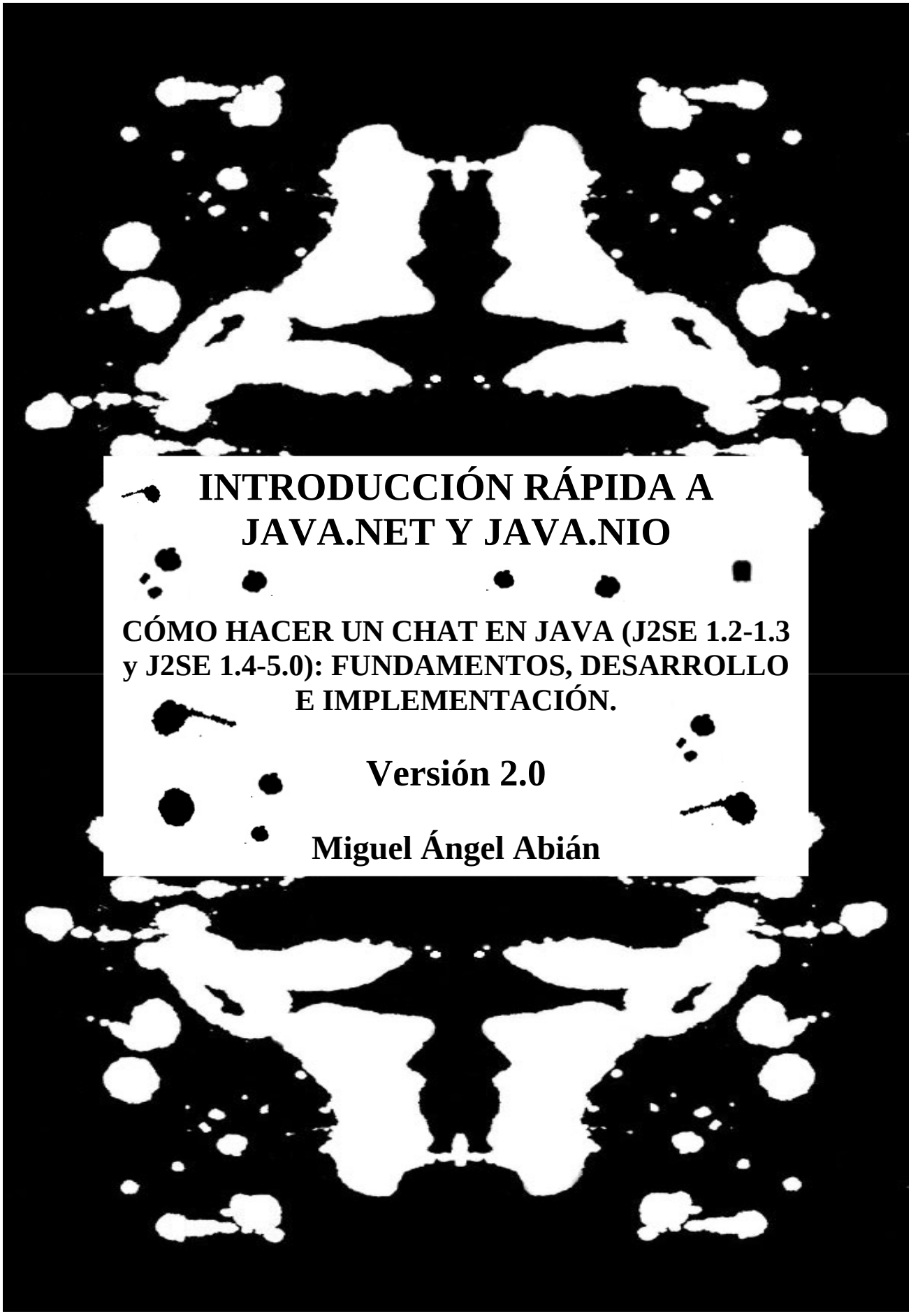

INTRODUCCIÓN RÁPIDA A JAVA.NET Y JAVA.NIO

CÓMO HACER UN CHAT EN JAVA (J2SE 1.2-1.3 y J2SE 1.4-5.0): FUNDAMENTOS, DESARROLLO E IMPLEMENTACIÓN.

Versión 2.0

Miguel Ángel Abián

ÍNDICE

INTRODUCCIÓN RÁPIDA A java.net Y java.nio

CÓMO HACER UN CHAT EN JAVA (J2SE 1.2-1.3 y J2SE 1.4-5.0): FUNDAMENTOS, DESARROLLO E IMPLEMENTACIÓN

Fecha de creación: **20.09.2002**
Revisión **2.0** (**29.11.2004**)

Miguel Ángel Abián
mabian ARROBA aidima PUNTO es

No cabe duda de que la única razón por la que Java es popular es porque está diseñado y pensado para desarrollar programas que se ejecutan en el navegador.
A. Nicolau. A survey of distributed languages (1996)

No se puede conducir un automóvil mirando el retrovisor.
Mire hacia atrás con ira. No hay nada bueno que mirar.
M. McLuhan (1970)

Estoy ya de muerte y llamo a mis amigos. Y dicen: Bueno va a vivir cinco horas.
Y con plena conciencia de mí mismo llamo a un sacerdote... Confieso todo en voz alta. Llamo a mis amigos ateos: comunistas y tal. Están allí presentes.

–Me acuso de tal, creo en ti, Dios mío, tomen ejemplo de mi muerte. Ustedes han compartido conmigo creencias nefastas. Miren cómo muero.

Y muero y voy al infierno, porque es todo una broma que les hago a mis amigos.
Luis Buñuel Portolés (1980)

1. INTRODUCCIÓN

El propósito de este trabajo de divulgación es ofrecer al lector o la lectora una introducción rápida a los paquetes **java.net** y **java.nio**, que forman parte de la J2SE (*Java 2 Standard Edition*). Mediante una aplicación de charla electrónica (*chat*) se ilustrarán los conceptos básicos de redes y las clases más relevantes de estos paquetes.

El paquete **java.net**, incluido en la J2SE desde las primeras versiones de Java, ofrece al programador un conjunto de clases e interfaces para establecer comunicaciones y trabajar con los recursos de red. La simplificada interfaz de objetos de este paquete para la programación en red facilita mucho el trabajo de los desarrolladores. Lenguajes como C o C++ son muy eficaces para trabajar en red, pero el código necesario resulta bastante más complicado y susceptible de errores que el equivalente en Java.

El paquete **java.nio** apareció por primera vez en la J2SE 1.4 (conocida antes como *Merlin*). En la versión 5.0 de la J2SE (conocida en sus primeras betas como 1.5 y como *Tiger*), se ha mantenido este paquete sin apenas modificaciones con respecto a la 1.4. **java.nio** introduce una nueva API (interfaz de programación de aplicaciones) de entrada y salida (E/S), conocida como **NIO** (*New Input/Output*). NIO se orienta hacia la transmisión

de grandes bloques de datos. Si nos restringimos a la programación en red, la característica más relevante del nuevo paquete reside en que permite crear sockets sin bloqueo, es decir, sockets que no bloquean el curso del normal del programa mientras realizan operaciones de E/S.

Como ejemplo que servirá para mostrar la aplicación inmediata de estos paquetes, usaré una aplicación de charla electrónica, que servirá también como hilo conductor del tutorial. Además, incluiré algunos ejemplos de uso para las clases más relevantes de estos paquetes. En este tutorial repetiré mucho la expresión “por ejemplo”; pues he preferido abordarlo desde un punto de vista práctico y concreto, y reducir la teoría al mínimo imprescindible para que se comprendan los ejemplos.

En el código del *chat*, se tratarán exhaustivamente **el control de las excepciones**, con el fin de aportar la máxima información posible al usuario, y **el cierre de flujos y sockets**. Algunos autores de libros, artículos y tutoriales sobre aplicaciones para redes parecen creer, quizá por optimismo o exceso de confianza, que la memoria disponible siempre es infinita, que los recursos del sistema operativo son inagotables y que basta tratar todas las excepciones como si fueran una sola. Otros confían fervorosamente en que el recolector de basura de Java intervendrá –a lo *deus ex machina*– para limpiar las pelusillas que van dejando. Pues bien: esas creencias son las que hunden una aplicación en red. Si quiere que una aplicación resista los envites del mundo real y que los usuarios no apuesten sobre cuántos minutos aguantará antes de colgarse (práctica poco caritativa, pero muy frecuente), le recomiendo que deseche las ideas anteriores.

Para sacar el máximo partido posible del tutorial, se requiere que el lector tenga un conocimiento básico de Java (compilación, sintaxis, entrada y salida) y conozca cómo trabajar con hilos de procesos (*threads*). Si bien estos conocimientos resultan necesarios (pueden consultarse en la sección de tutoriales de javaHispano), no pueden independizarse de un factor mucho más intangible: la curiosidad. Sin ella, de nada servirá todo lo demás.

En el apartado 2 del tutorial se introducen los fundamentos necesarios para entender las clases de Java que se irán explicando. Se da una rápida introducción a los conceptos elementales de redes (sockets, datagramas, DNS, protocolos IP, UDP, TCP).

En el apartado 3 se realiza una breve panorámica del paquete **java.net**, se explican con detalle las tres clases necesarias para el desarrollo del chat –**InetAddress**, **ServerSocket** y **Socket**– y se incluyen ejemplos de uso.

En el apartado 4 se presenta el código necesario para realizar un *chat* por consola y en el apartado 5 se presenta el código que lo implementa con una interfaz gráfica para el usuario.

