aportar atomicidad a parte del código. Esto se puede realizar mediante la creación de una sección crítica.

2.5.2.3 Sección crítica

Se denomina *sección crítica* a una región de código en la cual se accede de forma ordenada a variables y recursos compartidos, de forma que se puede diferenciar de aquellas zonas de código que se pueden ejecutar de forma asíncrona. Este concepto se puede aplicar tanto a hilos como a procesos concurrentes, la única condición es que compartan datos o recursos.

Cuando un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su correspondiente sección crítica, ordenando de esta forma la ejecución concurrente. Esto significa:

- Si otro proceso quiere ejecutar su sección crítica, se bloqueará hasta que el primer proceso finalice la ejecución de su sección crítica.
- Se establece una relación antes-después en el orden de ejecución de la sección crítica. Esto garantiza que los cambios en los datos son visibles a todos los procesos.

El *problema de la sección crítica* consiste en diseñar un protocolo que permita a los procesos cooperar. Cualquier solución al problema de la sección crítica debe cumplir:

- **Exclusión mutua**: si un proceso está ejecutando su sección crítica, ningún otro proceso puede ejecutar su sección crítica. Es la característica principal.
- **Progreso**: si ningún proceso está ejecutando su sección crítica y hay varios procesos que quieren entrar en su sección crítica, solo aquellos procesos que están esperando para entrar pueden participar en la decisión de quién entra definitivamente. Esta decisión no puede posponerse indefinidamente, esperando a un proceso más prioritario, por ejemplo.
- **Espera limitada**: debe existir un número limitado de veces que se permite a otros procesos entrar en su sección crítica después de que otro proceso haya solicitado entrar en la suya y antes de que se le conceda. En caso contrario se produciría inanición.

En definitiva, el código correspondiente a la sección crítica debe terminar en un tiempo determinado y deben existir mecanismos internos del sistema que eviten la inanición, para que el resto de procesos solo tengan que esperar un período determinado de tiempo para entrar a ejecutar sus correspondientes secciones críticas.

Para su implementación se necesita un mecanismo de sincronización tanto que actúe tanto antes de entrar en la sección crítica como después de salir de ejecutarla. Esto asegura la utilización en exclusiva de los datos o recursos compartidos. La manera de implementar las secciones críticas puede variar de un sistema operativo a otro. Java oculta estas diferencias, proporcionando formas sencillas de implementar las secciones críticas: los semáforos y monitores.

2.5.2.4 Semáforos

Los semáforos se pueden utilizar para controlar el acceso a un determinado recurso formado por un número finito de instancias. Un semáforo se representa como una variable entera donde su valor representa el número de instancias libres o disponibles en el recurso compartido y una cola donde se almacenan los procesos o hilos bloqueados esperando para usar el recurso. En la fase de **inicialización**, se proporciona un valor inicial al semáforo igual al número de recursos inicialmente disponibles. Posteriormente, se puede acceder y modificar el valor del semáforo mediante dos operaciones atómicas:

- *wait* (espera). Un proceso que ejecuta esta operación disminuye el número de instancias disponibles en uno, ya que se supone que la va a utilizar. Si el valor es menor que 0, significa que no hay instancias disponibles. En ese sentido, el proceso queda en estado Bloqueado hasta que el semáforo se libere cuando haya instancias. El valor negativo del semáforo especifica cuántos procesos están bloqueados esperando por el recurso.

```
wait(Semaphore S) {
 S.valor--;
 if (S.valor < 0) {
 Añadir el proceso o hilo a la lista S.cola
   Bloquear la tarea
 }
}
```

- *signal* (señal). Un proceso, cuando termina de usar la instancia del recurso compartido correspondiente, avisa de su liberación mediante la operación *signal*. Para ello aumenta el valor de instancias disponibles en el semáforo. Si el valor es negativo o menor que 0, significará que hay procesos en estado Bloqueado por lo que despertará a uno de ellos obteniéndolo de la cola. Si existen varios procesos esperando, solamente uno de ellos pasará a estado *Runnable.* Esto ocurre cuando el número de recursos es limitado, por ejemplo, una plaza de *parking*. Si un coche abandona el *parking*, solamente otro puede ocupar la plaza dejada sin ser necesario avisar a todos los coches. El hilo que se despierta cuando se hace un *signal* es aleatorio y depende de la implementación de los semáforos y del sistema operativo subyacente.

```
signal(Semaphore S) {
 S.valor++;
 if(S.valor <= 0) {
   Sacar una tarea P de la lista S.cola
   Despertar a P
 }
}
```

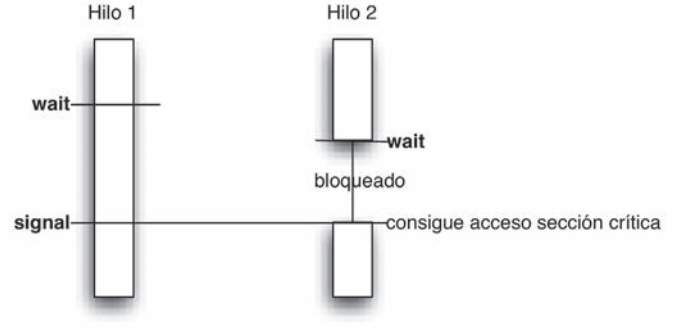


Figura 2.3. *Utilización de* wait *y* signal *para acceder a la sección crítica*

Las operaciones *wait* y *signal* deben ser atómicas para evitar los problemas anteriormente vistos. Para ello, en ordenadores con un único procesador se utiliza internamente la inhibición de interrupciones. En sistemas multiprocesador, donde las instrucciones de cada núcleo se pueden entrelazar de cualquier forma, la solución pasa por utilizar instrucciones hardware especiales, como *TestAndSet* o *Swap*, o soluciones software, como el algoritmo de Peterson.

ACTIVIDADES 2.5

- Existen varios mecanismos para proporcionar una sección crítica dentro de los diferentes sistemas operativos. Busca información sobre los métodos clásicos (algoritmo de Peterson [1981], *TestAndSet* y *Swap*) para comprender su funcionamiento.

  Como podrás ver, estas propuestas son complicadas de usar por parte del programador, además de ser dependientes de la arquitectura del ordenador subyacente. En su lugar se utilizan herramientas que permiten abstraerse de esos problemas, como semáforos, mutex, monitores, etc. en función del sistema operativo y del lenguaje de programación.

Un semáforo binario, también denominado *mutex* (*MUTual EXclusion*, "exclusión mutua" en español) es un indicador de condición que registra si un único recurso está disponible o no. Un *mutex* solo puede tomar los valores 0 y 1. Si el semáforo vale 1, entonces el recurso está disponible y se accede a la zona de compartición del recurso mientras que si el semáforo es 0, el hilo debe esperar.

Los *mutex* son un mecanismo liviano de sincronización idóneo para hilos, ya que posibilitan tanto la exclusión mutua en los accesos a los recursos como la ordenación en el acceso a los mismos pudiendo establecer relaciones antes-después. Para ello, un *mutex* representa un cerrojo sobre una parte de código. Este cerrojo se puede ver como si tuviéramos una cerradura cerrada con una llave puesta para acceder a esa sección de código. Cuando un hilo accede a esa sección, adquiere el semáforo binario, abre la cerradura, la cierra por dentro y se lleva la llave. Hasta que el hilo no finaliza la ejecución de la sección de código, no vuelve a dejar la llave en su sitio, imposibilitando que ningún otro hilo pueda acceder tanto a esa sección como a cualquier otra que requiera esa llave específica (*mutex*). Si la sección de código es aquella que utiliza los recursos compartidos, los semáforos binarios permiten entonces resolver de forma sencilla el problema de la sección crítica. Esta solución fue propuesta por Dijkstra en 1968.

¿SABÍAS QUE...?

Dijkstra es uno de los investigadores más importantes dentro del mundo de la informática, destacando principalmente por sus aportaciones en el campo de la computación distribuida. Como solución a problemas de coordinación de información, propuso la utilización de semáforos. Además propuso soluciones a múltiples problemas relacionados, como el problema del camino más corto (conocido como "algoritmo de Dijkstra"), la cena de filósofos y el algoritmo del banquero.

En Java, la utilización de semáforos se realiza mediante el paquete *java.util.concurrent* y su clase *Semaphore* correspondiente.

ACTIVIDADES 2.6

En este ejemplo se utiliza un único semáforo con valor 1 para crear una sección crítica.

```
import java.util.concurrent.Semaphore;

class Acumula {
    public static int acumulador = 0;
}

class Sumador extends Thread {

    private int cuenta;
    private Semaphore sem;

    Sumador(int hasta, int id, Semaphore sem) {
              this.cuenta = hasta;
              this.sem = sem;
    }

    public void sumar (){
              Acumula.acumulador++;
    }

    public void run() {

         for (int i=0; i< cuenta; i++){
              try {
                   sem.acquire();
              } catch (InterruptedException e) {
                   e.printStackTrace();
              }
              sumar();
              sem.release();
         }
    }
}

public class SeccionCriticaSemaforos {

    private static Sumador sumadores[];
    private static Semaphore semaphore = new Semaphore(1);

    public static void main (String[] args){

         int n_sum = Integer.parseInt (args[0]);
         sumadores = new Sumador[n_sum];
         for (int i= 0; i < n_sum; i++) {
              sumadores[i] = new Sumador(100000000,i, semaphore);
```

```
                sumadores[i].start();
            }

            for (int i= 0; i < n_sum; i++) {
                try {
                    sumadores[i].join();
                } catch (InterruptedException e) {
                        e.printStackTrace();
                }
            }

            System.out.println ("Acumulador: " + Acumula.acumulador);
        }
    }
```

Sin la utilización del semáforo, el resultado acumulador depende de la ejecución en concreto que realicen los hijos, pudiéndose obtener resultados erróneos en algunas ejecuciones. Comprueba que esto sucede comentando el semáforo y analiza a qué se debe.

2.5.2.4.1 Clase Semaphore

A la hora de utilizar semáforos en Java se utiliza la clase *Semaphore.* Se presenta, por simplicidad, una tabla con los métodos recogidos más importantes:

Método	Tipo de retorno	Descripción
Semaphore(int valor)	*void*	Inicialización del semáforo. Indica el valor inicial del semáforo antes de comenzar su ejecución
acquire()	*void*	Implementa la operación *wait*
release()	*void*	Implementa la operación *signal.* Desbloquea un hilo que esté esperando en el método *wait*()

2.5.2.5 Monitores

Un *monitor* es un conjunto de métodos atómicos que proporcionan de forma sencilla exclusión mutua a un recurso. Los métodos indicados permiten que cuando un hilo ejecute uno de los mismos, solamente ese hilo pueda estar ejecutando un método del monitor.

Funciona de forma similar a los semáforos binarios, pero proporciona una mayor simplicidad ya que lo único que tiene que hacer el programador es ejecutar una entrada del monitor. Mientras que un monitor no puede ser utilizado incorrectamente, los semáforos dependen del programador ya que debe proporcionar la correcta secuencia de operaciones para no bloquear el sistema.

Para utilizar un monitor en Java se utiliza la palabra clave *synchronized* sobre una región de código para indicar que se debe ejecutar como si de una sección crítica se tratase. Existen dos formas de utilizar la palabra clave *synchronized*: los métodos y las sentencias sincronizadas.

2.5.2.5.1 Métodos sincronizados

Los métodos sincronizados son un mecanismo para construir una sección crítica de forma sencilla. La ejecución por parte de un hilo de un método sincronizado de un objeto en Java imposibilita que se ejecute a la vez otro método sincronizado del mismo objeto por parte de otro hilo, cumpliendo así con los requisitos de las secciones críticas.

Cuando un hilo invoca un método sincronizado, adquiere automáticamente el monitor que el sistema crea específicamente para todo el objeto que contiene ese método. Solo lo libera cuando el método finaliza o si lanza una excepción no capturada. Ningún otro hilo podrá ejecutar ningún método sincronizado del mismo objeto mientras el monitor de ese objeto no sea liberado, es decir, el cerrojo está cerrado. Esto implica que se bloqueen los métodos que afectan a los recursos compartidos del objeto (indicados con *synchronized*).

ACTIVIDADES 2.7

- Los métodos *static* están asociados a una clase, en vez de al objeto propiamente dicho. Comprueba qué sucede cuando un método sincronizado está definido como *static*.

Para crear un método sincronizado, solo es necesario añadir la palabra clave *synchronized* en la declaración del método, sabiendo que los constructores ya son síncronos por defecto (y no pueden ser marcados como *synchronized*) ya que solo el hilo que lo llama tiene acceso a ese objeto mientras lo está creando.

EJEMPLO 2.11

Creación de métodos sincronizados:

```
public class Contador {
     private int c = 0;

     public void Contador(int num) {
        this.c = num;
     }

     public synchronized void increment() {
         c++;
     }

     public synchronized void decrement() {
        c--;
     }

     public synchronized int value() {
        return c;
     }
}
```

ACTIVIDADES 2.8

- Crea una clase Java que utilice 5 hilos para contar el número de vocales que hay en un determinado texto. Cada hilo se encargará de contar una vocal diferente, actualizando todos los hilos la misma variable común que representa el número de vocales totales. Para evitar condiciones de carrera se deben utilizar métodos sincronizados.

Así, si todos los métodos de lecturas y modificación sobre un objeto al cual accedan varios hilos están sincronizados, se evitan condiciones de carrera e inconsistencias de memoria. Sin embargo, hay que tener en cuenta que la sincronización en este caso no permite la ejecución de otro método sincronizado del mismo objeto al mismo tiempo mientras un método sincronizado de ese objeto se esté ejecutando.

ACTIVIDADES 2.9

- Las variables definidas como *final*, que no se pueden modificar después de construir el objeto, se pueden leer de forma segura (es decir, sin provocar condiciones de carrera) a través de métodos no sincronizados.

  Prueba mediante un ejemplo que no se necesita crear métodos sincronizados para evitar condiciones de carrera al acceder a las mismas.

2.5.2.5.2 Sentencias sincronizadas

Los métodos sincronizados utilizan un monitor que afecta a todo el objeto correspondiente, bloqueando todos los métodos sincronizados del mismo. Esto provoca, por ejemplo, que la ejecución de métodos de solo lectura (sin modificación de datos compartidos) no pueda paralelizarse si existe algún método sincronizado que sí modifique los datos y se quiera mantener el orden entre modificaciones y lecturas.

La utilización de *synchronized* en una sentencia o región específica de código es mucho más versátil y permite una sincronización de grano fino. Esta funcionalidad permite especificar el objeto que proporciona el monitor en vez de ser el objeto por defecto que se está ejecutando como ocurre en métodos sincronizados. De esta forma, se pueden crear nuevos objetos que se compartirán entre los hilos. Al bloquearse únicamente los hilos al acceder a esos nuevos objetos, lo que se consigue es bloquear a los hilos únicamente en las secciones de código especificadas por el programador (secciones críticas).

EJEMPLO 2.12

Utilización de sentencias sincronizadas.

```
class GlobalVar {
    public static int c1 = 0;
    public static int c2 = 0;
}

class TwoMutex extends Thread{

    private Object mutex1 = new Object();
    private Object mutex2 = new Object();

    public void inc1() {
        synchronized(mutex1) {
        GlobalVar.c1++;
        }
    }

    public void inc2() {
        synchronized(mutex2) {
            GlobalVar.c2++;
        }
    }

    public void run()
    {
        inc1();
        inc2();
    }
}

public class MutualExclusion {

    public static void main(String[] args) throws InterruptedException {
        int N = Integer.parseInt (args[0]);
        TwoMutex hilos[];
        System.out.println ("Creando " + N + " hilos");

        hilos= new TwoMutex[N];
        for (int i= 0; i < N; i++)
        {
            hilos[i] = new TwoMutex();
            hilos[i].start();
        }
        for (int i= 0; i < N; i++) {
            hilos[i].join();;
        }
        System.out.println ("C1 = " + GlobalVar.c1);
        System.out.println ("C2 = " + GlobalVar.c2);
    }
}
```

Recordemos que un hilo no puede acceder a una sección protegida por un monitor que ha conseguido otro hilo para ejecutar. Sin embargo, un hilo puede acceder a una sección de código de un monitor que ya posee. Permitir que un hilo pueda adquirir un monitor que ya tiene es lo que se denomina **sincronización reentrante**. Esto describe una situación en la cual un hilo está ejecutando código sincronizado, que, directa o indirectamente, invoca un método que también contiene código sincronizado, y ambos conjuntos de código utilizan el mismo monitor.

¿SABÍAS QUE...?

Se puede utilizar también JNI (*Java Native Interface*) para programar aplicaciones multihilo, además de la comunicación multiproceso vista con anterioridad. En este caso, hay que tener especial cuidado en su realización ya que los métodos nativos se deben programar como si fueran métodos sincronizados. En este sentido, los métodos nativos, no deben modificar variables globales sensibles de forma no protegida y los accesos a las variables deben ser coordinados mediante el método correspondiente que permita el lenguaje de programación nativo para resolver el problema de la sección crítica.

2.5.2.6 Condiciones

En concreto, a veces el hilo que se está ejecutando dentro de una sección crítica no puede continuar, porque no se cumple cierta *condición* que solo podría cambiar otro hilo desde dentro de su correspondiente sección crítica. En este caso es preciso que el hilo que no puede continuar libere temporalmente el cerrojo que protege la sección crítica mientras espera a que alguien le avise de la ocurrencia de dicha condición. Este proceso debe ser atómico y cuando el hilo retoma la ejecución lo hace en el mismo punto donde lo dejó (dentro de la sección crítica). En definitiva, una condición es una variable que se utiliza como mecanismo de sincronización en un monitor.

Para implementar condiciones se pueden utilizar operaciones conocidas. Llamar a la operación *wait* libera automáticamente el cerrojo sobre la sección crítica que se está ejecutando. Para avisar de la ocurrencia de la condición por la que espera, se puede utilizar *signal*. Sin embargo, esta operación no provoca que los hilos notificados empiecen a ejecutarse en ese preciso instante, sino que el hilo notificado no puede ejecutarse hasta que el hilo que lo notificó no libere el cerrojo de la sección crítica por la cual el hilo notificado está esperando.

Si no se tiene en cuenta que el hilo que esperaba no empieza a ejecutarse inmediatamente cuando recibe la notificación se pueden provocar comportamientos anómalos. Como se ve en la Figura 2.4, el hilo no se pone en ejecución hasta que consigue el cerrojo correspondiente, por el cual pelea en igualdad de condiciones con el resto de hilos. Esto podría provocar que otro hilo ejecute antes y le robe los recursos necesarios (condición) por los que el otro hilo esperó. Por eso mismo, la comprobación de la condición de espera no se debe realizar en una sentencia *if*, sino en un *while*. Cuando el hilo vuelva ejecutar después de realizar la operación *wait* siempre debe comprobar si la condición por la que esperó y le notificaron se cumple a su vuelta para poder seguir con la ejecución.

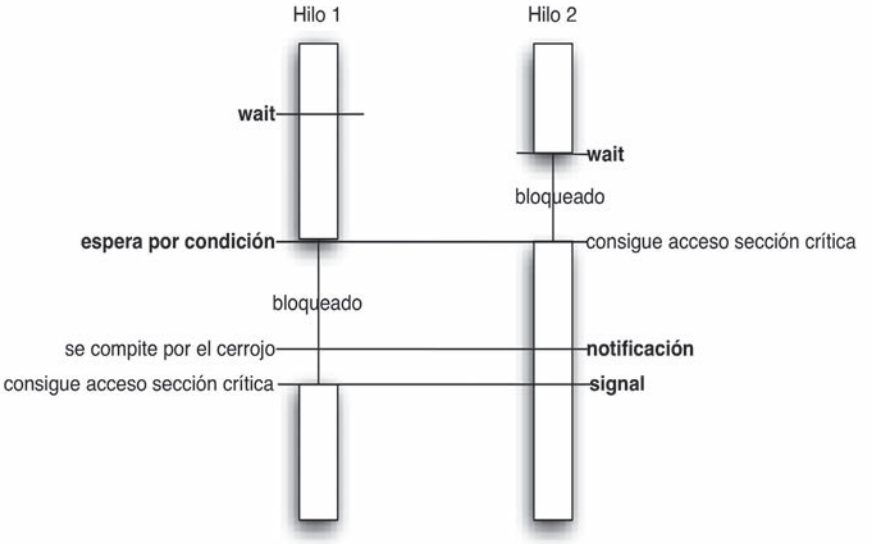


Figura 2.4. *Utilización de varios hilos con condiciones*

La implementación de condiciones en Java se realiza mediante la utilización de la clase *Object*. Cualquier hilo puede utilizar la operación *signal* en cualquier objeto en el cual otro hilo ejecutó la operación *wait*. Es decir los *wait* y *signal* tienen relación únicamente sobre el mismo objeto. Un hilo que espere con un *wait* sobre un objeto no se despertará por *signal* de otros hilos en otros objetos. Los métodos que corresponden a la implementación de las operaciones *wait* y *signal* para condiciones son *wait*() y *notify*(). Dichos métodos siempre se deben ejecutar sobre un bloque sincronizado, es decir, dentro de un monitor.

```
synchronized public void comprobacion_ejecucion()
{
     // Seccion critica
     while (condicion) //no pueda continuar
     {
          wait();
       }
      // Seccion critica
}

synchronized public void aviso_condicion()
{
      // Seccion critica
      if (condicion_se_cumple)
          notify();
      // Seccion critica
}
```

2.5.2.6.1 Clase Object

A la hora de utilizar condiciones en Java se utiliza la clase *Object*. Se presenta, por simplicidad, una tabla con los métodos recogidos más importantes explicados a lo largo del capítulo sabiendo que deben ejecutarse desde bloques sincronizados:

Método	Tipo de retorno	Descripción
wait()	*void*	Implementa la operación *wait*. El hilo espera hasta que otro hilo invoque *notify* o *notifyAll()*
notify()	*void*	Implementa la operación *signal.* Desbloquea un hilo que esté esperando en el método *wait()*
notifyAll()	*void*	Despierta a todos los hilos que estén esperando para que continúen con su ejecución. Todos los hilos esperando por *wait* reanudan su ejecución. Por ejemplo, si hay un proceso escritor, escribiendo datos, cuando finalice puede avisar a todos los procesos lectores para que continúen su ejecución a la vez (ya que pueden operar todos a la vez)

EJEMPLO 2.13

Utilización de condiciones. La condición de *clase comenzada* no sería necesaria ya que se podría gestionar fácilmente con *wait* y *notifyAll* de forma sencilla:

```
class Bienvenida {

    boolean clase_comenzada;

    public Bienvenida(){
              this.clase_comenzada = false;
    }

    // Hasta que el profesor no salude no empieza la clase,
    // por lo que los alumnos esperan con un wait
    public synchronized void saludarProfesor(){
         try {
              while (clase_comenzada == false){
                    wait();
              }
              System.out.println("Buenos días, profesor.");
          } catch (InterruptedException ex) {
              ex.printStackTrace();
          }
     }
```

EJEMPLO 2.13 (cont.)

```
     // Cuando el profesor saluda avisa a los alumnos con notifyall de su llegada
      public synchronized void llegadaProfesor(String nombre){
           System.out.println("Buenos días a todos. Soy el profesor " + nombre);
           this.clase_comenzada = true;
           notifyAll();
     }
}

class Alumno extends Thread{
     Bienvenida saludo;

     public Alumno(Bienvenida bienvenida){
          this.saludo = bienvenida;
     }

     public void run(){
          System.out.println("Alumno llegó.");
          try {
               Thread.sleep(1000);
               saludo.saludarProfesor();
           } catch (InterruptedException ex) {
             System.err.println("Thread alumno interrumpido!");
             System.exit(-1);
           }
      }
}

class Profesor extends Thread{
     String nombre;
     Bienvenida saludo;

      public Profesor(String nombre, Bienvenida bienvenida){
             this.nombre = nombre;
             this.saludo = bienvenida;
       }

       public void run(){
             System.out.println(nombre + " llegó.");
             try {
                     Thread.sleep(1000);
                     saludo.llegadaProfesor(nombre);
             } catch (InterruptedException ex) {
             System.err.println("Thread profesor interrumpido!");
               System.exit(-1);
             }
       }
}
```

EJEMPLO 2.13 (cont.)

```
public class ComienzoClase {

    public static void main(String[] args) {

        // Objeto compartido
        Bienvenida b = new Bienvenida();

        int n_alumnos = Integer.parseInt (args[0]);
        for (int i=0; i< n_alumnos; i++){
              new Alumno(b).start();
         }
        Profesor profesor = new Profesor("Osvaldo Ramirez",b);
        profesor.start();
    }
}
```

2.6 PROGRAMACIÓN DE APLICACIONES MULTIHILO

La **programación multihilo** permite la ejecución de varios hilos al mismo tiempo, tantos hilos como núcleos tenga el procesador, para realizar una tarea común. A la hora de realizar un programa multihilo cooperativo, se deben seguir las mismas fases que para la programación de aplicaciones multiproceso, aunque su comprensión difiere por las diferencias existentes entre hilos y procesos. Los pasos serían los siguientes:

Descomposición funcional. Es necesario identificar previamente las diferentes tareas que debe realizar la aplicación y las relaciones existentes entre ellas.

Partición. La comunicación entre hilos se realiza principalmente a través de memoria. Los tiempos de acceso son muy rápidos por lo que no supone una pérdida de tiempo significativa. Hay que tener en cuenta que la existencia de secciones críticas bloquea a los procesos para su sincronización, provocando una pérdida del rendimiento que se puede conseguir. Es conveniente minimizar la dependencia de sincronización existente entre los hilos, para aumentar la ejecución paralela o ejecución asíncrona.

Implementación. Se utiliza la clase *Thread* o la interfaz *Runnable* como punto de partida.

ACTIVIDADES 2.10

Este libro ha contado los conceptos de bajo nivel para entender en profundidad los hilos y cómo operan. A partir de este funcionamiento básico, se puede implementar cualquier solución a problemas multihilo. Para facilitar la programación de aplicaciones complejas, a partir de la versión 5 de Java se creó un nuevo paquete de datos *java.util.concurrent* donde aparecen clases de más alto nivel, que se abstraen del bajo nivel mostrado por las clases vistas en el capítulo.

Lee el siguiente tutorial, donde se muestran las nuevas clases *http://docs.oracle.com/javase/tutorial/essential/concurrency/highlevel.html* para comprender los nuevos conceptos probando su funcionalidad. La clase *Semaphore* pertenece a este paquete.

2.7 CASO PRÁCTICO

En este caso práctico se va a desarrollar una solución multitarea al problema clásico de la cena de filósofos.

En una mesa redonda hay N filósofos sentados. En total tiene N palillos para comer arroz, estando cada palillo compartido por dos filósofos, uno a la izquierda y otro a la derecha. Como buenos filósofos, se dedican a pensar, aunque de vez en cuando les entra hambre y quieren comer. Para poder comer, un filósofo necesita utilizar los dos palillos que hay a sus lados.

Para implementar este problema se debe crear un programa principal que cree N hilos ejecutando el mismo código. Cada hilo representa un filósofo. Una vez creado, se realiza un bucle infinito de espera. Cada una de los hilos tendrá que realizar los siguientes pasos:

1. Imprimir un mensaje por pantalla "Filósofo *i* pensando", siendo *i* el identificador del filósofo.
2. Pensar durante un cierto tiempo aleatorio.
3. Imprimir un mensaje por pantalla "Filósofo *i* quiere comer".
4. Intentar coger los palillos que necesita para comer. El filósofo 0 necesitará los palillos 0 y 1, el filósofo 1, los palillos 1 y 2, y así sucesivamente.
5. Cuando tenga el control de los palillos, imprimirá un mensaje en pantalla "Filósofo *i* comiendo".
6. El filósofo estará comiendo durante un tiempo aleatorio.
7. Una vez que ha finalizado de comer, dejará los palillos en su sitio.
8. Volver al paso 1.

Sin embargo, se pueden producir interbloqueos si por ejemplo todos los filósofos quieren comer a la vez. Si todos consiguen coger el palillo de su izquierda ninguno podrá coger el de su derecha. Para ello se plantean varias soluciones:

- Permitir que como máximo haya N–1 filósofos sentados a la mesa.
- Permitir a cada filósofo coger sus palillos solamente si ambos palillos están libres.
- Solución asimétrica: un filósofo impar coge primero el palillo de la izquierda y luego el de la derecha. Un filósofo par los coge en el orden inverso.

Implementar una solución al problema de los filósofos, solución que no presente un problema de interbloqueo. Por sencillez, se recomienda utilizar el método propuesto de solución asimétrica.

RESUMEN DEL CAPÍTULO

El capítulo ha mostrado el funcionamiento de los sistemas operativos en lo relativo a la ejecución de tareas. Se ha definido el concepto de tarea o hilo como una secuencia de código en ejecución que se puede ejecutar en paralelo con otras tareas siempre que sea dentro del contexto del mismo proceso. La utilización de hilos permite aprovechar procesadores que tienen varios núcleos, ya que permite que cada uno de los núcleos ejecute al mismo tiempo una instrucción de uno de los hilos que pertenecen al mismo proceso (ya que en un momento determinado solo puede haber un único proceso en ejecución). Los hilos perteneciente al mismo proceso comparten entre sí tanto código como memoria (variables) y otros recursos que están asignados al proceso.

Los hilos se pueden estar ejecutando siempre que el proceso esté en ejecución. Sin embargo, como el sistema operativo gestiona procesos, no hilos, no se sabe si en un momento determinado el *thread* tiene recursos hardware suficientes (un núcleo disponible) para su ejecución. Eso hace que su estado habitual sea *Runnable* indicando que puede estarse ejecutando en función de los recursos disponibles. Además de eso, los hilos pueden bloquearse por motivos de sincronización.

Al igual que los procesos, durante la ejecución de un programa se pueden arrancar nuevos hilos (que tienen que estar previamente programados) en el espacio de código del proceso. Cualquier proceso tiene un hilo principal que representa la ejecución básica del proceso. La operación *create* permite arrancar nuevos hilos y *join* esperar la finalización de los mismos. Esto permite realizar múltiples tareas de forma eficiente al mismo tiempo (no es necesario realizar cambios de contexto, como sí lo sería si estuvieran expresadas mediante procesos, y su ejecución es en paralelo, y no de concurrente) pero hay que tener especial cuidado con su gestión. Al compartir recursos, un hilo podría afectar la ejecución de otro hilo existiendo problemas de comunicación y sincronización. Dichos problemas van desde que el resultado de la ejecución de un programa dependa del orden en concreto en que se realicen los accesos a memoria (condiciones de carrera o inconsistencias de memoria), hasta el bloqueo o inanición de hilos que trabajan conjuntamente en función del código generado.

El acceso compartido a memoria suele generar los problemas anteriores, por lo que acceder en exclusión mutua a las zonas compartidas puede solucionar el problema. Cuando un hilo está ejecutando su sección crítica, ningún otro hilo puede ejecutar su correspondiente sección crítica, ordenando de esta forma la ejecución. La implementación de una sección crítica puede variar de un sistema operativo a otro, utilizando diversos mecanismos como semáforos, *mutex* y monitores. En general, para su implementación, se necesita un mecanismo de sincronización que indique la entrada a la sección crítica (operación *wait*) y su salida (operación *signal*). Sin embargo, a veces el hilo que está ejecutando dentro de una sección crítica no puede continuar porque no se cumple cierta condición que solo podría cambiar otro hilo desde su correspondiente sección crítica. En este caso, es preciso que el hilo que no pueda continuar libere temporalmente la sección crítica y espere hasta la ocurrencia de dicha condición. Mediante el uso de secciones críticas y condiciones se puede gestionar la programación multihilo.

EJERCICIOS PROPUESTOS

1. Escribe una clase llamada *Orden* que cree dos hilos y fuerce que la escritura del segundo sea siempre anterior a la escritura por pantalla del primero.

Ejemplo de ejecución:

```
Hola, soy el thread número 2
Hola, soy el thread número 1
```

2. Escribe una clase llamada *Check* que cree dos *threads* que accedan simultáneamente a un *buffer* de 10.000 enteros. Uno de ellos lee en el *buffer* y el otro escribe en el mismo. El *thread* escritor debe escribir el mismo valor en todos los elementos del *buffer* incrementando en uno el valor en cada pasada. El *thread* lector debe ir comprobando que todos los números del *buffer* son iguales, mostrando un mensaje de error en caso contrario o un mensaje de correcto si la condición se cumple. El código a realizar utilizará un monitor para acceder al *buffer* si se indica un parámetro al ejecutar el programa. En caso contrario, los *threads* accederán al *buffer* sin hacer uso del monitor.

3. Escribe una clase llamada *Relevos* que simule una carrera de relevos de la siguiente forma:

- Cree 4 *threads*, que se quedarán a la espera de recibir alguna señal para comenzar a correr. Una vez creados los *threads*, se indicará que comience la carrera, con lo que uno de los *threads* deberá empezar a correr.
- Cuando un *thread* termina de correr pone algún mensaje en pantalla y espera un par de segundos, pasando el testigo a otro de los hilos para que comience a correr, y terminando su ejecución (la suya propia).
- Cuando el último *thread* termine de correr, el padre mostrará un mensaje indicando que todos los hijos han terminado.

Ejemplo de ejecución:

```
Todos los hilos creados.
Doy la salida!
Soy el thread 1, corriendo . . .
Terminé. Paso el testigo al hijo 2
Soy el thread 2, corriendo . . .
Terminé. Paso el testigo al hijo 3
Soy el thread 3, corriendo . . .
Terminé. Paso el testigo al hijo 4
Soy el thread 4, corriendo . . .
Terminé!
Todos los hilos terminaron.
```

4. Escribe una clase llamada *SuperMarket* que implemente el funcionamiento de N cajas de un supermercado. Los M clientes del supermercado estarán un tiempo aleatorio comprando y con posterioridad seleccionarán de forma aleatoria en qué caja posicionarse para situarse en su cola correspondiente. Cuando les toque el turno serán atendidos procediendo al pago correspondiente e ingresando en la variable Resultados del supermercado. Se deben crear tantos *threads* como clientes haya y los parámetros M y N se deben pasar como argumentos al programa. Para simplificar la implementación, el valor de pago de cada cliente puede ser aleatorio en el momento de su pago.

5. Escribe una clase llamada *ModernSuperMarket* que implemente el funcionamiento de N cajas de supermercado. Los mismos M clientes del supermercado realizarán el mismo proceso que en el ejercicio anterior, situándose cuando han realizado la compra, en este caso, en una única cola. Cuando cualquier caja esté disponible, el primero de la cola será atendido en la caja correspondiente. Calcula el tiempo medio de espera por cliente y compáralo con el tiempo medio que se obtendría en el ejercicio anterior.

¿Cuál de las dos alternativas es más eficiente? ¿Cuál elegirías si tú tuvieras un supermercado? Razona la respuesta.

6. Escribe una clase llamada *Parking* que reciba el número de plazas del *parking* y el número de coches existentes en el sistema. Se deben crear tantos *threads* como coches haya. El *parking* dispondrá de una única entrada y una única salida. En la entrada de vehículos habrá un dispositivo de control que permita o impida el acceso de los mismos al *parking*, dependiendo del estado actual del mismo (plazas de aparcamiento disponibles). Los tiempos de espera de los vehículos dentro del *parking* son aleatorios. En el momento en el que un vehículo sale del *parking*, notifica al dispositivo de control el número de la plaza que tenía asignada y se libera la plaza que estuviera ocupando, quedando así estas nuevamente disponibles. Un vehículo que ha salido del *parking* esperará un tiempo aleatorio para volver a entrar nuevamente en el mismo. Por tanto, los vehículos estarán entrando y saliendo indefinidamente del *parking*. Es importante que se diseñe el programa de tal forma que se asegure que, antes o después, un vehículo que permanece esperando a la entrada del *parking* entrará en el mismo (no se produzca inanición).

Ejemplo de ejecución:

```
ENTRADA: Coche 1 aparca en 0.
Plazas libre: 5
Parking: [1] [2] [3] [0] [0] [0]
ENTRADA: Coche 2 aparca en 1.
Plazas libre: 4
Parking: [1] [2] [3] [0] [0] [0]
ENTRADA: Coche 3 aparca en 2.
Plazas libre: 3
Parking: [1] [2] [3] [0] [0] [0]
ENTRADA: Coche 4 aparca en 3.
Plazas libre: 2
Parking: [1] [2] [3] [4] [0] [0]
ENTRADA: Coche 5 aparca en 4.
Plazas libre: 1
Parking: [1] [2] [3] [4] [5] [0]
SALIDA: Coche 2 saliendo.
Plazas libre: 2
```

7. Escribe una clase llamada *ParkingCamion* que reciba el número de plazas del *parking*, el número de coches y el número de camiones existentes en el sistema. Dicha clase debe realizar lo mismo que la clase *Parking* pero debe permitir a su vez aparcar camiones. Mientras un automóvil ocupa una plaza de aparcamiento dentro del *parking*, un camión ocupa dos plazas contiguas de aparcamiento. Hay que tener especial cuidado con la inanición de camiones, que puede producirse si están saliendo coches indefinidamente y asignando la nueva plaza a los coches que esperan en vez de esperar a que haya un hueco para el camión (un camión solo podrá acceder al *parking* si hay, al menos, dos plazas contiguas de aparcamiento libre).

Ejemplo de ejecución:

```
ENTRADA: Coche 1 aparca en 0.
Plazas libre: 5
Parking: [1] [2] [3] [0] [0] [0]
ENTRADA: Camión 101 aparca en 2.
Plazas libre: 3
Parking: [1] [101] [101] [0] [0] [0]
ENTRADA: Coche 2 aparca en 4.
Plazas libre: 2
Parking: [1] [101] [101] [2] [0] [0]
ENTRADA: Coche 3 aparca en 5.
Plazas libre: 1
Parking: [1] [101] [101] [2] [3] [0]
SALIDA: Coche 1 saliendo.
Plazas libre: 2
Parking: [0] [101] [101] [2] [3] [0]
```

TEST DE CONOCIMIENTOS

1 Cuando varios hilos acceden a datos compartidos, y el resultado de la ejecución depende del orden concreto en que se accede a los datos compartidos se dice que:

a) Hay una sección crítica.
b) Hay una condición de carrera.
c) Eso no puede suceder, los hilos no pueden compartir datos.
d) Ninguna de las anteriores.

2 Cualquier solución al problema de la sección crítica debe cumplir las condiciones de:

a) Exclusión mutua, progreso y espera limitada.
b) Exclusión mutua, espera limitada y retención.
c) Envejecimiento e inanición.
d) Exclusión mutua, aislamiento y espera limitada.

3 Indica cuál de las siguientes afirmaciones sobre los monitores es FALSA:

a) Permiten resolver el problema de la sección crítica.
b) Pueden ser binarios o contadores.
c) Se pueden utilizar para garantizar el orden de ejecución entre procesos.
d) Son un mecanismo de sincronización.

4 Dado el siguiente fragmento de código sobre un Objeto en particular para realizar una sección crítica:

```
notify();
SECCIÓN CRÍTICA
wait();
```

¿Cuál de las siguientes afirmaciones es cierta?

a) El código mostrado no asegura retención.
b) El código mostrado no asegura exclusión mutua.
c) Es una solución valida para el problema de la sección crítica cuando el código se ejecuta en un único procesador.
d) El código mostrado permite resolver el problema de la sección crítica.

5 Indica cuál de las siguientes afirmaciones sobre los semáforos es FALSA:

a) Permiten resolver el problema de la sección crítica.
b) Pueden ser binarios o contadores.
c) No se pueden utilizar para garantizar el orden de ejecución entre procesos.
d) Son un mecanismo de sincronización.

6 Dado el siguiente fragmento de código, y suponiendo que el objeto *Semaphore S* tiene como valor inicial 2:

```
S.release(S);
Funcion();
S.acquire(S);
```

¿Cuántos procesos pueden ejecutar al mismo tiempo la tarea Función?

a) 0
b) 2
c) 3
d) Ninguna de los anteriores. Depende de la ejecución.