En el apartado 6 se explica la nueva API de Entrada/Salida de Java (**NIO**), prestándose atención a su utilidad para la programación en red. Se explican las clases más útiles –**ServerSocketChannel**, **SocketChannel**, **Selector** y **SelectionKey**– y se dan varios ejemplos de cómo usarlas.

Por último, en el apartado 7 se actualiza el servidor de *chat* del apartado 4 mediante las clases del nuevo paquete **java.nio**, que proporcionan importantes ventajas en cuanto a tiempos de respuesta, ahorro de recursos y escalabilidad.

En esta segunda versión del tutorial he incluido mucho material ausente en la primera versión (unas noventa páginas) y he reescrito todo el texto original. El principal motivo para quintuplicar su tamaño original estriba en que la primera versión se escribió pensando en su publicación como artículo. Como se publicó como tutorial (seguramente por motivos de

formato, pues no lo escribí en HTML ni con la plantilla XML de javaHispano –no existía en 2002–), he pensado desde entonces que tenía que completarlo si no quería que la palabra "tutorial" le viniera grande. He esperado casi dos años para publicar esta nueva versión por dos motivos:

- Quería ver cómo aceptaba NIO la comunidad de programadores.
- Quería probar a fondo la versión final de la J2SE 5.0 (antes 1.5) para comprobar en qué estado quedaba **java.nio** en ella.

Desde la aparición de la primera versión de este tutorial, he publicado en javaHispano la primera parte del tutorial *Java y las redes. Introducción a las redes, a java.io, java.zip, java.net, java.nio, a RMI y a CORBA*. Este trabajo es mucho más completo y extenso que el que usted lee ahora, y trata muchas materias ausentes aquí. Aunque ambos tutoriales guardan similitudes, existen diferencias importantes:

- *Introducción rápida a java.net y java.nio. Cómo hacer un chat en Java* se orienta más hacia quienes deseen echar un vistazo a **java.net** y a **java.nio** para ver si les interesa o no. En él se tratan sólo las clases y métodos imprescindibles para programar un *chat* (que pueden servir para muchas otras aplicaciones cliente-servidor). La teoría se ha reducido al mínimo.
- *Java y las redes. Introducción a las redes, a java.io, java.zip, java.net, java.nio, a RMI y a CORBA* es un trabajo más completo, donde se abordan las clases y métodos más comunes para realizar aplicaciones en red con J2SE –servidores web, de FTP, navegadores, aplicaciones RMI, aplicaciones CORBA–, además de materias muy útiles para el desarrollo de aplicaciones en red (E/S de Java, serialización de objetos, internacionalización, compresión). Además, en él se comparan distintas tecnologías (sockets, RMI, CORBA, servicios web) y se estudian los fundamentos teóricos que hay tras cada una, así como tras las redes.

Como lector empedernido y más allá de toda redención, siempre me ha desagradado que las ideas de un autor se repitan exactamente igual en sus obras (demasiados escritores y filósofos se adelantaron a su época utilizando sin recato el "cortar y pegar" antes de que existieran los ordenadores). Por no hacer a los demás lo que no quiero que me hagan a mí, he intentado que los materiales de *Introducción rápida a java.net y java.nio* y *Java y las redes* sean lo más distintos posible: he abordado los conceptos desde distintas perspectivas; salvo en algunos párrafos, he procurado que la redacción sea distinta; casi siempre he usado figuras diferentes; incluyo ejemplos cortos distintos.

Lo único casi idéntico en ambos tutoriales es la aplicación de charla electrónica. Es un ejemplo tan bueno para ver en funcionamiento lo básico de **java.net** y **java.nio** que no he podido resistirme a incluirla por duplicado. En *Java y las redes* podría usar, en su lugar, una aplicación del estilo de los juegos en red; pero incluir una interfaz gráfica mínimamente profesional distraería la atención del lector sin experiencia en redes y alargaría en exceso la extensión del código.

Me hubiera resultado muy cómodo abandonar a su suerte la primera versión del tutorial *Introducción rápida a java.net y java.nio* y dedicarme sólo a *Java y las redes*; pero habría dejado desatendidos a los lectores que sólo quieren aprender lo básico de **java.net** y **java.nio** o que tienen curiosidad por echar un vistazo rápido a lo que Java puede ofrecer en cuanto a programación en red. Como sé –por los correos recibidos– que hay muchos lectores así, he optado por escribir la segunda versión de *Introducción rápida a java.net y*

java.nio. Conforme vayan saliendo versiones de Java con mejoras que atañan a **java.net** o **java.nio**, iré actualizando el tutorial.

Imagine un anuncio en que aparece lo siguiente: un cadáver descansa sobre una camilla, tapado con una sábana blanca; del tobillo izquierdo cuelga una etiqueta donde se lee el nombre de una conocida marca de vodka. Imagine ahora otro en que se muestra una pantalla negra donde aparecen rótulos blancos cada diez minutos, del tipo "¿Se aburre?", "¿No ve que esto no mejora?", "¿Por qué no va a darse una vuelta?", "Qué aburrido es esto". Imagine, por último, un anuncio en que un hombre encorbatado y embutido en un traje caro se dirige a la cámara y dice: "Sí, vendemos un producto a sabiendas de que es perjudicial [*carraspeo*]. No lo vean raro: es nuestro negocio. Además, intentamos captar a los adolescentes, que constituyen el segmento más vulnerable del mercado. Cuando ellos, de aquí a veinte o treinta años, empiecen a sufrir las consecuencias de nuestro producto, nosotros ya estaremos jubilados. Como ahora los países occidentales no ven con buenos ojos que aceleremos la muerte de sus ciudadanos [*suspiro de lamento por los viejos buenos tiempos*], hemos de expandir el negocio hacia países subdesarrollados o en vías de desarrollo; ya saben, hay que diversificar [*sonrisa cómplice*]. Si sus gobiernos intentan impedir, por motivos sanitarios, la entrada de nuestro producto, les amenazamos con represalias comerciales y demandas ante organismos internacionales. Así evitamos que crean que pueden proteger impunemente la salud de sus gobernados por encima de nuestros intereses".

Estos anuncios, como puede imaginar, jamás se verán en televisión; pues atentan contra un principio elemental de la publicidad: los productos, por perjudiciales o inútiles que sean, deben presentarse desde un punto de vista positivo y favorable. En algunos casos, la publicidad sobreactúa y presenta los productos de una forma tan optimista que hace creer a mucha gente que la libertad es un coche más veloz, con más caballos de potencia; que la libertad de expresión consiste en tener un teléfono móvil con acceso a Internet y cámara fotográfica; que unos cilindros con un poderoso insecticida son muestras de amistad; que las penas de amor se solucionan con una colonia (recomendaría París para esto último...) o que un tranquilizante ayuda "al recién llegado que no puede hacer amigos, al ejecutivo que no puede resignarse a jubilarse, a la suegra que no aguanta a la nuera". En realidad, por supuesto, hay pocos objetos industriales que hagan más libre a nadie, más bien al contrario.

Para compensar los excesos de la publicidad, los ciudadanos deberían recibir una educación que les permitiera rechazar los intentos de adoctrinamiento involuntario. A falta de ella, el consumidor que no desee ser engañado cuando adquiere un bien debe hacerse una pregunta muy sencilla, tan simple que se olvida a menudo: ¿se adecua la realidad a la publicidad? La pregunta es válida para cualquier producto: alimentos, coches de segunda mano, nuevos fármacos, nuevas tecnologías. En el caso de Java, la cuestión se divide en dos: ¿consigue Java simplificar mi trabajo como programador o desarrollador?, ¿lo simplifica más que otros lenguajes y plataformas?

Contestar seriamente a estas cuestiones se aparta mucho de los objetivos de este artículo, aparte de que duplicaría el tamaño del tutorial. Ahora bien, las respuestas no son un coto reservado a los expertos o a los que se denominan como tales. Cualquiera puede hacerlo. ¿Cómo? Leyendo documentos de distintas fuentes, comparando con otras tecnologías, haciendo pruebas, desdeñando las tretas de los publicistas y juzgando por uno mismo. Implica un cierto trabajo, pero nada más. Incluso si usted se ve obligado a utilizar una sola tecnología en su quehacer diario, lo anterior le ayudará a saber qué puntos débiles tiene.

Aquí sólo expondré, de manera muy abreviada, las características por las que recomiendo Java para la programación en red, sin omitir la comparación con otros rivales (para evitar suspicacias, mantendré las citas en su idioma original):

1) Java es un lenguaje orientado a objetos. C++ está OO, pero es mucho más "impuro" (permite datos definidos por el usuario que no son objetos, así como la mezcla de código OO con código modular o no modular). C# está en la misma línea que Java en cuanto a OO, pero incluye –por eficacia– estructuras de datos que no son objetos. Tanto Java como C# son mucho más fáciles de usar que C/C++.

2) Independencia de la plataforma. Los *bytecodes* producidos al compilar un programa Java pueden ejecutarse en cualquier plataforma con una máquina virtual de Java (Windows, HP-UNIX, Solaris, AIX, Zos, Linux, Mac, FreeBSD, NetBSD, teléfonos móviles). Dado que las redes contienen ordenadores de muchos fabricantes, la independencia respecto a la plataforma constituye una característica muy valiosa. Por ejemplo, los *bytecodes* de un programa Java compilado en Windows pueden enviarse por FTP (*File Transfer Protocol*, protocolo de transferencia de archivos) a un sistema UNIX, ejecutarse y probarse allí, y luego enviarse a un teléfono móvil para que vuelvan a ejecutarse.