7 Dado el siguiente fragmento de código ejecutado por un hilo:

```
syncronyzhed(Object){
…
if (<<se cumple condicion>>)
    wait();
FUNCIÓN}
```

¿Cuál de las siguientes afirmaciones es cierta?

a) El *thread* solo ejecutará FUNCIÓN cuando no se cumpla la condición.

b) El *thread* podría ejecutar FUNCIÓN aunque esta se cumpla, debido a la espera del hilo por conseguir el monitor.

c) El *thread* podría ejecutar FUNCIÓN aunque esta se cumpla, debido que es un problema inherente al uso paralelo de *threads*.

d) El *thread* nunca conseguirá ejecutar FUNCIÓN.

8 Una condición de carrera:

a) Sucede por permitir que varios procesos manipulen variables compartidas de forma concurrente.

b) Existe cuando varios procesos acceden a los mismos datos en memoria y el resultado de la ejecución depende del orden concreto en que se realicen los accesos.

c) No necesita de mecanismos software para ser impedida, ya que siempre que se ejecuta el código de procesos concurrentes con los mismos argumentos, estos se ejecutarán de la misma forma.

d) a y b son correctas.

9 Dado el siguiente fragmento de código suponiendo que el objeto *Semaphore sem* tiene como valor inicial 2:

```
sem.acquire();
```

¿Qué sucederá?

a) El proceso que ejecuta la operación se bloquea hasta que otro ejecute una operación *sem.release()*;

b) El proceso continuará adelante sin bloquearse, y si previamente existían procesos bloqueados a causa del semáforo, se desbloqueará uno de ellos.

c) Tras hacer la operación, el proceso continuará adelante sin bloquearse.

d) Un semáforo jamás podrá tener el valor 2, si su valor inicial era 0 (cero) y se ha operado correctamente con él.

10 Si se usa un bloque sincronizado para lograr la sincronización de procesos:

a) Siempre se deben incluir variables de condición, pues el bloque únicamente proporciona exclusión mutua.

b) Las operaciones *wait* y *notify* se utilizan dentro de un mismo objeto.

c) Las operaciones *wait* y *notify* se utilizan en objetos separados.

d) La variable de condición se especifica siempre con una condición *if*.

3 Programación de comunicaciones en red

OBJETIVOS DEL CAPÍTULO

- ✓ Aprender los conceptos básicos de la computación distribuida.
- ✓ Conocer los protocolos básicos de comunicación entre aplicaciones y los principales modelos de computación distribuida.
- ✓ Aprender a programar aplicaciones que se comuniquen con otras en red mediante *sockets*.
- ✓ Desarrollar de forma práctica los principios fundamentales del modelo cliente/servidor.

3.1 CONCEPTOS BÁSICOS: COMUNICACIÓN ENTRE APLICACIONES

En los orígenes de la informática, allá por la década de 1960 (o incluso antes), la computación estaba concebida como grandes ordenadores centrales localizados en lugares específicos, como universidades, laboratorios, etc., y aislados entre sí. Con la proliferación de los ordenadores personales y la red Internet, décadas después, surgió una nueva forma de concebir la computación, en la que múltiples computadores colaboran entre sí, comunicándose a través de una red: la computación distribuida. Esta área de la computación abarca mutitud de aplicaciones, desde el clásico correo electrónico hasta la moderna computación en la nube. Para poder desarrollar estas aplicaciones es necesario conocer sus fundamentos y las técnicas que se utilizan para programarlas.

3.1.1 COMPUTACIÓN DISTRIBUIDA

Hoy en día, un gran número de los sistemas computacionales que existen siguen el **modelo de computación distribuida**. Desde las aplicaciones a través de Internet y móviles que se usan a diario, pasando por los juegos *on-line* y las redes de distribución de contenido multimedia (música, vídeo, etc.), hasta la mayoría de grandes superordenadores modernos, todos ellos son, de hecho, ejemplos de **sistemas distribuidos**. Las tres características fundamentales de todo sistema distribuido son:

- ✓ Está formado por **más de un elemento computacional** distinto e independiente. Un elemento computacional puede ser un procesador dentro de una máquina, un ordenador dentro de una red, una aplicación funcionando a través de Internet, etc. Ninguno de estos elementos comparte memoria con el resto.
- ✓ Los elementos que forman el sistema distribuido **no están sincronizados** entre sí (cada uno tiene su propio reloj).
- ✓ Todos los elementos del sistema **están conectados a una red de comunicaciones** que les permite comunicarse entre sí.

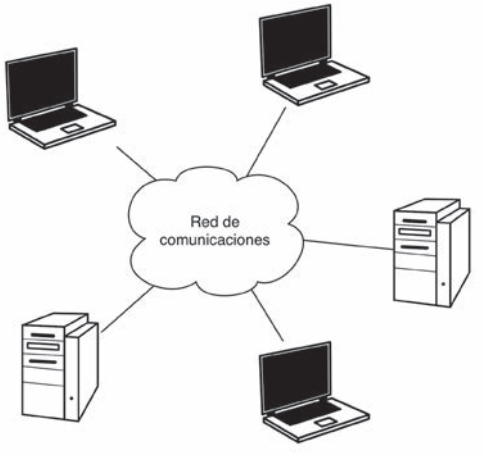


Figura 3.1. *Ejemplo de un sistema distribuido*

¿SABÍAS QUE...?

Una de las formas de computación distribuida más de moda en la actualidad es la llamada "computación en la nube" (*cloud computing*). En este modelo computacional, un proveedor de recursos ofrece servicios a un usuario, normalmente a través de Internet. Ejemplos de estos servicios son los de almacenamiento de datos (como DropBox), servicio de música (como Spotify) o aplicaciones de ofimática (como Google Docs/Drive). El proveedor abstrae los mecanismos internos con los que se proporcionan estos servicios, de tal manera que el usuario accede a ellos de manera transparente. Este proceso de abstracción se conoce normalmente como "virtualización de servicios".

3.1.2 COMUNICACIÓN ENTRE APLICACIONES

En un sistema distribuido, las aplicaciones que lo forman se comunican entre sí para conseguir un objetivo. Todo este proceso de comunicación involucra una serie de conceptos fundamentales, que debemos tener siempre presentes: **mensaje**, **emisor**, **receptor**, **paquete**, **canal de comunicación** y **protocolo de comunicaciones**.

¿SABÍAS QUE...?

Cuando un usuario navega por Internet haciendo una búsqueda en el buscador Google, por ejemplo, la aplicación de navegación web (el navegador) que esté usando, y que reside en su ordenador o móvil, se comunica con un servidor de páginas web, que es otra aplicación que reside en los ordenadores centrales de Google. Esta comunicación se realiza a través de la red Internet, y el resultado es que el usuario obtiene la información que había solicitado.

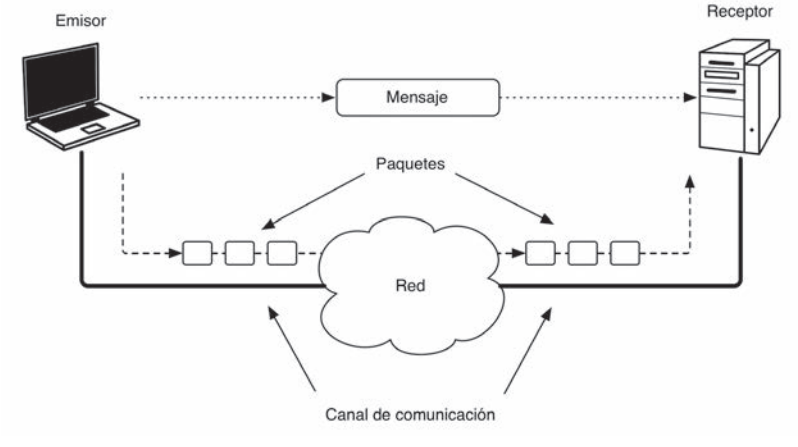


Figura 3.2. *Elementos de la comunicación entre aplicaciones*

3.1.2.1 Mensaje

El **mensaje** es la información que se intercambia entre las aplicaciones que se comunican. Por ejemplo, podría ser una solicitud ("deseo ver el contenido de la página web *www.google.es*"), y su respuesta ("contenido de la web").

3.1.2.2 Emisor

Es la aplicación que envía el mensaje.

3.1.2.3 Receptor

Es la aplicación que recibe el mensaje. Dependiendo del modelo de comunicación distribuida escogido (cliente/servidor, comunicación en grupo, P2P, etc.), un mensaje puede tener más de un receptor. Esto se verá en detalle a lo largo de este capítulo.

3.1.2.4 Paquete

Para transmitir un mensaje a través de una red de comunicaciones, este debe dividirse en uno o más paquetes. Un paquete es la unidad básica de información que intercambian dos dispositivos de comunicación.

3.1.2.5 Canal de comunicación

Es el medio por el que se transmiten los paquetes, que conecta el emisor con el receptor.

3.1.2.6 Protocolo de comunicaciones

Es el conjunto de reglas que fijan cómo se deben intercambiar paquetes entre los diferentes elementos que se comunican entre sí. Un **protocolo** define, por un lado, la secuencia de paquetes que se deben intercambiar para transmitir los mensajes y, por otro, el formato de los mensajes. Las características de un protocolo de comunicaciones se verán en detalle más adelante en este capítulo.

ACTIVIDADES 3.1

- Busca información sobre la historia de la computación distribuida. Averigua qué era ARPANET y cuál es su relación con la red Internet que conocemos actualmente.
- Otro paradigma de computación relacionado con la computación distribuida es la computación paralela. Busca información sobre esta e identifica las diferencias fundamentales entre ambas.
- ¿Qué tipos de redes de comunicaciones conoces? ¿Cuál es la diferencia entre una red de área local (LAN) y una de área extendida (WAN)?

3.2 PROTOCOLOS DE COMUNICACIONES: IP, TCP, UDP

Para que las diferentes aplicaciones que forman un sistema distribuido puedan comunicarse, debe existir una serie de mecanismos que hagan posible esa comunicación. Estos mecanismos están formados por elementos de dos posibles tipos:

- *Elementos hardware*: son aquellos dispositivos físicos necesarios para la comunicación en sí misma, tales como interfaces de red, encaminadores de tráfico (*routers*), etc.
- *Elementos software*: son aquellas herramientas, bibliotecas de programación, componentes del sistema operativo y demás piezas de software necesarias para hacer posible la comunicación.

Todos estos componentes se organizan en lo que se denomina una **jerarquía** o **pila de protocolos**. Esta pila contiene todos los elementos hardware y software necesarios para la comunicación, y establece cómo trabajan juntos para conseguirla. En la actualidad, la pila de protocolos usada en la mayoría de sistemas distribuidos se conoce como **pila IP**, y es la que se utiliza en la red Internet.

¿SABÍAS QUE...?

Un *router* o encaminador de tráfico es una máquina cuya función es dirigir los mensajes que se propagan a través de una red. Un *router* funciona como un nodo de distribución, captando los mensajes que le llegan y eviándolos por la red en la dirección adecuada, dependiendo de su destinatario. Sin la ayuda de los *routers*, los mensajes que se propagan por redes como Internet nunca llegarían a su destino.

¿SABÍAS QUE...?

Una dirección IP es un código numérico que identifica a una máquina de manera única dentro de una red de comunicaciones. En una red, la dirección IP equivale a la dirección postal (ciudad, calle, número, código postal, etc.) de una casa en el mundo real. Sin una dirección IP asignada, una máquina no puede recibir ni enviar mensajes.

3.2.1 PILA DE PROTOCOLOS IP

Como se ha mencionado anteriormente, el protocolo IP[1] es el protocolo estándar de comunicaciones en Internet y en la mayoría de sistemas distribuidos. Define además una pila de protocolos completa con los siguientes niveles, por orden ascendente: **nivel de red**, **nivel de Internet**, **nivel de transporte** y **nivel de aplicación**.

Figura 3.3. *Pila de protocolos IP*

3.2.1.1 Nivel de red

Es el nivel más bajo. Lo componen los elementos hardware de comunicaciones y sus controladores básicos. Se encarga de trasmitir los paquetes de información. Está construido sobre una red de comunicaciones de área local (LAN)[2] como la red interna de una empresa, una red de área extendida (WAN)[3] como Internet, o una combinación de ambas.

3.2.1.2 Nivel de Internet

También llamado **nivel IP**. Se sitúa justo encima del nivel de red, y lo componen los elementos software que se encargan de dirigir los paquetes por la red, asegurándose de que lleguen a su destino.

3.2.1.3 Nivel de transporte

Se sitúa justo encima del nivel de Internet. Lo componen los elementos software cuya función es crear el canal de comunicación, descomponer el mensaje en paquetes y gestionar su transmisión entre el emisor y el receptor. En la pila de protocolos IP existen dos protocolos de transporte fundamentales: TCP y UDP. Ambos se verán en detalle a lo largo de este capítulo.

1 Las siglas en inglés de *Internet Protocol*.

2 Las siglas en inglés de *Local Area Network*.

3 Las siglas en inglés de *Wide Area Network*.

3.2.1.4 Nivel de aplicación

Por último, se encuentra el nivel de aplicación, justo encima del nivel de transporte. Lo componen las aplicaciones que forman el sistema distribuido, que hacen uso de los niveles inferiores para poder transferir mensajes entre ellas.

3.2.1.5 Funcionamiento de la pila de protocolos

Las aplicaciones de un sistema distribuido se sitúan en el nivel superior (nivel de aplicación). Cuando una aplicación emisora decide enviar un mensaje se produce la siguiente secuencia de operaciones (ver Figura 3.4):

1. La aplicación del emisor entrega el mensaje al nivel inmediatamente inferior, es decir, el nivel de transporte.
2. El protocolo del nivel de transporte descompone el mensaje en paquetes, y los pasa al nivel inferior (nivel de Internet).
3. El nivel de Internet localiza al receptor del mensaje y calcula la ruta que deben seguir los paquetes para llegar a su destino. Una vez hecho esto, entrega los paquetes al nivel inferior (nivel de red).
4. El nivel de red transmite los paquetes hasta el receptor.
5. Una vez los paquetes van llegando al receptor, el nivel de red los recibe y los pasa a su nivel superior (nivel de Internet).
6. El nivel de Internet comprueba que los paquetes recibidos han llegado al destinatario (receptor) correcto. Si es así, los envía al nivel superior (nivel de transporte).
7. El nivel de transporte agrupa los paquetes recibidos para formar el mensaje. Una vez el mensaje ha sido reconstruido, lo envía al nivel superior (nivel de aplicación).
8. Por último, la aplicación receptora recibe el mensaje.

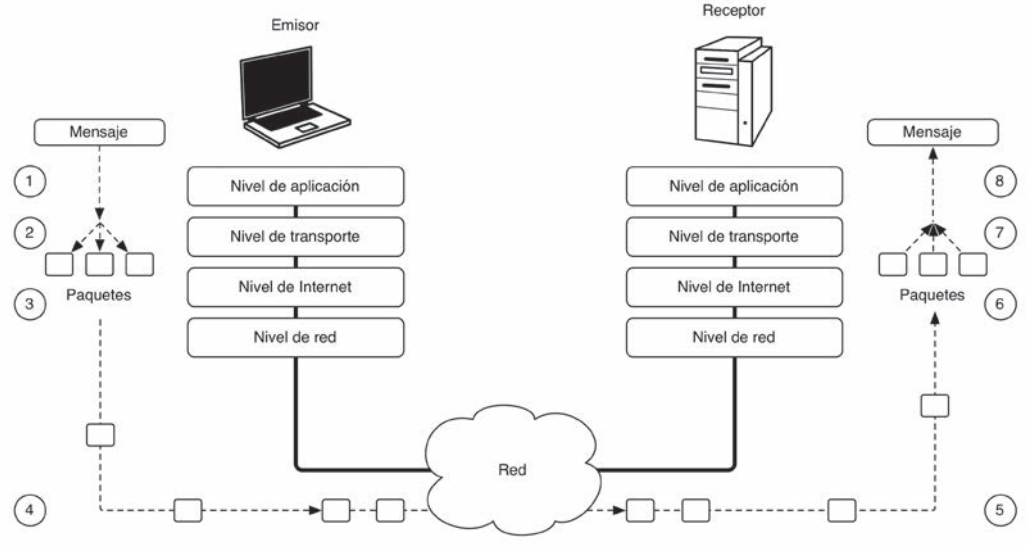


Figura 3.4. *Transmisión de mensajes usando la pila de protocolos IP*

¿SABÍAS QUE...?

El modelo OSI es un estándar para las comunicaciones en red muy similar a la pila IP. Fue propuesto por la Organización Internacional de Estandarización para crear un estándar universal de comunicaciones, y define una arquitectura en capas muy parecida a la jerarquía IP. No obstante, su aceptación ha sido desigual, y en la actualidad IP es el estándar *de facto,* usado en la mayoría de sistemas distribuidos modernos.

ACTIVIDADES 3.2

- Busca más información sobre la pila de protocolos OSI. ¿En qué se diferencia de la pila IP?
- ¿Sabes cómo se transmiten los mensajes a través de Internet? Busca información sobre cómo funcionan los mecanismos de encaminamiento del protocolo IP en el nivel de Internet.

3.2.2 PROTOCOLO TCP

El **protocolo de transporte TCP**[4] es el más comúnmente utilizado en la pila IP, hasta el punto de que muchas veces se denomina a esta como "pila TCP/IP". Como cualquier protocolo de transporte, TCP se encarga de subdividir los mensajes que recibe del nivel del aplicación, creando paquetes que se envían al nivel de Internet, y combinar los paquetes recibidos de este mismo nivel para formar mensajes que se pasan al nivel superior (nivel de aplicación). Las características principales de TCP son:

- ✓ Garantiza que los datos no se pierden, siempre y cuando la comunicación sea posible. Esto quiere decir que, siempre que los niveles inferiores de la pila de protocolos funcionen, los mensajes enviados llegarán a su destino.
- ✓ Garantiza que los mensajes llegarán en orden, siempre y cuando la comunicación sea posible. Esto implica que, no solo todos los mensajes que se envían llegarán (como se indica en el punto anterior), además lo harán siempre en el orden en que fueron enviados.
- ✓ Se trata de un **protocolo orientado a conexión**. Esto repercute en la forma que se crea y se gestiona el canal de comunicación, y se explica en detalle a continuación.

3.2.2.1 Protocolos orientados a conexión

Un **protocolo orientado a conexión** es aquel en que el canal de comunicaciones entre dos aplicaciones permanece abierto durante un cierto tiempo, permitiendo enviar múltiples mensajes de manera fiable por el mismo.

4 Las siglas en inglés de *Transmission Control Protocol*, es decir, "protocolo de control de transmisión".

Cuando se transmite información usando un protocolo de transporte orientado a conexión, como TCP, se llevan a cabo las siguientes operaciones:

1 **Establecimiento de la conexión**: este paso debe realizarse siempre al inicio de las comunicaciones, y sirve para crear el canal de comunicación, que permanecerá abierto hasta que uno de los extremos lo cierre.

2 **Envío de mensajes**: este paso se puede realizar tantas veces como se desee, siempre y cuando la comunicación siga siendo posible (los niveles inferiores de la pila de protocolos sigan funcionando). El canal de comunicaciones se reutiliza para el envío de cada mensaje, haciendo posible garantizar la llegada de cada paquete que estos llegan en el orden correcto.

3 **Cierre de la conexión**: este es el paso final, y se realiza solo cuando se desea interrumpir las comunicaciones. Una vez cerrado el canal, si se desea reanudar las comunicaciones, hay que volver a empezar, repitiendo el primer paso.

Por el contrario, **un protocolo no orientado a conexión** es aquel en el que el canal de comunicación se crea de forma independiente para cada mensaje. No hay, por tanto, establecimiento de conexión al principio ni cierre al final.

3.2.3 PROTOCOLO UDP

El protocolo de transporte UDP[5] es otro de los componentes fundamentales de la pila IP. Funciona de manera similar a TCP, pero tiene una serie de características fundamentales diferentes:

- ✓ Al contrario que TCP, se trata de un **protocolo no orientado a conexión**. Esto lo hace más rápido que TCP, ya que no es necesario establecer conexiones, etc.
- ✓ No garantiza que los mensajes lleguen siempre.
- ✓ No garantiza que los mensajes lleguen en el mismo orden que fueron enviados.
- ✓ Permite enviar mensajes de 64 KB **como máximo**.
- ✓ En UDP, los mensajes se denominan "datagramas" (*datagrams* en inglés).

Como se puede ver, se trata de un protocolo de transporte mucho menos fiable que TCP, fundamentalmente porque no es orientado a conexión. Además, solo permite enviar mensajes pequeños (64 KB). No obstante, es mucho más ligero y eficiente que TCP, por lo que también se utiliza mucho en la computación distribuida.

ACTIVIDADES 3.3

- Busca información sobre otros protocolos importantes que formen parte de la pila de protocolos IP. ¿Para qué sirve el protocolo ICMP? ¿En qué nivel de la pila se ubica?

5 Las siglas en inglés de *User Datagram Protocol*, es decir, "protocolo de datagramas de usuario".

3.3 *SOCKETS*

Los *sockets* son el mecanismo de comunicación básico fundamental que se usa para realizar transferencias de información entre aplicaciones, ya sea a través de redes internas (LAN) o Internet. Proporcionan una abstracción de la pila de protocolos, ofreciendo una interfaz de programación sencilla con la que las diferentes aplicaciones de un sistema distribuido pueden intercambiar mensajes.

¿SABÍAS QUE...?

Los *sockets* aparecieron por primera vez la versión 4.2 del sistema operativo UNIX BSD, en el año 1981. Desde entonces se han vuelto el mecanismo básico estándar de comunicación entre procesos distribuidos en la mayoría de entornos, incluyendo todos los sistemas operativos UNIX y derivados (Linux, Mac OS X, Android, etc.) y Windows. En la actualidad, la mayoría de lenguajes de alto nivel, como C#, Java o Python, ofrecen interfaces de programación para usar *sockets*.

3.3.1 FUNDAMENTOS

Un *socket* (en inglés, literalmente, un "enchufe") representa el extremo de un canal de comunicación establecido entre un emisor y un receptor. Para establecer una comunicación entre dos aplicaciones, ambas deben crear sus respectivos *sockets*, y conectarlos entre sí. Una vez conectados, entre ambos *sockets* se crea una "tubería privada" a través de la red, que permite que las aplicaciones en los extremos envíen y reciban mensajes por ella. El procedimiento concreto por el cual se realizan estas operaciones depende del tipo de *socket* que se desee utilizar.

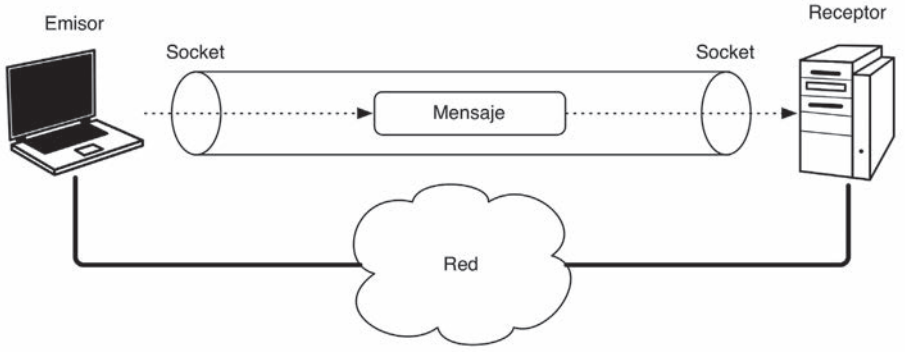


Figura 3.5. *Ejemplo de comunicación usando sockets*

Para enviar mensajes por el canal de comunicaciones, las aplicaciones *escriben* (en inglés *write*) en su *socket*. Para recibir mensajes *leen* (en inglés *read*) de su *socket*.

3.3.1.1 Direcciones y puertos

Para que las diferentes aplicaciones que forman parte de un sistema distribuido puedan enviarse mensajes, deben poder localizarse dentro de la red de comunicaciones a la que están conectadas. Esto es equivalente a cuando una persona desea enviar una carta a otra por correo ordinario: debe conocer la dirección del destinatario (ciudad, calle, número, código postal, etc.) e indicarlo en el sobre.

En las redes de comunicaciones que usan la pila de protocolos IP, las diferentes máquinas conectadas se distinguen por su **dirección IP**. Una dirección IP es un número que identifica de forma única a cada máquina de la red, y que sirve para comunicarse con ella. Es algo así como el "número de teléfono" de la máquina, dentro de la red. Actualmente existen dos versiones del protocolo IP, denominadas IPv4 (IP versión 4) e IPv6 (IP versión 6). IPv4 es la versión que se usa en Internet y en la mayoría de redes de área local. IPv6 es una versión más moderna y flexible de IP, pensada para sustituir a IPv4 en un futuro, cuando esta quede obsoleta definitivamente.

¿SABÍAS QUE...?

Inicialmente, a cada máquina que se conectaba a Internet se le asignaba una dirección IP fija, que usaba para todas sus comunicaciones. Con la proliferación de los accesos a Internet en casas y oficinas, cada vez resultaba más difícil seguir este modelo, ya que el número de direcciones disponible está limitado. Para evitar este problema se pasó a un modelo de asignación dinámico, en el que la mayoría de máquinas (especialmente las de conexiones domésticas) reciben una dirección IP distinta cada vez que se conectan. Esto permite compartir la misma dirección entre varias máquinas, siempre y cuando no estén conectadas a la vez. Los proveedores de acceso a Internet disponen de una lista de direcciones IP, que asignan a sus clientes cuando se conectan. Cuando un cliente se desconecta, la IP queda libre, y el proveedor puede asignarla a otro usuario. Aun así, este modelo tiene sus limitaciones, ya que el número máximo de direcciones IP existentes sigue siendo fijo. Una de las ventajas que aporta IPv6 es que dispone de un rango de direcciones muchísimo más amplio que IPv4, permitiendo muchas más máquinas conectadas a Internet simultáneamente.

En IPv4 las direcciones IP están formadas por secuencias de 32 bits, llamadas *palabras*. Cada palabra está, a su vez, dividida en 4 grupos de 8 bits, llamados *octetos*. Cada octeto consta de 8 dígitos binarios (bits), y con él podemos representar números desde el 0 (en binario 00000000) hasta el 255 (en binario 11111111). Cuando escribimos las direcciones IPv4, representamos cada octeto separado por puntos, y escribimos su valor en decimal, para que sea más cómodo.

¿SABÍAS QUE...?

En Windows 7 se puede ver y cambiar la dirección IP que tienen asignados todos nuestros dispositivos de red. Para ello hay que abrir el Panel de control y seleccionar la categoría de **Redes e Internet**. Una vez dentro se selecciona el **Centro de redes y recursos compartidos** y, en el panel de la izquierda, la opción de **Cambiar configuración del adaptador**. Esto nos muestra una lista de todos los adaptadores de red que existen en nuestra máquina (Ethernet, Wi-Fi, etc.). Haciendo doble clic en cualquiera de ellos se mostrará su configuración, y si dentro de esta se pulsa el botón **Detalles** aparecerá un diálogo con los datos de configuración IP del adaptador.

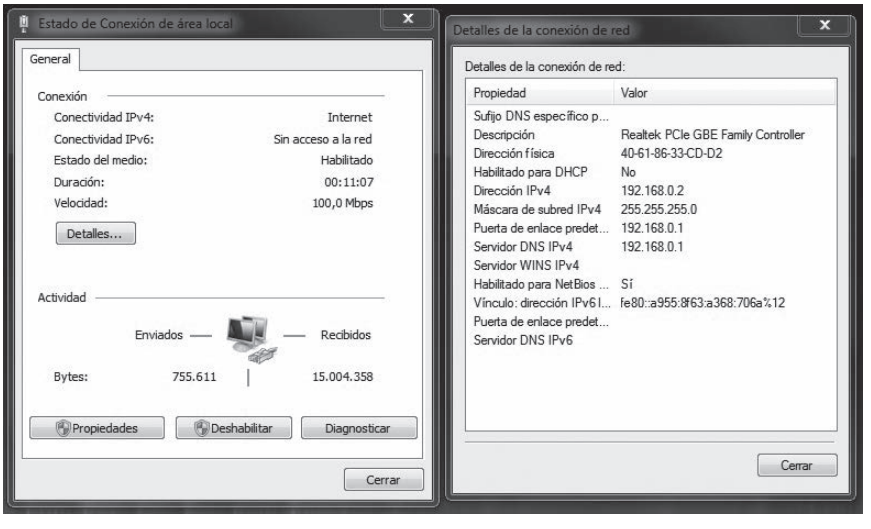


Figura 3.6. *Propiedades de un adaptador de red en Windows 7*

EJEMPLO 3.1

1. Nuestra máquina tiene la siguiente dirección IPv4:
   - 01100100110100110101110100011001
2. Separada en octetos, quedaría de la siguiente forma:
   - 01100100 11010011 01011101 00011001
3. Ahora escribimos cada octeto en decimal:
   - 100 211 93 25

El resultado final es la dirección 100.211.93.25

¿SABÍAS QUE...?

Un servidor de DNS (*Domain Name Service*) es una máquina cuya función es traducir nombres simbólicos de máquinas por sus direcciones IP en la red. Una aplicación puede realizar una petición a un servidor DNS para resolver un nombre, por ejemplo *www.google.es*, y obtener la dirección IP asociada, por ejemplo 173.194.34.223.

Usando la dirección IP, una aplicación puede localizar la máquina en la que reside el receptor de su mensaje, pero en una misma máquina puede haber más de una aplicación funcionando y usando *sockets* para comunicarse con el exterior. Entonces, ¿cómo distinguir entre todas las aplicaciones que pueden estar ejecutando en la máquina destino?

La solución es utilizar **puertos**. Un puerto es un número que identifica a un *socket* dentro de una máquina. Cuando una aplicación crea un *socket*, esta debe especificar el número de puerto asociado a dicho *socket*. Podríamos ver la dirección IP como el "nombre de la calle" a la que estamos enviando nuestra carta, y el puerto como el "número de vivienda". Si no especificamos ambos, nuestro mensaje no llegará a su destinatario. Los números de puerto se representan con 16 dígitos binarios (bits), y pueden, por tanto, tomar valores entre 0 (16 ceros en binario) y 65.535 (16 unos en binario).

Nunca puede haber más de un *socket* asignado a un mismo puerto en una máquina. Cuando una aplicación crea un *socket* y lo asigna a un puerto determinado, normalmente se dice que ese *socket* está *escuchando* (en inglés, *listening*) por ese puerto.

ACTIVIDADES 3.4

- El procedimiento de abstracción de las comunicaciones que usan los *sockets* (simular una "tubería" sobre la que se "lee" y se "escribe") se parece a otros mecanismos que se usan habitualmente en programación ¿A cuáles te recuerda?
- Busca información sobre cómo se gestionan las direcciones IPv4. ¿Podemos asignar cualquier dirección a una máquina? ¿Qué son las direcciones locales? ¿Qué es una dirección de *broadcast*?

3.3.1.2 Tipos de *sockets*

Existen dos tipos básicos de *sockets*: *sockets stream* y *sockets datagram*, dependiendo de su funcionalidad y del protocolo de nivel de transporte que utilizan.

3.3.1.2.1 *Sockets stream*

Los *sockets stream* son orientados a conexión y, cuando se utilizan sobre la pila IP, hacen uso del protocolo de transporte TCP. Son fiables (los mensajes que se envían llegan a su destino) y aseguran el orden de entrega correcto. Un *socket stream* se utiliza para comunicarse siempre con el mismo receptor, manteniendo el canal de comunicación abierto entre ambas partes hasta que se termina la conexión.

Cuando se establece una conexión usando *sockets stream*, se debe seguir una secuencia de pasos determinada. En esta secuencia uno de los elementos de la comunicación debe ejercer el papel de **proceso servidor** y otra, el de **proceso cliente**. El proceso servidor es aquel que crea el *socket* en primer lugar y espera a que el cliente se conecte. Cuando el proceso cliente desea iniciar la comunicación, crea su *socket* y lo conecta al servidor, creando el canal de comunicación. En cualquier momento, cualquiera de los dos procesos (cliente o servidor) puede cerrar su *socket*, destruyendo el canal y terminando así la comunicación.

Proceso cliente

Para usar *sockets stream*, un proceso cliente debe seguir los siguientes pasos:

1 Creación del *socket*. Esto crea un *socket* y le asigna un puerto. Normalmente el número concreto de puerto que usa el *socket* de un proceso cliente no es importante, por lo que se suele dejar que el sistema operativo lo asigne automáticamente (normalmente le da el primero que esté libre). A este *socket* se le llama "*socket* cliente".

2 Conexión del *socket* (en inglés *connect*). En este paso se localiza el *socket* del proceso servidor, y se crea el canal de comunicación que une a ambos. Para poder establecer la conexión el proceso cliente debe conocer la dirección IP del proceso servidor y el puerto por el que este está escuchando.

3 Envío y recepción de mensajes. Una vez establecida la conexión, el proceso cliente puede enviar y recibir mensajes mediante operaciones de escritura y lectura (*write* y *read*) sobre su *socket* cliente.

4 Cierre de la conexión (en inglés, *close*). Si desea terminar la comunicación, el proceso cliente puede cerrar el *socket* cliente.

Proceso servidor

En el caso de un proceso servidor, los pasos que se deben seguir para poder comunicarse con el proceso cliente son los siguientes:

1 Creación del *socket*. Esto crea un *socket*, de forma similar al caso del proceso cliente. A este *socket* se le llama "*socket* servidor".

2 Asignación de dirección y puerto (en inglés *bind*). En el caso del proceso servidor es importante que la dirección IP y el número de puerto del *socket* estén claramente especificados. De lo contrario, el proceso cliente no será capaz de localizar al servidor. La operación *bind* asigna una dirección IP y un número de puerto concreto al *socket* servidor. Lógicamente, la dirección IP asignada debe ser la de la máquina donde se encuentra el proceso servidor.

3 Escucha (en inglés *listen*). Una vez se ha creado el *socket* y se le ha asignado un número de puerto, se debe configurar para que escuche por dicho puerto. La operación *listen* hace que el *socket* servidor quede preparado para aceptar conexiones por parte del proceso cliente.

4 Aceptación de conexiones (en inglés *accept*). Una vez el *socket* servidor está listo para aceptar conexiones, el proceso servidor debe esperar hasta que un proceso cliente se conecte (con la operación *connect*). La operación *accept* bloquea al proceso servidor esperando por una conexión por parte del proceso cliente. Cuando llega una petición de conexión, **se crea un nuevo *socket* dentro del proceso servidor**. Este nuevo *socket* es que el que queda conectado con el *socket* del proceso cliente, estableciendo un canal de comunicación estable entre ambos. El nuevo *socket* se utilizará para enviar y recibir mensajes con el proceso cliente. El *socket* servidor queda libre, por lo que puede seguir escuchado, a la espera de nuevas conexiones.

5 Envío y recepción de mensajes. Una vez establecida la conexión, el proceso servidor puede enviar y recibir mensajes mediante operaciones de escritura y lectura (*write* y *read*) sobre su nuevo *socket*. **No se usa el *socket* servidor para realizar esta tarea**, ya que ha quedado fuera de la conexión.

6 Cierre de la conexión (en inglés *close*). Si desea terminar la comunicación, el proceso servidor puede cerrar el nuevo *socket*. El *socket* servidor sigue estando disponible para nuevas conexiones.

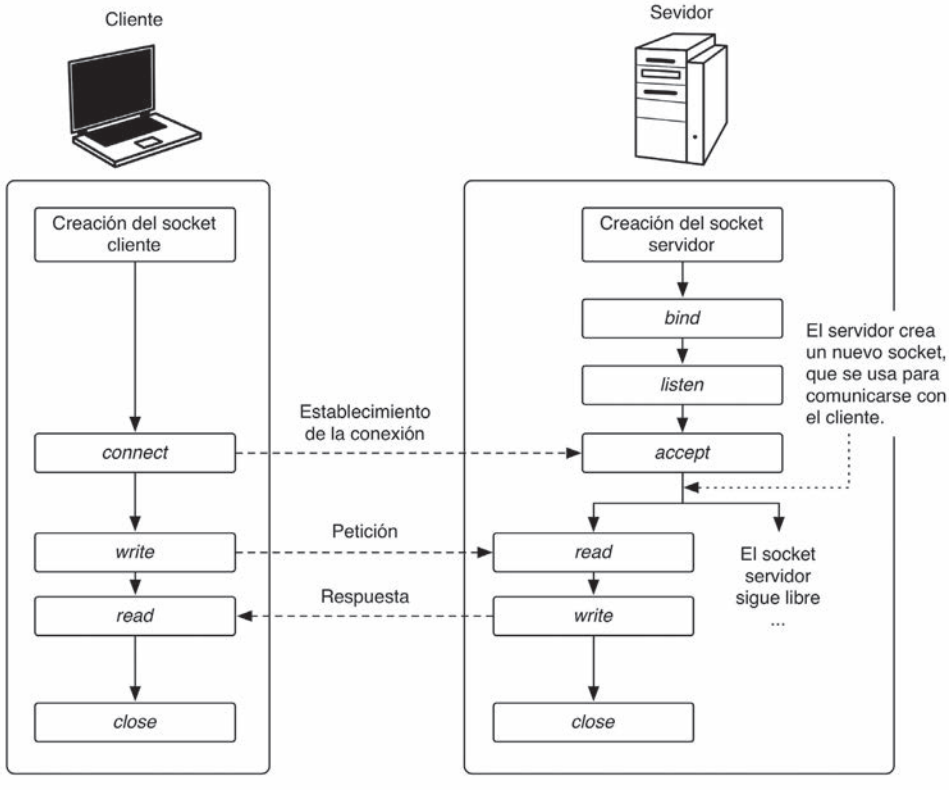


Figura 3.7. *Operaciones realizadas durante las comunicaciones con sockets stream*

3.3.1.2.2 Sockets datagram

Los *sockets datagram* no son orientados a conexión. Pueden usarse para enviar mensajes (llamados "datagramas") a multitud de receptores, ya que usan un canal temporal para cada envío. No son fiables ni aseguran orden de entrega correcto. Cuando se usan sobre la pila IP, hacen uso del protocolo de transporte UDP.

Cuando se usan *sockets datagram* no existe diferencia entre proceso servidor y proceso cliente. Todas las aplicaciones que usan *sockets datagram* realizan los siguientes pasos para enviar mensajes:

1 Creación del *socket*. Esto crea un *socket*, de forma similar al caso de los *sockets stream*.

2 Asignación de dirección y puerto (en inglés *bind*). En el caso de que se desee usar el *socket* para recibir mensajes, es importante que la dirección IP y el número de puerto del *socket* estén claramente especificados. De lo contrario los emisores no serán capaces de localizar al receptor. Al igual que en el caso de los *sockets stream*, la operación *bind* asigna una dirección IP y un número de puerto concreto al *socket*. La dirección IP asignada debe ser la de la máquina donde se encuentra la aplicación.

3 Envío y recepción de mensajes. Una vez creado el *socket*, se puede usar para enviar y recibir datagramas. En el caso de los *sockets* datagram existen dos operaciones especiales, denominadas "enviar" (en inglés *send*) y "recibir" (en inglés *receive*). La operación *send* necesita que se le especifique una dirección IP y un puerto, que usará como datos del destinatario del datagrama.

4 Cierre de la conexión (en inglés *close*). Si no se desea usar más el *socket*, se puede cerrar usando la operación *close*.

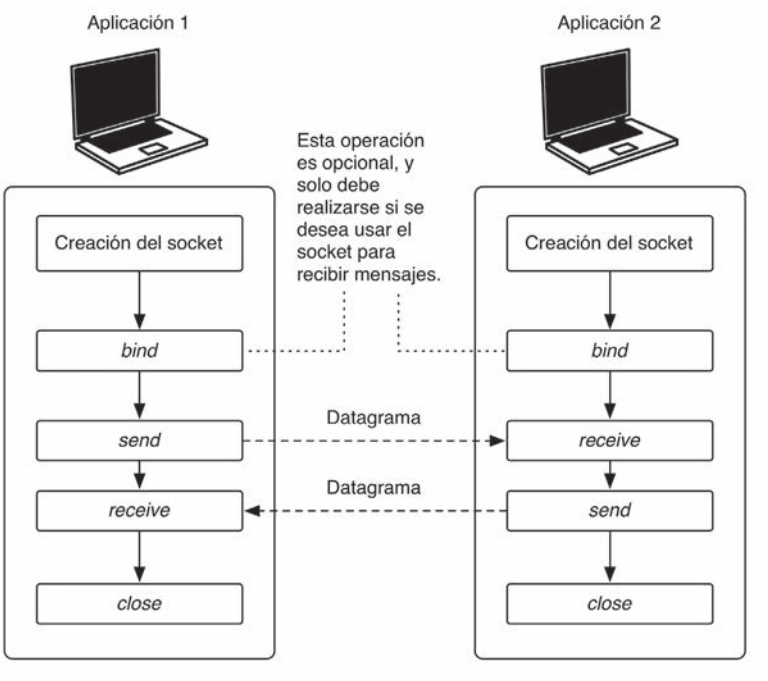


Figura 3.8. *Operaciones realizadas durante las comunicaciones con sockets datagram*

¿SABÍAS QUE...?

Los *sockets datagram* no son orientados a conexión, por lo que **se puede usar un mismo *socket* para enviar mensajes a distintos receptores**. Simplemente es necesario especificar la dirección y puerto de destino en cada operación *send*. No es necesario realizar la operación *close* para cerrar el canal, como en los *sockets stream*, porque no hay un canal permanente creado entre emisor y receptor. La operación *close* se realiza solamente cuando ya no se desea seguir usando el *socket*.

ACTIVIDADES 3.5

- Busca información sobre los tipos de *sockets* que se utilizan en las aplicaciones de Internet más comunes. ¿Qué tipo de *sockets* se suelen usar para navegar por la Web? ¿Y para resolver nombres de máquinas (DNS)?

3.3.2 PROGRAMACIÓN CON *SOCKETS*

En la mayoría de lenguajes de programación de alto nivel existen bibliotecas para crear, destruir y operar con *sockets* sobre la pila de protocolos IP. En Java existen tres clases principales que permiten la comunicación por *sockets*:

- *java.net.Socket*, para la creación de *sockets stream* cliente.
- *java.net.ServerSocket*, para la creación de *sockets stream* servidor.
- *java.net.DatagramSocket*, para la creación de *sockets datagram*.

En las siguientes secciones veremos cómo se usan en los escenarios típicos de comunicación.

3.3.2.1 La clase *Socket*

La clase *Socket* (*java.net.Socket*) se utiliza para crear y operar con *sockets stream* clientes. Sus métodos más importantes se describen en la siguiente tabla:

Método	Tipo de retorno	Descripción
`Socket()`	`Socket`	Constructor básico de la clase. Sirve para crear *sockets stream* clientes
`connect(SocketAddress addr)`	`void`	Establece la conexión con la dirección y puerto destino
`getInputStream()`	`InputStream`	Obtiene un objeto de clase *InputStream* que se usa para realizar operaciones de lectura (*read*)
`getOutputStream()`	`OutputStream`	Obtiene un objeto de clase *OutputStream* que se usa para realizar operaciones de escritura (*write*)
`close()`	`void`	Cierra el *socket*

El Ejemplo 3.2 muestra un programa en Java sencillo que opera con un *socket stream* cliente. En él se crea el *socket* usando la clase *Socket*, se conecta a un *socket stream* servidor que se encuentre en la misma máquina (*localhost*) y escuchando por el puerto 5555 y se le envía un mensaje con el texto "mensaje desde el cliente". Una vez realizadas estas operaciones, se cierra el *socket* y el programa termina.

EJEMPLO 3.2

Proceso cliente usando *sockets stream*:

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.Socket;

public class ClienteSocketStream {

    public static void main(String[] args) {
        try {
            System.out.println("Creando socket cliente");

            Socket clientSocket = new Socket();

            System.out.println("Estableciendo la conexión");

            InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
            clientSocket.connect(addr);

            InputStream is = clientSocket.getInputStream();
            OutputStream os = clientSocket.getOutputStream();

            System.out.println("Enviando mensaje");

            String mensaje = "mensaje desde el cliente";
            os.write(mensaje.getBytes());

            System.out.println("Mensaje enviado");

            System.out.println("Cerrando el socket cliente");

            clientSocket.close();

            System.out.println("Terminado");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Como se puede ver en el Ejemplo 3.2, para indicar la dirección IP y el número de puerto del *socket stream* servidor al que se desea conectar, el método *connect()* hace uso de un objeto de clase *java.net.InetSocketAddress*. Esta clase se utiliza en Java para representar direcciones de *sockets*, es decir, pares de dirección IP y número de puerto. Tal y como se hace en el ejemplo, la dirección IP se puede sustituir por un nombre de máquina (en el ejemplo, *localhost*), en cuyo caso se tratará de resolver dicho nombre y obtener su dirección IP asociada.

3.3.2.2 La clase ServerSocket

La clase *ServerSocket* (*java.net.ServerSocket*) se utiliza para crear y operar con *sockets stream* servidor. Sus métodos más importantes se describen en la siguiente tabla:

Método	Tipo de retorno	Descripción
`ServerSocket()`	`Socket`	Constructor básico de la clase. Sirve para crear *sockets stream* servidores
`ServerSocket(String host, int port)`	`Socket`	Constructor alternativo de la clase. Se le pasan como argumentos la dirección IP y el puerto que se desean asignar al *socket*. Este método realiza las operaciones de creación del *socket* y *bind* directamente
`bind(SocketAddress bindpoint)`	`void`	Asigna al *socket* una dirección y número de puerto determinado (operación *bind*)
`accept()`	`Socket`	Escucha por el *socket* servidor, esperando conexiones por parte de clientes (operación *accept*). Cuando llega una conexión devuelve un nuevo objeto de clase *Socket,* conectado al cliente
`close()`	`void`	Cierra el *socket*

La clase *ServerSocket* no tiene un método independiente para realizar la operación *listen*. Esto no quiere decir que los *sockets stream* implementados por esta clase no realicen dicha operación. La operación *listen* se realiza de forma conjunta a *accept*, cuando se ejecuta el método *accept()*.

El Ejemplo 3.3 muestra un programa sencillo en Java que opera con un *socket stream* servidor. En él se crea el *socket* usando la clase *ServerSocket*, se le asigna la dirección IP de la misma máquina (*localhost*) y el puerto 5555 y se realiza la operación *accept*, a la espera de conexiones por parte de *sockets* clientes. Cuando llega una conexión, el método *accept()* establece el canal, creando un nuevo *socket* (de clase *Socket*) y devolviéndolo. El proceso servidor utiliza este nuevo *socket* para recibir un mensaje (de tamaño 25 bytes) y lo imprime por su salida estándar, convertido en un *String*. Una vez realizadas estas operaciones se cierra el nuevo *socket* y el *socket* servidor y el programa termina. Este programa está pensado para operar conjuntamente con el que figura en el Ejemplo 3.2.

EJEMPLO 3.3

Proceso servidor usando *sockets stream*:

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.ServerSocket;


public class ServeridorSocketStream {

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

              System.out.println("Creando socket servidor");

              ServerSocket serverSocket = new ServerSocket();

              System.out.println("Realizando el bind");

              InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
              serverSocket.bind(addr);

              System.out.println("Aceptando conexiones");

              Socket newSocket = serverSocket.accept();

              System.out.println("Conexión recibida");

              InputStream is = newSocket.getInputStream();
              OutputStream os = newSocket.getOutputStream();

              byte[] mensaje = new byte[25];
              is.read(mensaje);

              System.out.println("Mensaje recibido: "+new String(mensaje));

              System.out.println("Cerrando el nuevo socket");

              newSocket.close();

              System.out.println("Cerrando el socket servidor");

              serverSocket.close();
```

EJEMPLO 3.3 (cont.)

```
                System.out.println("Terminado");

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

Como se puede ver en los Ejemplos 3.2 y 3.3, los *sockets stream* utilizan métodos estándar *read()* y *write()* de objetos de las clases *InputStream* y *OutputStream* para enviar y recibir mensajes.

ACTIVIDADES 3.6

- Copia los ejemplos anteriores que usan *sockets stream* y modifícalos para que el mensaje enviado sea "Mensaje extendido desde el programa cliente" y el puerto por el que escuche el servidor sea el 6666.

3.3.2.3 La clase DatagramSocket

La clase *DatagramSocket* (*java.net.DatagramSocket*) se utiliza para crear y operar con *sockets datagram*. Sus métodos más importantes se describen en la siguiente tabla:

Método	Tipo de retorno	Descripción
`DatagramSocket()`	`DataGramSocket`	Constructor básico de la clase. Sirve para crear *sockets datagram*
`DatagramSocket(SocketAddress bindaddr)`	`DataGramSocket`	Constructor de la clase con operación *bind* incluida. Sirve para crear *sockets* datagrama asociados a una dirección y puerto especificado
`send (DatagramPacket p)`	`void`	Envía un datagrama
`receive (DatagramPacket p)`	`void`	Recibe un datagrama
`close()`	`void`	Cierra el *socket*

EJEMPLO 3.4

Proceso que envía un mensaje usando *sockets datagram*:

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class EmisorDatagram {

    public static void main(String[] args){
        try {
            System.out.println("Creando socket datagram");

            DatagramSocket datagramSocket = new DatagramSocket();

            System.out.println("Enviando mensaje");

            String mensaje = "mensaje desde el emisor";

            InetAddress addr = InetAddress.getByName("localhost");
            DatagramPacket datagrama =
                new DatagramPacket(mensaje.getBytes(),
                                   mensaje.getBytes().length,
                                   addr, 5555);
            datagramSocket.send(datagrama);

            System.out.println("Mensaje enviado");

            System.out.println("Cerrando el socket datagrama");

            datagramSocket.close();

            System.out.println("Terminado");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

El Ejemplo 3.4 muestra un programa sencillo en Java que opera con un *socket datagram* para enviar mensajes. En él se crea el *socket* usando la clase *DatagramSocket*. Posteriormente se crea un datagrama con el mensaje "mensaje desde el emisor" y se fija su destinatario con la dirección *localhost* y el puerto 5555. Finalmente, se envía el datagrama usando el método *send()* y se cierra el *socket*.

ACTIVIDADES 3.7

- Copia el ejemplo anterior que usa *sockets datagram* y modifícalo para que el mensaje enviado sea "Mensaje extendido desde el programa cliente" y el nombre o IP de la máquina destino se pase como argumento del programa.

EJEMPLO 3.5

Proceso que recibe un mensaje y luego lo envía usando *sockets datagram*:

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.InetSocketAddress;

public class Ej32ReceiveSend {

    public static void main(String[] args) {
         try {
              System.out.println("Creando socket datagrama");

              InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
              DatagramSocket datagramSocket = new DatagramSocket(addr);

              System.out.println("Recibiendo mensaje");

              byte[] mensaje = new byte[25];
              DatagramPacket datagrama1 = new DatagramPacket(mensaje, 25);
              datagramSocket.receive(datagrama1);

              System.out.println("Mensaje recibido: "+new String(mensaje));

              System.out.println("Enviando mensaje");

              InetAddress addr2 = InetAddress.getByName("localhost");
              DatagramPacket datagrama2 =
                   new DatagramPacket(mensaje, mensaje.length, addr2, 5556);
              datagramSocket.send(datagrama2);

              System.out.println("Mensaje enviado");

              System.out.println("Cerrando el socket datagrama");

              datagramSocket.close();

              System.out.println("Terminado");

         } catch (IOException e) {
              e.printStackTrace();
         }
    }
}
```

El Ejemplo 3.5 muestra un programa sencillo en Java que opera con un *socket datagram* para enviar y recibir mensajes. En él se crea el *socket* usando la clase *DatagramSocket* y se le asigna la dirección *localhost* y el puerto 5555. Posteriormente, se espera la recepción de un mensaje por dicho *socket* (de hasta 25 bytes) y, una vez recibido, se imprime por la salida estándar en forma de *String*. A continuación se crea un nuevo datagrama con el mensaje recibido y se fija su destinatario con la dirección *localhost* y el puerto 5556. Finalmente, se envía el datagrama usando el método *send()* y se cierra el *socket*. Este programa está pensado para operar conjuntamente con el que figura en el Ejemplo 3.4.

Como se puede ver en los Ejemplos 3.4 y 3.5, el manejo del mensaje y la dirección y puerto del destinatario se realiza en los *sockets datagram* de forma bastante distinta al caso de los *sockets stream*. En primer lugar, en Java los datagramas se representan utilizando objetos de la clase *DatagramPacket* (*java.net.DatagramPacket*). Un objeto de esta clase contiene un datagrama de un tamaño fijo, asociando a un destinatario o emisor (dirección y puerto) concreto. El método *send()* de la clase *DatagramSocket* permite enviar datagramas representados por objetos de la clase *DatagramPacket*, siempre y cuando estos hayan sido correctamente inicializados con su destinatario, tal y como se ve en el Ejemplo 3.4. A su vez, el método *receive()* de la clase *DatagramSocket* permite recibir datagramas y almacenarlos en objetos de la clase *DatagramPacket*, tal y como se ve en el Ejemplo 3.5. Esta operación rellena los valores del objeto pasado como parámetro, almacenando en él el datagrama recibido y su remitente (dirección y puerto). Además, el constructor de la clase *DatagramPacket* utiliza un objeto de la clase *InetAddress* (*java.net.InetAddress*) para representar la dirección IP del emisor o destinatario del datagrama. Esta clase representa direcciones IP de forma independiente. Como se puede ver en ambos ejemplos, el número de puerto se debe indicar al constructor de la clase *DatagramPacket* como un parámetro separado.

ACTIVIDADES 3.8

- Consulta la documentación oficial de Oracle sobre el lenguaje Java (*http://docs.oracle.com/javase/*). Busca información sobre las clases *Socket*, *ServerSocket*, *DatagramSocket* y *DatagramPacket* y echa un vistazo a los demás métodos que ofrecen. ¿Ves alguno que pudiera resultarte útil?

3.4 MODELOS DE COMUNICACIONES

La pila de protocolos IP y los *sockets* son las herramientas fundamentales sobre las cuales se desarrolla la práctica totalidad de aplicaciones distribuidas que existen en la actualidad. Como hemos visto, usando *sockets* podemos crear programas que colaboran entre ellos, enviando y recibiendo mensajes. Los diferentes protocolos de nivel de transporte (TCP, UDP) permiten, además, controlar características específicas de cómo se realiza esta comunicación, teniendo en cuenta aspectos como la fiabilidad y la eficiencia.

Los *sockets*, no obstante, son solo la herramienta básica, es decir, aquello que permite enviar y recibir mensajes. A la hora de desarrollar aplicaciones distribuidas debemos tener en cuenta aspectos de más alto nivel. Dependiendo de cuál sea el propósito de nuestra aplicación, y cómo vaya a funcionar internamente, deberemos escoger un modelo de comunicaciones distinto.

Un **modelo de comunicaciones** es una arquitectura general que especifica cómo se comunican entre sí los diferentes elementos de una aplicación distribuida. Un modelo de comunicaciones normalmente define aspectos como cuántos elementos tiene el sistema, qué función realiza cada uno, etc. Los modelos de comunicaciones más usados en la actualidad son el *cliente/servidor* y el de *comunicación en grupo*. También existen modelos más sofisticados ampliamente usados, como las *redes peer-to-peer*.[6]

3.4.1 MODELO CLIENTE/SERVIDOR

El **modelo cliente/servidor** es el más sencillo de los comúnmente usados en la actualidad. En este modelo, un proceso central, llamado *servidor*, ofrece una serie de servicios a uno o más procesos *cliente*. El proceso *servidor* debe estar alojado en una máquina fácilmente accesible en la red, y conocida por los *clientes*. Cuando un *cliente* requiere sus servicios, se conecta con el *servidor*, iniciando el proceso de comunicación.

EJEMPLO 3.6

El modelo cliente/servidor se parece mucho a la interacción normal que realizamos cuando realizamos compras en un comercio convencional, por ejemplo, una panadería. La persona que entra en la panadería es el *cliente*, y el panadero o vendedor, el *servidor*. El *cliente* realiza una petición, una barra de pan por ejemplo, y el *servidor* se la proporciona.

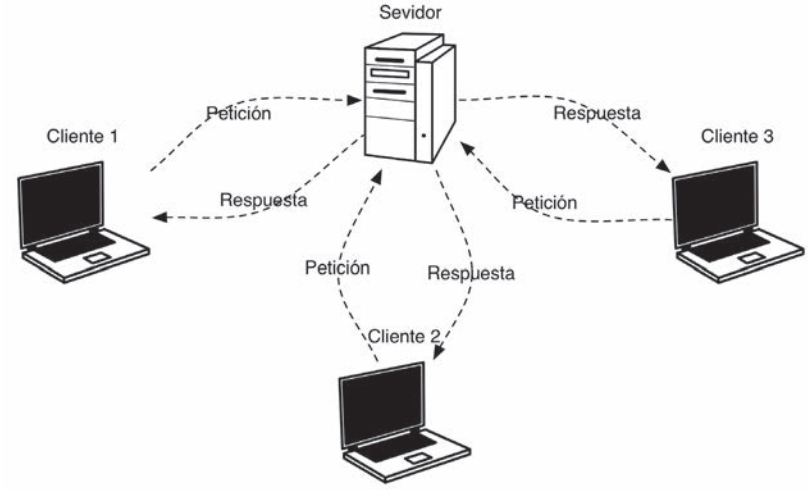


Figura 3.9. *Ejemplo de comunicación cliente/servidor*

6 El término inglés *peer-to-peer* se podría traducir como "de igual a igual" o "entre iguales".

¿SABÍAS QUE...?

El modelo cliente/servidor es el más utilizado en la actualidad en la red Internet. Más del 90 % de las aplicaciones distribuidas que usamos habitualmente usan ese modelo, entre ellas las páginas web y el correo electrónico.

El modelo cliente/servidor se basa en un mecanismo de comunicación mediante petición y respuesta. Los *clientes* realizan peticiones al *servidor*, que las resuelve, devolviendo el resultado al *cliente* correspondiente en la respuesta.

EJEMPLO 3.7

Ejemplos típicos de aplicaciones cliente/servidor son las aplicaciones móviles de mensajería instantánea, como GTalk o WhatsApp. Cuando un usuario desea enviar un mensaje a otro, abre su aplicación de GTalk o WhatsApp en su móvil o *tablet*. Esta es la aplicación *cliente*. Una vez ha seleccionado a la persona con la que desea hablar en la pantalla y ha escrito el mensaje, pulsa el botón de **enviar**. En ese momento la aplicación *cliente* se conecta a través de Internet con el *servidor*, alojado en los ordenadores centrales de Google o WhatsApp, y le envía un mensaje indicando el texto escrito, el destinatario y demás información. El *servidor* recibe el mensaje, lo procesa y responde al *cliente* indicando si ha podido o no hacerlo llegar a la persona a la que iba destinado.

Uno de los aspectos claves del modelo cliente/servidor, es que en él existen dos roles claramente diferenciados:

- **El servidor**. Es la pieza central del modelo, y la que hace que todo funcione. Ofrece un servicio específico a los *clientes* y debe ser fácilmente localizable en la red, conociendo su dirección IP (o nombre asociado) y número de puerto.
- **El cliente**. Cumple una función muy distinta al *servidor*, ya que es el que solicita los servicios y obtiene los resultados.

En el modelo clásico cliente/servidor estos roles son fijos y, por tanto, no cambian nunca: el *servidor* siempre es *servidor* y los *clientes* siempre son *clientes*. Existen modelos más avanzados en el que estos roles pueden cambiar, dependiendo de la situación. Estos modelos se verán más adelante.

3.4.1.1 Comparativa con *sockets stream*

Es muy fácil darse cuenta de que los conceptos básicos del modelo cliente/servidor se parecen mucho a los de los *sockets stream*. En ambos casos existen dos tipos de elementos con roles distintos, llamados *cliente* y *servidor*. En efecto, el funcionamiento de los *sockets stream* encaja perfectamente con el modelo cliente/servidor, ya que de hecho fueron diseñados con este modelo de comunicaciones en mente. No obstante, esto no significa que el modelo cliente/servidor deba ser siempre implementado usando *sockets stream*. El modelo cliente/servidor es una arquitectura de tipo abstracto, que puede ser implementada utilizando diferentes tipos de mecanismos de comunicación. Los *sockets stream* son uno de estos posibles mecanismos, pero no dejan de ser una alternativa más. A la hora de escoger el

mecanismo de comunicación que vamos a usar en nuestra aplicación debemos tener en cuenta no solo el modelo de comunicación, también otros aspectos como los mencionados anteriormente de fiabilidad, eficiencia, etc.

3.4.2 MODELO DE COMUNICACIÓN EN GRUPO

El **modelo de comunicación en grupo** es la alternativa más común al modelo cliente/servidor. A diferencia de este, en el modelo de comunicación en grupo no existen roles diferenciados, es decir, no existe un elemento llamado *servidor*, con funciones específicas, y un elemento llamado *cliente* con otras funciones distintas. En la comunicación en grupo existe un conjunto de dos o más elementos (procesos, aplicaciones, etc.) que cooperan en un trabajo común. A este conjunto se le llama *grupo*, y los elementos que lo forman se consideran todos iguales, sin roles ni jerarquías definidas.

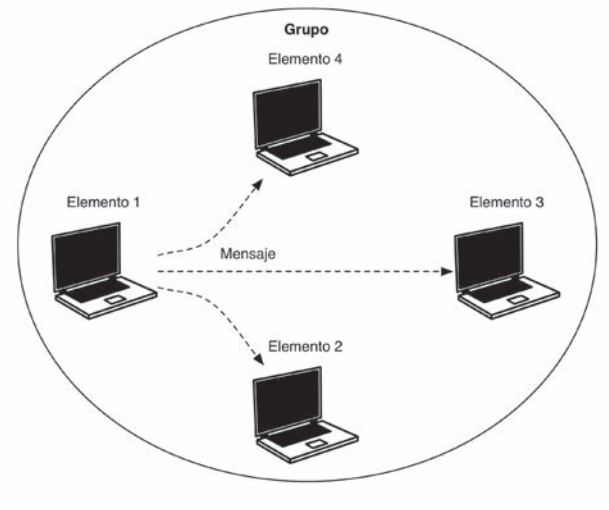


Figura 3.10. *Ejemplo de comunicación en grupo*

EJEMPLO 3.8

El modelo de comunicación en grupo se parece mucho a una conversación entre un grupo de amigos. En la conversación, cada participante escucha los comentarios de los demás y aporta los suyos propios, que los demás escuchan. La conversación va progresando gracias a la interacción entre todos los miembros del grupo.

En el modelo de comunicación en grupo los mensajes se transmiten mediante lo que se denomina *radiado*. El *radiado* implica que los mensajes se envían de manera simultánea a los distintos miembros del grupo, en vez de establecer comunicaciones punto-a-punto, como en el modelo cliente/servidor.

¿SABÍAS QUE...?

En sus orígenes, la mayoría de juegos *on-line* utilizaban el modelo cliente/servidor. Todos los jugadores debían conectarse a un servidor central, donde se alojaba la partida y se coordinaba la interacción entre ellos. El problema de este modelo es que el servidor constituye un *punto único de fallo*, esto es, un componente del sistema que, cuando falla, hace que el sistema entero deje de funcionar. En un juego *on-line*, la caída del servidor implica que la partida se interrumpe, y todos los jugadores se quedan fuera. Además, muchos juegos *on-line* generan gran cantidad de tráfico el la red, lo que contribuye a saturar el servidor y aumenta la probabilidad de que este falle. Estos problemas han hecho que, en la actualidad, muchos juegos *on-line* busquen alternativas basadas en la comunicación en grupo. Muchos juegos *on-line* actuales, como *World of Warcraft* o *Guild Wars 2*, siguen usando el modelo cliente/servidor, pero otros, como los últimos títulos de las sagas *Call of Duty* o *Battlefield*, incorporan mecanismos de comunicación en grupo para transmitir la información entre los jugadores de una misma partida y aumentar la tolerancia a fallos en los servidores.

3.4.2.1 Comunicación en grupo, *sockets multicast* y bibliotecas de comunicación

El modelo de comunicación en grupo se puede implementar usando *sockets stream* o *datagram,* dependiendo de cuáles sean las necesidades de la aplicación distribuida.

Usando *sockets stream*, cada elemento del grupo debe establecer una conexión con todos los demás, es decir, que cada elemento debe tener un *socket* distinto para comunicarse con cada uno de los demás miembros del grupo. Esto implica la creación de un gran número de *sockets* y conexiones, especialmente en grupos grandes. Pese a todo, los *sockets stream* son fiables, por lo que la comunicación en grupo lo será también.

Usando *sockets datagram*, cada elemento del grupo necesitará un único *socket*, que usará para enviar mensajes a todos los demás elementos del grupo de la misma forma. Este procedimiento requiere de muchos menos *sockets*, pero resulta menos fiable, al usar *sockets* no orientados a conexión.

Además de estos procedimientos básicos, el protocolo IP incorpora un mecanismo automático que se puede utilizar para realizar operaciones de comunicación en grupo, denominado *multicast*. Al contrario que los mecanismos de comunicación que hemos visto hasta ahora, denominados *unicast*, el *multicast* consiste en enviar el mismo mensaje a varios destinatarios de forma simultánea, utilizando una dirección IP especial denominada *dirección de IP multicast*. Para poder recibirlo, los *sockets* de todos procesos receptores deben estar configurados usando la misma *dirección IP multicast*. El mensaje enviado es automáticamente replicado y entregado a cada uno de los destinatarios. En IPv4 se reserva una serie de direcciones específicas para realizar *multicast*, que van desde 224.0.0.0 hasta 239.255.255.255. El uso de *IP multicast* permite la comunicación en grupo de forma muy eficiente, pero supone un riesgo para la seguridad de la red, ya que cualquiera puede enviar cientos de mensajes y saturar a todos los miembros de un grupo. Por esta razón, el tráfico *multicast* suele estar restringido en Internet, y solo se usa en redes de área local.

¿SABÍAS QUE...?

En el protocolo IP existen 4 tipos de envío de mensajes:

- *Unicast*: consiste en el envío de mensajes entre un único emisor y un receptor determinado. El uso de *sockets* TCP y UDP que hemos visto en este capítulo emplea este mecanismo.
- *Multicast*: consiste en el envío de mensajes a un grupo de destinatarios, distinguidos por una dirección específica (*dirección IP multicast*).
- *Broadcast:* consiste en el envío simultáneo de un mensaje a todos los miembros de una red local.
- *Anycast*: similar a *multicast o broadcast*, pero el mensaje es entregado solamente al destinatario más cercano en la red.

Ya sea usando *sockets unicast* o *multicast*, implementar un modelo de comunicación en grupo no es una tarea trivial. Se deben tener en cuenta muchos posibles *sockets*, direcciones especiales, etc. Es por esto que la mayoría de aplicaciones distribuidas que usan comunicación en grupo no la implementan directamente, sino que hacen uso de bibliotecas de comunicación. Estas bibliotecas son herramientas software específicamente diseñadas para implementar modelos de comunicación sofisticados, como la comunicación en grupo. Ofrecen multitud de ventajas al desarrollador, y facilitan la implementación, configuración y depuración de su aplicación. Un ejemplo típico de estas herramientas es la biblioteca *JGroups*, desarrollada para el lenguaje Java.

3.4.3 MODELOS HÍBRIDOS Y REDES *PEER-TO-PEER* (P2P)

Los dos modelos de comunicaciones vistos hasta ahora (cliente/servidor y comunicación en grupo) son la base de la mayoría de arquitecturas de comunicaciones modernas. Las aplicaciones distribuidas más avanzadas, no obstante, suelen tener requisitos de comunicaciones muy complejos, que requieren de modelos de comunicaciones más sofisticados que estos. En muchos casos, los modelos de comunicaciones reales implementados en estas aplicaciones mezclan conceptos del modelo cliente/servidor y la comunicación en grupo, dando lugar a enfoques híbridos.

3.4.3.1 Limitaciones de los modelos fundamentales

Los modelos de comunicación fundamentales (cliente/servidor y comunicación en grupo) presentan dos limitaciones fundamentales:

- ✓ Reparto de roles fijo.
- ✓ Mecanismo de comunicación único.

El reparto de roles fijo significa que la función de un elemento del sistema no cambia con el tiempo, pase lo que pase en este. En el modelo cliente/servidor, por ejemplo, el *servidor* siempre es *servidor* y los *clientes* siempre son *clientes*. Esto resulta muy poco flexible, restringiendo las posibilidades de la aplicación.

El mecanismo de comunicación único significa que los elementos que forman el sistema distribuido se comunican siempre de la misma forma. En el modelo de comunicación en grupo, por ejemplo, los elementos siempre se comunican con mensajes dentro del grupo, sin poder establecer conexiones separadas entre ellos u otros circuitos de comunicación paralelos.

En general, como vemos, los modelos de comunicación fundamentales son potentes pero poco flexibles. Al crear modelos híbridos podemos intentar aprovechar las ventajas de cada uno, y evitar sus inconvenientes.

EJEMPLO 3.9

¿Cómo se crea un modelo híbrido? Para hacernos una idea, imaginemos el problema mencionado anteriormente de los juegos *on-line*. Necesitamos disponer de un punto de encuentro central, donde los jugadores puedan conectarse e iniciar una partida. Para esto necesitamos aplicar un modelo cliente/servidor, en el que los jugadores actúan como clientes que se conectan a un servidor central para encontrarse y crear la partida. Pero una vez arrancada la partida, mantenerla alojada en el servidor puede ser un problema, ya que podríamos saturarlo si tenemos muchos usuarios, y si falla, la partida se vería interrumpida. Lo que hacemos en este punto es pasar a un modelo de comunicación en grupo, puesto que los jugadores ya se conocen (el servidor les da a todos la dirección IP y número de puerto de cada uno). Por lo tanto, en este modelo híbrido el servidor actúa solo como mecanismo de enlace, poniendo en contacto a unos jugadores con otros. Una vez comienza la partida, el modelo de comunicación en grupo controla la interacción entre ellos. Al no haber un punto central, los jugadores pueden salir de la partida si lo desean, sin que esto afecte al resto. Incluso si el servidor se cayese, la partida seguiría, ya que los jugadores ya se conocen entre sí y están dentro del mismo grupo.

3.4.3.2 Redes *peer-to-peer* (P2P)

Las llamadas *redes peer-to-peer*, o P2P, son uno de los modelos de comunicación híbrida más potentes que existen en la actualidad. Deben su fama a las redes de intercambio de archivos, como el antiguo Napster o el más reciente Bittorrent, pero se usan con éxito también en aplicaciones comerciales, como Spotify.

¿SABÍAS QUE...?

Aunque los orígenes de la tecnología P2P se remontan a la década de 1980, la primera red *peer-to-peer* de éxito mundial fue el sistema Napster, lanzado en 1999. En sus orígenes, Napster era un servicio que permitía compartir archivos a través de Internet entre todos los usuarios de la red, y estaba especialmente enfocado a la distribución gratuita de música, en forma de ficheros MP3. Por este motivo, se encontró rápidamente con la oposición de las grandes compañías discográficas, que acusaron a los creadores de Napster de infringir las leyes de derechos de autor. Esto provocó que la red terminara siendo desactivada en 2001, apenas dos años después de su puesta en funcionamiento. Pese a su corta vida, la influencia de Napster en la cultura de Internet ha sido enorme, sentando las bases de muchos de los sistemas de distribución de archivos basados en P2P más modernos, como eDonkey2000 o Bittorrent, e incluso aplicaciones comerciales de éxito, como iTunes o Spotify.

Una red P2P está formada por un grupo de elementos distribuidos que colaboran con un objetivo común. La diferencia fundamental con el modelo de comunicación en grupo convencional es que, en las redes P2P, cualquier elemento puede desempeñar los roles de *servidor* o *cliente*, como si de un modelo cliente/servidor se tratase. Esto hace que las redes P2P puedan ofrecer servicios de forma similar al modelo cliente/servidor, lo que las vuelve mucho más potentes en el entorno de Internet. Cualquier aplicación puede conectarse a la red como un *cliente*, localizar un *servidor* y enviarle una petición. Si permanece en la red P2P, con el tiempo ese mismo *cliente* puede hacer a su vez de *servidor* para otros elementos de la red.

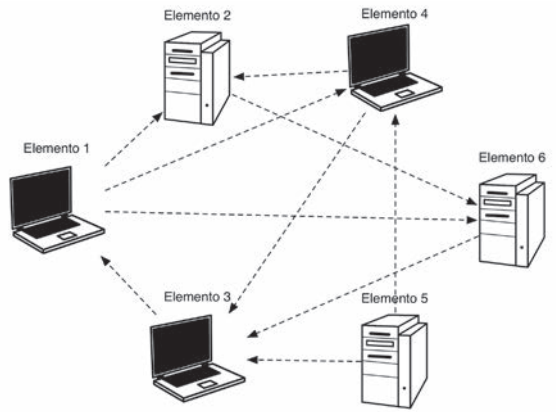


Figura 3.11. *Ejemplo de una red* peer-to-peer

EJEMPLO 3.10

El ejemplo más típico de una red *peer-to-peer* es una red de compartición de archivos entre iguales, como Bittorrent. En un sistema distribuido de estas características, cada usuario que se conecte a la red comparte una serie de archivos, identificados usando algún mecanismo estándar. En la red Bittorrent, por ejemplo, se utilizan unos ficheros especiales, denominados *torrents*, para identificar de manera única a cada archivo compartido en la red. Cuando un usuario desea descargar un archivo, utiliza el fichero *torrent* asociado para localizar a los otros usuarios que lo tienen, y se conecta a ellos para descargarlo. En este contexto, el usuario que descarga actúa como *cliente* y los que proporcionan la información, como *servidores*. En cuanto el *cliente* ha descargado un fragmento de dicho archivo, ya puede compartirlo, por lo que podría recibir una conexión de otro *cliente* que solicitase ese mismo fragmento. Si esto ocurriese, el usuario estaría actuando simultáneamente como *cliente* y como *servidor*, dependiendo de los otros elementos de la red con los que se estuviese comunicando.

La ventaja fundamental de las redes P2P es que eliminan los problemas más importantes del modelo cliente/servidor tradicional. Al no existir un único *servidor*, el sistema es mucho más tolerante a fallos, ya que puede seguir funcionando aunque algún elemento se desconecte. Además, el hecho de que cualquier elemento puede actuar como servidor permite repartir la carga de forma equilibrada entre todos los elementos, haciendo que el sistema sea aún más robusto.

¿SABÍAS QUE...?

En la última década, las redes P2P han causado mucha polémica, debido a su uso masivo como mecanismos para difundir de manera gratuita contenidos con *copyright*, como música o películas. El hecho de que todos los elementos de la red actúen simultáneamente como *servidor* y/o *cliente* hace mucho más difícil identificar a las personas que difunden estos archivos, lo cual puede ser problemático en aquellos países donde la legislación no permita estas prácticas. No obstante, no se debe considerar a las redes P2P exclusivamente como tecnología para "bajarse películas gratis". Las redes P2P han supuesto un enorme avance en sistemas distribuidos, y en la actualidad se usan con éxito en multitud de áreas. Algunas de las aplicaciones distribuidas más exitosas de los últimos años, como Skype o Spotify, funcionan gracias al modelo P2P.

ACTIVIDADES 3.9

- Consulta información sobre las redes *peer-to-peer.* Además de las mencionadas en este capítulo, ¿qué otras aplicaciones conoces que hagan uso de este modelo de comunicaciones?

RESUMEN DEL CAPÍTULO

Una aplicación distribuida es un sistema en el que varios elementos computacionales (máquina, programa, etc.) colaboran entre sí, comunicándose a través de una red. La base de esta comunicación es el paso de mensajes entre aplicaciones, dentro de la cual se distingue al emisor (el que envía el mensaje), el receptor (el que lo recibe), el canal de comunicaciones y el protocolo, que es el conjunto de reglas que se deben seguir para poder intercambiar los mensajes. El mensaje enviado suele dividirse en paquetes.

La base de las comunicaciones en la mayoría de redes modernas es la pila de protocolos IP. Se trata de una arquitectura organizada en cuatro capas. Cada capa (red, Internet, transporte, aplicación) tiene una función específica. Los programas se sitúan en el nivel más alto (aplicación) y hacen uso de los protocolos de nivel de transporte para intercambiar mensajes. Los protocolos de nivel de transporte más importantes son TCP (orientado a conexión) y UDP (no orientado a conexión).

El mecanismo estándar para la programación de comunicaciones entre aplicaciones son los *sockets*. Los *sockets* permiten abstraer los detalles del sistema de comunicaciones, centrándose en características funcionales. Los dos tipos básicos de *sockets* existentes son los *sockets stream* (orientados a conexión) y los *sockets datagram* (no orientados a conexión). Cuando operan sobre la pila IP, los primeros usan TCP y los segundos UDP como protocolo de transporte. En Java, las clases básicas que se usan para programar con *sockets* son *Socket*, *ServerSocket* y *DatagramSocket*.

Cuando se desarrolla una aplicación distribuida, esta debe seguir un modelo de comunicaciones. El modelo más usado es el cliente/servidor, en el que una aplicación servidor proporciona servicios a una o más aplicaciones cliente. La comunicación entre clientes y servidor se articula siguiendo un modelo petición-respuesta. También existe el modelo de comunicaciones en grupo, en el que todos los elementos del sistema colaboran de manera igualitaria, y modelos de comunicaciones híbridos, que mezclan características de más de un modelo básico. Un ejemplo típico de modelo híbrido son las redes *peer-to-peer*.

EJERCICIOS PROPUESTOS

1. Escribe una pareja de programas (A y B) que transfieran un fichero entre ellos. El programa A deberá leer un fichero de texto del disco y enviarlo a B. Be recibirá el contenido del fichero y lo imprimirá por su salida estándar. Utiliza para ello *sockets stream*.

2. Escribe un programa que conteste a preguntas. El programa creará un *socket stream* y aguardará conexiones. Cuando llegue una conexión, leerá los mensajes recibidos, byte a byte, hasta que encuentre el carácter ASCII "?" (signo de final de interrogación). Cuando esto ocurra, construirá una frase con todos los bytes recibidos y contestará con un mensaje. El contenido del mensaje dependerá de la frase recibida:

- Si la frase es "¿Cómo te llamas?", responderá con la cadena "Me llamo Ejercicio 2".
- Si la frase es "¿Cuántas líneas de código tienes?", responderá con el número de líneas de código que tenga.
- Si la frase es cualquier otra cosa, responderá "No he entendido la pregunta".

Escribe, además, dos programas de prueba que verifiquen que el programa responde bien a todas las preguntas.

3. Escribe un programa que responda a saludos usando *sockets datagram*. El programa escuchará por el *socket* mensajes que contengan la cada de texto "Hola". Cuando reciba uno, responderá a su emisor con otro mensaje que contenga la cadena "¿Qué tal?". Escribe además un programa adicional para probar el funcionamiento de este.

4. Escribe una pareja de programas (A y B) que usen *sockets datagram* para intercambiar un mensaje llamado *token*. Al arrancarse, el programa A enviará un mensaje al B con la palabra "token". Cuando el B la reciba, enviará de vuelta a A un mensaje con la palabra "recibido", y terminará. Cuando A reciba el mensaje de B, terminará también.

5. Escribe un programa que cuente el número de conexiones que vaya recibiendo. Este programa dispondrá de un *socket stream* servidor. Cada vez que un *socket* cliente se conecte, este le enviará un mensaje con el número de clientes conectados hasta ahora. Así pues, el primer cliente que se conecte recibirá un 1, el segundo un 2, el tercero un 3, etc.

6. Crea una versión generalizada de los programas del Ejercicio 4, para hacer posible que el token se pase entre un grupo de 2 o más programas, en forma de anillo. Cada uno de los programas se debe arrancar indicando como argumentos de entrada su posición en el anillo y el tamaño del anillo. El programa que se encuentre en la posición numero 1 (el primero), generará el mensaje "token" y se lo enviará al programa 2. Cuando este lo reciba se lo pasará al 3, y así sucesivamente. Cuando lo reciba el último programa, lo enviará de vuelta al número 1. Cuando el número 1 lo reciba, la secuencia se interrumpirá (el *token* habrá dado una vuelta completa al anillo). Se deben cumplir además las siguientes restricciones:

- Todos los programas deben tener el mismo código fuente. Se trata, por tanto, del mismo programa, pero ejecutado con distintos parámetros.
- El programa debe permitir un número variable de elementos en el anillo. El tamaño del anillo se especificará de antemano.

Se puede programar usando *sockets stream* o *sockets datagram*.

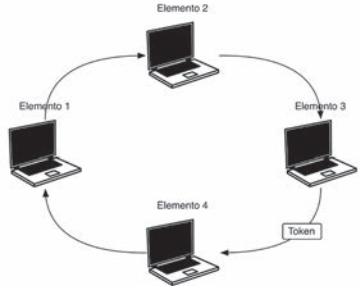


Figura 3.12. *Ejemplo de un anillo de procesos (Ejercicio 6)*

TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes NO es una característica fundamental de TODOS los sistemas distribuidos?
- a) Los elementos que forman el sistema no están sincronizados.
- b) Está formado por más de un elemento computacional.
- c) Los elementos que lo forman siguen el modelo de comunicación cliente/servidor.
- d) Los elementos que lo forman están conectados a una red de comunicaciones.

2 De los siguientes, ¿cuál es el elemento fundamental de la comunicación entre aplicaciones?
- a) Emisor.
- b) Paquete.
- c) Canal.
- d) Todos los anteriores.

3 En la pila de protocolos IP, ¿cuál es la función principal del nivel de Internet?
- a) Crear el canal de comunicación, descomponer el mensaje en paquetes y gestionar la transmisión entre emisor y receptor.
- b) Garantizar la seguridad de las comunicaciones mediante algoritmos de cifrado asimétrico.
- c) Proporcionar los elementos hardware de comunicaciones y sus controladores básicos, sobre los que se efectúan las comunicaciones.
- d) Dirigir los paquetes por la red y hacer que estos lleguen a su destino.

4 De las siguientes afirmaciones, ¿cuáles son ciertas acerca del protocolo TCP?
- a) Es un protocolo orientado a conexión.
- b) Se sitúa en el nivel de transporte de la pila de protocolos IP.
- c) Garantiza que los datos no se pierden y que lleguen en el mismo orden en que han sido enviados, siempre y cuando la comunicación sea posible.
- d) Permite enviar mensajes de 64 KB como máximo.

5 En la pila de protocolos IP, ¿cuál es la función principal del nivel de transporte?
- a) Crear el canal de comunicación, descomponer el mensaje en paquetes y gestionar la transmisión entre emisor y receptor.
- b) Garantizar la seguridad de las comunicaciones mediante algoritmos de cifrado asimétrico.
- c) Proporcionar los elementos hardware de comunicaciones y sus controladores básicos, sobre los que se efectúan las comunicaciones.
- d) Dirigir los paquetes por la red y hacer que estos lleguen a su destino.

6 De las siguientes afirmaciones sobre los mecanismos de funcionamiento de los *sockets*, ¿cuáles son ciertas?
- a) En los *sockets datagram*, no siempre es necesario realizar la operación *bind* antes de poder usarlos.
- b) Existen dos clases de *sockets stream*: los *sockets* servidor y los *sockets* cliente.
- c) La operación *accept* crea un nuevo *socket*, conectado al *socket* cliente que realizó el *connect*.
- d) Los *sockets datagram* solo sirven para enviar mensajes.

7 En Java, ¿qué clase debemos usar para crear *sockets stream* cliente?
- a) *ServerSocket*.
- b) *DatagramSocket*.
- c) *Socket*
- d) *MegaSocket*.

8 En Java, cuando el método *accept()* de un *ServerSocket* termina, ¿qué devuelve?

a) Un nuevo objeto de clase *ServerSocket*, conectado al *socket* del cliente.
b) El mensaje enviado desde el *socket* cliente al realizar la operación *connect()*.
c) Un objeto de clase *Socket*, conectado al *socket* del cliente.
d) Un objeto de clase *DatagramPacket*.

9 De los siguientes modelos de comunicaciones, ¿cuál es el más usado en la actualidad?

a) Comunicación en grupo.
b) Cliente/servidor.
c) Punto a punto.
d) Ninguno de los anteriores.

10 ¿Qué roles distintos puede desempeñar simultáneamente un elemento de una red *peer-to-peer*?

a) Uno solo: cliente o servidor.
b) Uno solo: en las redes *peer-to-peer* no hay roles diferenciados.
c) Tres: cliente, servidor y moderador.
d) Dos: cliente y servidor.

4 Generación de servicios en red

OBJETIVOS DEL CAPÍTULO

- ✓ Aprender el concepto de "servicio" y su aplicación en el contexto de la computación distribuida.
- ✓ Aprender a programar aplicaciones distribuidas siguiendo el modelo cliente/servidor.
- ✓ Entender las características de un protocolo de nivel de aplicación y conocer los protocolos de nivel de aplicación más usados en la actualidad.
- ✓ Conocer de la existencia de tecnologías avanzadas de comunicación, más allá de los *sockets*, y aprender a programar aplicaciones usando Java RMI.
- ✓ Aprender qué son los servicios web y cuáles son sus principales ventajas.

4.1 SERVICIOS

En el capítulo anterior se han visto los conceptos fundamentales de la computación distribuida y los mecanismos básicos de comunicación en red. Dentro de estos conceptos fundamentales, se ha desarrollado el modelo de comunicaciones cliente/servidor, el más usado en la actualidad en la computación distribuida. Al definir este modelo, se dice que uno de sus elementos clave es el *servidor*, una aplicación que proporciona servicios a uno o más *clientes*. Pero ¿qué es exactamente un servicio? ¿Qué características tiene? A lo largo de este capítulo se profundizará en este y otros conceptos relacionados, y se aprenderá a desarrollar aplicaciones complejas que proporcionen servicios.

4.1.1 CONCEPTO DE SERVICIO

Desde un punto de vista básico, todo sistema presenta dos partes fundamentales: **estructura** y **función**. La estructura está formada por aquellos componentes que conforman el sistema, es decir, las piezas que unidas lo forman. En informática en general, y en los sistemas distribuidos en particular, la estructura del sistema serán aquellos componentes hardware y software que, conectados entre sí, forman el ordenador, sistema distribuido, o lo que estemos analizando. La función, por otro lado, es aquello para lo que está pensado el sistema, es decir, para qué sirve y/o se usa.

EJEMPLO 4.1

La descomposición conceptual en estructura y función se puede aplicar a cualquier sistema tecnológico, ya sea en el campo de la informática o fuera de ella. Un electrodoméstico, una lavadora por ejemplo, es un sistema que consta de estructura y función, como cualquier otro. En este ejemplo, la estructura de la lavadora son la piezas mecánicas y electrónicas que lo forman (carcasa, tambor, puerta, controlador electrónico, motor interno…). La función de la lavadora es, obviamente, lavar la ropa.

EJEMPLO 4.2

Determinar la estructura y función de un sistema distribuido es igual de sencillo que determinar la de cualquier otro sistema. Si pensamos en una aplicación de mensajería instantánea, como WhatsApp, por ejemplo, la estructura estaría formada por las aplicaciones cliente y servidor, es decir, el software que lo compone. La función de este sistema es, sencillamente, permitir que los usuarios puedan comunicarse enviando mensajes de texto, imágenes, vídeos, etc.

Todo sistema tiene, por tanto, una estructura y una función. Como se puede observar, la estructura es algo concreto, relacionado con los componentes hardware y software que forman el sistema. La función, en cambio, suele ser una serie de conceptos abstractos, que explican "lo que hace" el sistema, pero no "cómo lo hace". Para que un sistema realice su función, se deben especificar los mecanismos concretos que este proporciona a sus usuarios. A este conjunto de

mecanismos concretos que hacen posible el acceso a la función del sistema se le denomina **servicio**. Un sistema puede proporcionar uno o más servicios, en función de cuál sea su función.

Además, normalmente, este conjunto de mecanismos, al que se llama "servicio", se especifica de manera concreta e impone restricciones específicas de utilización. Cualquier usuario del sistema ha de seguir una serie de procedimientos determinados a la hora de acceder al sistema y al usarlo. Esta serie de procedimientos se denomina **interfaz del servicio** y es el punto de contacto entre el sistema y el usuario.

EJEMPLO 4.3

Continuando con el ejemplo de la lavadora, el servicio proporcionado por esta se define como el conjunto de mecanismos que esta realiza para llevar a cabo su función. En este caso podemos hablar de operaciones como lavado con agua caliente, lavado en frío, programa para tejidos sintéticos, etc. La interfaz del servicio son los componentes con los que interactúa el usuario, como la puerta de carga, el cuadro de mandos, etc. Estos condicionan los procedimientos concretos que el usuario debe seguir para obtener el servicio. Para lavar en frío, por ejemplo, el usuario debe abrir la puerta, cargar la ropa en el tambor, cerrar la puerta, llenar la cubeta del detergente, ajustar el programa adecuado en el panel y pulsar el botón de encendido.

EJEMPLO 4.4

En el ejemplo de la aplicación de mensajería instantánea (Ejemplo 4.2), los servicios son las funciones específicas de la aplicación, como buscar a un amigo, enviarle un mensaje, enviarle una foto, etc. La interfaz del servicio es la propia interfaz de la aplicación cliente, que marca la manera concreta en la que se realizan dichas operaciones. Para enviar una foto, por ejemplo, estas operaciones podrían ser seleccionar al destinatario en la lista de contactos, pulsar el botón de **Enviar foto**, seleccionar una fotografía de la galería de imágenes y pulsar el botón de **Enviar**.

ACTIVIDADES 4.1

- Identifica estructura, función, servicio e interfaz de servicio en los siguientes ejemplos:
  a. Un procesador de textos (como Word o LibreOffice Writer).
  b. Un autobús urbano.
  c. Una plataforma de juegos *on-line*, como Steam o XBOX Live.
  d. Una cámara fotográfica.

4.1.2 SERVICIOS EN RED

Durante las comunicaciones entre los diferentes elementos de una aplicación distribuida, se hace uso de multitud de servicios distintos, que cooperan entre sí para conseguir el paso de mensajes entre emisores y receptores. La pila de protocolos IP es, en efecto, un conjunto de sistemas independientes, pero montados unos sobre otros para realizar una tarea compleja. Cada nivel de la jerarquía (red, Internet, transporte y aplicación) proporciona un servicio específico, tal y como se vio en el capítulo anterior, y ofrece una interfaz de servicio al nivel superior, a través de la cual interactúa con este. Además, para que las comunicaciones puedan llevarse a cabo, cada nivel de la pila dispone de su propio protocolo de comunicaciones, que gobierna la interacción a ese nivel con los demás elementos del sistema distribuido.

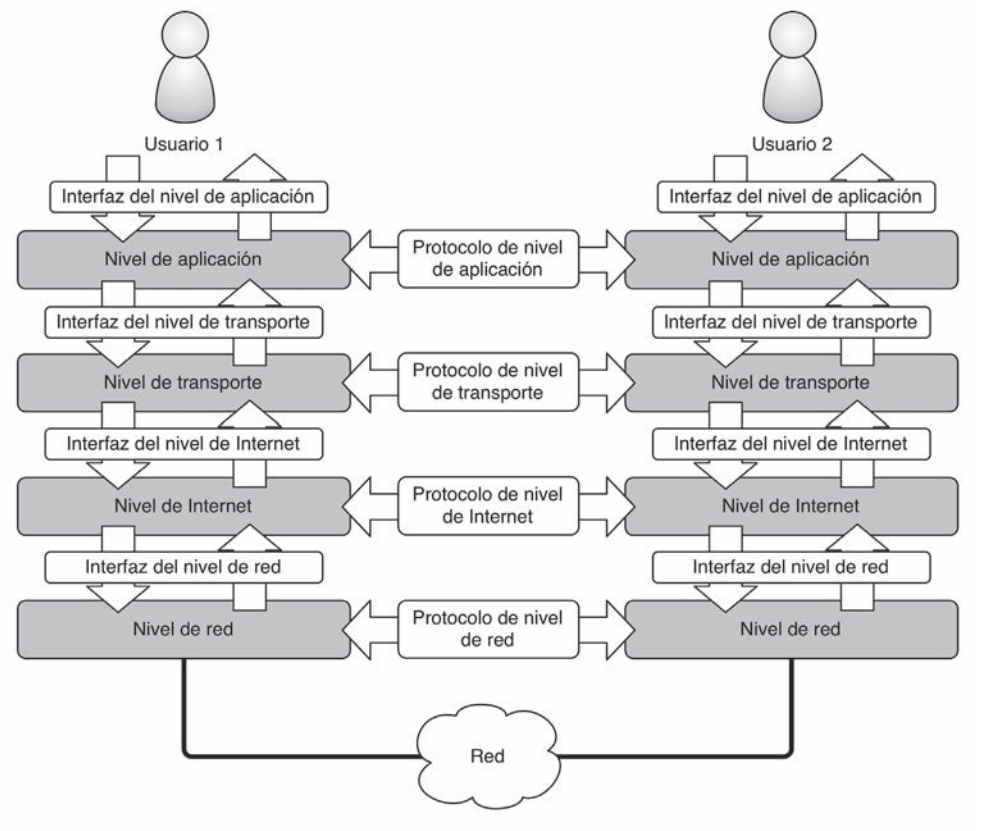


Figura 4.1. *Comunicación entre dos usuarios, usando la pila de protocolos IP*

Se puede considerar, por tanto, un servicio en red como cualquier servicio que se ubique en cualquier nivel de la pila. Las tecnologías de comunicaciones del nivel de red son servicios, los mecanismos de encaminamiento del nivel de Internet son servicios, etc.

¿SABÍAS QUE...?

El sistema de *sockets* estudiado en el capítulo anterior proporciona servicios del nivel de transporte. Su función es hacer llegar mensajes entre un emisor y un receptor, ya sea por un modelo orientado a conexión (*sockets stream*) o por medio de datagramas (*sockets datagram*). La interfaz del servicio está formada, en este caso, por las bibliotecas de programación que permiten hacer uso de los *sockets* dentro de nuestros programas. Los protocolos TCP y UDP son ejemplos de protocolos de nivel de transporte asociados a este servicio.

4.1.3 SERVICIOS DE NIVEL DE APLICACIÓN

El nivel más alto de la pila IP lo componen las aplicaciones que forman el sistema distribuido. Estas aplicaciones, al igual que en el resto de niveles, ofrecen una interfaz de servicio para que los usuarios las usen, y disponen de un protocolo de nivel de aplicación que gobierna las comunicaciones entre ellas. La mayoría de aplicaciones distribuidas más comunes se ubican en este nivel, como las páginas web, el correo electrónico o los juegos *on-line*.

Se define un **protocolo de nivel de aplicación** como el conjunto de reglas que gobiernan la interacción entre los diferentes elementos de una aplicación distribuida. A la hora de desarrollar un servicio distribuido, definir este protocolo es uno de los pasos fundamentales. En una aplicación cliente/servidor este protocolo especifica cómo se realiza la interacción entre el servidor y los clientes. A su vez, los diferentes elementos de una aplicación cliente/servidor (los clientes y el servidor) se pueden ver como sistemas independientes, con su estructura y función. En este sentido, el servidor es la pieza clave, ya que es la proporciona el servicio deseado a los clientes. Su estructura estará formada por los componentes software con los que está programado. Su función será aquella que realiza para los clientes, y su servicio el procedimiento mediante el cual la realiza. El protocolo de nivel de aplicación define cómo se interactúa con el cliente y es, por tanto, la interfaz de servicio del servidor.

¿SABÍAS QUE...?

Un servidor web es una aplicación que proporciona páginas web a los clientes que lo solicitan, normalmente navegadores web como Firefox o Internet Explorer. Si se considera el servidor web como un sistema, el servicio proporcionado por este es el acceso a la página o páginas que contiene. La interfaz de servicio del servidor web está gobernada por el protocolo de nivel de aplicación que se usa en Internet para el tráfico web, denominado HTTP *(Hipertext Transfer Protocol*). Este protocolo se estudiará (entre otros) en este capítulo.

4.2 PROGRAMACIÓN DE APLICACIONES CLIENTE Y SERVIDOR

Como se ha visto, el modelo de comunicaciones cliente/servidor es el más usado en el desarrollo de aplicaciones distribuidas. Los mecanismos de transmisión de mensajes explicados en el capítulo anterior, los *sockets stream* y *sockets datagram*, son la tecnología base sobre la que se implementa este modelo. A la hora de programar una aplicación siguiendo el modelo cliente/servidor, se deben definir de forma precisa los siguientes aspectos: **funciones del servidor**, **tecnología de comunicaciones** y **protocolo de nivel de aplicación**. A continuación veremos en detalle cada uno de estos aspectos.

4.2.1 FUNCIONES DEL SERVIDOR

A la hora de programar una aplicación distribuida siguiendo el modelo cliente/servidor, el primer paso debe ser siempre definir de forma clara las funciones del *servidor*. Esto equivale a contestar preguntas como: ¿para qué sirve nuestro servidor? Esta definición debe hacerse de manera clara y sin ambigüedades, para facilitar el desarrollo posterior y evitar problemas. Algunas de las preguntas clave que debemos hacernos a la hora de definir nuestro servidor son:

- ¿Cuál es la función básica de nuestro servidor?
- El servicio que proporciona nuestro servidor, ¿es rápido o lento?
- El servicio que proporciona nuestro servidor, ¿puede resolverse con una simple petición y respuesta o requiere del intercambio de múltiples mensajes entre este y el cliente?
- ¿Debe ser capaz nuestro servidor de atender a varios clientes simultáneamente?

El resultado de contestar estas y otras preguntas similares debe ser el hacernos una idea clara de cómo va a ser nuestro servidor, qué va a hacer y cómo va a interactuar con los clientes.

EJEMPLO 4.5

Se desea crear una aplicación cliente/servidor de consulta horaria. Como hemos dicho, el primer paso será definir las funciones del servidor. Analizamos el problema y finalmente decidimos que nuestro servidor tendrá las siguientes características:

- La función básica de nuestro servidor será "dar la hora". Cuando un cliente se conecte, realizará una petición de servicio ("preguntará la hora"). El servidor les responderá con un mensaje en el que figure la hora actual.
- Se trata de un servicio rápido, ya que solo "da la hora".
- La interacción con el cliente se limita a un único intercambio petición-respuesta ("¿Qué hora es?", "Son las 4:10").
- El servidor debe ser capaz de atender a múltiples clientes a la vez.

EJEMPLO 4.6

Se desea crear una aplicación cliente/servidor que funcione como una calculadora. Al igual que en ejemplo anterior, analizamos el problema y decidimos que nuestro servidor tendrá las siguientes características:

- La función básica de nuestro servidor será realizar operaciones aritméticas sencillas. Esto incluye sumas, restas, multiplicaciones y divisiones. Cuando un cliente se conecte, realizará una petición de servicio, indicando la operación que desea realizar ("suma", "resta", etc.) y los operandos de la operación. El servidor les responderá con un mensaje en el que figure el resultado de la operación. El cliente podrá seguir realizando operaciones aritméticas, siguiendo el mismo proceso.
- Nuestro servicio debe mantener una conexión abierta con cada cliente, ya que este debe enviar varios mensajes para solicitar una operación.
- La interacción con el cliente puede ser compleja, con múltiples mensajes intercambiados.
- Al igual que en el ejemplo anterior, el servidor debe ser capaz de atender a múltiples clientes a la vez.

ACTIVIDADES 4.2

- Se desea implementar un servicio de información meteorológica. Los clientes indicarán el nombre de la ciudad en la que viven, y el servidor les dará la temperatura prevista para el día siguiente. Basándose en los ejemplos anteriores, defina las características fundamentales que debe tener este servicio.

4.2.2 TECNOLOGÍA DE COMUNICACIONES

Una vez definidas las características de nuestro servidor, el siguiente paso es escoger la tecnología de comunicaciones que es necesario utilizar. Los dos mecanismos básicos de comunicación que se han estudiado son los *sockets stream* y los *sockets datagram*. Como ya se ha visto, cada tipo de *socket* presenta una serie de ventajas e inconvenientes. Es necesario escoger el tipo de *socket* adecuado, teniendo en cuenta las características del servicio que proporciona nuestro servidor. Los *sockets stream* son más fiables y orientados a conexión, por lo que se deben usar en aplicaciones complejas en las que clientes y servidores intercambian muchos mensajes. Los *sockets datagram* son menos fiables, pero más eficientes, por lo que es preferible usarlos cuando las aplicaciones son sencillas, y no es problema que se pierda algún mensaje.

EJEMPLO 4.7

Continuando con el Ejemplo 4.5, se escoge el tipo de *socket* más adecuado para nuestro servidor de consulta horaria. Dado que se trata de un servicio rápido, que se resuelve con un único intercambio petición-respuesta y los mensajes son cortos, escogemos utilizar *sockets datagram* para este caso.

EJEMPLO 4.8

Continuando ahora con el Ejemplo 4.6, escogemos en este caso *sockets stream* para nuestro servicio calculadora. Los *sockets stream* son más adecuados para este caso, ya que son orientados a conexión, lo que nos permite mantener un canal abierto con cada cliente mientras se realizan las operaciones aritméticas. Además, la fiabilidad de los *sockets stream* nos garantiza que los mensajes no se perderán por el camino, por lo que el servidor siempre recibirá los operandos y operadores aritméticos de forma correcta.

4.2.3 DEFINICIÓN DEL PROTOCOLO DE NIVEL DE APLICACIÓN

Llegados a este punto, se han definido de forma clara las funciones y características de nuestro servidor y hemos escogido el tipo de *socket* que vamos a usar. El último paso, antes de empezar a escribir nuestro programa, es definir el protocolo de nivel de aplicación.

Un **protocolo de nivel de aplicación** es el conjunto de reglas que gobiernan la interacción entre los diferentes elementos de una aplicación distribuida. En el caso del modelo de comunicaciones cliente/servidor, el protocolo de nivel de aplicación definirá explícitamente el formato de los mensajes que se intercambian entre cliente y servidor, así como las posibles secuencias en las que estos pueden ser enviados y recibidos. Dicho de otra forma, la definición del protocolo de nivel de aplicación debe contener todos los posibles tipos de mensajes (tanto peticiones como respuestas) que pueden ser enviados y recibidos, indicando cuándo se puede enviar cada uno y cuándo no.

EJEMPLO 4.9

Continuando con el Ejemplo 4.7, definimos el protocolo de nivel de aplicación para nuestro servicio de consulta horaria. Para ello, identificamos todos los posibles mensajes que se pueden intercambiar, y definimos su formato:

1. Mensaje 1: petición de hora. Se trata de la petición de hora por parte de un cliente.
   - **¿Quién lo envía?** El cliente.
   - **¿Cuándo se envía?** En cualquier momento, un cliente puede enviar este mensaje al servidor para realizar una petición.
   - **¿Qué contiene?** Una cadena de texto como la palabra "hora". Esto le servirá al servidor para comprobar que se trata de una petición correcta.
2. Mensaje 2: respuesta de hora. Se trata de la respuesta que envía el servidor al cliente, con la hora actual.
   - **¿Quién lo envía?** El servidor.
   - **¿Cuándo se envía?** Como respuesta a un cliente que ha realizado una petición de hora.
   - **¿Qué contiene?** Una cadena de texto con la hora, por ejemplo "10:25".

Cuando se define el protocolo de nivel de aplicación de un sistema distribuido complejo, es muy importante especificar de manera clara todas las secuencias posibles de intercambio de mensajes que pueden ocurrir en el sistema y cómo este reacciona. Asegurarnos de que el protocolo está correctamente definido y es exhaustivo es fundamental para evitar posteriormente comportamientos inesperados por parte de nuestra aplicación.

EJEMPLO 4.10

Continuando con el Ejemplo 4.8, definimos el protocolo de nivel de aplicación para nuestro servicio calculadora. Para ello, se identifican todos los posibles mensajes que se pueden intercambiar, y se define su formato:

1. Mensaje 1: inicio de operación. Se trata de la petición de inicio de operación por parte de un cliente.
   - **¿Quién lo envía?** El cliente.
   - **¿Cuándo se envía?** En cualquier momento, un cliente puede enviar este mensaje al servidor para realizar una petición, siempre y cuando otra operación no esté ya en curso por parte del mismo cliente. El cliente debe terminar dicha operación antes de comenzar una nueva.
   - **¿Qué contiene?** Una cadena de texto con un código que indique la operación que desea realizar. Los cuatro códigos posibles son "+" (para la suma), "-" (para la resta), "*" (para la multiplicación) y "/" (para la división). Al igual que en el caso anterior, esto le servirá al servidor para comprobar que se trata de una petición correcta.
2. Mensaje 2: primer operando. Se trata del primer parámetro de la operación aritmética que desea realizar el cliente.
   - **¿Quién lo envía?** El cliente.
   - **¿Cuándo se envía?** Justo después del mensaje 1, para indicar el primer operando de la operación.
   - **¿Qué contiene?** Un número entero, por ejemplo 567.
3. Mensaje 3: segundo operando. Se trata del segundo parámetro de la operación aritmética que desea realizar el cliente.
   - **¿Quién lo envía?** El cliente.
   - **¿Cuándo se envía?** Justo después del mensaje 2, para indicar el segundo operando de la operación.
   - **¿Qué contiene?** Un número entero, por ejemplo 12.
4. Mensaje 4: resultado de la operación. Se trata de la respuesta que envía el servidor al cliente, con el resultado de la operación aritmética.
   - **¿Quién lo envía?** El servidor.
   - **¿Cuándo se envía?** Después de recibir el mensaje 3, como respuesta a un cliente que ha realizado una petición.
   - **¿Qué contiene?** Un número entero con el resultado, por ejemplo 2.387.

4.2.3.1 Protocolos, sesión y estado

En una aplicación cliente/servidor, se llama *sesión* a la secuencia de mensajes que se intercambian entre cliente y servidor desde que se establece la conexión entre ellos hasta que se cierra. En algunos servicios (como el del Ejemplo 4.9), la interacción entre ambas partes se realiza de manera rápida, con un único intercambio de tipo pregunta-respuesta. En estos casos, el control de la sesión no es importante, ya que la interacción es sencilla. En otro (como el del Ejemplo 4.10), esta interacción es más compleja, ya que requiere de múltiples mensajes entre ambas partes. En estos casos, la sesión es mucho más importante, ya que la secuencia en la que se envían y reciben los mensajes es clave. El protocolo de nivel de aplicación refleja esto, y en este sentido se pueden distinguir dos tipos de protocolos:

- **Protocolos sin estado** (en inglés *stateless*): son aquellos en los que la secuencia concreta en la que se reciben los mensajes no es importante, ya que no afecta al resultado de estos. El servidor se limita a recibir las peticiones del cliente y a responderlas una por una, de forma independiente.
- **Protocolos con estado** (en inglés *stateful*): son aquellos en los que la secuencia concreta en la que se reciben los mensajes es importante, ya que afecta al resultado final de las peticiones. En estos casos, el servidor debe almacenar información intermedia durante la sesión, denominada el *estado de la sesión*. Conocer este estado es imprescindible para poder resolver las peticiones de manera correcta.

4.2.4 IMPLEMENTACIÓN

Una vez se han especificado las características del servidor, seleccionado la tecnología de comunicaciones y definido el protocolo de nivel de aplicación ya solo queda implementar la aplicación.

EJEMPLO 4.11

Continuando con el Ejemplo 4.9, esta es la implementación del servidor de nuestra aplicación de servicio horario:

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;
import java.net.InetSocketAddress;
import java.net.SocketException;
import java.util.Date;

public class HoraServer {

    public static void main(String[] args) {

         System.out.println("Arrancando servidor de hora.");
```