Los lenguajes incluidos en la plataforma .Net de Microsoft también se compilan a un código intermedio independiente de la plataforma; pero, por ahora, .Net no se ha extendido a tantas plataformas como Java. Aunque la documentación de Microsoft se abstiene de usar el término "máquina virtual", .Net se basa en una, conocida como CLR (*Common Language Runtime*, entorno de ejecución común de lenguajes). A diferencia de Microsoft, Sun sólo ha fomentado un lenguaje para su máquina virtual (Java), aunque puede admitir tantos como .Net.

Por el momento, el CLR de Microsoft sólo está disponible comercialmente para la plataforma Windows. Existe una versión reducida que puede ejecutarse en dispositivos PocketPC (.Net Compact). Para el sistema operativo FreeBSD existe una versión reducida del CLR (llamada Rotor), pero su uso se restringe a fines no comerciales.

El proyecto Mono (http://www.go-mono.org/) se encarga de desarrollar para Linux una implementación, independiente de Microsoft y de código abierto, de C#, del CLR y de las bibliotecas de .Net; es decir, su objetivo consiste en desarrollar una versión abierta de la plataforma .Net. Por el momento, ha conseguido avances muy notables, pero aún no ofrece una versión que pueda usarse en un entorno de producción. Por ejemplo, mientras escribo esto (noviembre de 2004), la implementación de WinForms es muy deficiente: su falta de eficacia y los frecuentes errores la hacen incluso inferior al AWT de Java. La posición de Microsoft respecto a Mono no es clara: resulta obvio que las licencias de Microsoft no están pensadas para que nadie reproduzca todas las funciones de .Net, pero la empresa no ha condenado el intento ni ha tomado medidas legales en contra.

3) Licencia de Java. Si consideramos a Java como una plataforma (especificación del lenguaje, *bytecode*, máquina virtual Java y bibliotecas estándar), hemos de concluir que su licencia no es de código abierto. Cualquiera puede descargar y usar gratuitamente la MVJ de Sun, pero no se permite modificar el código y redistribuirlo (tampoco se permite distribuir la MVJ de Sun con aplicaciones comerciales, aunque sí el JRE –*Java Runtime Environment*, entorno de ejecución de Java–).

La especificación del lenguaje, en cambio, sí puede considerarse de código abierto: uno puede estudiarla y crear sus propias variantes de Java, aunque no sean totalmente compatibles con la especificación. GJ y AspectJ son variantes del lenguaje Java: la primera añadió genéricos a Java antes de que apareciera la versión 5.0; la segunda incluye características para la programación orientada a aspectos.

La implementación oficial de Java (MVJ, compilador a *bytecode*, bibliotecas estándar) no es abierta: pertenece a Sun. Si alguien descarga el código fuente de la J2SE y lo modifica, no puede distribuirlo libremente para uso comercial (al menos, no de forma legal). Para ello, el producto resultante debe pasar unas pruebas de compatibilidad y se debe obtener una licencia de Sun. Asimismo, si alguien utiliza para uso comercial (no educativo, privado ni de investigación) la implementación de Sun de un paquete de Java, debe solicitar una licencia comercial a la empresa.

Para propósitos de investigación, existe ahora una licencia JRL (*Java Research License*, licencia de investigación de Java) que permite crear y distribuir código basado en el de Sun, aunque no pase las pruebas de compatibilidad o no sea compatible con la implementación de Sun. Esta licencia permite la distribución legal de lenguajes derivados de Java, siempre que no se usen con fines comerciales.

Ahora bien, Sun permite actualmente la implementación por terceros de la MVJ, del compilador o de las bibliotecas –así como su distribución con la licencia que se estime oportuna, sin que sea obligatorio obtener una licencia suya o pagar regalías–, siempre que no se utilice su código. El gobierno brasileño, por ejemplo, está desarrollando una versión libre de la J2SE (Javali); además, existen muchas MVJ de código abierto (véase http://www.gnu.org/software/classpath/stories.html#jvm).

Por ejemplo, Kafee (http://www.kaffe.org/) es una implementación de código libre (licencia GNU GPL) de la MVJ y de las bibliotecas de clases necesarias para tener un JRE. Ha sido desarrollada desde cero, sin usar el código de Sun, y se incluye en muchas distribuciones de Linux y BSD (Red Hat, Mandrake, SuSE, Debian, Gentoo, Conectiva, PLD, Ark Linux, FreeBSD, NetBSD, OpenBSD). Tal y como se dice en la página web de Kaffe, "Kaffe is not an officially licensed version of the Java virtual machine. In fact, it contains no Sun source code at all, and was developed without even looking at the Sun source code. It is legal -- but Sun controls the Java trademark, and has never endorsed Kaffe, so technically, **Kaffe is not Java**“. Una MVJ implementada desde cero no puede usar el término “Java” (es propiedad de Sun); para ello, debe pasar unas pruebas de compatibilidad y se debe conseguir una licencia de Sun para ella (Jikes RVM, por ejemplo, cumple ambas condiciones).

Muchas voces piden, incluso dentro de Sun, que la implementación oficial de Java sea de código abierto. Las reticencias a escucharlas provienen de la sabiduría del gato escaldado: Sun tuvo problemas con Microsoft porque ésta intentó fragmentar Java produciendo una versión específica para Windows e incompatible con las restantes. Cuando preguntaron al vicepresidente de software de Sun, Jonathan Schwartz, qué le parecía la propuesta de IBM de hacer que Java sea de código abierto, contestó: "If IBM wants to allow incompatible implementations, I've seen that movie. It's called 'Microsoft licenses Java from Sun.' It forked the Java community, set us back years, and is now the subject of intense antitrust litigation. I'm not going to let that happen" (el juicio acabó con un acuerdo entre las dos partes: el dinero siempre lima asperezas). El futuro dirá si la implementación de Sun se convierte en código abierto o no.

Nota: La implementación libre más conocida de las bibliotecas estándar de Java (**java.***, **javax.***) es GNU CLASSPATH, una implementación construida desde cero. Casi todas las máquinas virtuales libres o de código abierto usan GNU CLASSPATH.

Si bien el esfuerzo que hay tras GNU CLASSPATH es meritorio, esta implementación está retrasada con respecto a las versiones de Java que va sacando Sun y falla en algunas bibliotecas (Swing, por ejemplo, no dispone de una implementación completa). Estos problemas se transmiten a todas las MVJ libres o abiertas que usan GNU CLASSPATH, lo cual restringe su utilización en entornos empresariales.

Con la licencia JRL de Sun, se puede completar GNU CLASSPATH con las bibliotecas que aún no implementa; pero hay que tener en cuenta que los productos resultantes sólo se podrán usar en proyectos de investigación.

Si Sun decidiera algún día que Java debe ser de código abierto, bastaría con que distribuyera las bibliotecas estándares de Java bajo alguna de las licencias de código abierto. Sólo con eso, los proyectos relacionados con MVJ abiertas evitarían reescribirlas y podrían dedicarse a mejorar las implementaciones de Sun y a incorporar sus propias bibliotecas.

En el caso de la plataforma .Net, la sintaxis y semántica del lenguaje C# se especifica en la norma ISO/IEC 23270:2003 (las páginas web de las organizaciones ECMA e ISO son http://www.ecma-international.org/ y http://www.iso.org/, respectivamente). La norma ISO/IEC 23271:2003 define la CLI (*Common Language Infrastructure*, infraestructura común de lenguajes) en que las aplicaciones escritas en distintos lenguajes pueden ejecutarse en distintos sistemas sin necesidad de reescribirlas. (Java no cuenta con una estandarización garantizada por un organismo oficial: ¿es eso un inconveniente? No lo creo, pues C++ lleva décadas de estandarización –ISO, ANSI– y sigue figurando en casi todos los diccionarios de informática como sinónimo de “incompatibilidad” o de “falta de interoperabilidad”: siguen existiendo bibliotecas incompatibles entre las distintas implementaciones de C++ y no parece que este problema vaya a desaparecer, salvo que se extinga el lenguaje.)

Microsoft envió una carta al ECMA en la que afirma que otras implementaciones de C# o del CLR "grant on a non-discriminatory basis, to any party requesting it, licenses on reasonable terms and conditions, for its patent(s) deemed necessary for implementing the ECMA standard”. Sin embargo, no hay que olvidar que la plataforma tiene muchos otros componentes aparte de C# y de la CLR (WinForms, ADO .Net, ASP .Net, etc.) y que

> The .Net Framework includes valuable intellectual property belonging to Microsoft. Just like any other company, we will review any action that may affect our intellectual property rights before making a decision about how to proceed. Any other speculation is just that, speculation. (Respuesta de Microsoft a la pregunta de si había patentes sobre tecnologías de .Net y, en ese caso, si Microsoft iba a renunciar a cualquier intento de limitar el proyecto Mono u otros intentos de implementar .Net; publicada en la revista *Java Developer’s Journal*, noviembre de 2002, volumen 7, número 11.)

De aquí se deduce que Microsoft tiene la libertad de impedir legalmente el desarrollo de cualquier implementación de .Net independiente. Dependiendo de la estrategia de Microsoft, ya se verá si la plataforma .Net es *realmente* multiplataforma.

En contraste, las licencias actuales de Java ofrecen dos grandes ventajas:

a) Permiten, sin resquicicios legales, que haya implementaciones libres o de código abierto de la plataforma de Java, lo cual garantiza la supervivencia de Java aunque Sun quebrara (empresas mucho más grandes han caído; pasó ya la época en que la definición de *gran empresa* era "Dícese de aquella empresa demasiado grande como para quebrar"). Incluso si Sun cambiara su estrategia respecto a Java, esas implementaciones independientes podrían seguir existiendo.

b) Favorece que desarrolladores, investigadores y usuarios se involucren en el desarrollo de las futuras versiones de Java.