EJEMPLO 4.11 (cont.)

```
            DatagramSocket datagramSocket = null;
            try {
                 InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
                 datagramSocket = new DatagramSocket(addr);
            } catch (SocketException e) {
                 e.printStackTrace();
            }

            while (datagramSocket != null) {
                 try {
                      System.out.println("Esperando mensaje");

                      byte[] buffer = new byte[4];
                      DatagramPacket datagrama1 = new DatagramPacket(buffer, 4);
                      datagramSocket.receive(datagrama1);

                      String mensaje = new String(datagrama1.getData());

                      InetAddress clientAddr = datagrama1.getAddress();
                      int clientPort = datagrama1.getPort();

                      System.out.println("Mensaje recibido: desde " +
                      clientAddr + ", puerto " + clientPort);
                      System.out.println("Contenido del mensaje: " + mensaje);

                      if (mensaje.equals("hora")) {

                           System.out.println("Enviando respuesta");

                           Date d = new Date(System.currentTimeMillis());
                           byte[] respuesta = d.toString().getBytes();
                           DatagramPacket datagrama2 =
                                new DatagramPacket(respuesta, respuesta.length,
                                                   clientAddr, clientPort);
                           datagramSocket.send(datagrama2);

                           System.out.println("Mensaje enviado");
                      } else {
                           System.out.println("Mensaje recibido no reconocido");
                      }

                 } catch (IOException e) {
                      e.printStackTrace();
                 }
            }
            System.out.println("Terminado");
       }
  }
```

EJEMPLO 4.12

Ejemplo de cliente para nuestro servidor de consulta horaria (Ejemplo 4.11):

```
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.DatagramSocket;
import java.net.InetAddress;

public class HoraClient {

    public static void main(String[] args) {
        try {
            System.out.println("Creando socket datagrama");

            DatagramSocket datagramSocket = new DatagramSocket();

            System.out.println("Enviando petición al servidor");

            String mensaje = new String("hora");

            InetAddress serverAddr = InetAddress.getByName("localhost");
            DatagramPacket datagrama1 =
                new DatagramPacket(mensaje.getBytes(),
                                   mensaje.getBytes().length,
                                   serverAddr, 5555);
            datagramSocket.send(datagrama1);

            System.out.println("Mensaje enviado");

            System.out.println("Recibiendo respuesta");

            byte[] respuesta = new byte[100];
            DatagramPacket datagrama2 =
                new DatagramPacket(respuesta, respuesta.length);
            datagramSocket.receive(datagrama2);

            System.out.println("Mensaje recibido: "+new String(respuesta));

            System.out.println("Cerrando el socket datagrama");

            datagramSocket.close();

            System.out.println("Terminado");

        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

ACTIVIDADES 4.3

- Observa el Ejemplo 4.9, que define el protocolo de nivel de aplicación para el servidor de consultas horarias. ¿Qué modificaciones habría que introducir para que diese también la fecha, dependiendo de lo que solicitase el cliente?
- Basándote en lo desarrollado en la actividad anterior, modifica el Ejemplo 4.11 para que acepte peticiones de hora o fecha.

4.2.4.1 Utilización de hilos en aplicaciones cliente/servidor

La mayor parte de servidores están pensados para atender a múltiples clientes simultáneamente. Un servidor moderno que proporcione servicio a muchos usuarios, como el que da soporte al correo electrónico de GMail, por ejemplo, debe ser capaz de antender a miles de clientes simultáneamente. Esto quiere decir que, si al servidor le llega una nueva petición mientras está atendiendo a un cliente, se deben cumplir las siguientes condiciones:

- La nueva petición no debe interferir con la que está en curso. Dicho de otra forma, los clientes deben percibir que operan con el servidor ellos solos, independientemente de cuántos clientes haya.
- La nueva petición debe ser atendida lo antes posible, incluso de manera simultánea a la que está ya en curso. Esto evita que unos clientes tengan que esperar por otros.

Para que estas condiciones se cumplan, la mayoría de servidores optan por un implementar un enfoque *multihilo*. Un **servidor multihilo** es un servidor en el que a cada cliente se le atiende en un hilo de ejecución independiente. Cuando un nuevo cliente envía una petición, el servidor arranca un hilo específico para atender las peticiones de este cliente. Este hilo se dedica exclusivamente a interactuar con el cliente en cuestión. Mientras tanto, el hilo principal del servidor queda libre, esperando a nuevos clientes.

Los *sockets stream* son especialmente adecuados para implementar servidores multihilo. Cada vez que el *socket* servidor recibe un intento de conexión, la operación *accept* crea un nuevo *socket*, conectado con el cliente. Al implementar nuestro servidor, el procedimiento típico consiste en arrancar entonces un nuevo hilo, y darle a este el nuevo *socket* que se acaba de crear. De esta forma, el nuevo hilo dispone de un canal de comunicación ya abierto con el cliente, a través del cual pude responder a sus peticiones. El hilo principal y el *socket* servidor permanecen libres, realizando la operación *accept* a la espera de nuevas conexiones.

EJEMPLO 4.13

Continuando con el Ejemplo 4.10, esta es la implementación del servidor de nuestra aplicación de calculadora. Fíjate en que hace uso de múltiples hilos para poder atender a varios clientes de manera simultánea:

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.ServerSocket;
import java.net.Socket;
```

EJEMPLO 4.13 (cont.)

```
public class CalcServer extends Thread {

    private Socket clientSocket;

    public CalcServer(Socket socket) {
         clientSocket = socket;
    }

    public void run() {

         try {

              System.out.println("Arrancando hilo");

              InputStream is = clientSocket.getInputStream();
              OutputStream os = clientSocket.getOutputStream();

              System.out.println("Esperando mensaje de operación");

              byte[] buffer = new byte[1];
              is.read(buffer);
              String operacion = new String(buffer);

              System.out.println("Operaciùn recibida: " + new String(operacion));

              if (operacion.equals("+") || operacion.equals("-") ||
                   operacion.equals("*") || operacion.equals("/")) {

                   System.out.println("Esperando primer operador");

                   int op1 = is.read();

                   System.out.println("Primer operador: " + op1);

                   System.out.println("Esperando segundo operador");

                   int op2 = is.read();

                   System.out.println("Segundo operador: " + op2);

                   System.out.println("Calculando resultado");

                   int result = Integer.MIN_VALUE;

                   if (operacion.equals("+")) {
                        result = op1 + op2;
```

EJEMPLO 4.13 (cont.)

```
                    } else if (operacion.equals("-")) {
                         result = op1 - op2;
                    } else if (operacion.equals("*")) {
                         result = op1 * op2;
                    } else if (operacion.equals("/")) {
                         result = op1 / op2;
                    }

                    System.out.println("Enviando resultado");

                    os.write(result);

               } else {
                    System.out.println("Operaciùon no reconocida");
               }

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

          System.out.println("Hilo terminado");
     }

     public static void main(String[] args) {

          System.out.println("Creando socket servidor");

          ServerSocket serverSocket = null;

          try {
               serverSocket = new ServerSocket();

               System.out.println("Realizando el bind");

               InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
               serverSocket.bind(addr);

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

          System.out.println("Aceptando conexiones");

          while (serverSocket != null) {
               try {
                    Socket newSocket = serverSocket.accept();
```

EJEMPLO 4.13 (cont.)

```
                    System.out.println("Conexión recibida");

                    CalcServer hilo = new CalcServer(newSocket);
                    hilo.start();

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

          System.out.println("Terminado");
     }
}
```

EJEMPLO 4.14

Ejemplo de cliente para nuestro servidor de calculadora (Ejemplo 4.13):

```
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import java.net.Socket;


public class CalcClient {

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

          System.out.println("Creando socket cliente");

          Socket clientSocket = new Socket();

          System.out.println("Estableciendo la conexión");

          InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
          clientSocket.connect(addr);

          InputStream is = clientSocket.getInputStream();
          OutputStream os = clientSocket.getOutputStream();

          System.out.println("Enviando petición de suma");
```

EJEMPLO 4.14 (cont.)

```
            os.write("+".getBytes());

            System.out.println("Enviando primer operando");

            os.write(59);

            System.out.println("Enviando segundo operando");

            os.write(130);

            System.out.println("Recibiendo resultado");

            int result = is.read();

            System.out.println("Resultado de la suma: "+result);

            System.out.println("Cerrando el socket cliente");

            clientSocket.close();

            System.out.println("Terminado");

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

ACTIVIDADES 4.4

- Observa los Ejemplos 4.13 y 4.14, con la implementación de la aplicación cliente/servidor de calculadora. A la vista del código del servidor, ¿cómo habría que modificar el cliente para que, a continuación, solicitase una operación de división (10/2, por ejemplo)? ¿Le basta al cliente con enviar nuevos mensajes o debe reiniciar la conexión? ¿Por qué?

4.3 PROTOCOLOS ESTÁNDAR DE NIVEL DE APLICACIÓN

En la actualidad existen multitud de aplicaciones distribuidas de uso generalizado. Las páginas web o el correo electrónico son ejemplos típicos de sistemas estándar que siguen el modelo cliente/servidor para organizar sus comunicaciones. En la mayoría de estos casos, el cliente y el servidor de cualquiera de estos sistemas son aplicaciones independientes, desarrolladas por diferentes personas/compañías y muchas veces ni siquiera implementadas usando el mismo lenguaje de programación. La definición exhaustiva de un protocolo de nivel de aplicación es fundamental en estos casos, para garantizar que clientes y servidores pueden comunicarse de manera efectiva. Para la mayoría de aplicaciones distribuidas comunes se han definido, a lo largo de los años, protocolos de nivel de aplicación estándar, que facilitan la tarea de desarrollar servidores y clientes para ellas. A continuación se estudiarán algunos de los protocolos estándar de nivel de aplicación más importantes.

¿SABÍAS QUE...?

La *World Wide Web* es probablemente el conjunto de aplicaciones distribuidas más usado en el mundo. Actualmente en Internet se pueden encontrar muchas aplicaciones servidoras de páginas web distintas, como Apache, Internet Information Service o Google Web Server. A su vez, hay una gran variedad de clientes web (comúnmente llamados "navegadores"), como Firefox, Opera, Safari, Internet Explorer, etc. Gracias al protocolo estándar de nivel de aplicación HTTP, cualquier servidor puede comunicarse con cualquier navegador.

¿SABÍAS QUE...?

Una interacción entre un cliente web y un servidor es un ejemplo típico de una aplicación distribuida en la que cada elemento ha sido desarrollado por una entidad distinta, usando lenguajes y tecnologías diferentes. La aplicación servidora más usada, Apache, está escrita en lenguaje C y ha sido desarrollada por el proyecto Apache. El navegador web Firefox, desarrollado por Mozilla, está escrito en C++ y Javascript UI. El navegador web de Microsoft, Internet Explorer, está escrito también en C++, aunque versiones anteriores fueron desarrolladas en Visual Basic.

4.3.1 TELNET

El **Telnet** es un protocolo de nivel de aplicación diseñado para proporcionar comunicación bidireccional basada en texto plano (caracteres ASCII) entre dos elementos de una red. Este protocolo simula una conexión virtual a una terminal de texto, como las antiguas terminales que se usaban para conectarse a los grandes ordenadores en los centros de datos. Telnet forma parte de la pila de protocolos IP, y utiliza el protocolo de nivel de transporte TCP (*sockets stream*) para intercambiar mensajes entre ambos extremos de la comunicación.

Tradicionalmente, el Telnet se ha usado para conectarse remotamente a máquinas, creando una sesión de línea de mandatos como las *Shell* de los sistemas operativos UNIX o el "símbolo de sistema" de Windows (C:\>). Usando un cliente Telnet, un usuario se puede conectar a una máquina que esté en otro punto de la red y hacer uso de ella de manera similar a si estuviese físicamente junto a ella. En una interacción típica por Telnet, el usuario escribe las órdenes usando su teclado, y el cliente las envía a la máquina destino como cadenas de texto por un *socket stream*. Al llegar al receptor, el servidor de Telnet recoge las cadenas de texto recibidas y las ejecuta como órdenes en la terminal. Posteriormente envía por el mismo *socket stream* la salida que ha producido la ejecución de la orden, para que el usuario pueda verla en su extremo. Telnet es un protocolo con estado (*stateful*). El número de puerto por defecto para servidores Telnet es el 23.

¿SABÍAS QUE...?

Para acceder al símbolo de sistema en Windows 7 basta con abrir el menú de inicio y seleccionar **Todos los programas**. Dentro de Accesorios hay un acceso directo a este.

Figura 4.2. *Símbolo de sistema en Windows 7*

El principal problema del protocolo Telnet es que las cadenas de texto se envían de forma clara por la red, sin ningún tipo de protección o cifrado. Esto hace que cualquier aplicación que monitorice las comunicaciones pueda leer perfectamente los mensajes que se intercambian cliente y servidor. Esto puede suponer un problema de seguridad grave, cuando se transfiere información sensible como contraseñas, etc. Por esta razón, el uso de este protocolo está desaconsejado en la actualidad, salvo en redes locales y entornos controlados en los que existen mecanismos de seguridad adicionales.

¿SABÍAS QUE...?

El protocolo de nivel de aplicación estándar Telnet fue definido en 1973 y ampliamente usado hasta la década de los 90, cuando fue sustituido por SSH (que se verá a continuación).

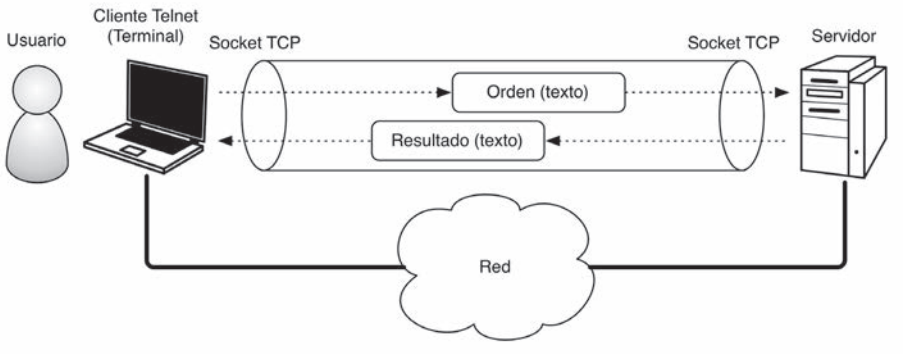


Figura 4.3. *Comunicación usando el protocolo de nivel de aplicación Telnet*

4.3.2 SSH (*SECURE SHELL*)

SSH es un protocolo de nivel de aplicación muy similar a Telnet. Sus características básicas y funcionamiento son esencialmente iguales, y fue diseñado con el mismo propósito. Se trata, en cambio, de un protocolo mucho más moderno (la primera versión es de 1995), que fue desarrollado precisamente para suplir las carencias de Telnet. La principal característica de SSH que lo diferencia de Telnet (y su principal virtud) es que la información transferida entre cliente y servidor está cifrada. Esto aumenta enormemente la seguridad de las comunicaciones ya que, aunque se monitorice el tráfico, no se puede acceder de forma directa a los mensajes, pues se envían codificados. SSH es el protocolo de nivel de aplicación recomendado en la actualidad para realizar sesiones de líneas de mandatos en máquinas remotas, especialmente si la comunicación se realiza a través de Internet u otra red no segura. SSH es un protocolo con estado (*stateful*). El número de puerto por defecto para servidores SSH es el 22.

ACTIVIDADES 4.5

- Busca en la Web más información sobre los protocolos Telnet y SSH. ¿Qué ventajas adicionales ofrece el segundo frente al primero?

4.3.3 FTP (*FILE TRANSFER PROTOCOL*)

FTP es un protocolo de nivel de aplicación diseñado para la transferencia de archivos a través de una red de comunicaciones. Al igual que Telnet y SSH, FTP utiliza el protocolo de transporte TCP (*sockets stream*) para establecer un canal de comunicación entre cliente y servidor y realizar la transferencia de datos. FTP establece un par de conexiones simultáneas entre cliente y servidor, que usa de forma distinta. La primera (conexión de control) se usa para enviar órdenes al servidor y obtener información. La segunda (conexión de datos) se utiliza para transferir los archivos. FTP permite tanto la descarga de archivos (del servidor al cliente) como la subida de estos (del cliente al servidor). Además, incluye mecanismos para listar y navegar por el árbol de directorios, cambiar las propiedades de los archivos (como el nombre o los permisos de acceso), etc. FTP es un protocolo con estado (*stateful*). El número de puerto por defecto para servidores FTP es el 21.

¿SABÍAS QUE...?

Al igual que ocurre con Telnet, el protocolo de nivel de aplicación FTP no está diseñado para ser seguro, y las comunicaciones se realizan sin protección. Esto puede causar toda clase de problemas de seguridad, sobre todo en redes públicas como Internet. Existe una extensión al protocolo, llamada FTPS (*FTP Secure*), que incorpora añadidos que lo hacen más seguro. Además existe otra alternativa segura, llamada SFTP. A diferencia de FTPS, SFTP no es una extensión de FTP, es un protocolo de transferencia de archivos distinto, basado a su vez en SSH.

ACTIVIDADES 4.6

- Uno de los clientes de FTP más usados en Windows es FileZilla (*https://filezilla-project.org/*). Descárgalo y úsalo para conectarte a un servidor de FTP. Comprueba que puedes navegar por los directorios, descargar archivos, etc.

4.3.4 HTTP (*HYPERTEXT TRANSFER PROTOCOL*)

HTTP es, probablemente, el protocolo de nivel de aplicación más importante de la actualidad. La mayoría del tráfico que se realiza en la *World Wide Web*, la parte más visible de la red Internet, utiliza este protocolo para controlar la transferencia de información entre las diferentes aplicaciones implicadas (fundamentalmente navegadores y servidores web).

¿SABÍAS QUE...?

La mayoría de la gente considera los términos "Internet" y "*World Wide Web*" (o simplemente "Web") como sinónimos. Esto se debe a que la gran mayoría de usuarios de la red Internet utiliza casi de forma exclusiva las páginas web para acceder a ella. Esto, por supuesto, es incorrecto. Internet es una red global de comunicaciones que utiliza la pila IP como arquitectura fundamental de comunicaciones. La web es el conjunto de aplicaciones distribuidas que, sobre Internet, proporcionan el servicio de acceso a documentos de *hipertexto*, comúnmente llamados "páginas web".

Según su definición, la función de este protocolo es facilitar la transmisión de documentos de *hipertexto* entre diferentes aplicaciones ubicadas dentro de una red. Un documento de *hipertexto* es un texto expresado en un código especial, diseñado para ser mostrado en un dispositivo electrónico como un ordenador o un *e-book*. El código usado por el documento de *hipertexto* suele contener mecanismos que permiten formatear su contenido de manera sofisticada, permitiendo organizarlo en columnas, insertar imágenes, etc. La característica más importante de un documento de *hipertexto* es que este contiene *hiperenlaces* (en inglés *hiperlinks*) a otros documentos. Un usuario que esté visualizando el documento debe poder acceder a estos *hiperenlaces* de forma sencilla, haciendo clic en ellos con el ratón, por ejemplo. Las páginas web son un caso típico de documentos de *hipertexto*.

¿SABÍAS QUE...?

El protocolo de nivel de aplicación HTTP se ocupa de la transferencia de documentos de *hipertexto*, pero no especifica cómo debe estar organizado el contenido dentro de ellos. En la Web se utiliza principalmente el lenguaje de marcas HTML (*HyperText Markup Language*) para codificar las páginas.

Como el resto de protocolos de nivel de aplicación estándar vistos hasta ahora, HTTP sigue el modelo de comunicaciones cliente/servidor, y gestiona el paso de mensajes por el mecanismo petición-respuesta. HTTP forma parte de la pila de protocolos IP y está diseñado para utilizar un protocolo de nivel de transporte fiable y orientado a conexión. Aunque normalmente funciona sobre TCP, no restringe su uso a este, y puede funcionar también sobre alternativas de transporte menos fiables, como UDP. El número de puerto por defecto para servidores HTTP es el 80.

¿SABÍAS QUE...?

Al igual que Telnet y FTP, el protocolo de nivel de aplicación HTTP no está diseñado para ser seguro, y la transferencia de información se realiza sin ningún tipo de protección. Existe una extensión de HTTP, denominada HTTPS (*HTTP Secure*), que incorpora mecanismos de cifrado de las comunicaciones. HTTPS es la base del comercio electrónico a través de la Web, ya que permite transferir información sensible (contraseñas, números de tarjeta de crédito, etc.) de manera confidencial.

A diferencia de otros protocolos vistos anteriormente, como Telnet y SSH, HTTP es un protocolo sin estado (*stateless*). No obstante, algunos servidores HTTP implementan mecanismos para almacenar el estado de la sesión. El método más usado para conseguir este efecto consiste en almacenar pequeños fragmentos de información en la aplicación cliente (normalmente el navegador web). A estos fragmentos de información se les conoce habitualmente como *cookies*.

4.3.4.1 Sesiones, recursos y peticiones HTTP

Se conoce como una **sesión HTTP** a una secuencia de intercambios petición-respuesta entre un cliente y un servidor. Normalmente, el cliente inicia una sesión estableciendo una conexión TCP (*socket stream*) con el puerto 80 del servidor. Una vez establecida la conexión, el servidor espera la llegada de peticiones HTTP por parte del cliente, y las responde. Las respuestas del servidor contienen información sobre el estado de realización de la petición (si se ha

podido llevar a cabo con éxito o no, etc.). Pueden además incluir contenido adicional (si se ha solicitado), como páginas web. Este contenido se almacena en lo que se conoce como *cuerpo* del mensaje (en inglés *message body*).

Un servidor HTTP organiza la información que contiene (fundamentalmente documentos de *hipertexto*, es decir, páginas web) usando *recursos*. Un recurso puede ser cualquier documento que contenga el servidor y suele estar identificado por una clave única, que se usa en los *hiperenlaces* que se refieren a él. En la Web, estas claves se llaman *Uniform Resource Locators*[7] o URL, comúnmente conocidas como *direcciones web*.

HTTP define toda una serie de peticiones que los clientes pueden realizar durante una sesión. En la versión 1.1 de HTTP (la utilizada actualmente en la Web) son las siguientes:

- **GET**: se utiliza para solicitar recursos, como páginas web identificadas por su URL. Esta es la primera petición que suele realizar un navegador web cuando se conecta a un servidor.
- **HEAD**: similar a GET, pero la respuesta nunca contiene cuerpo del mensaje. Esta petición no permite obtener recursos, pero es útil para obtener información sobre el estado del servidor, sin tener que descargar un documento completo.
- **POST**: se utiliza para solicitar al servidor la incorporación de nuevo contenido a un recurso existente, identificado por su URL. Esta petición se usa de forma habitual en páginas web que permiten a los usuarios agregar contenido, como foros o las secciones de comentarios de muchos artículos en periódicos *on-line* y blogs.
- **PUT**: se utiliza para solicitar la incorporación de un nuevo recurso al servidor, identificado por una nueva URL (proporcionada como parte de la petición). Si la URL ya existe en el servidor, este sustituirá el recurso existente por el nuevo enviado en el PUT.
- **DELETE**: se utiliza para solicitar la eliminación de un recurso.
- **OPTIONS**: se utiliza para solicitar una lista de las peticiones (GET, PUT, POST, etc.) que el servidor acepta sobre una determinada URL.
- **TRACE**: se utiliza para solicitar al servidor que le devuelva la petición que acaba de recibir del mismo cliente, a modo de eco. Esto es útil para detectar posibles modificaciones de la petición realizadas por elementos intermedios de la comunicación entre el cliente y el servidor.
- **CONNECT**: se utiliza para convertir la conexión entre el cliente y servidor en un túnel TCP/IP. Esto facilita la transmisión de datos cifrados.
- **PATCH**: se utiliza para realizar modificaciones parciales a un recurso.

No todos los servidores HTTP están obligados a aceptar todas las posibles peticiones. Dependiendo del recurso, determinadas peticiones pueden no ser aceptables, como POST o DELETE en páginas web que no se pueden modificar. Para poder operar correctamente, un servidor debería ser capaz al menos de aceptar peticiones GET y HEAD. También se recomienda que acepte la operación OPTIONS, cuando sea posible.

7 Se podría traducir como "localizadores uniformes de recursos".

ACTIVIDADES 4.7

- Busca información sobre la estructura de las peticiones HTTP. ¿Cómo es una cabecera HTTP? ¿Qué formato tiene una petición GET? ¿Se pueden leer de manera más o menos cómoda?

4.3.4.2 Códigos de estado

Como ya se ha explicado, todas las respuestas HTTP contienen información sobre el estado de realización de la petición. La primera línea de texto del mensaje de respuesta, llamada *línea de estado* (en inglés *status line*) contiene un código numérico de estado y una pequeña frase explicativa. El estándar HTTP en su versión 1.1 define multitud de códigos de estado, abarcando todas las posibles respuestas a las diferentes peticiones. Estos códigos son normalmente números de tres dígitos, y se dividen en 5 categorías:

- **Información**: son los códigos que empiezan por 1, y se usan para informar al cliente de aspectos diversos. El código 100 (*continue*), por ejemplo, se utiliza para solicitar al cliente que continúe enviando información, como parte de una petición de POST, por ejemplo.
- **Éxito**: son los códigos que empiezan por 2, y se usan para indicar al cliente que su petición ha sido recibida y procesada correctamente. Ejemplos típicos de esta categoría son el 200 (*OK*), el mensaje de éxito estándar, o el 202 (*accepted*), que indica que la petición ha sido recibida y aceptada, pero aún no ha sido procesada.
- **Redirección**: son los códigos que empiezan por 3, y se usan para indicar al cliente que debe realizar operaciones adicionales para completar su petición. Un ejemplo es el código 303 (*see other*), que se utiliza para redirigir al cliente a una nueva URL.
- **Error del cliente**: son los códigos que empiezan por 4, y se usan para indicar al cliente que ha cometido un error. Ejemplos típicos de esta categoría son el código 400 (*bad request*), que indica que la petición es incorrecta, o el 403 (*forbidden*), que indica que la petición no está permitida (se ha intentado hacer un POST sobre un recurso de solo lectura, por ejemplo). La mayoría de códigos de estado del estándar HTTP 1.1 pertenecen a esta categoría.
- **Error del servidor**: son los códigos que empiezan por 5, y se usan para indicar al cliente que el servidor ha experimentado un error y no puede completar la petición. Ejemplos de esta categoría son el 500 (*internal server error*) y el 503 (*service unavailable*).

¿SABÍAS QUE...?

El código de estado HTTP más reconocido por la mayor parte de usuarios de la Web es el 404 (*not found*). Este código indica al cliente que el recurso solicitado no se encuentra en el servidor, y en la mayoría de situaciones se origina por el uso de un *hiperenlace* antiguo, que contiene una URL que ya no existe. Prácticamente todas las personas que usan la Web de manera frecuente se han encontrado en algún momento con un mensaje que contenía este código de estado.

ACTIVIDADES 4.8

- Busca en la Web más información sobre el protocolo HTTP. ¿Existe (o ha existido) alguna vez alguna alternativa a este para la transferencia de páginas web?

4.3.5 POP3 (*POST OFFICE PROTOCOL*, VERSIÓN 3)

El protocolo de nivel de aplicación POP está diseñado para que las aplicaciones clientes de *e-mail*, como Thuderbird o Outlook, accedan los mensajes alojados en los servidores de correo electrónico. La mayoría de servidores comerciales de *e-mail*, como el GMail de Google o el Hotmail de Microsoft, soportan este protocolo. La versión actual y más extendida de POP es la 3, conocida como POP3.

POP3 se basa en el protocolo de transporte TCP (*sockets stream*) y proporciona peticiones básicas para acceso, descarga y borrado de mensajes. POP3 es un protocolo sin estado (*stateless*). El número de puerto por defecto para servidores POP3 es el 110.

¿SABÍAS QUE...?

POP3 es uno de los dos protocolos de nivel de aplicación estándar para clientes de correo electrónico. La alternativa a POP3 es IMAP (*Internet Message Access Protocol*). En muchos aspectos, IMAP es un protocolo mucho más sofisticado que POP3, aunque sus funciones principales son parecidas. Todos los clientes y servidores de correo electrónico comúnmente usados en la actualidad soportan ambos protocolos.

ACTIVIDADES 4.9

- Busca información sobre el protocolo IMAP ¿Qué características adicionales ofrece frente a POP3?
- Considera dos posibles escenarios: 1) acceso al correo electrónico desde el interior de una red corporativa, y 2) acceso al correo electrónico a través de Internet. ¿Qué protocolo de acceso (POP3 o IMAP) te parece más adecuado en cada caso?
- Hoy en día, la mayoría de proveedores de *e-mail*, como GMail o Hotmail, proporcionan aplicaciones para acceder al correo electrónico a través de una página web (cliente de correo web). En un contexto como este, ¿qué ventajas aporta usar un cliente de correo convencional, que implemente el protocolo POP3?
- Busca en la página web de tu proveedor de correo (GMail, Hotmail, etc.) cómo habilitar el acceso POP3 desde un cliente como Thunderbird. Descarga Thunderbird y configúralo para acceder a tu correo por POP3.

4.3.6 SMTP (*SIMPLE MAIL TRANSFER PROTOCOL*)

SMTP es el protocolo de nivel de aplicación estándar para el envío de mensajes de correo electrónico en Internet. Todos los *e-mail* que circulan por la red lo hacen gracias a este protocolo. Además, SMTP es el protocolo que usan los clientes de correo electrónico, como Thunderbird o Outlook, para entregar los mensajes que se desea enviar a los servidores que proporcionan el servicio de correo, como GMail o Yahoo! Mail.

SMTP se basa en el protocolo de transporte TCP (*sockets stream*) y proporciona peticiones para el envío de mensajes de correo electrónico entre un emisor y un receptor. SMTP es un protocolo sin estado (*stateless*). El número de puerto por defecto para servidores SMTP es el 587.

¿SABÍAS QUE...?

Ninguno de los protocolos que se usan habitualmente para la transferencia de correos electrónicos a través de Internet (POP3, SMTP, etc.) incorpora mecanismos de seguridad como el cifrado de las comunicaciones. En el caso del protocolo POP3, existe una extensión a este denominada STARTTLS, que permite cifrar las comunicaciones entre cliente y servidor. El caso de SMTP es mucho más complejo, ya que incorporar mecanismos de seguridad implicaría sustituir todos los servidores de correo electrónico del mundo para que todo el tráfico de mensajes por la red fuese seguro. En la práctica esto no ha ocurrido, por lo que los mensajes se envían casi siempre en claro a través de Internet. Esto es muy importante, ya que quiere decir que **el correo electrónico es un método de comunicación no seguro**, y nunca se debería usar para enviar información confidencial como números de tarjeta de crédito, etc. Si un usuario desea que sus correos electrónicos vayan cifrados, debe realizar esta tarea por sí mismo y asegurarse de que los destinatarios de dichos mensajes tengan los medios para descifrarlos.

4.3.7 OTROS PROTOCOLOS DE NIVEL DE APLICACIÓN IMPORTANTES

Además de los vistos hasta ahora, existen muchos otros protocolos de nivel de aplicación ampliamente utilizados por las aplicaciones distribuidas actuales. Algunos ejemplos de estos protocolos son:

- **DHCP** (*Dynamic Host Configuration Protocol*): es un protocolo que se usa para la configuración dinámica de dispositivos dentro de una red. Su uso más habitual es la asignación dinámica de direcciones IP.
- **DNS** (*Domain Name System*): es un protocolo que se utiliza para la resolución de nombres simbólicos de máquinas en la red. Usando DNS se puede obtener la dirección IP de una máquina, a partir de su nombre.
- **NTP** (*Network Time Protocol*): se trata de un protocolo diseñado para la sincronización de relojes entre máquinas de una red.
- **TLS** (*Transport Layer Security*): es un protocolo diseñado para incorporar características criptográficas a los protocolos de nivel de transporte, como TCP. TLS y su predecesor, SSL (*Secure Sockets Layer*) son la base de la mayoría de extensiones de seguridad de muchos protocolos de nivel de aplicación como el FTPS o el HTTPS.

¿SABÍAS QUE...?

En la mayoría de sistemas operativos (Windows, Linux, Mac OS X, etc.) existe una herramienta llamada *nslookup* que sirve para realizar consultas al sistema de resolución de nombre simbólicos DNS. Por ejemplo, en Mac OS X podemos resolver el nombre simbólico *www.google.es* de la siguiente forma:

```
$ nslookup www.google.es

Server:  8.8.8.8
Address: 8.8.8.8#53

Non-authoritative answer:
Name:    www.google.es
Address: 173.194.41.223
Name:    www.google.es
Address: 173.194.41.215
Name:    www.google.es
Address: 173.194.41.216
```

ACTIVIDADES 4.10

- Busca en la Web información sobre el protocolo NTP. ¿En qué se basa? ¿Cuál es su principal uso?
- DHCP es un protocolo que se usa mucho en redes de área local, para la asignación de direcciones IP locales. ¿Es este el único uso que se hace de este protocolo? Busca información sobre él en la Web e identifica sus funciones principales.

4.4 TÉCNICAS AVANZADAS DE PROGRAMACIÓN DE APLICACIONES DISTRIBUIDAS

Hasta ahora se han visto los fundamentos del desarrollo de servicios en red, las técnicas básicas de implementación de estos y algunos de los más importantes protocolos de nivel de aplicación de la actualidad. Todos los servicios y protocolos de nivel de aplicación vistos se basan en la pila IP para llevar a cabo las comunicaciones y utilizan protocolos de transporte como TCP y UDP para gestionar el envío de sus mensajes. En el capítulo anterior se ha visto en detalle la utilización de *sockets stream* y *sockets datagram*, que son las herramientas básicas para hacer uso de dichos protocolos. No obstante, desde el punto de vista del desarrollo de aplicaciones de alto nivel, las interfaces de programación que normalmente se usan para operar con *sockets* (como la que ofrece Java que se estudió en el capítulo previo) resultan poco prácticas, ya que requieren el formateo completo de los mensajes y el control detallado de las comunicaciones.

Existen multitud de alternativas al uso directo de *sockets* a la hora de programar aplicaciones distribuidas. Cada una de estas alternativas ofrece ventajas distintas, por lo que se deben analizar y considerar en detalle antes de empezar a desarrollar una aplicación de alto nivel. La mayoría de estas tecnologías avanzadas de comunicación son, en realidad, una capa extra de software que se coloca entre el nivel de transporte y el nivel de aplicación de la pila de protocolos. Por debajo, la comunicación sigue ocurriendo mediante *sockets stream* o *sockets datagram*, pero su uso se abstrae al programador para facilitar su trabajo y proporcionar funcionalidades de comunicación complejas de manera cómoda y eficiente.

4.4.1 INVOCACIÓN DE MÉTODOS REMOTOS

La primera de las técnicas avanzadas de comunicación que se van a estudiar es la invocación de métodos remotos. Esta tecnología se basa en la idea de que en un programa orientado a objetos la invocación de métodos es, desde un punto de vista teórico, un proceso de comunicación. Dicho de otra forma, cuando en un programa orientado a objetos el objeto A invoca un método del objeto B, esto se puede analizar como si el objeto A enviase un mensaje a B, B realizase una serie de operaciones, y, finalmente, devolviese el resultado a A mediante otro mensaje. Una invocación de un método remoto implica que el objeto B se encuentra en un lugar distinto de la red que el objeto A y, por lo tanto, para poder invocar su método debe ocurrir un intercambio de mensajes a través de la red. En el lenguaje Java, la invocación a métodos remotos se conoce por sus siglas en inglés: **RMI** (*Remote Method Invocation*).

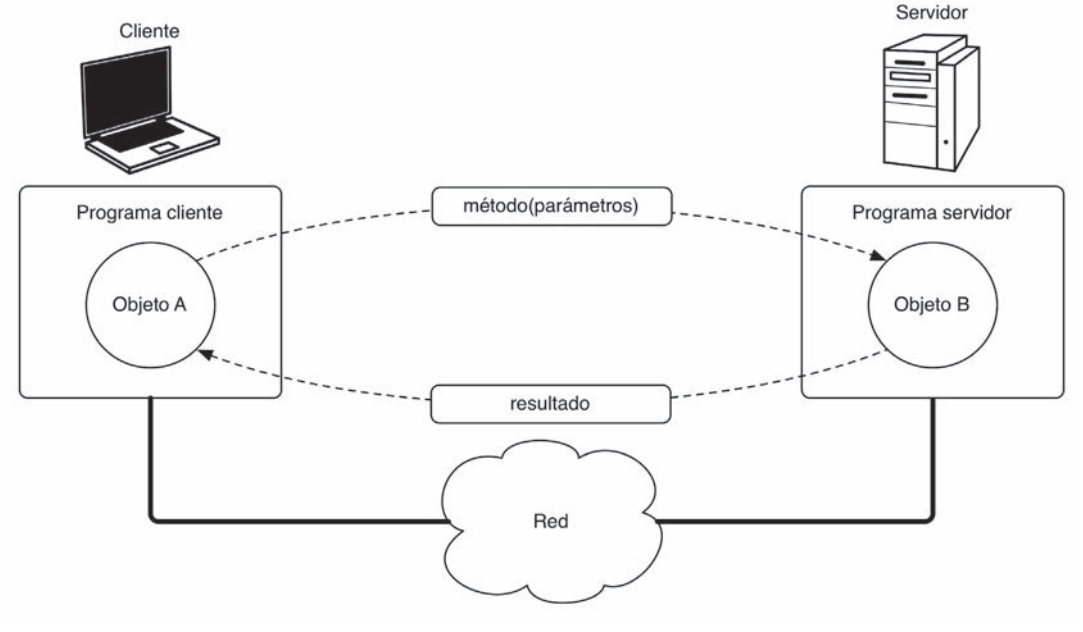


Figura 4.4. *Ejemplo de invocación de un método remoto*

¿SABÍAS QUE...?

Existe otra técnica de comunicación entre aplicaciones muy similar a la invocación de métodos remotos, denominada *llamada a procedimiento remoto* o RPC (en inglés *Remote Procedure Call*). Las RMI y las RPC son tecnologías muy parecidas. La diferencia clave entre ellas es el paradigma de programación en el que se basan. RMI está basado en la programación orientada a objetos, y RPC en la programación estructurada, y de ahí la diferencia de nomenclatura.

En una invocación a un método remoto existen cuatro componentes fundamentales: **objeto servidor**, **objeto cliente**, **método invocado** y **valor de retorno**.

4.4.1.1 Objeto servidor

Se trata del objeto cuyo método es invocado. Se llama **objeto servidor** porque es el que recibe la petición (llamada al método) y la procesa. Al objeto servidor se le llama habitualmente también **objeto remoto**, ya que es al que se accede de forma remota.

4.4.1.2 Objeto cliente

Se trata del objeto que invoca el método. Para ello, envía una petición al objeto servidor a través de la red, indicando el método invocado y sus parámetros. Una vez finalizada la invocación recibe un mensaje del objeto servidor con el resultado obtenido.

4.4.1.3 Método invocado

Para que se pueda realizar la invocación del método, este y sus parámetros se convierten en un mensaje, que se envía por la red al objeto servidor, a modo de petición.

4.4.1.4 Valor del retorno

Una vez se ha concluido la ejecución del método, el resultado de este se convierte en un mensaje que se envía al objeto cliente.

4.4.1.5 Infraestructura de comunicaciones para la invocación de métodos remotos

Para poder llevar a cabo la invocación de métodos remotos se necesita una capa de software adicional entre la aplicación de alto nivel y el nivel de transporte de la pila IP. La función de esta capa es traducir las llamadas a métodos remotos y valores de retorno de estos a mensajes que se envíen por *sockets* y, de forma simétrica, traducir los mensajes recibidos a valores de retorno y llamadas a métodos. Esa capa de software debe incluir además mecanismos para poder crear objetos remotos y localizarlos desde otros puntos de la red, de forma que se pueda operar con ellos desde las aplicaciones de alto nivel. El objetivo final de esta herramienta es que la programación de aplicaciones usando invocación de métodos remotos sea lo más parecida posible al desarrollo de aplicaciones sin comunicaciones por red.

Los componentes básicos de esta capa de software son:

- **Stubs**: los *stubs* son objetos que sustituyen a los objetos remotos dentro del programa donde se encuentra el objeto cliente. Tienen métodos similares al objeto servidor, pero no realizan las mismas funciones. En su lugar, cuando *stub* recibe la invocación de un método, lo que hace es construir un mensaje y enviarlo por la red al objeto servidor. Una vez hecho esto, espera hasta que el objeto remoto responda. Cuando recibe respuesta la devuelve, como si él mismo hubiese realizado la operación. De esta forma, el objeto cliente no tiene que realizar ninguna operación especial para invocar un método remoto. Tan solo invocar el método del objeto *stub*.
- **Registro de objetos remotos**: el registro de objetos remotos es un servicio de nivel de aplicación cuya función consiste en controlar todos los objetos remotos que existen en el sistema. Cuando un programa crea un objeto remoto, el primer paso que debe realizar es apuntarlo en el registro. De esta forma, cuando un objeto cliente necesite contactar con un objeto servidor, lo buscará en el registro. En el momento en que un objeto *stub* quiera configurarse para conectarse a un objeto remoto, el registro le dará la información necesaria.

ACTIVIDADES 4.11

- Como se ha explicado, la invocación de métodos remotos (RMI) está íntimamente relacionada con la llamada a procedimientos remotos (RPC). Busca en la Web información sobre la segunda (RPC) e identifica los elementos que intervienen en la comunicación en esta tecnología. ¿Qué diferencias observas con respecto a RMI? ¿Existe alguna correspondencia entre los componentes básicos de RMI y RPC?

4.4.1.6 Invocación de métodos remotos: proceso detallado

Para poder realizar invocación de métodos remotos se debe seguir una serie de pasos. La secuencia de pasos completa (incluyendo servidor y cliente) es:

1. (En el servidor) Arranque del registro de objetos remotos. Si no está arrancado ya, este debe ser siempre el primer paso.
2. (En el servidor) Creación del objeto servidor. El objeto se crea, como cualquier otro.
3. (En el servidor) Inscripción del objeto en el registro de objetos remotos. Esto se suele hacer indicando un nombre identificador para el objeto. Al realizar esta operación, el objeto servidor se convierte en un objeto remoto.
4. (En el cliente) Localización del objeto remoto en el registro. Para esto se debe conocer la dirección y el puerto del registro. El cliente debe conectarse a él y solicitar la información de conexión con el objeto servidor. Este proceso suele incluir directamente la creación del objeto *stub*.
5. (En el cliente) Invocación de métodos del objeto *stub*. Desde el punto de vista del programa cliente, estas son llamadas normales, pero dentro de ellas se produce la comunicación con el objeto remoto.
6. (Entre servidor y cliente) Intercambio de mensajes entre *stub* y objeto servidor.

7 (En el cliente) Obtención del valor de retorno de la invocación al método remoto.

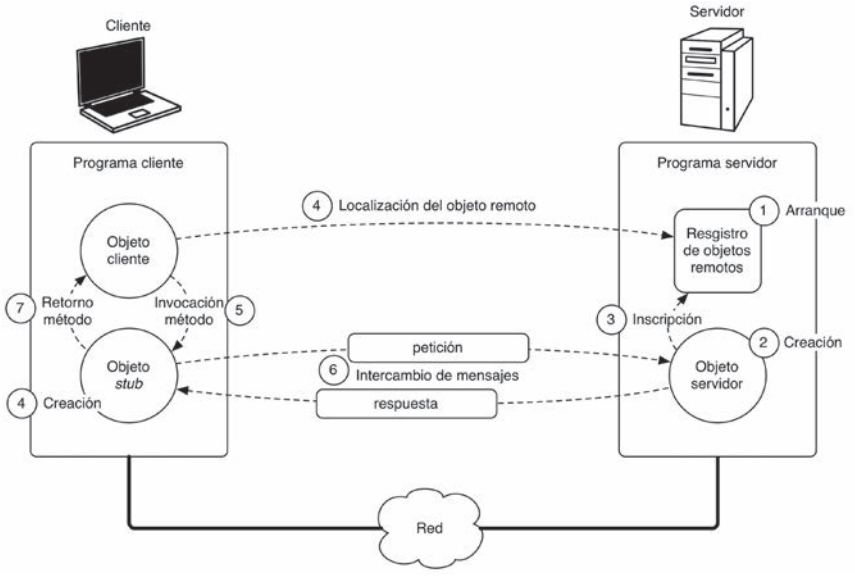


Figura 4.5. *Proceso detallado de invocación de un método remoto*

4.4.1.7 Programación de aplicaciones cliente/servidor basadas en invocación de métodos remotos

La biblioteca estándar de Java proporciona una serie de clases predefinidas que sirven para desarrollar aplicaciones distribuidas basadas en el modelo de invocación a métodos remotos (RMI). Estas clases se encuentran dentro del paquete *java.rmi*, y a continuación se verá cómo usar las más importantes.

Para programar una aplicación cliente/servidor usando Java RMI, se deben seguir los mismos pasos que para cualquier otra aplicación de este tipo:

1 **Definición de las funciones servidor**. Esto se hace de forma similar a como ya se ha visto, determinando sus características fundamentales y propiedades del servicio proporcionado.

2 **Selección de la tecnología de comunicaciones**. En este caso, esta será la invocación de métodos remotos (RMI).

3 **Definición del protocolo de nivel de aplicación**. En este caso, los mensajes que se intercambian entre cliente y servidor van englobados dentro de las llamadas a métodos remotos. Por tanto, esta tarea implica la especificación de dichos métodos. En Java RMI esto significa codificar la interfaz que deberá implementar la clase del objeto remoto. A esto se le llama la **interfaz remota**.

4 **Implementación de la clase del objeto servidor**, incluyendo la implementación de los métodos remotos.

5 **Implementación de la clase del objeto cliente**.

4.4.1.7.1 Interfaz remota

La interfaz remota es una interfaz Java que contiene los métodos que el cliente invocará de forma remota. Al ser una interfaz y no una clase, los métodos no están implementados, solo definidos. Esta interfaz debe extender a su vez de la interfaz Java *Remote* (*java.rmi.Remote*). Además, los métodos remotos definidos deben lanzar la excepción *RemoteException* (*java.rmi.RemoteException*).

EJEMPLO 4.15

Se va a retomar el Ejemplo 4.6, en el que se planteaba el desarrollo de una aplicación cliente/servidor con funciones de calculadora sencilla. En este caso, se va implementar una aplicación de las mismas características, pero usando ahora Java RMI como tecnología de comunicaciones. Como se ha visto, el primer paso es implementar la interfaz remota del objeto servidor:

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface RMICalcInterface extends Remote {

 public int suma(int a, int b) throws RemoteException;
 public int resta(int a, int b) throws RemoteException;
 public int multip(int a, int b) throws RemoteException;
 public int div(int a, int b) throws RemoteException;
}
```

ACTIVIDADES 4.12

- Copia el Ejemplo 4.15 en un editor de texto y modifícalo para que la interfaz opere con números reales en vez de enteros. Agrega además a la interfaz la operación potencia (a^b).

4.4.1.7.2 Implementación del objeto servidor

Una vez se dispone de la interfaz remota, se debe implementar la clase del objeto servidor. Este debe incluir una implementación de todos los métodos definidos en la interfaz. Además, ya sea en la misma clase o en otra, se deben implementar los mecanismos para poner en funcionamiento el registro de objetos remotos e inscribir el objeto servidor en él.

EJEMPLO 4.16

Continuando con el ejemplo de la aplicación calculadora, este es el código del servidor:

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
```

EJEMPLO 4.16 (cont.)

```
public class RMICalcServer implements RMICalcInterface {

    public int suma(int a, int b) throws RemoteException {
         System.out.println("Objeto remoto: Sumando "+a+" y "+b+"...");
         return (a + b);
    }

    public int resta(int a, int b) throws RemoteException {
         System.out.println("Objeto remoto: Restando "+a+" y "+b+"...");
         return (a - b);
    }

    public int multip(int a, int b) throws RemoteException {
         System.out.println("Objeto remoto: Multiplicando "+a+" por "+b+"...");
         return (a * b);
    }

    public int div(int a, int b) throws RemoteException {
         System.out.println("Objeto remoto: Dividiendo "+a+" entre "+b+"...");
         return (a / b);
    }

    public static void main(String[] args) {

         System.out.println("Creando el registro de objetos remotos...");

         Registry reg = null;
         try {
              reg = LocateRegistry.createRegistry(5555);
         } catch (Exception e) {
              System.out.println("ERROR: No se ha podido crear el registro");
              e.printStackTrace();
         }

         System.out.println("Creando el objeto servidor e inscribiéndolo en el registro...");
         RMICalcServer serverObject = new RMICalcServer();

         try {
              reg.rebind("Calculadora",
              (RMICalcInterface) UnicastRemoteObject.exportObject(serverObject, 0));
         } catch (Exception e) {
              System.out.println("ERROR: No se ha podido inscribir el objeto servidor.");
              e.printStackTrace();
         }
    }
}
```

El Ejemplo 4.16 muestra un código típico de un programa servidor usando Java RMI. En este ejemplo se debe prestar atención a los siguientes aspectos:

- La clase del objeto remoto implementa la interfaz remota. Esto hace que implemente todos los métodos definidos en dicha interfaz, que son la representación del protocolo de nivel de aplicación.
- El método *main* crea el registro de objetos, escuchando por el puerto 5555. Como ya se ha explicado, el registro de objetos remotos es un servicio que se usa para localizar y gestionar objetos remotos. Las clases *java.rmi.registry.Registry* y *java.rmi.registry.LocateRegistry* permiten realizar tareas como crear un registro, localizar un registro existente, etc., desde dentro del programa.
- Una vez arrancado el registro, el método *main* crea un objeto servidor y lo inscribe en este. Para ello hace uso de la clase *java.rmi.server.UnicastRemoteObject*, que sirve para representar objetos remotos y operar con ellos. El método *rebind()* de la clase *Registry* permite realizar esta inscripción. Cuando se inscribe un objeto remoto en el registro, se le debe dar un nombre único para identificarlo. En el Ejemplo 4.16 este nombre es "Calculadora".

ACTIVIDADES 4.13

- Copia el Ejemplo 4.16 en un editor de texto y modifícalo para que implemente la interfaz definida en la Actividad 4.8, incluyendo la operación con números reales y el método potencia.

4.4.1.8 Implementación del objeto cliente

Por ultimo está la clase del objeto cliente, que contiene la localización del objeto remoto y la invocación de sus métodos.

EJEMPLO 4.17

Continuando con el ejemplo de la aplicación calculadora, este es el código del cliente:

```
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

public class RMICalcClient {

    public static void main(String[] args) {

        RMICalcInterface calc = null;
        try {
            System.out.println("Localizando registro de objetos remotos...");
            Registry registry = LocateRegistry.getRegistry("localhost", 5555);
```

EJEMPLO 4.17 (cont.)