4) Descarga dinámica de clases. Java permite descargar los *bytecodes* de una clase mediante una conexión de red, crear una instancia de la clase e incorporar el nuevo objeto a la aplicación en ejecución. En consecuencia, las aplicaciones en red pueden distribuirse de forma muy sencilla (mediante servidores web donde se almacenan los *bytecodes*, navegadores y *applets*) y son capaces de incorporar objetos desconocidos en en tiempo de compilación. C++ no ofrece unas capacidades dinámicas como las de Java. Smalltalk, sí; pero no ofrece tantas facilidades para la programación en red.

5) Inclusión estándar de varias API para la programación en red. El paquete **java.net** ofrece una API de alto nivel para trabajar con el protocolo HTTP, con los URL y con sockets.

La plataforma .Net también proporciona una, sospechosamente parecida a la de Java. Por ejemplo, cualquier programador de Java que use .Net enseguida se dará cuenta de lo semejantes que son las clases **System.Net.Sockets.TCPClient** y **java.net.Socket**. La asociación entre flujos y sockets que usa **java.net** para transmitir y recibir datos también ha sido adoptada por .Net.

Además de **java.net**, la versión estándar de Java incorpora el paquete **java.rmi**, que hace posible llamar a métodos de objetos en ordenadores remotos como si estuvieran en uno local, y permite trabajar con CORBA (*Common Object Request Broker Architecture*: arquitectura común de intermediación de solicitudes de objetos), una especificación para arquitecturas distribuidas de objetos definida por el OMG (*Object Management Group*, Grupo de gestión de objetos).

La plataforma .Net incluye su .Net Remoting., una plataforma de objetos distribuidos comparable, en esencia, a CORBA o RMI. Su funcionamiento interno guarda parecido con el de RMI.

C++ puede trabajar con sockets y con CORBA; pero exige un mayor esfuerzo para el programador: la legibilidad del código es menor y las API tradicionales no están tan orientadas a objetos como las de Java. En el caso de .Net, Microsoft siempre ha rechazado CORBA y la estandarización que supone, incluso cuando no existían los servicios web. Cuando CORBA comenzó, Microsoft apostó por su propia tecnología de objetos distribuidos (DCOM), idónea para trabajar en entornos Windows. DCOM se implementó también en entornos UNIX, pero no gozó de aceptación entre los desarrolladores: la API orientada a objetos de CORBA siempre ha sido superior a DCOM. Sun, en cambio, siempre ha apostado por CORBA: sus soluciones empresariales se basan en esta arquitectura o son compatibles con ella.

6) Java se mantiene al día en cuanto a tendencias en tecnologías distribuidas. Aunque Microsoft apostó por las tecnologías de servicios web antes que Sun, ahora Java dispone de una serie de herramientas excelentes para trabajar con estas nuevas tecnologías.

7) Comunidad de usuarios. Java dispone de una vasta comunidad, espontánea e internacional, de usuarios preocupados por difundir el lenguaje y por perfeccionarlo. No me extiendo más al respecto, pues javaHispano constituye buena prueba de ello.

8) Java se enseña en muchas universidades y escuelas de Informática. De hecho, éste es el motivo por que Microsoft incluye una copia de Java (Visual J#) en su plataforma .Net: no quiere verse excluida de las universidades. Como prueba, se puede leer esto en la publicidad de Visual Studio .Net 2005: "Visual J# 2005 Express Edition is an implementation of the Java language and is an ideal tool for anyone with prior Java experience or students using Java in school". Por si alguien se siente tentado a usar Visual J#, no está de más indicar que todas las novedades que se han ido introduciendo desde la versión 1.1.4 de Java (NIO, mejoras para gráficos 2D, 3D, etc.) no existen en esa réplica anticuada de Java.

9) Disponibilidad de herramientas gratuitas y de código abierto. Java dispone de herramientas de desarrollo gratuitas de excelente calidad (Eclipse, NetBeans; puede ver una lista de herramientas de código abierto en http://java-source.net/). Cualquier persona que empiece a programar en Java dispondrá de excelentes herramientas gratuitas que le ayudarán en cualquier etapa de su trabajo. Esto se traduce en: a) ahorro de costes para las empresas y los desarrolladores; b) libertad de elección del consumidor (en muchos casos, el usuario incluso puede hacerse su herramienta de desarrollo a medida, mediante *plug-ins* –extensiones– o modificando directamente el código fuente). C++ no goza de tantas herramientas gratuitas como Java: incluso hay herramientas de pago para C++ que son inferiores en prestaciones a algunas gratuitas de Java.

En el caso de .Net, la herramienta de desarrollo oficial es Visual Studio .Net; una solución excelente y completamente integrada con Windows y SQL Server; tiene en contra su precio (unos 2500 dólares la versión Enterprise Architect de Visual Studio .Net 2003) y que sólo está disponible para entornos Windows. Borland tiene para .Net su C#Builder, una excelente herramienta también muy cara (unos 2500 dólares la versión Architect). (Estos precios son para particulares y se han consultado en octubre de 2004.)

Existen herramientas de código abierto para trabajar con .Net; por ejemplo en http://www.icsharpcode.net/OpenSource/SD/ se puede encontrar un IDE (*Integrated Developement Environment*, entorno integrado de desarrollo) para Visual Basic .Net y C# llamado **SharpDevelop**, gratuito y de código abierto (licencia GPL). Como usted mismo puede comprobar, tanto este SharpDeveloper como otros IDE de código abierto quedan muy lejos de igualar las funciones de Visual Studio .Net o de algunos entornos de desarrollo para Java (en parte porque Java lleva más tiempo en el mercado que .Net y en parte porque Java es apoyado abiertamente por muchas empresas, lo que fomenta la competitividad en cuanto a herramientas).

10) Estabilidad de Java (como lenguaje y como plataforma). Cualquier código escrito en Java hace unos años sigue funcionando en las versiones más recientes de Java, y es bastante probable que lo mismo suceda en el futuro. Sun intenta que la plataforma mejore sin eliminar la compatibilidad con las versiones anteriores.

El contraste con Microsoft es claro: las tecnologías que ha patrocinado Microsoft han sido muchas, y casi siempre incompatibles con las elegidas antes para sus productos. Cualquier empresa que haya confiado, por ejemplo, en Visual Basic 6.0, para desarrollar

sus aplicaciones internas, se habrá dado ya cuenta de que el dinero invertido en formación y en desarrollo de nada servirá para pasar la aplicación a .Net (las aplicaciones deberán ser reescritas desde cero, salvo que sean triviales).

Si bien es cierto que las tecnologías informáticas avanzan a pasos agigantados y de forma incierta, también lo es que muchas empresas no pueden permitirse tirar a la basura sus aplicaciones y empezar de cero. Hasta el momento, Java ha sido sorprendentemente estable (con mejoras continuas, eso sí) en un mundo cambiante.

El final de la introducción no me corresponde a mí, sino a la recepcionista del hotel de Luxemburgo donde me hospedo ahora. Mientras alternaba la escritura del tutorial con miradas a la iluminada calle Joseph Junck, he recordado que cuando me registré en el hotel, hace cuatro días, la recepcionista me dijo en francés: “Está usted en un país muy pequeño, pero aquí nos caben muchos lugares interesantes”. Espero que algo parecido pueda decirse de este tutorial, pese a sus muchos defectos y omisiones. Buenas noches.

2. FUNDAMENTOS DE REDES

Hay una palabra que abunda en cualquier libro o artículo sobre redes: **protocolo**. Con ella, se designa una descripción o especificación de como deben comunicarse dos componentes de red, ya sean de hardware o de software.

La historia de las telecomunicaciones siempre ha discurrido paralela a la de los protocolos, aunque éstos no siempre han sido técnicos. Así, por ejemplo, cuando se instalaron los primeros teléfonos en Estados Unidos, los usuarios no sabían qué hacer cuando descolgaban el auricular. En consecuencia, permanecían mudos; muchos, enfadados por no saber cómo obrar, colgaban el auricular y juraban odio eterno al nuevo invento. Al final, se decidió que las telefonistas debían decir con amabilidad "Hello, speaking" ("Hola, al habla") cuando alguien establecía comunicación. Luego, decían al usuario que tenía una llamada o le preguntaban a qué número quería llamar.

Este ejemplo, que ahora parece lejano e ingenuo, ilustra la consecuencia de la falta de protocolos: **incomunicación**. Sin un protocolo tan sencillo como "Diga 'Hola, al habla' cuando alguien descuelgue un auricular", no había comunicación entre el emisor y el receptor.

Los protocolos técnicos de comunicaciones suelen ser difíciles de entender, porque se redactan de forma precisa, con el fin de evitar las ambigüedades. Por lo general, los protocolos no especifican ninguna implementación concreta: se pueden materializar de muy diversas maneras.

En telecomunicaciones, una **familia de protocolos** es un conjunto de protocolos que trabajan de manera conjunta para enviar datos de un dispositivo a otro. En el caso concreto de las redes de ordenadores (cualquier ordenador conectado a una red es un nodo de ella), una familia de protocolos es un conjunto de reglas, secuencias, formatos de mensaje y procedimientos que los ordenadores de una red pueden usar y comprender cuando intercambian datos.

La familia de protocolos de red más conocida es TCP/IP: Internet se basa en ella. Esta familia es independiente de la plataforma usada o del vendedor: permite superar los abismos que median entre ordenadores, redes y sistemas operativos de distintos fabricantes.