```
                System.out.println("Obteniendo el stub del objeto remoto...");
                calc = (RMICalcInterface) registry.lookup("Calculadora");

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

            if (calc != null) {
                System.out.println("Realizando operaciones con el objeto remoto...");

                try {
                    System.out.println("2 + 2 = " + calc.suma(2, 2));
                    System.out.println("99 - 45 = " + calc.resta(99, 45));
                    System.out.println("125 * 3 = " + calc.multip(125, 3));
                    System.out.println("1250 / 5 = " + calc.div(1250, 5));
                } catch (RemoteException e) {
                    e.printStackTrace();
                }
                System.out.println("Terminado");
            }
        }
    }
```

De forma análoga al caso anterior, el Ejemplo 4.17 muestra un código típico de un programa cliente usando Java RMI. En este ejemplo se debe prestar atención a los siguientes aspectos:

- El método *main* hace uso de las clases *java.rmi.registry.Registry* y *java.rmi.registry.LocateRegistry* para localizar el registro de objetos remotos. La llamada al método *lookup()* de la clase *Registry* devuelve el objeto *stub* que el cliente utilizará para invocar remotamente a los métodos del objeto servidor.
- Para localizar el objeto remoto utiliza los datos de conexión del registro (*host*/dirección IP y número de puerto) y el nombre con el que se inscribió el objeto servidor en el registro (en el Ejemplo 4.17 ese nombre es "Calculadora").
- Una vez obtenido el *stub*, el cliente invoca los métodos remotos como si se tratase de métodos en un objeto local.

ACTIVIDADES 4.14

- Extiende el Ejemplo 4.17 para que utilice el método remoto potencia definido en las actividades anteriores y opere con números reales en lugar de enteros.

4.4.1.9 Invocación de métodos remotos y otros modelos de comunicaciones

Hasta ahora se ha visto el uso de la invocación de métodos remotos dentro del marco de las aplicaciones que siguen el modelo cliente/servidor. Esto no es solo debido al hecho de que el modelo cliente/servidor sea el más usado. Debido a la forma en que está diseñada, esta tecnología de comunicaciones resulta especialmente idónea para este modelo.

No obstante, es importante recordar que la invocación de métodos remotos en el fondo no es más que intercambio de mensajes, pero abstraído dentro de la interacción típica entre los objetos de un programa. Nada impide utilizar esta tecnología para implementar modelos de comunicación distintos, como la comunicación en grupo, o enfoques avanzados o híbridos.

ACTIVIDADES 4.15

- ¿Cómo implementarías el modelo de comunicaciones en grupo usando invocación de métodos remotos? ¿Qué ventajas e inconvenientes podría haber?

4.4.2 SERVICIOS WEB

Una de las técnicas de comunicaciones avanzadas más usadas de la actualidad son los llamados *servicios web*. Un servicio web es una aplicación distribuida, basada en el modelo de comunicaciones cliente/servidor, que utiliza el protocolo de nivel de aplicación HTTP para la transferencia de mensajes. El uso de este protocolo, el que se usa en la Web como ya se ha explicado, facilita la interoperabilidad entre clientes y servidores.

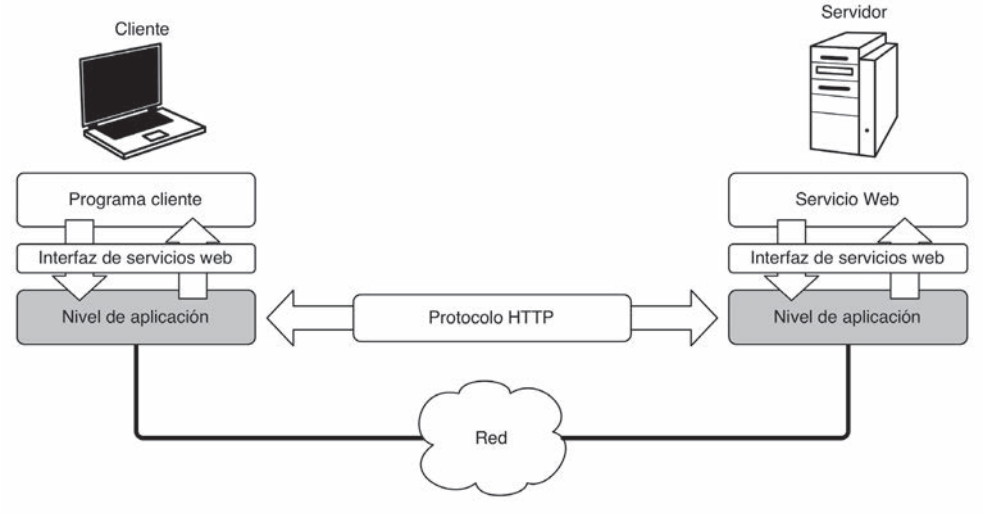


Figura 4.6. *Comunicación con servicios web*

¿SABÍAS QUE...?

Debido a la cantidad de amenazas de seguridad que existen en la red Internet, la mayoría de proveedores de acceso y empresas encargadas de gestionar el tráfico a través de la Red implementan políticas de control que restringen los protocolos de comunicaciones permitidos en determinados entornos. El protocolo HTTP es el que se utiliza para el tráfico web, y raramente se ve restringido. Los servicios web se aprovechan de esto, lo que los vuelve más accesibles y fiables.

El hecho de que los servicios web utilicen HTTP para enviar y recibir mensajes implica que los mensajes del protocolo de nivel de aplicación de un servicio web están, a su vez, encapsulados dentro de peticiones y respuestas HTTP. Dicho de otro modo, cuando un cliente de un servicio web realiza una petición a un servidor, esta se envía en el interior de un mensaje HTTP, como un GET, POST, o similar. A su vez, debemos recordar que HTTP utiliza habitualmente TCP como protocolo de transporte, por lo que estos mensajes irán también dentro de un mensaje TCP. Existen dos tipos de servicios web fundamentales, clasificados según sus características principales y los mecanismos que utilizan para representar los mensajes del protocolo de nivel de aplicación: **servicios web SOAP** y **servicios web REST**.

4.4.2.1 Servicios web SOAP

El primer tipo de servicios web que existieron fueron los llamados “servicios web SOAP”. Reciben este nombre debido al formato en que representan los mensajes, que sigue el estándar SOAP (en inglés *Simple Object Access Protocol*). En SOAP se utiliza el lenguaje XML para definir tanto el protocolo de mensajes como el contenido de estos. Esto hace que los mensajes sean fácilmente procesables de forma automática, a la vez que permite una alta flexibilidad a la hora de diseñarlos. La facilidad de procesamiento de los mensajes XML facilita además el desarrollo separado de servidores y clientes, y la fácil adaptación de las aplicaciones existentes, independientemente del lenguaje de programación en el que estuviesen desarrolladas. La descripción de interfaz de servicio de un servicio web SOAP se realiza usando un lenguaje basado en XML llamado WDSL (en inglés *Web Services Description Language*).

Un servicio web basado en SOAP debe cumplir los siguientes requisitos:

- Se debe expresar de manera formal y pública la interfaz del servicio. Esto se suele hacer usando WSDL.
- La arquitectura del servicio debe ser capaz de soportar realización y procesamiento de peticiones de forma asíncrona.
- Los servicios web basados en SOAP pueden o no tener estado (*stateful* o *stateless*).

La biblioteca de Java incluye un paquete de clases especiales para el desarrollo de servicios web basados en SOAP, denominado JAX-WS.

¿SABÍAS QUE...?

Como se ha explicado, SOAP establece que los mensajes entre cliente y servidor se codifiquen en el lenguaje XML. Esta decisión incorpora numerosas ventajas, como facilitar el procesamiento automático, la interoperabilidad, etc. No obstante, esto fuerza a que los mensajes se envíen siempre como documentos de texto (documentos XML), lo que puede suponer un mayor consumo de los recursos de red. Por esta razón, los servicios web basados en SOAP no están aconsejados para proporcionar servicios que requieran de la transferencia de una gran cantidad de información, como acceso a ficheros remotos.

4.4.2.2 Servicios web REST

Como alternativa a los servicios basados en SOAP, existen los servicios web REST. Estos servicios web no siguen el estándar SOAP, y por tanto no están forzados a utilizar XML para representar sus mensajes y su interfaz. En su lugar, utilizan REST (en inglés *Representational State Transfer*), un formato mucho más ligero y flexible, aunque sin las ventajas de interoperabilidad y facilidad en el procesamiento de XML.

ACTIVIDADES 4.16

- Busca en la Web información sobre el estándar WSRF (*Web Services Resource Framework*). ¿Qué relación tiene con los servicios web basados en SOAP?
- Busca información sobre el formato XML de los mensajes en SOAP. ¿Qué estructura tiene? ¿Resulta fácil de leer para una persona?
- Busca en la Web la descripción del lenguaje WSDL. ¿Qué elementos de un servicio web se definen usándolo?
- Busca en la Web más información sobre los servicios REST. ¿Por qué surgieron? ¿En qué situaciones se recomienda su uso frente a los servicios basados en SOAP?

4.5 CASO PRÁCTICO

Se desea programar una aplicación distribuida de buzón, siguiendo el modelo cliente/servidor. La idea consiste en que los clientes se conecten a un servidor, indicando su nombre. Una vez indicado su nombre, el cliente podrá consultar si tienen mensajes para él/ella, o dejar mensajes para otros usuarios. El servidor almacenará los mensajes entregados hasta que el usuario al que van dirigidos los consulte. Se pide:

- Especificar las funciones del servidor.
- Seleccionar la tecnología de comunicación adecuada (*sockets stream*, *sockets datagram*, llamada a métodos remotos, etc.).
- Diseñar el protocolo de nivel de aplicación.

- Implementar en Java el servidor.
- Implementar en Java un programa cliente que sirva para que diferentes usuarios consulten sus mensajes y dejen mensajes nuevos en el buzón.

Para simplificar el problema, se podrá asumir que los usuarios se identifican solo usando su nombre. No es necesario implementar gestión de contraseñas u otros mecanismos de seguridad.

RESUMEN DEL CAPÍTULO

Todo sistema consta de una estructura (los componentes que lo forman) y una función (aquello para lo que está destinado). El mecanismo específico mediante el cual un sistema realiza su función se denomina el *servicio* del sistema. Este servicio se proporciona mediante una interfaz de servicio. Los diferentes niveles de la pila IP se pueden considerar sistemas, que porporcionan servicios unos a otros. En el nivel más alto de la pila se encuentran los servicios del nivel de aplicación. El protocolo de nivel de aplicación es el protocolo que emplean las aplicaciones distribuidas.

Desarrollar una aplicación siguiendo el modelo de comunicaciones cliente/servidor requiere definir tres aspectos fundamentales: 1) la función del servidor, 2) la tecnología de comunicaciones a emplear, y 3) el protocolo de nivel de aplicación. Un servicio y su protocolo pueden ser sin estado (*stateless*), si el resultado del servicio no depende de la secuencia concreta de interacciones con el servidor, o con estado (*statefull*) en caso contrario.

Existen multitud de protocolos de nivel de aplicación considerados estándar, ya que los usan muchas aplicaciones distribuidas. Ejemplos de estos son Telnet y SSH (para establecer sesiones remotas), FTP (para transferir archivos) y HTTP, que es el protocolo de nivel de aplicación de la *World Wide Web*.

Aunque la base de las comunicaciones entre aplicaciones son los *sockets*, existen técnicas avanzadas para la programación de aplicaciones distribuidas. Todas estas se basan en los *sockets*, pero añaden una serie de mecanismos de abstracción sobre estos que hacen más fácil su desarrollo y añaden nuevas características. Uno de los ejemplos más importantes de estas técnicas son las llamadas a métodos remotos. Con ellas, la comunicación entre objetos de una aplicación distribuida se realiza como si se tratase de un invocación de métodos en local. El lenguaje Java incorpora un conjunto de herramientas para realizar este tipo de comunicación, llamadas RMI. La pieza clave de este conjunto es la interfaz *Remote*.

Por último, existen otras técnicas de comunicación aún más sofisticadas y potentes que RMI. La más importante de ellas son los servicios web, que hacen uso de protocolo HTTP para el intercambio de mensajes.

EJERCICIOS PROPUESTOS

1. Implementa una aplicación cliente/servidor que sirva documentos de texto. El servidor deberá almacenar documentos de texto, en ficheros *txt*. Cuando un cliente se conecte al servidor, especificará el número de fichero que desea obtener y el servidor le enviará su contenido, carácter a carácter. Usa para ello *sockets stream*.

2. Implementa un servicio de identificación de direcciones IP usando *sockets datagram*. Los clientes enviarán peticiones de identificación al servidor, que contestará con un mensaje que contenga la dirección IP del cliente como una cadena de texto.

3. Se pretende crear una aplicación cliente/servidor que almacene una agenda de contactos. Los clientes pueden conectarse al servidor y subir su información de contacto (nombre, dirección y teléfono), o modificarla si ya existe. Especifica las funciones del servidor, escoge la tecnología de comunicaciones adecuada y diseña (de forma conceptual) el protocolo de nivel de aplicación de este servicio.

4. Implementa un servicio de publicación de mensajes. Los clientes enviarán mensajes de texto al servidor, que los presentará por pantalla. Utiliza para ello invocación de métodos remotos.

5. Implementa un servicio de control de estado de procesos remotos mediante *heartbeat*. La función de este servicio es controlar que sus clientes están activos, y detectar cuándo se han desactivado. Para ello, los clientes deben registrarse en el servidor cuando arrancan. Una vez se han registrado, deben enviar un mensaje especial (latido del corazón o *heartbeat*) al servidor a intervalos de tiempo regulares (cada 10 segundos). Si al cabo de 20 segundos el servidor no ha recibido ningún mensaje por parte de un cliente, lo considerará muerto y avisará por pantalla. Implementar este servicio usando *sockets datagram*.

6. Repite el ejercicio anterior, introduciendo las siguientes modificaciones:

- La tecnología de comunicaciones empleada debe ser llamadas a métodos remotos.
- La duración del intervalo de tiempo del *heartbeat* se indicará como parámetro cuando el cliente se registre. De esta forma, el intervalo de un cliente puede ser de 10 s, otro de 5 s, otro de 20 s, etc.
- El tiempo que debe pasar antes de que el servidor declare muerto ha un cliente será el doble del intervalo de *heartbeat* especificado.
- El servidor debe poder detectar "resurrecciones", esto es, si un cliente que ha sido dado por muerto vuelve a enviar un mensaje de *heartbeat*, el servidor debe darse cuenta y notificarlo por su salida estándar.

7. Implementa un servicio de "operador telefónico". Un "operador telefónico" es un servicio que mantiene comunicación con un grupo de clientes, y que ayuda a que los clientes puedan comunicarse entre sí sin necesidad de conocer su dirección IP. Para ello, cuando un cliente se conecta al servidor indica su identificador personal. Después, cuando envía un mensaje, indica además el identificador personal del cliente al que va destinado. El servidor lo recoge y lo hace llegar al destinatario correcto. Implementa este servicio usando *sockets stream*.

TEST DE CONOCIMIENTOS

1 ¿Cuál de los siguientes NO es una parte fundamental de todo sistema?

a) Función.
b) Modelo de comunicaciones.
c) Estructura.
d) Ninguna de las anteriores.

2 En el contexto de la computación distribuida, ¿qué es un servicio?

a) El conjunto de procedimientos que debe llevar a cabo el cliente para acceder al servidor.
b) Los componentes hardware y software que forman el sistema.
c) El conjunto de mecanismos concretos que hacen posible el acceso a la función del sistema.
d) El protocolo de nivel de aplicación.

3 ¿Cuál de las siguientes afirmaciones es FALSA?

a) Un servicio en red es cualquier servicio que se ubique en cualquier nivel de la pila IP.
b) Un protocolo de nivel de aplicación es el conjunto de reglas que gobiernan la interacción entre los diferentes elementos de una aplicación distribuida.
c) La pila de protocolos IP es un conjunto de sistemas independientes, montados unos sobre otros para realizar una tarea compleja.
d) Todos los niveles de la pila IP ofrecen una interfaz de acceso al siguiente nivel, salvo el nivel de aplicación, por ser el último.

4 ¿Cuál de las siguientes tareas NO es un paso fundamental a la hora de programar un servidor?

a) Seleccionar la tecnología de comunicaciones adecuada.
b) Definir las funciones básicas del servidor.
c) Escoger el lenguaje de programación en el que se deberán implementar los clientes.
d) Diseñar el protocolo de nivel de aplicación.

5 ¿Cuál de las siguientes afirmaciones es FALSA?

a) Los protocolos sin estado no almacenan información sobre la evolución de la sesión, por lo que la secuencia en la que se realizan las peticiones no influye en su resultado.
b) Los protocolos con estado necesitan conocer en detalle la sesión, ya que los resultados de las peticiones dependen de cómo se haya desarrollado esta.
c) El hecho de que un protocolo tenga o no estado es independiente de la tecnología de comunicaciones que se use (*sockets stream*, *sockets datagram*, invocación de métodos remotos, etc.).
d) Ninguna de las anteriores.

6 ¿Cuál es la solución más habitual cuando se desea implementar aplicaciones cliente/servidor en las que el servidor pueda atender a muchos clientes de forma simultánea?

a) Usar *sockets datagram* y diseñar un protocolo de nivel de aplicación que incluya pocos mensajes.
b) Implementar un servidor multihilo, probablemente usando *sockets stream*.
c) Paralelizar el código del servidor usando MPI.
d) Ninguna de las anteriores.

7 ¿Cuál de las siguientes afirmaciones es FALSA?

a) El protocolo SSH ofrece comunicaciones seguras.
b) SFTP es una extensión del protocolo FTP que incorpora seguridad en las comunicaciones, gracias la a la incorporación de TLS.
c) Telnet es un protocolo con estado (*stateful*).
d) Ninguna de las anteriores.

8 ¿Cuál de las siguientes afirmaciones es FALSA sobre el protocolo HTTP?

a) Es el protocolo que se usa en la *World Wide Web*.
b) Es un protocolo sin estado (*stateless*).
c) La versión 1.1 define, entre otras, las peticiones GET, PUT y RESTART.
d) Los códigos de estado se agrupan en 5 categorías, dependiendo de su significado.

9 ¿Cuál de los siguientes pasos NO OCURRE durante la invocación de un método remoto?

a) Invocación del método del objeto *stub*.
b) Envío de la petición al objeto servidor.
c) Retorno del valor por parte del objeto *stub*.
d) Destrucción del objeto *stub*.

10 ¿Cuál de las siguientes afirmaciones sobre los servicios web es FALSA?

a) Existen dos tipos de servicios web, en función de si están basados en SOAP o en REST.
b) Basan sus comunicaciones en el protocolo HTTP.
c) Los servicios web REST representan los mensajes en lenguaje XML.
d) Un servicio web se puede expresar de manera formal usando WSDL.

5 Utilización de técnicas de programación segura

OBJETIVOS DEL CAPÍTULO

- ✓ Aprender los conceptos básicos relacionados con la seguridad de la información y la historia de la criptografía.
- ✓ Familiarizarse con los modelos criptográficos más importantes, como el modelo de clave privada y el modelo de clave pública.
- ✓ Aprender a programar usando los mecanismos de cifrado de la información más importantes.
- ✓ Conocer los algoritmos de cifrado más usados en la actualidad.
- ✓ Aprender los fundamentos de la programación de aplicaciones distribuidas que utilizan comunicaciones seguras.

5.1 CONCEPTOS BÁSICOS

Aunque la **criptografía** se define en el *Diccionario de la Real Academia de la Lengua Española* como "el arte de escribir con clave secreta o de un modo enigmático", esta definición no encaja en los tiempos actuales. Así, la criptografía puede verse más como la ciencia que trata de conservar los secretos o hasta el arte de enviar mensajes en clave secreta aplicándose a todo tipo de información, tanto escrita como digital, la cual se puede almacenar en un ordenador o enviar a través de la red.

Se denomina **encriptar** a la acción de proteger la información mediante su modificación utilizando una clave. En informática también se usan los términos codificar/descodificar y cifrar/descifrar como sinónimos de encriptar/desencriptar.

EJEMPLO 5.1

Supongamos que Pedro desea enviar un mensaje a Ana. Sin embargo, Pedro quiere enviarlo de forma privada, ya que es una carta amorosa, a través de Internet y no se fía, ya que piensa que el medio de comunicación no es seguro. En concreto, Pedro quiere evitar que el mensaje pueda ser leído por terceras personas que incluso podrían llegar a modificarlo. El objetivo de la criptografía consiste en proporcionar métodos para prevenir tales ataques.

Pedro tiene una opción sencilla, codificará el mensaje y obtendrá un mensaje cifrado el cual puede enviar tranquilamente a Ana (este mensaje cifrado es ilegible y difícilmente descifrable sin tener la clave correcta). Ana empleará un método de descodificación para conseguir obtener el mensaje original a partir del mensaje cifrado. Para ello, habitualmente necesitará alguna información secreta, por ejemplo una clave. Por supuesto, alguien podría interceptar el mensaje, pero sin la clave no podría obtener información útil de él.

5.1.1 APLICACIONES DE LA CRIPTOGRAFÍA

La criptografía es una disciplina con multitud de aplicaciones, sobre todo en el área de Internet. Entre otras se pueden destacar:

- **Identificación y autenticación**. Identificar a un individuo o validar el acceso a un servidor con más garantías que los sistemas de usuario y clave tradicionales.
- **Certificación**. Esquema mediante el cual agentes fiables validan la identidad de agentes desconocidos (como usuarios reales). El sistema de certificación es la extensión lógica del uso de la criptografía para identificar y autenticar cuando se emplea a gran escala.
- **Seguridad de las comunicaciones**. Permite establecer canales seguros para aplicaciones que operan sobre redes que no son seguras (Internet).

EJEMPLO 5.2

SSL y TLS son los protocolos más utilizados en la actualidad para proporcionar versiones seguras de protocolos de red (por ejemplo son los protocolos utilizado por HTTPS para proporcionar seguridad o un canal seguro a HTTP, como se verá posteriormente). SSL permite gestionar qué algoritmos se van a emplear e intercambiar las claves de encriptación y la autenticación de clientes y ordenadores, mientras que TLS es una evolución del anterior.

- **Comercio electrónico**. El empleo de canales seguros y mecanismos de identificación posibilita el comercio electrónico o *e-comerce* a través de Internet, ya que tanto las empresas como los usuarios tienen garantías de que las operaciones no van a ser espiadas ni modificadas, reduciéndose el riesgo de fraudes.

A lo largo del capítulo se analizará cómo conseguir cada una de ellas.

5.1.2 HISTORIA DE LA CRIPTOGRAFÍA

Aunque se puede pensar que la utilización de la criptografía es algo relativamente nuevo, se lleva utilizando desde la Antigüedad. Los egipcios y mesopotámicos ya utilizaban sistemas de transformación deliberada de la información desde antes del año 1500 a. C. utilizando símbolos no habituales con significados menos comunes para dificultar la comprensión de la misma. Los chinos, griegos y hebreos años más tarde, a partir del 700 a. C., empezaron a introducir métodos de protección de la información en el envío de sus mensajes a través de mensajeros. Así, se introdujeron métodos como la valija diplomática, donde se sellaba el contenedor de información y el sello indicaba si se había abierto, la ocultación de información en bolas de papel o de seda tragadas por el propio mensajero en China, el alfabeto hebreo invertido *Atbash*, o la escritura en tatuajes en la cabeza afeitada del mensajero en Grecia, esperando hasta el crecimiento de su pelo antes del envío al cual antes de partir se esperaba que le creciera el pelo. Como ejemplo, se puede destacar el procedimiento empleado por los espartanos en el siglo v a. C., denominado *escitalo lacedomonio* por Plutarco. Era un sistema muy sencillo y se basaba en enmascarar el significado real de un texto alterando el orden de los signos que lo forman. Para ello se escribía el mensaje sobre una tela envuelta en una vara. Una vez escrito y retirada la vara, el mensaje podía enviarse sin miedo a que fuera comprendido. Así, el mensaje solo podía leerse cuando se enrollaba sobre un bastón del mismo grosor, bastón que poseía el destinatario. Los generales espartanos lo utilizaron para enviarse mensajes en la guerra contra Atenas.

Otro sistema muy conocido fue el empleado por el mismísimo Julio César para enviar mensajes secretos a sus tropas. El sistema, denominado "método de Cesar", consiste en sumar un número determinado al número de orden de cada letra sustituyendo esa letra en el mensaje por la letra resultado que se obtiene de la operación.

EJEMPLO 5.3

En el "método Cesar", si dicho número fuera el 3, cada letra A (orden 1) del mensaje se sustituye por la letra de orden 4, la D. La B se sustituye por la E y así sucesivamente. En este sentido, un simple mensaje como HOLA quedaría codificado como LROD.

En este sistema, la clave es el número y debe ser conocido por el emisor y el receptor del mensaje. Aunque este sistema se ha utilizado hasta nuestra era, es un sistema demasiado simple, ya que para descifrarlo (conseguir conocer el mensaje), basta con probar todos los posibles números (desde 1 hasta el número máximo de letras [27 en castellano]) hasta que el mensaje decodificado tenga sentido.

¿SABÍAS QUE...?

Para dificultar el descifrado del mensaje, el ejército de Felipe II utilizó un sistema de cifrado con más de 500 símbolos y 6 tablas de códigos. Las letras, sílabas y trigramas más utilizados se sustituían por letras, números o hasta trazos especiales. De la misma forma, los nombres y palabras se sustituían por códigos numéricos y alfabéticos. Tal era el grado de confianza en el sistema que, cuando el ejército francés consiguió descifrarlo, la corte española llevó al rey de Francia ante el papa de Roma acusado de emplear magia negra.

La criptografía moderna, tal como se conoce hoy en día, se desarrolló durante la Segunda Guerra Mundial, coincidiendo con el desarrollo de las computadoras en los años 40 con el ordenador *Colossus*. Durante la Primera Guerra Mundial los ejércitos eran capaces de descubrir la mayoría de las claves utilizadas para cifrar los mensajes enemigos. Para ello bastaba con conseguir suficiente texto cifrado con una determinada clave. Empleando estos textos y mediante un análisis estadístico se podían reconocer patrones y deducir la clave. Sin embargo, durante la Segunda Guerra mundial, el ejército alemán empleó un mecanismo de cifrado mucho más complejo y difícil de descifrar. Se trataba de un cifrado rotatorio implementado sobre un dispositivo llamado "Enigma". Dicho dispositivo estaba constituido por un teclado parecido al de las máquinas de escribir. Sin embargo, sus teclas eran interruptores que activaban una luz que pasaba a través de varios rotores conectados entre sí, los cuales iban girando a cada tecla pulsada. Esto producía que mismas letras en el mensaje original tuvieran diferente resultado en el mensaje cifrado. Además, todo el mensaje obtenido variaba dependiendo de la configuración inicial de los rotores. Para hacerse una idea, el mensaje:

nczwvusxpnyminhzxmqxsfwxwlkjahshnmcoccakuqpmkcsmhkseinjusblkiosxckubhmllxcsjusrrdvkohulxwc cbgvliyxeoahxrhkkfvdrewezlxobafgyujqukgrtvukameurbveksuhhvoyhabcjwmaklfklmyfvnrizrvvrtkofdanjm olbgffleoprgtflvrhowopbekvwmuqfmpwparmfhagkxiibg

significaba:

"Señal de radio 1132/19. Contenido: Forzados a sumergirnos durante ataque, cargas de profundidad. Última localización enemiga: 8:30 h, cuadrícula AJ 9863, 220 grados, 8 millas náuticas. Siguiendo. Cae 14 milibares. NNO 4, visibilidad 10".

Para una información más detallada sobre el funcionamiento de Enigma se puede consultar la página: *http://es.wikipedia.org/wiki/Enigma_(máquina)*

¿SABÍAS QUE...?

Debido a la importancia del descifrado de Enigma, su historia ha sido contada en multitud de libros. Entre otros muchos, se pueden destacar las novelas *Criptonomicón* de Neal Stephenson y *Enigma* de Robert Harris, que cuenta como fue llevado a cabo el descifrado del código por las tropas inglesas y gran parte de sus dificultades. Con posterioridad, una adaptación del libro *Enigma* con el mismo nombre fue llevada al cine en 2001.

En definitiva, Enigma era una máquina que automatizaba considerablemente los cálculos que era necesario realizar para las operaciones de cifrado y descifrado de mensajes, convirtiéndola en prácticamente indescifrable. Los científicos aliados necesitaron obtener una de las maquinas Enigma y realizar titánicos esfuerzos para conocer el sistema de cifrado alemán. La máquina alemana se convirtió así en el talón de Aquiles del régimen nazi, un topo en el que el ejército alemán confiaba ciegamente y que, en definitiva, trabajaba para el enemigo. La posibilidad de leer los mensajes enviados entre las fuerzas alemanas es considerada por numerosos historiadores como una de las principales causas de la victoria aliada final.

¿SABÍAS QUE...?

Alan Turing fue un científico británico considerado como uno de los padres de la informática. Trabajó en descifrar los códigos alemanes provenientes de la máquina Enigma durante la Segunda Guerra Mundial, y a él se debe gran parte de la victoria aliada. Debido a sus investigaciones, en las cuales era necesario conseguir poder computacional para descifrar los códigos, diseñó uno de los primeros ordenadores, la máquina *Colossus*, supercalculadora de la época realizada con más de 1.500 tubos de vacío.

Además, Turing trató el problema de la inteligencia artificial, formalizando el funcionamiento de los ordenadores mediante la máquina de Turing y proponiendo un experimento estándar para averiguar si una máquina es inteligente. Dicho experimento se conoce como "test de Turing" y consiste en un juez situado en una habitación distinta a donde se encuentran una máquina y un ser humano. El juez debe descubrir cuál es la persona y cuál es la máquina, estando permitido mentir. Si la máquina es suficientemente inteligente, el juez no debería distinguir cuál es humano y cuál no. Sesenta años después ninguna máquina ha podido pasar dicho test.

A pesar de ser uno de los científicos más influyentes en la sociedad, fue procesado por ser homosexual, siguiendo las anticuadas leyes de esa época. Se suicidó en 1954, dos años después de ser condenado por este motivo. En 2009 el primer ministro británico, Gordon Brown, pidió públicamente disculpas en nombre del Gobierno de su país por la "lamentable forma en la que Turing había sido tratado" en los últimos años de su vida.

Tras la conclusión de la Segunda Guerra Mundial, la criptografía vivió un desarrollo teórico importante. A mediados de los años 70 del pasado siglo, la autoridad de estándares estadounidense NBS (National Bureau of Standards), ahora denominada National Institute of Standards and Technology publicó el primer diseño lógico de un cifrador que estaría llamado a ser el principal sistema criptográfico de finales de siglo: el DES (*Data Encryption Standard*). En esas mismas fechas ya se empezaba a gestar lo que sería la, hasta ahora, última revolución de la

criptografía teórica y práctica: los sistemas asimétricos. Desde entonces hasta hoy en día ha habido un crecimiento espectacular de la tecnología criptográfica, aunque, como es normal, la mayoría de estos avances se han mantenido en secreto. Muchas de las investigaciones en este campo se han tratado como secretos militares, si bien en los últimos años el interés del mundo académico e internauta por el análisis criptográfico está sacando a la luz nuevas aplicaciones y desarrollos teóricos.

Aunque se han mostrado diferentes formas de cifrar mensajes a través de la historia, es importante señalar que los algoritmos criptográficos tienden a degradarse con el tiempo. Es decir, los algoritmos caducan al igual que la fruta fresca. Todos los algoritmos criptográficos son vulnerables a los ataques de *fuerza bruta* (probar sistemáticamente con todas y cada una de las posibles claves de encriptación). A medida que pasa el tiempo, es más fácil romper los algoritmos debido al avance en la potencia de los computadores. Sin embargo, al mismo tiempo que avanza el poder de los computadores, también avanzan los nuevos métodos y técnicas criptográficas.

¿SABÍAS QUE...?

El análisis de contraseñas es un proceso altamente paralelizable. Frente a los procesadores más rápidos disponibles en el mercado han aparecido los nuevos chips de tarjetas gráficas o GPU (*Graphics Processing Unit*, en inglés, por similitud con el nombre de CPU, *Central Processing Unit*) que permiten realizar operaciones masivamente paralelas, ya que cuentan con varios múltiples núcleos (del orden de más de 100), lo que permite ejecutar un elevado número de hilos en paralelo. La utilización de esta nueva arquitectura ofrece un rendimiento una centena de veces superior al de una CPU para aquellos algoritmos criptográficos que puedan ejecutarse de forma eficiente en una GPU. Esto reduce el tiempo necesario para romper una clave de forma muy significativa.

5.2 CARACTERÍSTICAS DE LOS SERVICIOS DE SEGURIDAD

Para proteger la información, tanto para su envío como para su almacenamiento se deben utilizar métodos criptográficos. Las claves de acceso a los sistemas y la gestión de permisos que proporcionan los sistemas operativos ayudan en el proceso pero no son suficientes para proteger la información de manera adecuada. Es necesario el uso de la criptografía.

Existe una serie de características que se deben cumplir para proporcionar la seguridad necesaria para un caso en particular:

- **Confidencialidad**: se trata de asegurar que la comunicación solo pueda ser vista por los usuarios autorizados, evitando que ningún otro pueda leer el mensaje. Esto se requiere siempre que la información tenga que ser privada y esta característica suele ir acompañada de la autenticación de los usuarios que participan en la misma.

- **Integridad** de la información: se trata de asegurar que el mensaje no haya sido modificado de ningún modo por terceras personas durante su transmisión. La información recibida debe ser igual a la que fue emitida, ya que si no, por ejemplo, en acuerdos comerciales se podrían modificar productos, precios, etc.
- **Autenticación**: se trata de asegurar el origen, autoría y propiedad de la información de quien envía el mensaje. Esto se requiere siempre que sea importante conocer cuál es la fuente del mensaje.

EJEMPLO 5.4

Es fácilmente modificable el remite de una carta. Por ejemplo, Carlos podría escribir una carta a Juan indicando en el remite que la ha escrito Pedro. Si necesitamos conocer con exactitud quién envía el mensaje, se deben establecer métodos que permitan asegurar con exactitud quién lo realizó.

- **No repudio**: se trata de evitar que la persona que envía el mensaje o realiza una acción niegue haberlo hecho ante terceros. Esto es muy importante en acuerdos comerciales ya que si se cumple, el emisor, por ejemplo, no podrá decir que no envió el mensaje correspondiente pidiendo los productos indicados. Al igual que la característica de confidencialidad, necesita de la autenticación del origen de la información.

Habitualmente, el objetivo de un sistema seguro es que se cumplan todas, aunque esto queda en función de la necesidad concreta del usuario.

5.2.1 ESTRUCTURA DE UN SISTEMA SECRETO

Un sistema secreto actual se encuentra definido por dos funciones: la **función de cifrado** y la de **descifrado**. La **clave** es el parámetro que especifica una transformación concreta dentro de todas las posibles sustituciones que se podrían realizar con la función de cifrado.

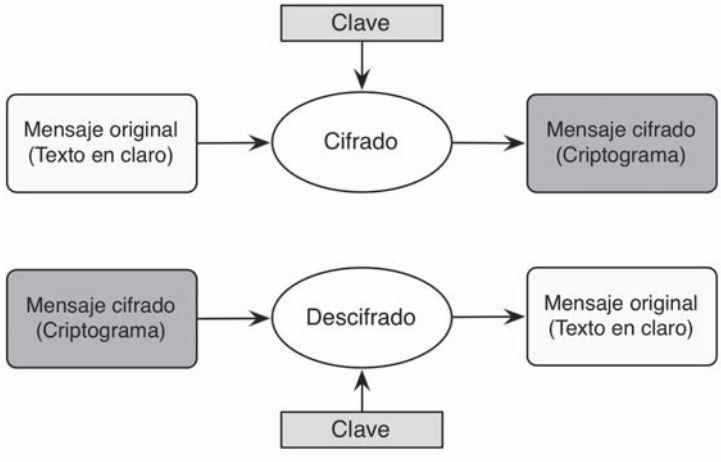


Figura 5.1. *Estructura de un sistema secreto*

La función de cifrado transforma, de forma reversible (mediante la función de descifrado), todos los posibles mensajes en claro en sus correspondientes mensajes cifrados o **criptogramas** (uno a uno) sabiendo que si se modifica la clave, el mensaje cifrado correspondiente debe ser diferente. Más aún, para dos mensajes similares, pero no iguales, la encriptación utilizando la misma clave debe dar mensajes cifrados no correlacionados, es decir, muy diferentes. Y de la misma forma, el mismo mensaje cifrado utilizando dos claves similares debe dar resultados no correlacionados. Además, debe ser imposible deducir la clave utilizada aunque se conozca la versión en claro y cifrada de cualquier mensaje.

Todo ello es necesario para dificultar la obtención de la clave mediante análisis estadístico. Shannon proponía que todos los cifrados se construyeran sobre la base de una confusión y difusión máximas. La **confusión** se refiere a distribuir las propiedades estadísticas (redundancia) de todos los elementos del mensaje (por ejemplo, alterar posición de caracteres, etc.) sobre el texto cifrado. La **difusión** se refiere a dificultar la relación entre la clave y el texto cifrado mediante complejos algoritmos de sustitución para dificultar su descifrado.

¿SABÍAS QUE...?

Shannon fue un ingeniero electrónico y matemático estadounidense, conocido por ser el padre de la teoría de la información, y al igual que Alan Turing investigó durante la Segunda Guerra Mundial, en su caso en los laboratorios Bell. Muchos de sus estudios, pensados inicialmente para todas las fuentes de información (telégrafo, teléfono, radio, etc.) han sido dirigidos hacia la criptografía y hacia el mundo de los ordenadores.

Según Shannon las características deseables que se deben cumplir en las funciones de cifrado y descifrado para proporcionar un sistema secreto serían:

1. El grado de protección determinará el trabajo y tiempo requeridos para poder vulnerar el sistema. El análisis estadístico del texto cifrado para descifrarlo debe suponer tal cantidad de trabajo que no sea rentable hacerlo por el envejecimiento de la propia información contenida en él.
2. Las claves deben ser de fácil construcción y sencillas.
3. Los sistemas secretos, una vez conocida la clave, deben ser simples pero han de destruir la estructura del mensaje en claro para dificultar su análisis.
4. Los errores de transmisión no deben originar ambigüedades.
5. La longitud del texto cifrado no debe ser mayor que la del texto en claro.

Sin embargo, dichas funciones se suponen conocidas por un atacante. Suelen ser públicas para que la gente pueda analizarlas y ver si les proporcionan la confianza suficiente para cifrar con ellas (para evitar, por ejemplo, puertas traseras que permitan descifrar mensajes al que diseñó el algoritmo). Así, la mayor parte del secreto no reside en ellas mismas sino en la clave utilizada. En general, los algoritmos criptográficos utilizan claves con un elevado número de bits (ceros y unos puestos uno detrás del otro). Normalmente se mide la calidad de un algoritmo por el esfuerzo requerido para descifrarlo. Como hemos visto anteriormente, se puede realizar un análisis estadístico de los mensajes

para ayudar a conseguir la clave pero también se puede tratar de encontrar la clave mediante fuerza bruta, es decir, probando todas y cada una de las posibles claves.

¿SABÍAS QUE...?

En febrero de 2013, una investigación interna de la agencia antidrogas norteamericana DEA afirmó que "es totalmente imposible interceptar los mensajes transmitidos a través de la aplicación iMessage entre dos dispositivos, incluso con la orden de un juez" debido al mecanismo de cifrado utilizado por la misma en los dispositivos Apple. Resulta impactante que hasta la fecha ni los Gobiernos sean capaces de descifrarlo teniendo recursos más que suficientes para probar por fuerza bruta, y más si lo comparamos con otras alternativas, como WhatsApp o Skype, las cuales son fácilmente rompibles. Esto da una idea de lo importantes que son los algoritmos de cifrado.

En este sentido, la seguridad depende tanto del algoritmo de cifrado como del nivel de secreto que se le dé a la clave. Si se pierde una clave, o es fácilmente averiguable (dependiente de los datos del poseedor de la clave: fecha de nacimiento, palabra relacionada, nombre relacionado, o hasta conjunción de todo ello) se pone en peligro todo el sistema. Saber elegir una clave y establecer métodos para cuidarla es un requisito muy importante para proporcionar un sistema seguro.

Además del nivel de seguridad de la clave, existen diferentes tipos de claves, cada una con sus ventajas e inconvenientes:

- **Claves simétricas** ($\mathbf{K_s}$): las claves de cifrado y descifrado son la misma. El problema que se plantea con su utilización es cómo transmitir la clave para que el emisor (que cifra la información) y el receptor de la información (descifra) tengan ambos la misma clave. Dan lugar a lo que se denomina **modelo de clave privada**.
- **Claves asimétricas** ($\mathbf{K_p}$ y $\mathbf{K_c}$): las claves de cifrado y descifrado son diferentes y están relacionadas entre sí de algún modo. Dan lugar al **modelo de clave pública**.

5.2.2 HERRAMIENTAS DE PROGRAMACIÓN BÁSICAS PARA EL CIFRADO

Al igual que para todo el resto de técnicas que se han visto hasta ahora, el lenguaje Java proporciona herramientas para el manejo de claves y mecanismos de cifrado. La mayoría de estas herramientas son clases e interfaces predefinidas, incluidas en los paquetes *java.security* y *javax.crypto*. Como se verá a lo largo de este capítulo, el uso de estas depende mucho de los sistemas de claves y mecanismos de cifrado empleados. No obstante, existe una serie de componentes básicos que es importante conocer desde el principio, ya que sirven como punto de partida. Son los siguientes:

- La interfaz *Key*.
- La interfaz *KeySpec*.
- La clase *Cipher*.

5.2.2.1 Interfaz Key

La interfaz *Key* (*java.security.Key*) representa una clave, que se puede usar para funciones de cifrado y descifrado. Al ser una interfaz, no implementa una funcionalidad concreta, sino que define las operaciones fundamentales que todo objeto que se use para almacenar claves debe implementar. Toda clave representada por un objeto de una clase que implemente la interfaz *Key* consta de tres partes fundamentales:

- El **algoritmo**: es el nombre de la función de cifrado y descifrado para la que está diseñada la clave. A lo largo de este capítulo se verán los ejemplos más comunes de algoritmos de este tipo.
- La **forma codificada**: es una representación de la clave. Esta representación está codificada siguiendo un formato específico, como el X.509, que se verá más adelante.
- El **formato**: es el formato en el que se encuentra la forma codificada de la clave.

Esta interfaz se utiliza para implementar clases que codifican claves de forma **opaca**. Esto significa que no proporcionan acceso a los componentes de la clave. Las claves en forma opaca se usan de forma directa por los algoritmos de cifrado y descifrado.

5.2.2.2 Interfaz KeySpec

La interfaz *KeySpec* (*java.security.spec.KeySpec*) se utiliza para representar claves en forma **transparente**. En oposición a la forma opaca, las claves en forma transparente sí ofrecen acceso a sus componentes, y se usan para poder diseminarlas (intercambiarlas entre diferentes entidades, como usuarios o aplicaciones).

5.2.2.2.1 Generadores y factorías de claves

A diferencia de la mayoría de objetos en Java, los objetos de las clases que implementan la interfaz *Key* no se suelen crear usando la operación *new*. En Java, las claves se generan usando objetos **generadores de claves** y **factorías de claves**. Los generadores de claves se utilizan para crear objetos de clases que implementan la interfaz *Key* (claves opacas). El ejemplo más típico de estas clases es *KeyGenerator* (*javax.crypto.KeyGenerator*). Las factorías de claves se utilizan para pasar de *Key* (clave opaca) a *KeySpec* (clave transparente) y viceversa. Análogamente, los generadores y factorías tampoco se crean usando *new*. Sus clases suelen disponer de un método estático llamado *getInstance()*, que sirve para crear instancias de estas clases.

EJEMPLO 5.5

Creación de una clave. Se usa el algoritmo DES, que se verá más adelante. En este ejemplo se usa además *SecretKey*, que implementa la interfaz *Key*, y la clase *SecretKeyFactory*, que está diseñada para operar con objetos *SecretKey*.

```
import java.security.spec.KeySpec;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESKeySpec;
```

EJEMPLO 5.5 (cont.)

```
public class CipherExample {

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

            System.out.println("Obteniendo generador de claves con cifrado DES");

            KeyGenerator keygen = KeyGenerator.getInstance("DES");

            System.out.println("Generando clave");

            SecretKey key = keygen.generateKey();

            System.out.println("Obteniendo factoría de claves con cifrado DES");

            SecretKeyFactory keyfac = SecretKeyFactory.getInstance("DES");

            System.out.println("Generando keyspec");

            KeySpec keyspec = keyfac.getKeySpec(key, DESKeySpec.class);

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

5.2.2.3 Clase Cipher

Los objetos de la clase *Cipher* (*javax.crypto.Cipher*) representan funciones de cifrado o descifrado. Se crean especificando el algoritmo que se desea utilizar. Posteriormente pueden ser configurados para realizar tanto operaciones de cifrado como descifrado, indicando en el proceso la clave necesaria. Sus métodos más importantes son:

Método	Tipo de retorno	Descripción
`getInstance(String transformation)`	`Static Cipher`	Método estático para crear objetos de clase *Cipher*. Recibe como parámetro el nombre del algoritmo de cifrado/descifrado que se desea emplear
`init(int opmode, Key key)`	`void`	Configura el objeto para que realice operaciones. Los modos de funcionamiento más habituales son *Cipher.ENCRIPT_MODE* para encriptar información y *Cipher.DECRIPT_MODE* para desencriptar. Recibe además como parámetro la clave que se desea utilizar
`dofinal(byte[] input)`	`byte[]`	Realiza la operación para la que ha sido configurado. Recibe como parámetro la secuencia de bytes que se desea cifrar/descifrar y devuelve el resultado de realizar la transformación correspondiente

EJEMPLO 5.6

Este ejemplo muestra el cifrado de un mensaje, usando la clase *Cipher*. Para ello, se emplea el algoritmo DES, que se verá más adelante.

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;

public class CipherExample {

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

              System.out.println("Obteniendo generador de claves con cifrado DES");

              KeyGenerator keygen = KeyGenerator.getInstance("DES");

              System.out.println("Generando clave");

              SecretKey key = keygen.generateKey();

              System.out.println("Obteniendo objeto Cipher con cifrado DES");

              Cipher desCipher = Cipher.getInstance("DES");

              System.out.println("Configurando Cipher para encriptar");

              desCipher.init(Cipher.ENCRYPT_MODE, key);

              System.out.println("Preparando mensaje");

              String mensaje = "Mensaje de prueba";

              System.out.println("Mensaje original: "+mensaje);

              System.out.println("Cifrando mensaje");

              String mensajeCifrado = new
              String(desCipher.doFinal(mensaje.getBytes()));

              System.out.println("Mensaje cifrado: "+mensajeCifrado);

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

5.3 MODELO DE CLAVE PRIVADA

La encriptación empleada habitualmente para transmitir mensajes entre ordenadores a través de la red se conoce con el nombre de "criptografía de clave secreta", "simétrica" o "privada". Tanto el emisor como el receptor del mensaje utilizan la misma clave K_s. Sin embargo, aunque este método presenta un buen rendimiento (se tarda poco en cifrar el mensaje) presenta algunos inconvenientes que son resueltos con la criptografía de clave pública, como veremos en la siguiente sección. En primer lugar está el cómo el emisor y el receptor obtienen la misma clave (sin enviarla sin cifrar por la red, por supuesto, ya que todo el mundo podría obtenerla). Y en segundo lugar está la necesidad de emplear claves distintas para comunicarse con cada uno de los posibles receptores de la información. En este sentido, un mismo emisor necesitaría tantas claves como posibles receptores tuviera, incrementándose la dificultad que supone gestionar esto. Además, la validez de una clave se pone en entredicho a medida que se va utilizando. Cuantos más mensajes se cifren con la misma clave, más expuesta estará a un análisis estadístico, por lo que es necesario cambiar las claves cada cierto tiempo.

Aun así, debido a su buen rendimiento se utiliza por un lado como apoyo a los sistemas de clave pública para asegurar confidencialidad de la información de forma rápida (mediante algoritmos de cifrado simétrico, como DES o AES, entre otros) y por otro lado para proporcionar integridad (mediante funciones *hash*).

5.3.1 ALGORITMOS DE CIFRADO SIMÉTRICO

5.3.1.1 Algoritmo DES

Es el algoritmo más extendido de clave simétrica. Basado en un sistema existente de IBM (con el nombre de Lucifer, con una clave de 128 bits), fue adoptado como estándar por el Gobierno de los Estados Unidos para comunicaciones no clasificadas en 1976, se usó hasta 1999. DES es un algoritmo que tomaba un texto de una longitud fija y lo transforma mediante una serie de complicadas operaciones en otro texto cifrado de la misma longitud. Dicho proceso sigue tres fases, como en la mayor parte de algoritmos simétricos:

- Permutación inicial: para dotar de confusión y difusión al algoritmo.
- Dieciséis (16) etapas en las que se aplica la misma función. La función es realizada mediante unas cajas de sustitución o cajas-S previamente definidas que comprimen la información, la permutan y la sustituyen. En su diseño radica la robustez del algoritmo.
- Permutación final inversa a la inicial.

  Emplea para ello una clave de 56 bits para modificar las transformaciones realizadas. La reducción de 128 bits a 56 bits, la corta longitud de la clave y las misteriosas cajas-S definidas por la propia agencia de inteligencia estadounidense NSA (National Security Agency) mediante criterios de diseño clasificados hicieron que recibiera muchas críticas, ya que se sospechaba que el algoritmo fue debilitado para que pudieran leer mensajes cifrados. Aun así, nunca pudo ser roto más que por fuerza bruta.

EJEMPLO 5.7

En 1998 se demostró que un ataque de fuerza bruta a un texto cifrado con el algoritmo DES era viable debido a la escasa longitud que emplea en su clave.

El número de 56 bits, con dos posibilidades por cada bit (0 o 1), implica 2^{56} posibles claves (72.057.594.037.927.936 posibilidades). Aunque más de 72.000 billones de posibilidades pueden parecer muchas, hoy en día un ordenador puede realizar un número enorme de operaciones por segundo, lo que hace que en un tiempo no demasiado elevado pruebe todas las claves posibles del algoritmo DES.

Por ese motivo se recomienda utilizar siempre cifrado mayor de 128 bits (por ejemplo para las conexiones Wi-Fi domésticas) para evitar que se pueda obtener la clave mediante fuerza bruta.

Debido a la rotura del algoritmo DES, IBM propuso el algoritmo 3DES en 1998 como solución. Como una clave 56 bits no era suficiente para evitar un ataque de fuerza bruta, 3DES hace un triple cifrado del DES alargando la clave hasta 192 bits sin necesidad de cambiar de algoritmo de cifrado. La mayoría de las tarjetas de crédito y otros medios de pago electrónicos utilizan el algoritmo 3DES para cifrar la información.

5.3.1.2 Algoritmo AES

El algoritmo **AES** (*Advanced Encryption Standard*), también llamado *Rijndael,* fue adoptado como estándar de cifrado por el Gobierno de los Estados Unidos en el año 2000 como sustituto del DES. Su desarrollo se llevó a cabo de forma pública y abierta y no de modo secreto como en el caso del DES. Al igual que el DES, AES es un sistema de cifrado por bloques, pero maneja longitudes de clave y de bloque variables, ambas comprendidas entre los 128 y los 256 bits.

AES es un algoritmo muy rápido y es fácil de implementar. Por ello se está utilizando a gran escala.

ACTIVIDADES 5.1

- Busca información acerca del algoritmo AES y entiende su funcionamiento interno.
- Además de DES, 3DES y AES, existen multitud de algoritmos de cifrado simétrico entre los que podemos destacar CAST, IDEA, Blowfish, etc. Elige uno de ellos y busca las diferencias existentes (número de bits en la clave, robustez, etc.) con DES y AES.

5.3.2 PROGRAMACIÓN DE CIFRADO SIMÉTRICO

La mayoría de las herramientas del lenguaje Java que se han visto hasta ahora para el uso de claves y cifrado se emplean en el modelo de cifrado simétrico.

Las más importantes son:

- La interfaz *SecretKey*, que implementa a su vez la interfaz *Key* y representa claves simétricas (opacas).
- La clase *KeyGenerator* se usa para generar claves simétricas. El método estático *getInstance()* sirve para obtener objetos de esta clase, indicando el algoritmo de cifrado como parámetro. Una vez obtenido un objeto, el método *generateKey()* crea claves de manera aleatoria. La clase de estas claves depende del algoritmo de cifrado especificado, pero todas ellas implementan la interfaz *SecretKey*.
- La clase *SecretKeyFactory* se emplea como clase de factoría de claves basadas en la interfaz *SecretKey*.
- La clase *SecretKeySpec* implementa la interfaz *KeySpec* y sirve como representación transparente de las claves simétricas.

EJEMPLO 5.8

Las siguientes líneas de código se emplean para crear una clave simétrica que se pueda usar junto con el algoritmo AES:

```
KeyGenerator keygen = KeyGenerator.getInstance("AES");
SecretKey key = keygen.generateKey();
```

En este caso, la llamada al método *generateKey()* produce un objeto de una clase que implementa la interfaz *SecretKey* y representa claves del algoritmo AES.

EJEMPLO 5.9

Otro ejemplo de cifrado y descifrado simétrico de un mensaje, en este caso el algoritmo AES. Como se puede ver, se utiliza la misma clave para ambas operaciones.

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
```

EJEMPLO 5.9 (cont.)

```
public class CipherExample2 {

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

            System.out.println("Obteniendo generador de claves con cifrado AES");

            KeyGenerator keygen = KeyGenerator.getInstance("AES");

            System.out.println("Generando clave");

            SecretKey key = keygen.generateKey();

            System.out.println("Obteniendo objeto Cipher con cifrado AES");

            Cipher aesCipher = Cipher.getInstance("AES");

            System.out.println("Configurando Cipher para encriptar");

            aesCipher.init(Cipher.ENCRYPT_MODE, key);

            System.out.println("Preparando mensaje");

            String mensaje = "Mensaje que se cifrará con AES";

            System.out.println("Mensaje original: "+mensaje);

            System.out.println("Cifrando mensaje");

            String mensajeCifrado = new
            String(aesCipher.doFinal(mensaje.getBytes()));

            System.out.println("Mensaje cifrado: "+mensajeCifrado);

            System.out.println("Configurando Cipher para desencriptar");

            aesCipher.init(Cipher.DECRYPT_MODE, key);

            System.out.println("Descifrando mensaje");

            String mensajeDescifrado = new
            String(aesCipher.doFinal(mensajeCifrado.getBytes()));

            System.out.println("Mensaje descifrado: "+mensajeDescifrado);

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

ACTIVIDADES 5.2

- Busca información acerca de la clase *SecretKeySpec*. ¿De qué métodos dispone para acceder a los componentes fundamentales de la clave?
- Modifica el ejemplo anterior para construir una representación transparente (clase *SecretKeySpec*) de la clave simétrica empleada (objeto de clase *SecretKey*). Utiliza los métodos identificados anteriormente para obtener los componentes fundamentales de la clave e imprimirlos por pantalla.

5.3.3 RESUMEN DE INFORMACIÓN (FUNCIÓN *HASH*)

Una **función *hash*** es un algoritmo matemático que resume el contenido de un mensaje en una cantidad de información fija menor. Las funciones *hash* se emplean en multitud de campos.

EJEMPLO 5.10

Las funciones hash se emplean para identificar archivos independientemente de su nombre o ubicación (necesario, por ejemplo, para el intercambio de archivos en redes P2P, como eMule, o por *torrent* mediante BitTorrent). Si no existieran funciones *hash* habría que buscar en cada posición de una base de datos, una por una. Para evitarlo, aplicamos una función *hash* al dato y lo almacenamos en la posición dada por dicha función. Cuando queremos buscar el dato, no hace falta recorrer toda la base de datos. Simplemente, volvemos a aplicar la función *hash* al dato y accedemos a la posición indicada.

Para que sea de utilidad, la función *hash* debe satisfacer varios requisitos:

1. La descripción de la función de descifrado debe ser pública. Podrían utilizar una clave simétrica K_s para realizar el resumen, aunque en la mayoría de los casos, las funciones hash utilizadas son sin clave, para evitar tener que distribuirla entre emisor y receptor.
2. El texto en claro puede tener una longitud arbitraria. Sin embargo, el resultado debe tener una longitud fija (resumen). La longitud depende del algoritmo utilizado, pero en todos los casos será muy pequeño: 64 bits, 128 bits, etc.
3. Frente a las funciones habituales de cifrado que son uno a uno, como se está generando un resumen de menor tamaño no es posible que para un mensaje de entrada haya un único mensaje cifrado de resumen. A esto se le denomina **colisión**. Sin embargo, debe ser muy difícil encontrar dos documentos (texto de entrada) cuyo valor final para la función *hash* sea el mismo o presenten una colisión.
4. **Función de un único sentido**: dado uno de estos valores cifrados, debe ser imposible producir un documento con sentido que dé lugar a ese *hash*.
5. Aun cuando se conozca un gran número de pares (texto en claro, resumen) debe ser difícil determinar la clave.
6. Rápida de calcular.

En definitiva, la función *hash* ideal ha de aproximarse a la idea de función aleatoria ideal. Existen varios algoritmos para implementar funciones *hash,* entre los que se pueden destacar MD5 y SHA-1, los cuales no requieren clave. MD5 genera resúmenes de 128 bits mientras que los de SHA-1 son de 160 bits.

Todos estos requisitos hacen que las funciones *hash* sirvan para proporcionar pruebas de la **integridad** de la transferencia de información. Debido a esto se las llama "códigos de autenticación de mensajes" o MAC (en inglés, *Message Authentication Codes*).

¿SABÍAS QUE...?

Las funciones *hash* como MD5 sirven, entre otras cosas, para comprobar si la descarga de un fichero a través de Internet ha sido correcta. Se descarga dicho archivo y el MD5 correspondiente (siempre ocupa 128 bits), que contiene el resumen del fichero origen calculado por quien colgó el fichero. Para comprobar si el fichero descargado es correcto se aplica la misma función MD5, y si el resultado es el mismo que el MD5 publicado, la descarga se ha realizado de forma correcta.

5.3.3.1 Programación de resúmenes (*HASHES*)

La biblioteca de Java incluye una clase específica para generación de resúmenes de información, llamada *MessageDigest* (*java.security.MessageDigest*). Los objetos de esta clase se usan para crear resúmenes de secuencias de bytes, usando algoritmos como MD5 y SHA-1. El modo de uso de esta clase es el siguiente:

1. *Obtención de una instancia*: de forma similar a los generadores y factorías de claves, los objetos de la clase *MessageDigest* se crean usando el método estático *getInstance()*. Al crear una instancia se debe especificar el algoritmo de resumen que se desea emplear.
2. *Introducción de datos*: una vez obtenida la instancia, se actualiza el contenido de esta con los datos de los que se desea calcular el resumen.
3. *Cómputo del resumen*: una vez actualizada la instancia con la información necesaria, se calcula el resumen usando el algoritmo seleccionado.

5.3.3.2 Clase MessageDigest

Los métodos más importantes de la clase *MessageDigest* se resumen en la siguiente tabla:

Método	Tipo de retorno	Descripción
`getInstance(String algorithm)`	`static MessageDigest`	Método estático para crear objetos de clase MessageDigest. Recibe como parámetro el nombre del algoritmo de resumen que se desea emplear
`update(byte[] input)`	`void`	Actualiza el contenido del objeto, incluyendo la información pasada como parámetro. Si este método se invoca varias veces va acumulando toda la información suministrada
`reset()`	`void`	Reinicia el objeto, eliminando toda la información introducida
`digest()`	`byte[]`	Realiza la operación de resumen sobre toda la información almacenada

EJEMPLO 5.11

Un ejemplo de uso de la clase *MessageDigest*, usando el algoritmo MD5.

```
import java.security.MessageDigest;

public class MessageDigestExample {

    public static void main(String[] args) {

         try {

              System.out.println("Obteniendo instancia");

              MessageDigest md = MessageDigest.getInstance("MD5");

              System.out.println("Actualizando contenido de la instancia");

              byte[] c1 = "Primera cadena".getBytes();
              byte[] c2 = "Segunda cadena".getBytes();
              byte[] c3 = "Tercera cadena".getBytes();

              md.update(c1);
              md.update(c2);
              md.update(c3);

              System.out.println("Calculando resumen");

              byte[] resumen = md.digest();

              System.out.println("Resumen: " + new String(resumen));

              System.out.println("Reiniciando instancia");

              md.reset();

              System.out.println("Actualizando contenido de la instancia");

              byte[] c4 = "Cuarta cadena".getBytes();

              md.update(c4);

              System.out.println("Calculando resumen");

              resumen = md.digest();

              System.out.println("Resumen: " + new String(resumen));

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

ACTIVIDADES 5.3

- Busca información acerca de la clase *MessageDigest*. ¿Qué otros algoritmos implementa?
- Modifica el ejemplo anterior para generar resúmenes usando al menos dos algoritmos distintos. Haz que el programa imprima ambos resúmenes por pantalla, y comprueba cómo estos cambian al seleccionar un algoritmo distinto.

5.4 MODELO DE CLAVE PÚBLICA

El modelo de clave pública se desarrolló en la década de 1970 para evitar los problemas derivados de la criptografía de clave simétrica, sobre todo el que tenía que ver con la distribución de las claves entre emisor y receptor. Para solucionarlo, en este sistema cada usuario tiene dos claves diferentes, las cuales están relacionas:

- **Clave privada** K_c: que solo él mismo conoce. Como suele estar formada por multitud de bits, por ejemplo 1.024 bits, no suele ser recordable. Habitualmente se almacena en un fichero protegido con contraseña, la cual se pide al usuario cuando desee firmar.
- **Clave pública** K_p: publicada para todo el mundo que lo desee.

Los sistemas de clave pública están basados en la existencia de funciones de sentido único relacionadas entre sí, lo que permite establecer la relación entre el par de claves. Las características de estas funciones son:

- Es fácil calcular la función directa $y = f(x)$
- Existe una función inversa $f^{-1}()$
- Es computacionalmente imposible obtener $f^{-1}()$ a partir de $f()$
- Si $f^{-1}(f()) = f(f^{-1}())$ se dice que conmutan.

En este sentido, el sistema de clave pública es asimétrico, frente al sistema simétrico de clave secreta, pues se emplean claves diferentes para encriptar y desencriptar la información. Si alguien desea enviar un mensaje, buscará la clave pública de aquel al que desea enviárselo, y lo cifrará con dicha clave. Esta clave, conocida por todos, no permite desencriptar el mensaje. La única forma de desencriptarlo es utilizando la clave privada del receptor, la cual él únicamente conoce asegurando la **confidencialidad** de la información. Además, esto también puede funcionar a la inversa si las claves conmutan, es decir, se puede encriptar un mensaje con la clave privada y desencriptarlo con la clave pública. Este procedimiento se denomina **firma digital**, y se estudiará en detalle en la siguiente sección. Sin embargo, este método no sirve para asegurar la confidencialidad del mensaje, aunque sirve para **autenticar** y **no repudiar** el mensaje, indicando que el que lo ha enviado es quien dice ser, ya que es el único que conoce la clave privada. De este modo, una vez producido el descifrado podemos estar seguros de que el mensaje ha sido enviado por la persona adecuada. Como ambas claves están relacionadas, para que estos sistemas sean seguros tiene que ser muy difícil (por no decir imposible) calcular una clave a partir de la otra.

Los sistemas de clave pública son computacionalmente mucho más costosos que los sistemas de clave privada. Por ello y para no poner en riesgo la clave privada cifrando con ella mucha información que podría ser posteriormente analizada, generalmente se emplea una combinación de ambos sistemas, clave privada y pública. Así, en el intercambio de información utilizando, por ejemplo, HTTPS, por rapidez se codifican los mensajes mediante algoritmos simétricos y una única clave K_s. Dicha clave se suele crear aleatoriamente por el emisor en función de la hora del sistema. Luego se utilizan sistemas de clave pública para codificar y enviar la clave simétrica K_s sin miedo a que sea vista por otro usuario, siendo el resto de mensajes intercambiados durante esa sesión cifrados con la clave K_s.

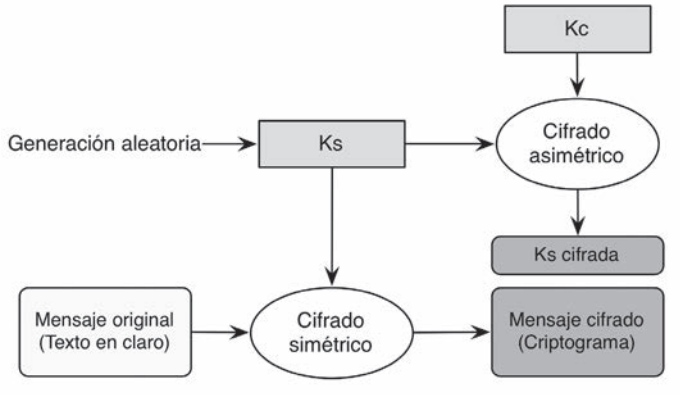


Figura 5.2. *Modelo híbrido de comunicación segura*

Los algoritmos de clave pública han demostrado su utilidad en redes de comunicación inseguras como Internet y, por ello, los nuevos DNI electrónicos se basan en esta tecnología. El más popular de los sistemas de clave pública es el RSA, incluido en la mayor parte del software para redes.

5.4.1 ALGORITMO RSA

El más popular de entre los algoritmos de clave pública, por su sencillez, es RSA. Debe su nombre a sus tres inventores: Ronald Rivest, Adi Shamir y Leonard Adleman, del MIT (en inglés, Massachusetts Technological Institute).

Se trata de un algoritmo asimétrico de cifrado por bloques, que utiliza una clave pública, conocida por todos, y otra privada, la cual es guardada en secreto por su propietario. Para conformar las claves el funcionamiento se basa en el producto de dos números primos muy grandes elegidos al azar. La pregunta que debemos responder para atacar este sistema es la siguiente: dado un número muy grande, ¿cuáles son los dos primos que se han multiplicado para obtenerlo? Esta pregunta no es fácil de responder, de hecho, representa un problema computacionalmente intratable siempre y cuando los números elegidos sean lo suficientemente grandes. Por tanto, la seguridad de este algoritmo radica en que no hay maneras rápidas conocidas de factorizar un número grande en sus factores primos. De modo que podemos asegurar que la seguridad de este sistema criptográfico y en consecuencia, de todos aquellos sistemas que lo emplean (tales como seguridad en sistemas operativos, seguridad en redes, comercio electrónico…) radica en la dificultad para determinar los factores primos de un número. Curiosamente, nadie ha conseguido probar o rebatir la seguridad del sistema RSA, por lo que se le considera uno de los algoritmos más seguros.

Para construir el texto cifrado, el proceso que realiza una identidad RSA es el siguiente:

1 Se eligen al azar y en secreto dos números primos p y q lo suficientemente grandes y se calcula:

- $n = p \cdot q$
- $\Phi(n) = (p-1) \cdot (q-1)$

2 Se eligen dos exponentes e y d tales que:

- e y $\Phi(n)$ no tengan números en común que los dividan, es decir $mcd(e, \Phi(n)) = 1$
- $e \cdot d = 1 \ mod \ \Phi(n)$, es decir, e y d son inversos multiplicativos en $\Phi(n)$

La clave pública es el par (e,n) y la clave privada (d,n) o (p,q). Para cifrar M lo único que se debe hacer es $M^e \ mod \ n = M'$, siendo el descifrado $M'^d \ mod \ n$. Como se puede observar, conmuta.

EJEMPLO 5.12

Elegiremos dos números pequeños para ver el proceso. Sin embargo, hay que tener en cuenta que RSA utiliza números primos muy grandes.

Sea $p=11$, primer número primo (privado), y $q=3$ segundo número primo (privado). Tenemos:

$n = p \cdot q = 33$
$(n) = (p-1) \cdot (q-1) = 10 \cdot 2 = 20$

Consideremos $e=3$ como exponente público, el cual $mcd(20,3) = 1$. Calculamos:

$d = e^{-1} \ (mod \ \Phi(n)) = 3^{-1} \ mod \ 20 = 7$, exponente privado.

En este sentido, tenemos:

- Clave pública K_p: $(3,33)$
- Clave privada K_c: $(7,33)$

Sea el número 7 el mensaje que deseamos cifrar. La función de cifrado con la clave pública $(3,33)$ es:

$7^3 \ mod \ 33 = 13$

Luego, lo que se envía como mensaje cifrado es 13. La función de descifrado utilizando la clave privada $(7,33)$ es:

$13^7 \ mod \ 33 = 7$

ACTIVIDADES 5.4

- Además de RSA, existen otros algoritmos de clave pública. Busca información sobre el algoritmo Diffie-Helman y descubre en qué se basa su robustez frente a RSA, el cual se basa en la diferente complejidad computacional entre encontrar números primos y factorizar números compuestos.

5.4.2 PROGRAMACIÓN DE CIFRADO ASIMÉTRICO

Al igual que en el caso del cifrado simétrico, la biblioteca de Java ofrece herramientas para trabajar con cifrado asimétrico. Estas herramientas nos permiten generar pares de claves y cifrar la información usándolas. Las clases e interfaces más importantes que se usan para estas tareas son:

- La interfaz *PublicKey* (*java.security.PublicKey*). Implementa a su vez la interfaz *Key*, y se emplea para representar claves públicas en el modelo de cifrado asimétrico.
- La interfaz *PrivateKey* (*java.security.PrivateKey*). Implementa a su vez la interfaz *Key*, y se emplea para representar claves privadas en el modelo de cifrado asimétrico.
- La clase *KeyPair* (*java.security.KeyPair*). Los objetos de esta clase se utilizan para almacenar pares de claves en el modelo de cifrado asimétrico.
- La clase *KeyPairGenerator* (*java.security.KeyPairGenerator*). Se utiliza para generar pares de claves, de forma similar a los generadores de claves simétricas.
- La clase *KeyFactory* (*java.security.KeyFactory*). Se usa como factoría de claves para claves del modelo de cifrado asimétrico.

5.4.2.1 Clase KeyPair

Los objetos de la clase *KeyPair* representan pares de claves. Sus métodos más importantes son:

Método	Tipo de retorno	Descripción
`KeyPair(PublicKey publicKey, PrivateKey privateKey)`	`KEyPair`	Constructor de la clase
`getPrivate()`	`PrivateKey`	Método para obtener la clave privada almacenada
`getPublic()`	`PublicKey`	Método para obtener la clave pública almacenada

5.4.2.2 Clase KeyPairGenerator

Los objetos de la clase *KeyPairGenerator* se utilizan para generar pares de claves. El método estático *getInstance()* se emplea para obtener una instancia del generador (indicando el algoritmo de cifrado asimétrico que se desea emplear). Una vez hecho esto, se pueden generar pares de claves usando *generateKeyPair()*.

Método	Tipo de retorno	Descripción
`getInstance(String algorithm)`	`static KeyPairGenerator`	Método estático para crear objetos de clase *KeyPairGenerator*. Recibe como parámetro el nombre del algoritmo de cifrado asimétrico que se desea emplear
`generateKeyPair()`	`KeyPair`	Genera aleatoriamente un par de claves (pública y privada) y las devuelve como un objeto de clase *KeyPair*

EJEMPLO 5.13

Programa de ejemplo que utiliza cifrado asimétrico (algoritmo RSA). Observa cómo se usa una clave del par para cifrar y la otra para descifrar.

```
import javax.crypto.Cipher;
import java.security.KeyPair;
import java.security.KeyPairGenerator;

public class AsymetricExample {

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

            System.out.println("Obteniendo generador de claves con cifrado RSA");

            KeyPairGenerator keygen = KeyPairGenerator.getInstance("RSA");

            System.out.println("Generando par de claves");

            KeyPair keypair = keygen.generateKeyPair();

            System.out.println("Obteniendo objeto Cipher con cifrado RSA");

            Cipher aesCipher = Cipher.getInstance("RSA");

            System.out.println("Configurando Cipher para encriptar
            usando la clave privada");

            aesCipher.init(Cipher.ENCRYPT_MODE, keypair.getPrivate());

            System.out.println("Preparando mensaje");

            String mensaje = "Mensaje de prueba del cifrado asimétrico";

            System.out.println("Mensaje original: "+mensaje);

            System.out.println("Cifrando mensaje");

            String mensajeCifrado = new
            String(aesCipher.doFinal(mensaje.getBytes()));

            System.out.println("Mensaje cifrado: "+mensajeCifrado);

            System.out.println("Configurando Cipher para desencriptar
            usando la clave p£blica");

            aesCipher.init(Cipher.DECRYPT_MODE, keypair.getPublic());

            System.out.println("Descifrando mensaje");
```

EJEMPLO 5.13 (cont.)

```
                String mensajeDescifrado = new
                String(aesCipher.doFinal(mensajeCifrado.getBytes()));

                System.out.println("Mensaje descifrado: "+mensajeDescifrado);

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

ACTIVIDADES 5.5

- Modifica el ejemplo anterior para cifrar el mensaje con la clave pública y descifrarlo con la privada. ¿Qué ocurre?
- Cuando se cifra un mensaje usando el modelo de clave asimétrica, en unas ocasiones se cifra con la clave pública (y descifra con la privada) y en otras se cifra con la privada (y descifra con la pública). ¿En qué ocasiones estará indicada cada una? ¿Por qué?

5.4.2.3 Clase KeyFactory

La clase *KeyFactory* es la factoría de claves por defecto para claves del modelo de cifrado asimétrico. Funciona de manera muy similar al resto de factorías de claves, permitiendo cambiar entre claves opacas y transparentes, y viceversa. Sus métodos más importantes son:

Método	Tipo de retorno	Descripción
`getInstance(String algorithm)`	`static KeyFactory`	Método estático para crear objetos de clase KeyFactory. Recibe como parámetro el nombre del algoritmo de cifrado asimétrico que se desea emplear
`generatePrivate(KeySpec keySpec)`	`PrivateKey`	Genera una clave privada opaca, a partir de su versión transparente
`generatePublic(KeySpec keySpec)`	`PublicKey`	Genera una clave pública opaca, a partir de su versión transparente
`getKeySpec(Key, key, Class<T> keySpec)`	`<T extends KeySpec>`	Genera una clave transparente, a partir de su versión opaca. Se le debe especificar como parámetro además la clase derivada de *KeySpec* que se desea usar para crear la clave transparente. Esto es necesario porque la estructura interna de las claves depende mucho del algoritmo de cifrado, por lo que sus propiedades fundamentales pueden cambiar

EJEMPLO 5.14

Programa de ejemplo que utiliza *KeyFactory* para obtener la información interna de un par de claves de cifrado asimétrico (algoritmo RSA). Observa cómo se usan las clases *RSAPublicKeySpec* y *RSAPrivateKeySpec*. Estas clases implementan la interfaz *KeySpec* y se usan para representar claves transparentes para el algoritmo RSA.

```
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.spec.RSAPrivateKeySpec;
import java.security.spec.RSAPublicKeySpec;

public class KeyFactoryExample {

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

            System.out.println("Obteniendo generador de claves con cifrado RSA");

            KeyPairGenerator keygen = KeyPairGenerator.getInstance("RSA");

            System.out.println("Generando par de claves");

            KeyPair keypair = keygen.generateKeyPair();

            System.out.println("Obteniendo la factoría de claves con cifrado RSA");

            KeyFactory keyfac = KeyFactory.getInstance("RSA");

            System.out.println("Obteniendo las especificaciones del par de claves");

            RSAPublicKeySpec publicKeySpec = keyfac.getKeySpec(keypair.getPublic(),
            RSAPublicKeySpec.class);
            RSAPrivateKeySpec privateKeySpec =
            keyfac.getKeySpec(keypair.getPrivate(), RSAPrivateKeySpec.class);

            System.out.println("CLAVE PÚBLICA");
            System.out.println("Mùdulo: "+publicKeySpec.getModulus());
            System.out.println("Exponente: "+publicKeySpec.getPublicExponent());

            System.out.println("CLAVE PRIVADA");
            System.out.println("Mùdulo: "+privateKeySpec.getModulus());
            System.out.println("Exponente: "+privateKeySpec.getPrivateExponent());

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

ACTIVIDADES 5.6

- Busca información sobre el algoritmo de cifrado asimétrico DSA. ¿Es similar a RSA?
- Modifica el ejemplo anterior para que el par de claves se genere utilizando el algoritmo DSA. Modifica además la parte que obtiene la versión transparente del par de claves y los componentes de estas que muestra.

5.4.3 FIRMA DIGITAL

La firma digital hace referencia a un método de criptografía que asocia la identidad del emisor al mensaje que se transmite a la vez que se comprueba la integridad del mensaje. Esta firma digital se consigue mediante la aplicación de una función *hash*. A continuación, se emplea un sistema de clave pública para codificar mediante la clave privada el resumen obtenido anteriormente para autenticar el mensaje. Al resumen cifrado con la clave privada se le denomina **firma digital**.

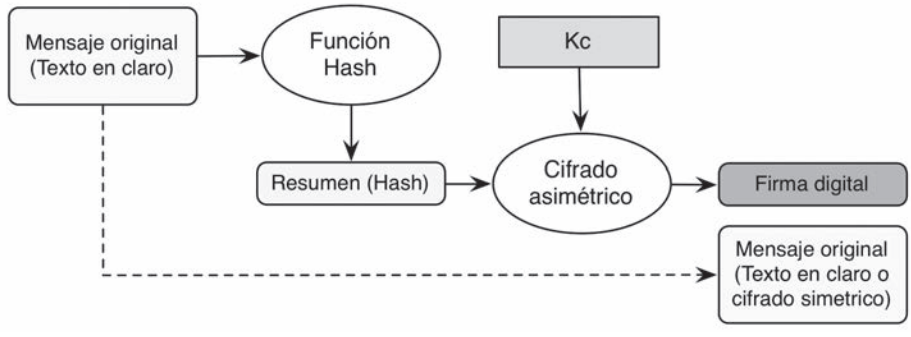


Figura 5.3. *Construcción de la firma digital*

De esta forma, cualquiera puede comprobar la firma usando la clave pública correspondiente del usuario que lo firmó. Para obtener las claves públicas correspondientes a los diferentes usuarios se utiliza el concepto de **certificado**, que se verá a continuación. Si el documento se modifica, la comprobación de la firma fallará, pero esto es precisamente lo que la verificación se supone que debe descubrir. El documento firmado, del que proviene el resumen, se puede enviar usando cualquier otro algoritmo de cifrado (cifrado simétrico por rapidez), o incluso ninguno si es un documento público.

En definitiva, la firma digital permite saber que la información no se ha modificado (en otro caso, se detecta el cambio porque el valor *hash* almacenado en el mensaje no coincide con el que se calcula al volver a hacer el resumen al descifrarlo) y quién ha enviado el mensaje (ya que está firmado con su clave privada).

5.4.3.1 Programación de firmado digital

La biblioteca de Java incluye la clase *Signagture* (*java.security.Signature*) para realizar operaciones de firmado digital. Esta clase se basa en un par de claves asimétricas previamente generadas (usando *KeyPairGenerator* u otro método análogo). Usando este par de claves y la función *hash* correspondiente, los objetos de la clase *Signature* pueden firmar secuencias de bytes, por un procedimiento similar al que se sigue para obtener resúmenes. Además, usando *Signature*, se puede verificar que una firma es válida, esto es, que los datos de partida no han sido modificados. La secuencia de operaciones concreta que se debe llevar a cabo depende de la operación que se desee realizar. Para firmar un mensaje se siguen los siguientes pasos:

1 Obtención de la clave privada. Esto se puede hacer generando un par nuevo de claves (con *KeyPairGenerator*), utilizando uno ya existente, obteniendo la clave opaca a partir de su versión transparente (*KeySpec*), etc.

2 Creación de una instancia de *Signature*. Para ello se debe conocer el algoritmo de firma digital que se desea emplear, y que fue usado para generar la clave privada (y su correspondiente clave pública asociada). Ejemplos típicos de estos algoritmos son DSA (*Digital Signature Algorithm*) o MD5withRSA (Cifrado asimétrico RSA + resumen MD5).

3 Inicialización de la instancia de *Signature* para firmar. Se le debe proporcionar la clave privada en formato opaco.

4 Inserción de datos en la instancia de *Signature*. Se actualiza el contenido del objeto con los datos que se desean firmar.

5 Generación de la firma. Esto genera un secuencia de bytes con la firma asociada a los datos introducidos.

Si se desea verificar una firma, se deben seguir los siguientes pasos:

1 Obtención de la clave pública. Esto se puede hacer generando un par nuevo de claves (con *KeyPairGenerator*), utilizando uno ya existente, obteniendo la clave opaca a partir de su versión transparente (*KeySpec*), etc.

2 Creación de una instancia de *Signature*. Para ello se debe conocer el algoritmo de firma digital que se desea emplear, y que fue usado para generar la clave privada (y su correspondiente clave pública asociada).

3 Inicialización de la instancia de *Signature* para verificar. Se le debe proporcionar la clave pública en formato opaco.

4 Inserción de datos en la instancia de *Signature*. Se actualiza el contenido del objeto con los datos que se desean firmar.

5 Verificación de la firma. Esto produce un resultado lógico (verdadero o falso) indicando si la firma proporcionada verfica la integridad de los datos introducidos.

Los métodos más importantes de la clase *Signature* son:

Método	Tipo de retorno	Descripción
`getInstance(String algorithm)`	`static Signature`	Método estático para crear objetos de clase *Signature*. Recibe como parámetro el nombre del algoritmo de firma digital que se desea emplear
`initSing(PrivateKey privateKey)`	`void`	Inicializa el objeto para la operación de firmado. Se le debe pasar como argumento la clave privada
`initVerify(PublicKey publicKey)`	`void`	Inicializa el objeto para la operación de verificación de firma. Se le debe pasar como argumento la clave pública
`update(byte[] input)`	`void`	Actualiza el contenido del objeto, incluyendo la información pasada como parámetro. Si este método se invoca varias veces va acumulando toda la información suministrada
`sign()`	`byte[]`	Firma los datos contenidos en el objeto, usando la clave privada con la que ha sido inicializado. Devuelve la firma resultante. Al finalizar este método el objeto queda reiniciado
`verify(byte[] signature)`	`boolean`	Comprueba si la firma pasada como parámetro verifica la integridad de los datos contenidos en el objeto, usando la clave pública con la que ha sido inicializado. Devuelve verdadero o falso, indicando el resultado de la verificación. Al finalizar este método el objeto queda reiniciado

EJEMPLO 5.15

Un ejemplo de uso de la clase *Signature*, usando el algoritmo DSA.

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.Signature;

public class SignatureExample {

    public static void main(String[] args) {
        try {
            System.out.println("Obteniendo generador de claves con cifrado DSA");

            KeyPairGenerator keygen = KeyPairGenerator.getInstance("DSA");

            System.out.println("Generando par de claves");

            KeyPair keypair = keygen.generateKeyPair();
```