Muchos protocolos de esta familia se han implementado para pequeños dispositivos (agendas electrónicas, teléfonos móviles), electrodomésticos, ordenadores personales y superordenadores. Por ejemplo, cualquier fabricante de teléfonos que permitan acceder a Internet implementa la familia de protocolos TCP/IP. La implementación dependerá del hardware específico de los teléfonos; pero, aun así, permitirá la comunicación con cualquier otro dispositivo –una estación de trabajo, pongamos por caso– que tenga en funcionamiento alguna implementación de la familia TCP/IP. Las implementaciones serán absolutamente distintas (en cuanto a hardware y software, escasa relación guardan un teléfono y una estación de trabajo); pero permitirán el intercambio de información a través de Internet. De hecho, Internet es la unión de miles de redes heterogéneas, con distintas topologías y medios de transmisión: gracias a que todas implementan la familia TCP/IP, pueden interoperar.

Los protocolos más conocidos de la familia TCP/IP se muestran en la figura 1. El estudio de todos ellos excede con creces el propósito del tutorial. De forma simplificada, se tratarán aquí los protocolos IP, TCP y UDP.

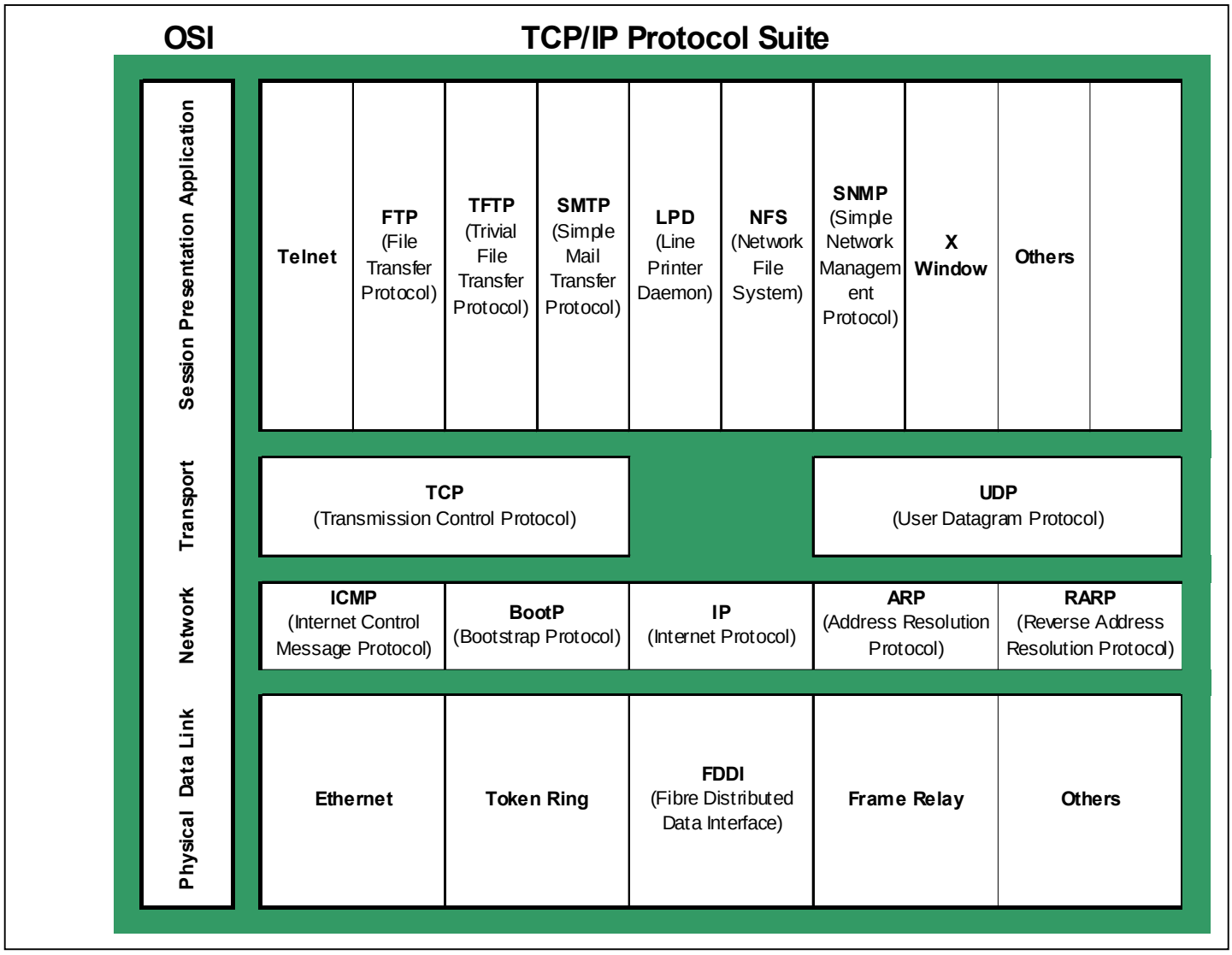

Figura 1. La familia de protocolos TCP/IP

Los componentes de una familia de protocolos suelen ubicarse en capas que agrupan funciones comunes, de modo que el objetivo común de la comunicación entre dos ordenadores se logra mediante la colaboración entre todas las capas. Por ejemplo, parece lógico agrupar el protocolo HTTP en una capa distinta a la de los protocolos que trabajan con voltajes, pues sus funciones son muy diferentes. El primero solicita una información con un formato determinado (HTML); los otros especifican como deben transmitirse los ceros y unos a través de los cables eléctricos.

TCP/IP se divide en cuatro capas:

- **La capa de aplicación.** Es la capa de nivel más alto; contiene todos los protocolos que las aplicaciones usan directamente para comunicarse con otras aplicaciones.
- **La capa de transporte.** Se encarga de establecer el proceso de comunicación entre dos nodos. Por extensión, se encarga de establecer las conexiones, transferir los datos, cerrar las conexiones, etc.

- **La capa de red o capa IP.** Proporciona los medios por los cuales un nodo puede enviar paquetes de datos (cuyo formato especifica un protocolo de esta capa) a través de una red, de manera independiente unos de otros. Esta capa se encarga de "encaminar" de forma individual cada paquete; de ahí que los paquetes puedan llegar a su destino en un orden distinto del que fueron enviados.
- **La capa de enlace.** Se encarga de enviar físicamente los paquetes de datos de un nodo a otro y de los detalles concernientes a la representación de los paquetes en el medio físico de transmisión. Esta capa admite múltiples protocolos, pues TCP/IP funciona sobre redes de muy distintas naturalezas (Ethernet, de fibra óptica, inalámbricas...), y cada una tiene su propio protocolo de enlace.

El protocolo IP (*Internet Protocol*, protocolo de Internet) es el protocolo de red utilizado para enviar datos entre los ordenadores conectados a Internet o que forman parte de intranets (redes basadas en los mismos protocolos TCP/IP que usa Internet). Su función básica consiste en enviar datos a través de Internet en *paquetes* (datagramas IP) que contienen de uno a mil quinientos bytes. IP es un protocolo sin conexión: no intercambia información de control antes de enviar datos al destino ni durante el envío. En consecuencia, no garantiza que los datagramas IP se entreguen en el orden en que se enviaron, ni siquiera que se entreguen. Los paquetes de datos pueden perderse o llegar fuera de orden; en estos casos, son otros los protocolos encargados de arreglar el desbarajuste, pues IP carece de medios para detectar los errores o corregirlos.

Todo datagrama IP consta de una cabecera y de una parte de datos. El esquema de la cabecera de un datagrama IP aparece en la figura 2.

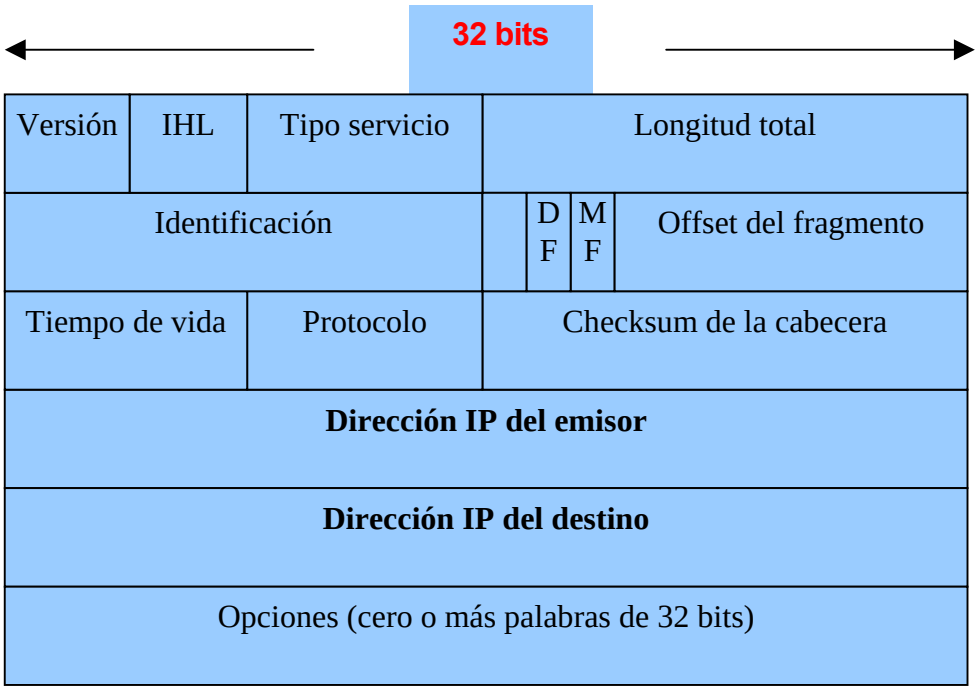

Figura 2. Cabecera de un datagrama IP (v4). Tras ella vienen los datos.