EJEMPLO 5.15 (cont.)

```
                System.out.println("Creando objeto signature");

                Signature signature = Signature.getInstance("DSA");

                System.out.println("Firmando mensaje");

                signature.initSign(keypair.getPrivate());

                String mensaje = "Mensaje para firmar";

                signature.update(mensaje.getBytes());

                byte[] firma = signature.sign();

                System.out.println("Comprobando el mensaje firmado");

                signature.initVerify(keypair.getPublic());

                signature.update(mensaje.getBytes());

                if (signature.verify(firma))
                System.out.println("El mensaje es auténtico :-)");

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

ACTIVIDADES 5.7

- Busca más información sobre la clase *Signature* ¿Qué otros algoritmos de firma digital soporta?
- Modifica el ejemplo anterior para que use el algoritmo MD5withRSA.
- La clase *Signature* ofrece un mecanismo muy cómodo para generar y verificar firmas digitales. No obstante, los procesos de firma y verificación de datos mediante firma digital pueden realizarse sin usar esta clase. El resto de las herramientas de programación en Java vistas hasta ahora deberían ser suficientes para realizar el mismo proceso, pero de manera manual y algo más tediosa ¿Se te ocurre cómo? ¿Qué otras clases estarían involucradas?

5.4.3.2 Certificados digitales

Un **certificado** digital o certificado electrónico es un documento firmado electrónicamente por alguien en quien se confía, que confirma la identidad de una persona o servidor vinculándolo con su correspondiente clave pública K_p. En definitiva, un certificado es un documento que permite a los usuarios y servicios identificarse en Internet para realizar trámites, presentando su correspondiente clave pública, confirmada por una **autoridad de certificación** de la cual nos fiamos.

¿SABÍAS QUE...?

El DNI tradicional puede verse como un ejemplo de un certificado en el mundo real no digital. El DNI es un documento que confirma que su portador tiene un determinado nombre, nació en una fecha determinada y vive en un sitio en concreto. Si alguien necesita identificarse puede mostrar el DNI indicando que es esa persona. Sin embargo, cuando se recibe un DNI, no estamos confiando en el documento en sí, sino en la autoridad que lo emitió, en este caso en concreto el Ministerio del Interior, de la cual suponemos que se encargó de comprobar que la persona que lo porta es quien dice el documento que es.

Un certificado electrónico sirve, por tanto, para:

- Firmar digitalmente para garantizar la integridad de los datos trasmitidos y su procedencia, y autenticar la identidad del usuario de forma electrónica ante terceros. Al presentar un certificado, estamos permitiendo que se puedan comprobar las firmas digitales que se emitan con la clave privada relacionada con el certificado.
- Cifrar datos para que solo el destinatario del documento pueda acceder a su contenido. Al tener un certificado, se puede cifrar un mensaje con la clave pública que aparece en él para el dueño del certificado sea el único que lo pueda descifrar con su clave privada.

El formato específico de certificado que se suele utilizar es X.509 el cual incluye:

- Identificación del usuario o servicio. Los certificados no solamente sirven para identificar usuarios, sino también servidores como *www.urjc.es*.
- Validez de certificado: período en el cual el certificado es válido.
- Clave pública del usuario K_p
- La firma de una autoridad de certificación (CA) en la que se confíe. En concreto es la firma digital (resumen) del certificado firmado por la CA.

EJEMPLO 5.16

Certificado X.509:

- **Identificación de CA**:

```
Data:
  Version: 3 (0x2)
  Serial Number: 2 (0x2)
  Signature Algorithm: md5WithRSAEncryption
  Issuer: O=Test, OU=Test, OU=simpleCA-server.com, CN=Simple CA
```

- **Validez de certificado**:

```
Validity
Not Before: Apr 2 09:43:30 2011 GMT
Not After: Apr 2 09:43:30 2012 GMT
```

- **Identificación de usuario:**

```
Subject: O=Test, OU=Test, OU=simpleCA-server.com, OU=server.com, CN=Alberto
Sanchez Campos
```

- **Clave pública**:

```
Subject Public Key Info:
Public Key Algorithm: rsaEncryption
RSA Public Key: (1024 bit)
Modulus (1024 bit):
 00:ac:60:e6:55:3b:a3:d0:a4:7e:3a:4f:dd:04:16:
 7d:fc:b3:26:f1:44:a3:d3:06:fc:e7:ae:d1:22:85:
 7d:d3:28:fb:6e:57:3e:33:7a:9d:b6:23:d8:dd:52:
 38:93:b9:49:af:59:f4:e4:45:dd:b7:e7:c8:a5:8c:
 91:ee:3e:67:6c:c5:9c:f2:cd:5e:ff:bf:43:5d:32:
 e3:dd:e4:65:39:dc:5c:05:af:4f:7e:d2:59:8c:d3:
 28:46:25:c0:9d:84:e1:22:24:b3:f1:33:d1:88:73:
 e6:89:9e:7f:72:d2:07:8a:63:d3:30:3b:39:0f:b7:
 70:b7:47:43:84:f3:ec:b1:79
Exponent: 65537 (0x10001)
X509v3 extensions:
Netscape Cert Type:
SSL Client, SSL Server, S/MIME, Object Signing
```

- **Firma de la CA**:

```
Signature Algorithm: md5WithRSAEncryption
8a:f1:93:7f:33:8e:e6:8b:0e:93:d4:97:6d:bc:07:f0:b8:da:
a7:74:8c:63:3a:c5:ff:6b:59:38:23:df:25:63:d9:f8:07:e1: …
```

5.4.3.3 Servicios en red seguros: protocolos SSL, TLS y SSH

Cada vez que accedemos a una página cuya URL empieza por HTTPS, estamos utilizando un protocolo de criptografía para acceder a los contenidos de dichas páginas de forma confidencial. Para ello el servidor presenta su certificado, que debe ser validado por el usuario si no se ha podido comprobar que está firmado de una CA confiable por el navegador. A partir de ese certificado se obtiene la clave pública del servidor, la cual se utiliza para enviarle la clave *simétrica* K_s de sesión que será utilizada para la comunicación. En este sentido, la comunicación se realiza mediante un modelo mixto, se transmite utilizando el modelo de clave pública la clave simétrica para poder enviar y recibir mensajes con el servidor mediante un modelo de clave privada. Este es solo un ejemplo de cómo se articulan la mayoría de comunicaciones seguras a través de Internet y otras redes de comunicaciones. Tal y como se comentó anteriormente, la base de la mayoría de estas comunicaciones son los protocolos SSL (*Secure Sockets Layer*) y su sucesor TLS (*Transport Layer Security*).

SSL y TLS ofrecen una interfaz de programación basada en *sockets stream*, muy similar a la vista en el Capítulo 3. Además de las funcionalidades proporcionadas por los *sockets stream* (en la pila IP, *sockets* TCP), SSL y TLS agregan las siguientes características de seguridad:

- Uso de criptografía asimétrica para el intercambio de claves de sesión.
- Uso de criptografía simétrica para asegurar la confidencialidad de la sesión.
- Uso de códigos de autenticación de mensajes (resúmenes) para garantizar la integridad de los mensajes.

Como se puede ver, SSL y TLS incorporan la mayoría de mecanismos de seguridad vistos hasta ahora. Estos protocolos se sitúan entre el nivel de aplicación y el de transporte, mejorando de manera transparente la seguridad de los protocolos de nivel de aplicación que los usan (como FTPS o HTTPS). El funcionamiento general de los *sockets stream* SSL/TLS se basa en el establecimiento y mantenimiento de una **sesión segura**. Una sesión segura es una conexión similar a la de un *socket stream*, pero en la que se cumplen tres condiciones básicas:

- La identidad del elemento servidor está garantizada (autenticación).
- La privacidad de las comunicaciones está garantizada (confidencialidad).
- Los mensajes que se intercambian no pueden ser alterados de ninguna manera (integridad).

Para garantizar la autenticación del servidor, SSL y TLS utilizan cifrado asimétrico. El servidor dispone de un **certificado de servidor** que emplea para que los clientes puedan confiar en que se están conectando al elemento adecuado.

Para garantizar la confidencialidad de las comunicaciones, SSL y TLS utilizan cifrado simétrico. Una vez la identidad del servidor ha sido validada usando su certificado, el cliente genera una clave de cifrado simétrico de forma aleatoria y la envía el servidor, cifrada con la clave pública del certificado servidor. Esto garantiza que solo el servidor pueda recibirla y utilizarla. A esta clave simétrica se la denomina **clave de sesión**. Una vez la clave de sesión está en posesión del servidor, ambas partes pueden intercambiar mensajes de manera confidencial.

Por último, para garantizar la integridad de los mensajes se utilizan resúmenes o MAC. Junto con los mensajes se envían sus resúmenes, para que ambas partes puedan comprobar si el mensaje ha sido alterado por el camino.

¿SABÍAS QUE...?

SSL y TLS utilizan una mezcla de cifrado asimétrico y simétrico por motivos de eficiencia. El cifrado asimétrico permite garantizar la autenticación, pero resulta muy costoso computacionalmente, por lo que no resulta viable para cifrar todos los mensajes de la sesión. En su lugar, se utiliza para intercambiar de forma segura un único mensaje con la clave de sesión. Esta clave se usa para cifrado simétrico, mucho menos costoso computacionalmente que el asimétrico.

Otro de los mecanismos más comúnmente usados en Internet para la realización de comunicaciones seguras es el protocolo SSH. Tal y como se explicó en el capítulo anterior, SSH es un protocolo de nivel de aplicación similar a Telnet, cuya función es proporcionar comunicación bidireccional basada en texto plano (caracteres ASCII) entre dos elementos de una red. SSH se usa habitualmente para establecer sesiones remotas de operación por línea de mandatos entre máquinas de una misma red de forma segura. Aunque SSH es un protocolo distinto a SSL y TLS, sus características de seguridad son similares a las de estos. SSH utiliza cifrado asimétrico para garantizar autenticación de las partes (soporta los algoritmos DSA y RSA), cifrado simétrico para garantizar confidencialidad de la sesión y resúmenes para garantizar integridad. SSH comprime la información enviada para aprovechar el ancho de banda de la red.

Existe además un protocolo de nivel de aplicación para transferir ficheros de manera sencilla y segura, llamado SFTP, basada en SSH. SFTP hereda de SSH las características de autenticación, confidencialidad, integridad y compresión de la información.

¿SABÍAS QUE...?

Existen multitud de aplicaciones que permiten establecer sesiones remotas usando SSH y SFTP. En Windows, las más habituales son PuTTY (para SSH) y WinSCP (para SFTP). Ambas son libres y gratuitas, y ofrecen muchas comodidades, como gestión de claves, exploración de archivos de manera visual, etc.

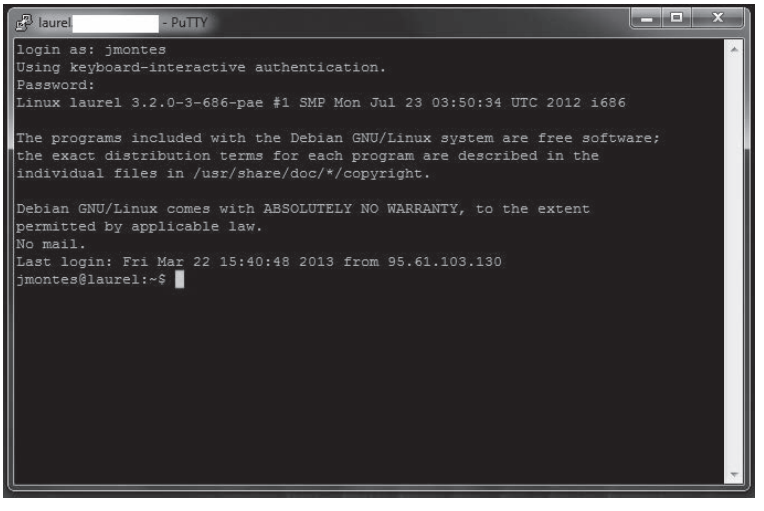


Figura 5.4. *Sesión remota por SSH usando PuTTY*

ACTIVIDADES 5.8

- Descarga PuTTY (*http://www.chiark.greenend.org.uk/~sgtatham/putty/*) y establece una sesión remota con un servidor a través de SSH.
- Descarga WinSCP (*http://winscp.net*) y establece una sesión remota con un servidor por SFTP. Descarga un fichero usando esta herramienta.

5.4.3.3.1 Programación con *Sockets* seguros

La biblioteca de Java dispone de herramientas para utilizar *sockets* basados en SSL, que incorporan las características de seguridad de dicho protocolo. Las clases más importantes relacionadas son:

- *SSLSocket* (*javax.net.ssl.SSLSocket*). Clase similar a *Socket*, pero incorporando SSL. Sirve para representar *sockets stream* cliente seguros.
- *SSLSocketFactory* (*javax.net.ssl.SSLSocketFactory*). Clase generadora de objetos *SSLSocket*.
- *SSLServerSocket* (*javax.net.ssl.SSLServerSocket*). Clase similar a *ServerSocket*, pero incorporando SSL. Sirve para representar *sockets stream* servidor seguros.
- *SSLServerSocketFactory* (*javax.net.ssl.SSLServerSocketFactory*). Clase generadora de objetos *SSLServerSocket*.

El mecanismo de uso de las clases *SSLSocket* y *SSLServerSocket* es análogo a *Socket* y *ServerSocket*. La única excepción es que estos objetos se crean usando las clases generadoras *SSLSocketFactory* y *SSLServerSocketFactory*.

EJEMPLO 5.17

Ejemplo de un programa que usa *sockets stream* servidor seguros.

```
import java.io.IOException;
import java.io.InputStream;
import java.net.InetSocketAddress;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLServerSocket;
import javax.net.ssl.SSLServerSocketFactory;

public class SecureServer {

    public static void main(String[] args) {
         try {
              System.out.println("Obteniendo factoría de sockets servidor");

              SSLServerSocketFactory serverSocketFactory =
              (SSLServerSocketFactory) SSLServerSocketFactory.getDefault();
```

EJEMPLO 5.17 (cont.)

```
                System.out.println("Creando socket servidor");

                SSLServerSocket serverSocket =
                (SSLServerSocket) serverSocketFactory.createServerSocket();

                System.out.println("Realizando el bind");

                InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
                serverSocket.bind(addr);

                System.out.println("Aceptando conexiones");

                SSLSocket newSocket = (SSLSocket) serverSocket.accept();

                System.out.println("Conexión recibida");

                InputStream is = newSocket.getInputStream();

                byte[] mensaje = new byte[25];
                is.read(mensaje);

                System.out.println("Mensaje recibido: "+new String(mensaje));

                System.out.println("Cerrando el nuevo socket");

                newSocket.close();

                System.out.println("Cerrando el socket servidor");

                serverSocket.close();

                System.out.println("Terminado");

            } catch (IOException e) {
                e.printStackTrace();
            }
        }
}
```

EJEMPLO 5.18

Ejemplo de un programa que usa *sockets stream* cliente seguros. Este programa sirve de cliente para el servidor del ejemplo anterior.

```
import java.io.IOException;
import java.io.OutputStream;
import java.net.InetSocketAddress;
import javax.net.ssl.SSLSocket;
import javax.net.ssl.SSLSocketFactory;
```

EJEMPLO 5.18 (cont.)

```
public class SecureClient {

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

              System.out.println("Obteniendo factoría de sockets cliente");

              SSLSocketFactory socketFactory =
              (SSLSocketFactory) SSLSocketFactory.getDefault();

              System.out.println("Creando socket cliente");

              SSLSocket clientSocket = (SSLSocket) socketFactory.createSocket();

              System.out.println("Estableciendo la conexión");

              InetSocketAddress addr = new InetSocketAddress("localhost", 5555);
              clientSocket.connect(addr);

              OutputStream os = clientSocket.getOutputStream();

              System.out.println("Enviando mensaje");

              String mensaje = "mensaje desde el cliente transmitido por SSL";
              os.write(mensaje.getBytes());

              System.out.println("Mensaje enviado");

              System.out.println("Cerrando el socket cliente");

              clientSocket.close();

              System.out.println("Terminado");

         } catch (IOException e) {
              e.printStackTrace();
         }
    }
}
```

Los ejemplos anteriores muestran un cliente y un servidor usando *sockets* SSL para comunicarse de manera segura. Para que esta comunicación se pueda establecer, se debe disponer de un certificado de servidor válido en el que confíe el cliente. Para facilitar la gestión de estos certificados, Java proporciona la herramienta *keytool*.

EJEMPLO 5.19

La herramienta *keytool* se puede usar para crear un certificado de servidor. Véase por ejemplo el siguiente mandato:

```
> keytool -genkey -keystore mySrvKeystore -keyalg RSA
```

Este mandato crea un certificado y lo almacena en un **fichero de almacén de claves**, o *keystore*. En este caso el fichero se llama *mySrvKeystore*. El algoritmo de cifrado con el que se ha generado el certificado es RSA.

Una vez generado el certificado, el programa servidor del ejemplo anterior se lanzaría con el siguiente mandato:

```
> java -Djavax.net.ssl.keyStore=mySrvKeystore
 -Djavax.net.ssl.trustStore=mySrvKeystore
 -Djavax.net.ssl.keyStorePassword=password
 SecureServer
```

Y el cliente con el siguiente mandato:

```
> java -Djavax.net.ssl.keyStore=mySrvKeystore
 -Djavax.net.ssl.trustStore=mySrvKeystore
 -Djavax.net.ssl.keyStorePassword=password
 Secureclient
```

En ambos casos, la opción *-Djavax.net.ssl.keyStore=mySrvKeystore* le indica a la máquina virtual de Java dónde se encuentra el fichero de almacén de claves. La opción *-Djavax.net.ssl.trustStore=mySrvKeystore* le indica a la máquina virtual de Java que los certificados contenidos en *mySrvKeystore* son de confianza. Por útlimo, la opción *-Djavax.net.ssl.keyStorePassword=password* proporciona la contraseña del almacén de claves.

ACTIVIDADES 5.9

- Fijándote en el ejemplo anterior, genera un nuevo almacén de claves que contenga un certificado que utilice cifrado DSA. Copia los ejemplos servidor y cliente que usan *sockets* SSL y ejecútalos usando el nuevo certificado.

5.4.3.4 Autoridades de certificación

Una **autoridad de certificación** (CA por sus siglas en inglés, *Certification Authority*) es una entidad de confianza, responsable de emitir y revocar los certificados electrónicos que se emplean en la criptografía de clave pública.

La firma de una autoridad de certificación (CA) asegura que:

- El certificado procede de la CA.
- Responde de que el nombre corresponde con quien dice que es.
- Afirma la relación de obligación existente entre el nombre y la clave pública.

¿SABÍAS QUE...?

En España, actualmente los certificados electrónicos son emitidos principalmente por entidades públicas que hacen de autoridad de certificación, como la Dirección General de la Policía (DGP), encargada del DNIe o DNI electrónico, o la Fábrica Nacional de Moneda y Timbre (FNMT).

El DNIe español al ser de uso obligado dispone de más de 32 millones de usuarios, lo que facilita su uso y la reducción de costes burocráticos para las Administraciones públicas y los usuarios.

Para conseguir el certificado correspondiente como persona de FNMT es necesario contar con DNI o NIE para poderse identificar en sus instalaciones.

A la hora de solicitar un certificado:

1 Utilizando ciertas funciones del propio navegador y de la página de la CA, se completan ciertos datos identificativos y se genera una pareja de claves pública/privada.

2 Con esa información se compone una petición CSR (*Certificate Signing Request*) que se envía a la CA en formato PKCS (*Public-Key Cryptography Standards*), estándar *de facto* para la utilización del modelo de clave pública.

3 La CA comprueba la información de identificación aportada, pudiendo ser requerido que vayan a comprobar su identidad a las oficinas de la propia CA en caso de ser una persona física.

4 Se firma el certificado que muestra la identificación junto a la clave pública y se envía al solicitante, para que lo instale en el navegador para su utilización. Habitualmente es necesario descargarlo en el mismo navegador sin actualizar desde donde se hizo la petición. El usuario mantiene siempre su clave privada en su poder, no habiendo sido necesario su envío a la CA.

5 Pueden revocarse certificados (por ejemplo, por pérdida del DNIe) estableciendo una *Certificate Revocation List* (CRL) que debería actualizarse en los navegadores.

Las CA disponen de sus propios certificados públicos, cuyas claves privadas asociadas son empleadas para firmar digitalmente los certificados que emiten. Y el certificado público de la CA, puede a su vez venir firmado por otra CA de rango superior de forma jerárquica. Una jerarquía de certificación consiste en una estructura jerárquica de CA en la que se parte de una CA que se autofirma, y en cada nivel existen una o más CA que pueden firmar certificados.

ACTIVIDADES 5.10

- Solicita un certificado a la FNMT y sigue todo el proceso hasta tener instalado tu propio certificado en el navegador. Intenta acceder a diferentes servicios de la Administración electrónica (Seguridad Social, recursos de multas, etc.) utilizando el certificado para ver su funcionamiento.

Para confiar en una CA hay que instalar en el navegador su certificado correspondiente en el repositorio de CA de confianza, y a partir de ese momento el propio navegador comprobará la validez de los certificados emitidos por esa CA, ya que tiene su clave pública. Será el navegador el responsable de su validación, teniendo que repetirlo por cada uno de los navegadores que existan en el sistema, si utilizamos varios: Google Chrome, Mozilla Firefox, Internet Explorer, etc. Si un certificado recibido no está validado por una CA de confianza, pedirá al usuario su conformidad con el certificado, mostrando el certificado recibido. El propio navegador puede traer por defecto CA de confianza reconocidas internacionalmente, como *VeriSign,* permitiendo que añadan más con posterioridad. Para ello se pueden localizar los certificados correspondientes en las páginas web de las CA o el propio certificado presentado por un servidor o usuario puede contener toda la cadena de certificación necesaria para ser instalado con confianza.

5.5 CONTROL DE ACCESO

A lo largo de este capítulo se ha hablado de la criptografía como el principal mecanismo de seguridad que disponen las aplicaciones. Los diferentes modelos criptográficos (clave privada y clave publica) y su tecnología derivada (certificados digitales, *sockets* seguros, etc.) permiten garantizar las características claves de los servicios de seguridad: confidencialidad, integridad, autenticación y no repudio. Estas características se refieren, en su mayor parte, a la forma en que las aplicaciones se comunican y su gestión de la información. Además, muchas veces es importante garantizar la seguridad en lo que respecta a los recursos externos a los que accede una aplicación, como ficheros leídos, dispositivos de la máquina donde se ejecuta la aplicación, etc. El conjunto de mecanismos que gestionan estos aspectos de seguridad se denominan **control del acceso**.

El control de acceso suele ser responsabilidad del sistema operativo sobre el que se ejecuta una aplicación. Usando diferentes mecanismos de control de acceso , el sistema operativo puede controlar lo que puede realizar una aplicación, evitando que ésta lleve a cabo operaciones no autorizadas.

¿SABÍAS QUE...?

El mecanismo de control de acceso más habitual en los sistemas operativos modernos es la gestión de permisos de acceso a ficheros. Este mecanismo de control de acceso permite al sistema operativo determinar si una aplicación tiene permiso para realizar operaciones sobre ficheros como leer, escribir, crear un fichero nuevo, etc. La manera en la que se gestionan estos permisos suele depender, entre otras cosas, del sistema operativo. Por ejemplo, el control de acceso a ficheros en Windows 7 y GNU Linux es bastante diferente.

La función principal de los mecanismos de control de acceso es asegurar que las aplicaciones que ejecutan en una máquina no produzcan violaciones de seguridad que afecten a su integridad o sus usuarios. Ejemplos típicos de aspectos que se gestionan usando control de acceso son:

- Operaciones sobre ficheros y directorios, como lectura, escritura, creación, ejecución, etc.
- Acceso a dispositivos hardware, como periféricos de video, audio, geolocalización (GPS), etc.
- Servicios de sistema, como comunicaciones en red, almacenamiento de datos, etc.
- Operaciones de administración del sistema, como creación de usuarios, instalación de software, etc.

En la mayor parte de sistemas el control de acceso se basa en la autenticación de usuarios y sus políticas de seguridad asociadas. La **autenticación de usuarios** emplea mecanismos criptográficos para identificar a los usuarios que hacen uso del sistema. Un usuario suele identificarse en el sistema usando un nombre y una contraseña, que lo identifican de forma únivoca. Cuando un usuario ejecuta una aplicación, ésta va asociada a él, a efectos de control de acceso. Una **política de seguridad** es un conjunto de reglas que definen qué puede hacer cada usuario del sistema, en términos de control de acceso.

¿SABÍAS QUE...?

En la mayoría de sistemas operativos de la familia UNIX y derivados, como GNU Linux o Mac OS X, la autenticación de usuarios se basa en el cálculo de resúmenes, *hashes* o *checksums*. Cuando se crea una cuenta de usuario, éste proporciona una contraseña. Guardar esa contraseña en el disco duro de la máquina puede ser peligroso, ya que si se produce un acceso no autorizado podría acceder a la misma. En su lugar, se almacena un *checksum* o *hash* de la contraseña, calculado usando un algoritmo estándar como MD5. Como las funciones *hash* de estos algoritmos no son invertibles, la contraseña no se verá comprometida aunque un usuario no autorizado lea el resumen. Cuando un usuario desea acceder al sistema, introducirá de nuevo su contraseña, que será cifrada usando el mismo algoritmo y comparada con el *checksum* almacenado, Si ambos coinciden, el usuario se habrá autenticado correctamente.

¿SABÍAS QUE...?

Android, el sistema operativo para móviles de Google, ofrece de serie mecanismos de control de acceso para gestionar multitud de operaciones, como:

- Acceso a geolocalización (usando GPS o red móvil).
- Acceso al estado de las conexiones de datos (3G, Wi-Fi, etc.).
- Acceso a Internet.
- Acceso al buzón de voz.
- Acceso al dispositivo de *bluetooth*.
- Lectura y envío de SMS.
- Realización de llamadas telefónicas.
- Escritura en la tarjeta de memoria.
- Borrado de archivos y aplicaciones.
- Activación y desactivación del flash/linterna.
- Acceso a la cámara (frontal y/o trasera).

Y muchas otras. Como es lógico, estas opciones estarán o no disponibles dependiendo de las características del aparato donde esté instalado el sistema operativo (si, por ejemplo, el móvil o tablet no tiene cámara, las opciones de cámara no estarán disponibles). Además el sistema permite gestionar de manera independiente este control de acceso para cada aplicación instalada.

Como se ha comentado, la mayoría de mecanismos de control de acceso los proporciona el sistema operativo. No obstante, existen mecanismos de programación avanzados que permiten definir políticas de seguridad de manera interna, y gestionar qué operaciones puede realizar una aplicación y cuales no. Estas políticas estarán supeditadas, obviamente, a los propios mecanismos de control de acceso del sistema operativo. En Java existe una clase estandar, denominada *SecurityManager*, que permite definir y gestionar dichas políticas.

ACTIVIDADES 5.11

- Busca en la documentación oficial de Java información sobre la clase *SecurityManager*. ¿Qué tipos de mecanismos de control de acceso permite implementar?

5.6 CASO PRÁCTICO

Se desea programar un servicio en red de descarga de información con transferencia segura. El servicio estará formado por dos aplicaciones:

- Un servidor seguro, que almacenará archivos de texto.
- Un cliente seguro, que podrá descargar archivos de texto.

El servidor almacenará una serie de archivos, identificados por su nombre. Cuando el cliente se conecte, indicará el nombre del archivo que desea descargar y, si existe, el servidor se lo enviará. Los requisitos de seguridad son los siguientes:

- La transferencia de la información se realizará mediante cifrado simétrico. La clave de sesión la generará el cliente.
- Previamente a la transferencia del fichero, se deberá realizar el intercambio de la clave de sesión. Este intercambio se realizará mediante cifrado asimétrico, usando un par de claves generado en el servidor.
- La integridad de la transferencia se verificará usando firma digital. Los mensajes deberán ir firmados por el servidor.

No se debe implementar el sistema usando *sockets* SSL. Todo el proceso de cifrado asimétrico, simétrico y de firma digital debe ser implementado explícitamente usando las bibliotecas de Java vistas a lo largo de este capítulo. Se puede escoger cualquier otra tecnología de comunicaciones vista (*sockets stream*, *sockets datagram*, RMI, etc.). Cualquier otro aspecto de diseño y programación queda a la libre decisión del desarrollador.

RESUMEN DEL CAPÍTULO

El capítulo ha mostrado la utilización de técnicas de programación segura mediante la criptografía. La utilización de la criptografía permite construir comunicaciones seguras obteniendo unas características de seguridad en la comunicación, como confidencialidad en los mensajes intercambiados, autenticación del emisor de los mensajes, integridad en el envío de datos y no repudio. Para su consecución, se ha definido el concepto de encriptación o cifrado como el mecanismo de protección de la información mediante su modificación utilizando una clave. En función del tipo de clave utilizado, simétrica o asimétrica, se establecen dos modelos diferentes de seguridad: de clave privada o de clave pública. El lenguaje Java ofrece una serie de clases básica para la gestión de claves y cifrado. Las clases *Key* y *Cipher* son la base de estos mecanismos. La primera sirve para representar claves, y la segunda para ejecutar algoritmos de cifrado.

En el modelo de clave privada, las claves de cifrado y descifrado son la misma. Sin embargo, se plantea el problema de cómo transmitirla para que el emisor y el receptor tengan ambos la misma clave y la necesidad de emplear claves distintas para comunicarse con cada uno de los posibles receptores de la información. Para ver su funcionamiento, se han analizado en detalle los algoritmos de clave simétrica DES y AES. De la misma forma, se ha explicado el concepto de función *hash*, como el mecanismo que resume el contenido de un mensaje en una cantidad de datos fija menor sin clave o utilizando para ello una clave simétrica. La biblioteca del lenguaje Java proporciona mecanismos para realizar procesos de generación de claves (generadores y factorías de claves) y generación de resúmenes (clase *MessageDigest*).

En el modelo de clave pública, las claves de cifrado y descifrado son diferentes (asimétricas) y están relacionadas entre sí de alguna forma. Así, por un lado tenemos la clave privada, que solo el usuario conoce, y la clave pública, publicada para todo el mundo mediante un certificado. En este sentido, un certificado es un documento firmado electrónicamente por alguien en quien se confía, denominado "autoridad de certificación", que confirma la identidad del usuario vinculándolo con su correspondiente clave pública. Esto permite que se cifre la información con una de las claves siendo descifrable unicamente con la otra. Si lo que se cifra es un resumen del mensaje con la clave privada del usuario, se denomina "firma digital", ya que permite asegurar la integridad, autenticación y no repudio en el envío del mismo. Para mostrar su funcionamiento, se ha analizado el algoritmo RSA en detalle y se han proporcionado herramientas Java para la creación y gestión de pares de claves para cifrado asimétrico (clase *KeyPair* y *KeyPairGenerator*). Los procesos de firma digital pueden igualmente realizarse en Java de forma sencilla, gracias a la clase *Signature*.

Los sistemas de clave pública son computacionalmente mucho más costosos que los sistemas de clave privada. Por ello habitualmente se suele emplear una combinación mixta de ambos sistemas. Así en los protocolos seguros de comunicación SSL, TLS y SSH se codifican los mensajes a enviar mediante una clave simétrica creada aleatoriamente por el emisor, la cual se envía al receptor mediante sistemas de clave pública. En Java existe un conjunto de clases predefinidas con las que se pueden establecer comunicaciones

usando *sockets stream* que incorporan seguridad mediante el protocolo SSL (sobre TCP). Para este propósito se usan las clases *SSLSocket*, *SSLServerSocket* y *SSLSocketFactory*. También es necesario gestionar los certificados que se usan durante la sesión SSL, para lo que se emplea la herramienta *keytool*.

La seguridad basada en criptografía se complementa con los mecanismos de control de acceso que proporcionan los sistemas operativos. Estos mecanismos permiten controlar aspectos de la interacción de las aplicaciones con el sistema, como el acceso a ficheros, dispositivos hardware, etc.

EJERCICIOS PROPUESTOS

1. Cifra el mensaje "HOLA, QUÉ TAL" con un desplazamiento de 6 caracteres, ¿cuál es texto cifrado? ¿Y si lo desplazamos 27? ¿Cómo podríamos atacar un sistema de cifrado tipo César?

2. Escribe un programa que lea un fichero de texto, lo cifre usando cifrado DES y lo escriba en un nuevo fichero. Para ello el programa debe generar una clave de usando el objeto generador correspondiente. Una vez el proceso de cifrado haya concluido, el programa debe acceder a la forma transparente de la clave y guardar sus componentes fundamentales en un archivo, de forma que la clave no se pierda y el fichero cifrado pueda ser descifrado en algún momento.

3. Escribe un programa que reciba como parámetro el nombre de un algoritmo de resumen, codifique usando dicho algoritmo todo lo que entre por su entrada estándar y devuelva el resultado por su salida estándar. El programa debe funcionar para cualquier algoritmo de resumen soportado por la clase *MessageDigest*.

4. Escribe un programa que utilice cifrado asimétrico para cifrar el contenido de un fichero, usando la clave privada. El resultado del cifrado y la clave pública se deben volcar en dos ficheros adicionales de salida. Posteriormente, escribe un segundo programa que cargue dichos ficheros y descifre el archivo cifrado. Utiliza cifrado RSA.

5. Escribe un par de programas similares a los del ejercicio anterior, pero que operen con firmas digitales usando el algoritmo DSA. El primer programa generará el par de claves, firmará el archivo de datos y generará dos ficheros de salida con la firma y la clave pública. El segundo programa cargará el fichero de datos, la firma y la clave pública y comprobará que el archivo no ha sido alterado. Verifica el correcto funcionamiento de los programas modificando el contenido del archivo de datos y comprobando si la firma sigue siendo válida.

6. Escribe una aplicación cliente/servidor que se comunique mediante *sockets* SSL. El servidor debe recibir mensajes de texto por parte de los clientes, pasarlos a mayúsculas e imprimirlos por pantalla.

TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes no es una característica de seguridad?

a) Confidencialidad.
b) Control de acceso.
c) Integridad.
d) No repudio.

2 Las funciones *hash*:

a) Deben aproximarse a la idea de aleatoriedad máxima posible o ideal.
b) Son de sentido único, es decir, se puede producir un documento con sentido que dé lugar a un *hash* especificado previamente.
c) Requieren bastante tiempo para producir el cifrado.
d) Tanto el texto como el resultado obtenido tienen una longitud fija.

3 ¿Cuál de los siguientes NO es un componente fundamental de una clave?

a) Forma codificada.
b) Parte pública.
c) Algoritmo.
d) Formato.

4 Indica cuál de las siguientes afirmaciones sobre el modelo de clave privada es FALSA:

a) Las claves de cifrado y descifrado son la misma.
b) Para su utilización se requieren claves simétricas.
c) El problema que se plantea con su utilización es cómo transmitir la clave para que el emisor y el receptor tengan ambos la misma clave.
d) Se tarda bastante más tiempo en cifrar el mensaje si lo comparamos con la criptografía de clave pública.

5 En el modelo de clave pública:

a) Las claves de cifrado y descifrado son la misma.
b) Las claves de cifrado y descifrado son diferentes y están relacionadas entre sí de algún modo.
a) Las claves de cifrado y descifrado son diferentes y no existe relación entre ellas para que un atacante no pueda descubrir una a partir de la otra.
a) Se tarda bastante menos tiempo en cifrar el mensaje si lo comparamos con la criptografía de clave privada.

6 ¿Para qué tipos de cifrado se usa la clase *Cipher* de Java?

a) Cifrado simétrico y asimétrico.
b) Solo cifrado asimétrico.
c) Solo cifrado simétrico.
d) Ninguna de las anteriores.

7 ¿Qué clase de la biblioteca de Java nos sirve para obtener la versión transparente de un par de claves asimétricas?

a) *SecretKeyFactory*.
b) *KeyGenerator*.
c) *KeyFactory*.
d) Ninguna de las anteriores.

8 Para la utilización de la firma digital:

a) Dicha firma se obtiene a partir del resumen del mensaje realizado mediante una función *hash* cifrado con la clave pública del usuario.
b) La clave pública del usuario se obtiene a partir del certificado.
c) El documento firmado, del que proviene el resumen, siempre se debe enviar cifrado mediante un algoritmo de cifrado simétrico.
d) El navegador siempre debe conocer la CA que firmó el certificado.

9 ¿Qué clase o clases Java se utilizan para las operaciones de firmado y verificación basadas en firma digital?

a) *KeyGenerator* para generar las claves y *Signature* para la firma y verificación.

b) *KeyGenerator* para generar las claves, *Signature* para firmar y *Verification* para verificar.

c) *Signature* para la generación de claves, firma y verificación.

d) Ninguna de las anteriores.

10 Indica cuál de las siguientes afirmaciones sobre las autoridades de certificación es FALSA:

a) Una CA es una entidad de confianza en la cual el usuario confía.

b) Las CA disponen de sus propios certificados, cuyas claves privadas asociadas son empleadas para firmar digitalmente los certificados que emiten.

c) El certificado de una CA, puede a su vez venir firmado por otra CA de rango superior de forma jerárquica.

d) Para confiar en una CA hay que instalar en el navegador su certificado correspondiente en el repositorio de CA de confianza.

Índice Alfabético

La presente obra está dirigida a los estudiantes del Módulo Profesional **Programación de Servicios y Procesos** de Grado Superior, en concreto para el Ciclo Formativo **Programación de Aplicaciones Multiplataforma**.

Los contenidos incluidos en este libro abarcan los conceptos básicos del funcionamiento de los sistemas para aprovechar los nuevos avances en los ordenadores, tanto en lo relativo a la ejecución de diferentes programas y tareas como en la comunicación a través de la Red, todo ello utilizando diferentes métodos y servicios a la vez que se asegura el nivel de seguridad necesario.

En este sentido, se muestran los conceptos de concurrencia y paralelismo. También se enseña a familiarizarse con la programación de múltiples procesos e hilos, entendiendo sus principios y formas de aplicación para obtener un mejor rendimiento del sistema para poder crear aplicaciones que optimicen los tiempos de respuesta para el usuario. De la misma forma, se explica el concepto de computación distribuida y cómo se puede aprovechar la comunicación entre aplicaciones y ordenadores a través de la Red para programar soluciones a problemas complejos. Además, debido al entorno de conexión, se explican los conceptos de seguridad de la información y se enseña a habituarse a la programación de aplicaciones que realicen una comunicación segura.

Los capítulos incluyen actividades y ejemplos con el propósito de facilitar la asimilación de los contenidos tratados. Así mismo, se incorporan test de conocimientos y ejercicios propuestos con la finalidad de comprobar que los objetivos de cada capítulo se han asimilado correctamente. Además, reúne los recursos necesarios para incrementar la didáctica del libro, tales como un glosario con los términos informáticos necesarios, bibliografía y documentos para ampliación de los conocimientos.

En la página web de **Ra-Ma** (**www.ra-ma.es**) se encuentra disponible el material de apoyo y complementario.

9 788499 642406

ra-ma.es

Ra-Ma®