Todos los ordenadores que se comunican con este protocolo se identifican mediante una o más direcciones IP. Por ahora, la versión en vigor de este protocolo es la IPv4 que utiliza 4 bytes (32 bits) para cada dirección IP. En formato binario, las direcciones IP se representan como cuatro octetos separados por puntos (por ejemplo, 10000111.10100101.00100011.11110000). En formato decimal, una dirección IP está formada por 4 números enteros separados por puntos, siendo cada entero menor o igual a 255 (por ejemplo, 135.165.35.240).

Como puede verse en la anterior figura, cada paquete IP tiene dos campos con direcciones IP: una corresponde a la IP del ordenador emisor; la otra, a la IP del ordenador de destino. Si un datagrama IP pierde la información del campo “Dirección IP de destino”, el protocolo IP es incapaz de entregarlo y acaba descartándolo (cuando termine el tiempo de vida del datagrama, será eliminado; así se evita sobrecargar las redes con paquetes errantes).

Una dirección IP también puede representarse mediante un URL (*Uniform Resource Locator*, localizador uniforme de recursos), que viene a ser un puntero a un recurso de la Web (su “dirección”). La principal ventaja de los URL frente a las direcciones IP es que son mucho más sencillos de recordar. Así, http://www.google.es/ es mucho más fácil de recordar (y de anotar) que 216.239.59.147.

El servicio DNS (*Domain Name System*, sistema de nombres de dominio) se encarga de realizar la conversión entre los URL y las direcciones IP (proceso conocido como “resolver las direcciones IP”). Si este servicio no existiera, habría que usar siempre direcciones IP; pues los ordenadores trabajan internamente con direcciones IP, no con URLs.

En un URL hay la siguiente información:

- El protocolo necesario para acceder al recurso.
- El nombre de la máquina donde se alberga el recurso.
- El número de puerto por el cual la máquina permite solicitar el recurso.
- La ubicación del recurso dentro del ordenador.

Un URL tiene la forma

esquema: localización-según-el-esquema

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

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

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

Nota: En una red, un recurso es cualquier entidad, física o lógica, susceptible de tener interés para el usuario. Puede ser un archivo, un documento, un vídeo, un archivo de música, una consulta a una base de datos, un disco duro compartido, una impresora compartida, etc.
Los recursos no tienen por qué estar almacenados en la red (es decir, en los discos duros de los ordenadores que la componen), sino que pueden crearse dinámicamente como respuesta a las peticiones de los usuarios. Por ejemplo, cuando se hace una consulta en el buscador Google, la página web que se genera no existía previamente: se ha creado como respuesta a la consulta.
La estructura de Internet es tan abierta y flexible que permite incorporar nuevos protocolos cuando se necesitan nuevos recursos. Si alguien inventa un nuevo recurso, se pueden incluir protocolos nuevos en la pila TCP/IP, de modo que permitan trabajar con el nuevo recurso. Por ejemplo, la necesidad de usar hipertexto para facilitar el acceso a la información llevó a inventar el protocolo HTTP y el lenguaje de formato HTML. Las páginas web, que ahora forman parte de la experiencia inmediata de cualquier usuario de la Red, fueron al principio un nuevo recurso de ella. Con el tiempo, dieron lugar a la Web.

La mayoría de las aplicaciones que se ejecutan en Internet (WWW, FTP, Telnet, correo electrónico, etc.) corresponden a una arquitectura denominada cliente-servidor. Se dice que una aplicación informática –definida como un conjunto de programas y datos– tiene una estructura cliente-servidor cuando los programas y datos que la constituyen se encuentran repartidos entre dos (o más) ordenadores.

En una arquitectura cliente-servidor, un ordenador (denominado cliente) contiene la parte de la aplicación que se encarga de la gestión de la interacción inmediata con el usuario y, posiblemente, de parte de la lógica y de los datos de la aplicación. Las operaciones o los datos que no residen en el cliente pero que éste necesita se solicitan a la otra parte de la aplicación mediante el protocolo de aplicación correspondiente. Esta parte se encuentra en un ordenador compartido (el servidor), por lo común más potente y de mayor capacidad que el cliente, y está formada por el resto de datos y funciones de la aplicación informática. Este equipo se encarga de "servir" –es decir, atender y contestar– las peticiones realizadas por los clientes. Los términos *cliente* y *servidor* se emplean tanto para referirse a los ordenadores como a las partes de la aplicación.

Atendiendo a la ubicación de la lógica y de los datos de la aplicación, se pueden establecer distintas clasificaciones dentro de esta arquitectura (lógica distribuida, presentación descentralizada, lógica descentralizada), pero un estudio en detalle no es necesario.

Figura 3. Una aplicación cliente-servidor de lógica distribuida.

Un ejemplo típico de arquitectura cliente-servidor lo constituye un servidor web y un navegador web. El ordenador cliente es el del usuario que desea acceder a una página web y tiene en ejecución un programa cliente (un navegador web, en este caso). Cuando un usuario introduce una dirección web en el navegador, éste solicita, a través del protocolo HTTP, la página web a la aplicación de servidor web que se ejecuta en el ordenador servidor donde reside la página. La aplicación de servidor web envía la página por Internet a la aplicación cliente (el navegador) que la ha solicitado. Una vez recibida, el navegador la muestra al usuario.

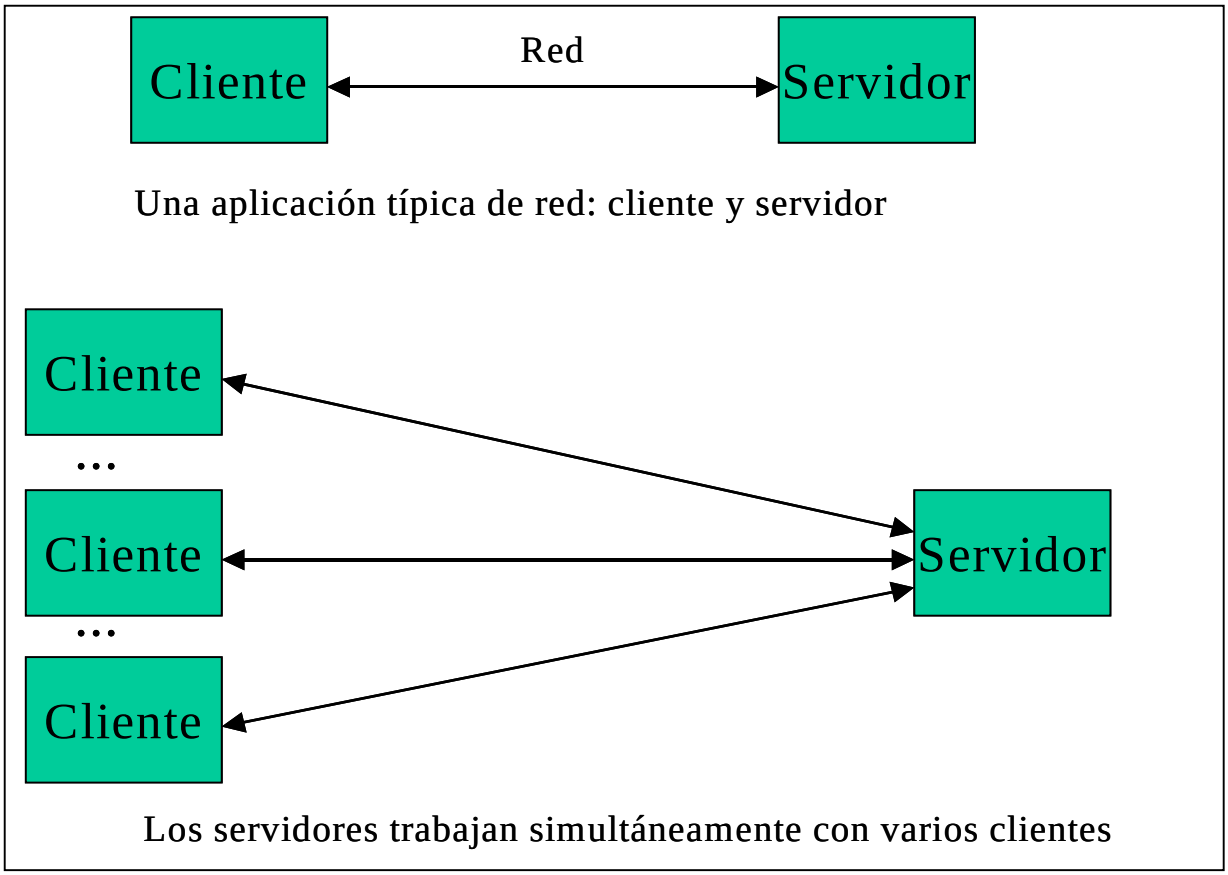

Figura 4. Esquema general de las aplicaciones de tipo cliente-servidor

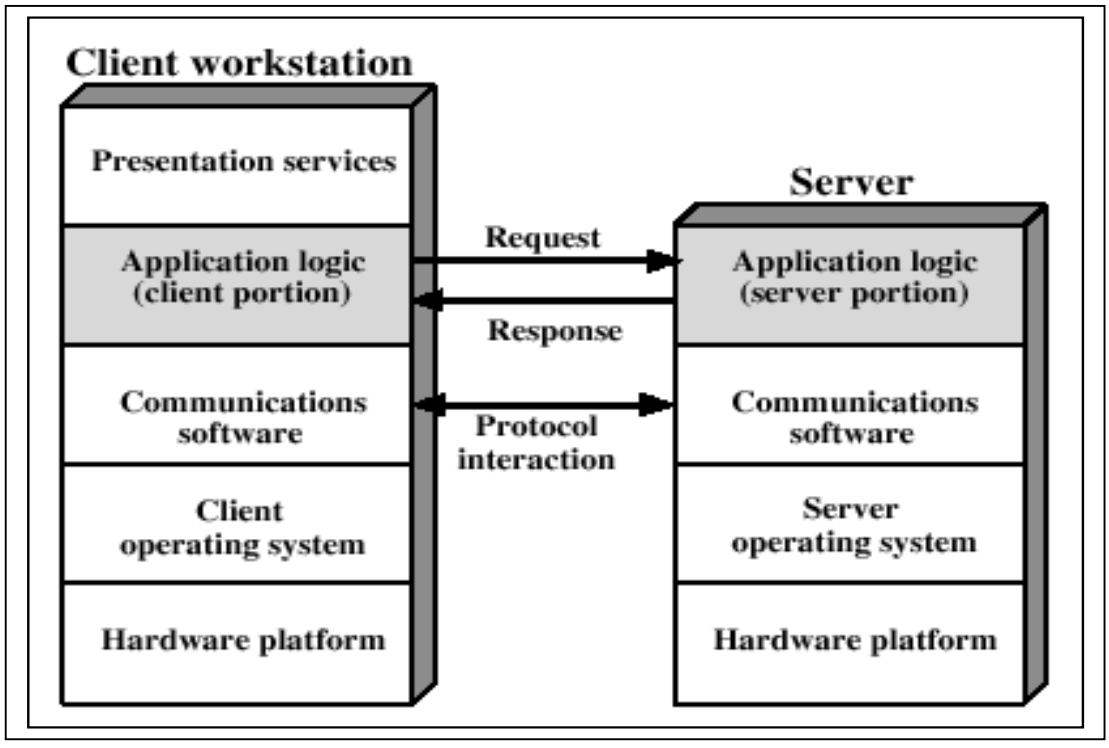

Figura 5. Esquema más detallado de las aplicaciones de tipo cliente-servidor

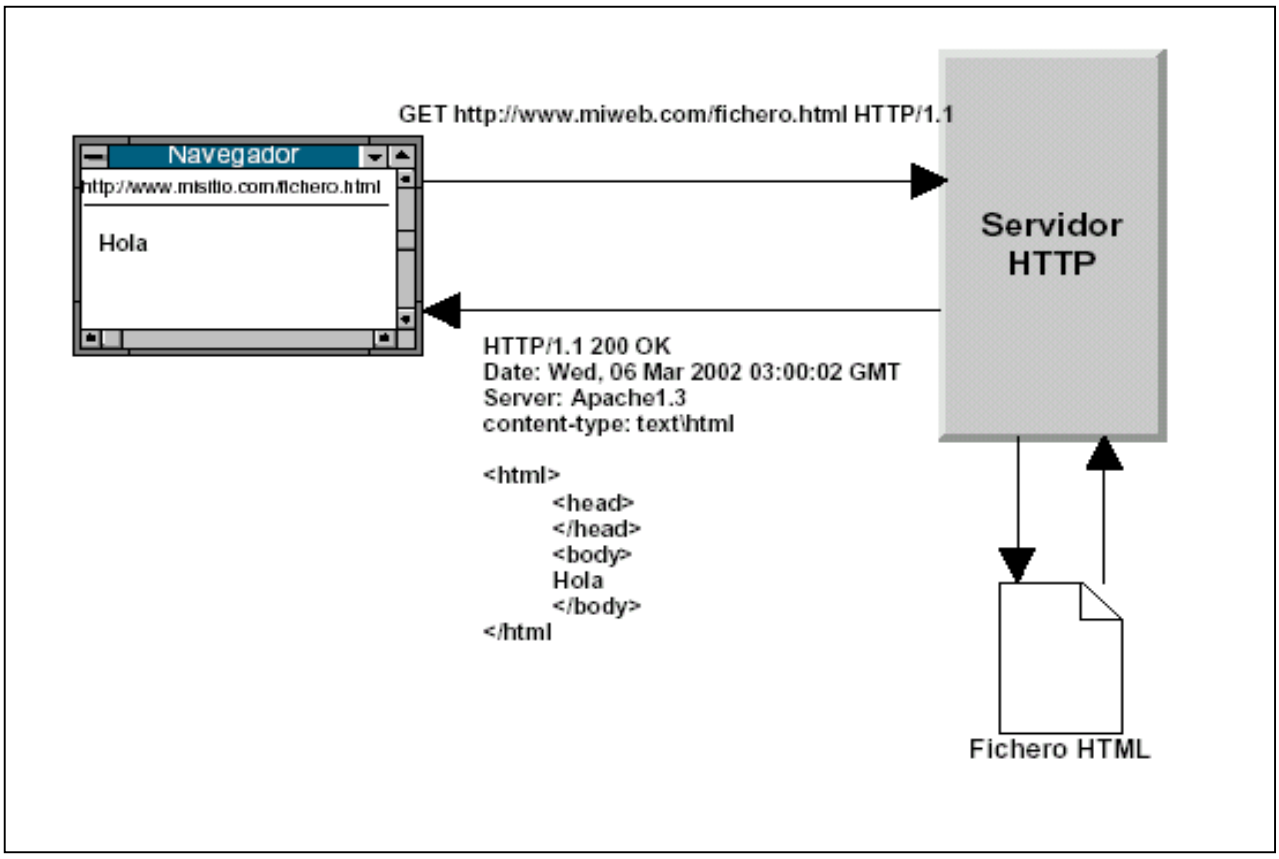

Figura 6. Esquema de una comunicación HTTP estándar, que sigue el modelo cliente-servidor

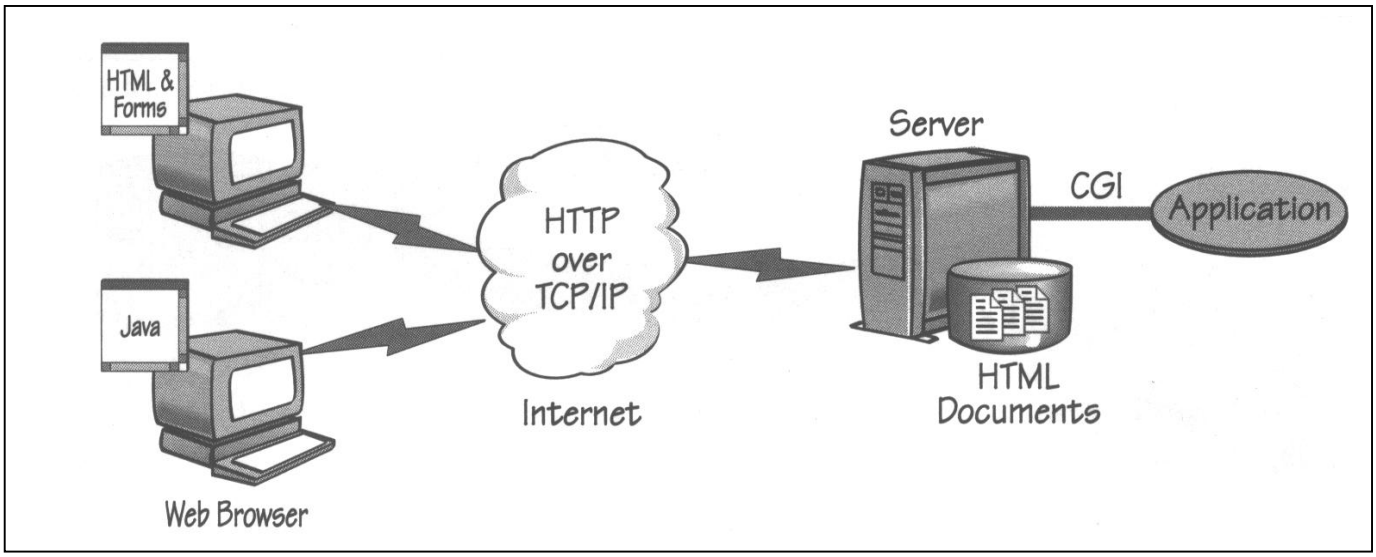

Figura 7. La Web, basada en HTTP, es la aplicación distribuida de tipo cliente-servidor más amplia y conocida

Al igual que el protocolo IP permite la comunicación entre ordenadores de Internet, los **puertos** permiten la comunicación entre las aplicaciones que se ejecutan en los ordenadores (toda aplicación proporciona uno o más servicios). En esencia, un puerto es una dirección de 16 bits asociada, por lo común, a una aplicación o servicio. Una interfaz de ordenador de red está dividida lógicamente en 65536 puertos (los primeros 1024 suelen

reservarse para servicios estándares y procesos del sistema operativo). La división no es física: estos puertos no tienen nada que ver con los puertos físicos de hardware.

En una arquitectura cliente-servidor (por ejemplo, el ejemplo anterior del servidor web y el navegador cliente), el programa servidor permanece *a la escucha* de peticiones de los clientes a través de los puertos que tiene activos. La información o los servicios solicitados son enviados por el servidor a los clientes, en concreto a los puertos que éstos han utilizado para realizar las peticiones. Por lo común, los servicios de Internet tienen puertos estándar por defecto. Por ejemplo, el protocolo HTTP utiliza el puerto estándar 80, y el protocolo FTP, el puerto estándar 21. Por ello, no es preciso escribir en un navegador http://www.aidima.es:21, basta escribir ftp://www.aidima.es.

En una comunicación a través de Internet, la dirección IP (de 32 bits) identifica de manera única el ordenador o nodo al que se dirige el mensaje y el número de puerto (de 16 bits) identifica a qué puerto (es decir, a qué aplicación o servicio) deben enviarse los datos.

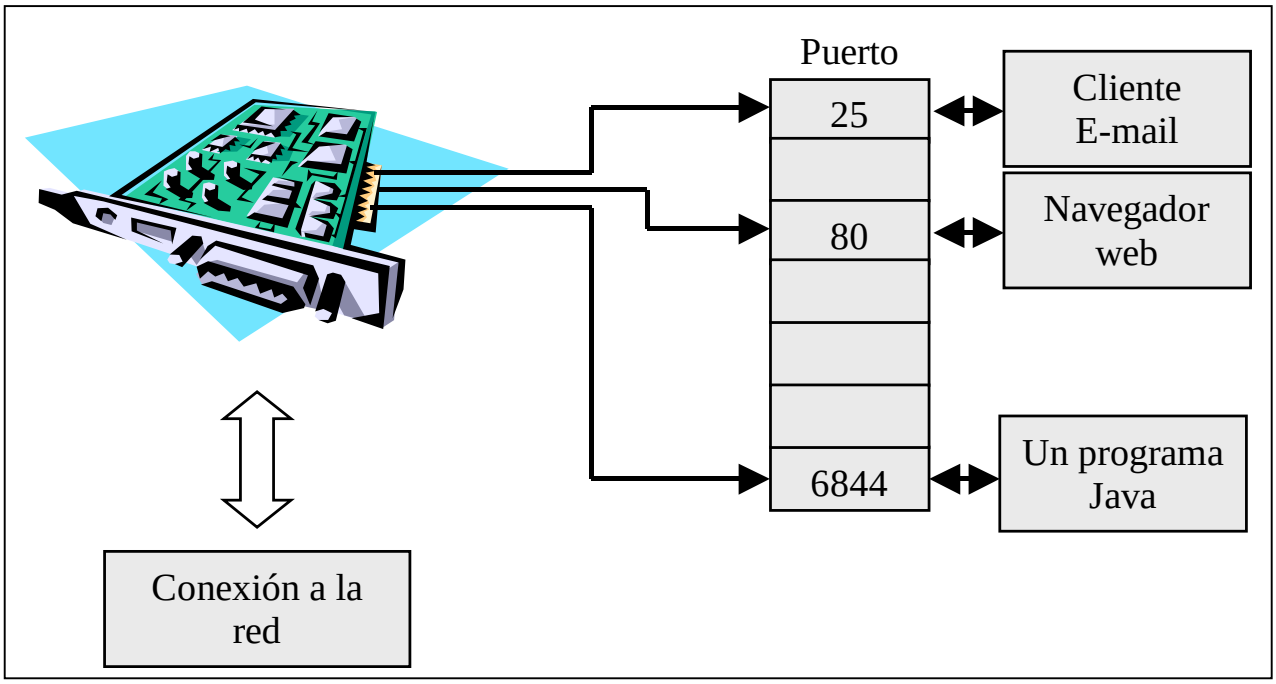

Figura 8. Uso simultáneo de varios puertos

Es posible utilizar puertos distintos de los estándares para los servicios de Internet, si bien no es lo habitual. Se podría, por ejemplo, asignar el puerto 2278 al protocolo HTTP, y el puerto 6756 al protocolo FTP. En este caso, los clientes necesitarían conocer los nuevos puertos asociados a cada servicio. Si no, el espectro de la incomunicación planearía de nuevo.

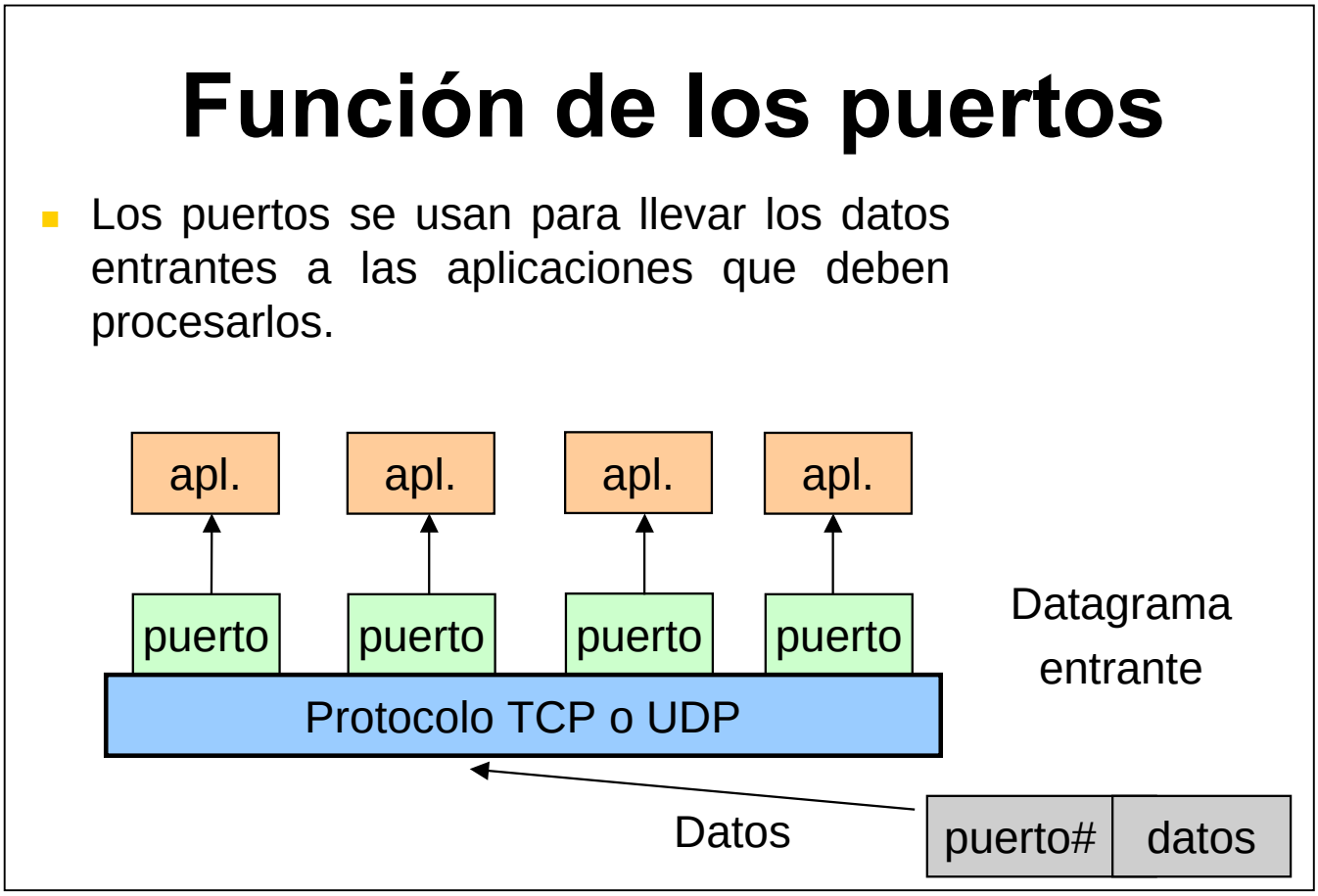

Figura 9. Función fundamental de los puertos

En la familia de protocolos TCP/IP, existen dos protocolos de transporte (TCP y UDP) que se encargan de enviar datos de un puerto a otro para hacer posible la comunicación entre aplicaciones.

El protocolo **TCP** (*Transmission Control Protocol*, protocolo de control de la transmisión) es **un servicio de transporte orientado a la conexión**. Mientras existe una comunicación TCP, hay un enlace entre el emisor y el receptor, enlace que permite el intercambio bidireccional de información. Este protocolo incorpora algoritmos de detección y corrección de errores en la transmisión de datos; y garantiza la entrega de los paquetes de datos TCP (datagramas TCP) al receptor, en el mismo orden en que los envió el emisor.

Como nada sale gratis, este protocolo funciona sacrificando velocidad por seguridad. Cuando un emisor envía un datagrama TCP, se comienza una cuenta atrás de cierto tiempo. Si el tiempo acaba sin que el emisor haya recibido un mensaje de confirmación de la aplicación TCP del receptor, el datagrama se vuelve a enviar.

TCP proporciona fiabilidad a IP. Con TCP, las comunicaciones *parecen* realizarse mediante flujos continuos de bytes. En realidad, la comunicación física se basa en paquetes discretos (datagramas); pero TCP simula un flujo lógico continuo de bytes. Internamente, TCP acepta un flujo de datos de una aplicación, fragmenta los datos en datagramas que no exceden de 65535 bytes (suele usarse un tamaño de 1500 bytes) y envía cada datagrama a la capa de red, donde será transformado en un paquete IP. Cuando los datagramas IP llegan al nodo de destino, éste los pasa a la aplicación TCP correspondiente, que reconstruye el flujo continuo de bytes.

Debido al enlace existente entre el emisor y el receptor durante una comunicación TCP, este protocolo suele compararse con el servicio telefónico: se establece la conexión, se transfieren los datos y, por último, se cierra la conexión.

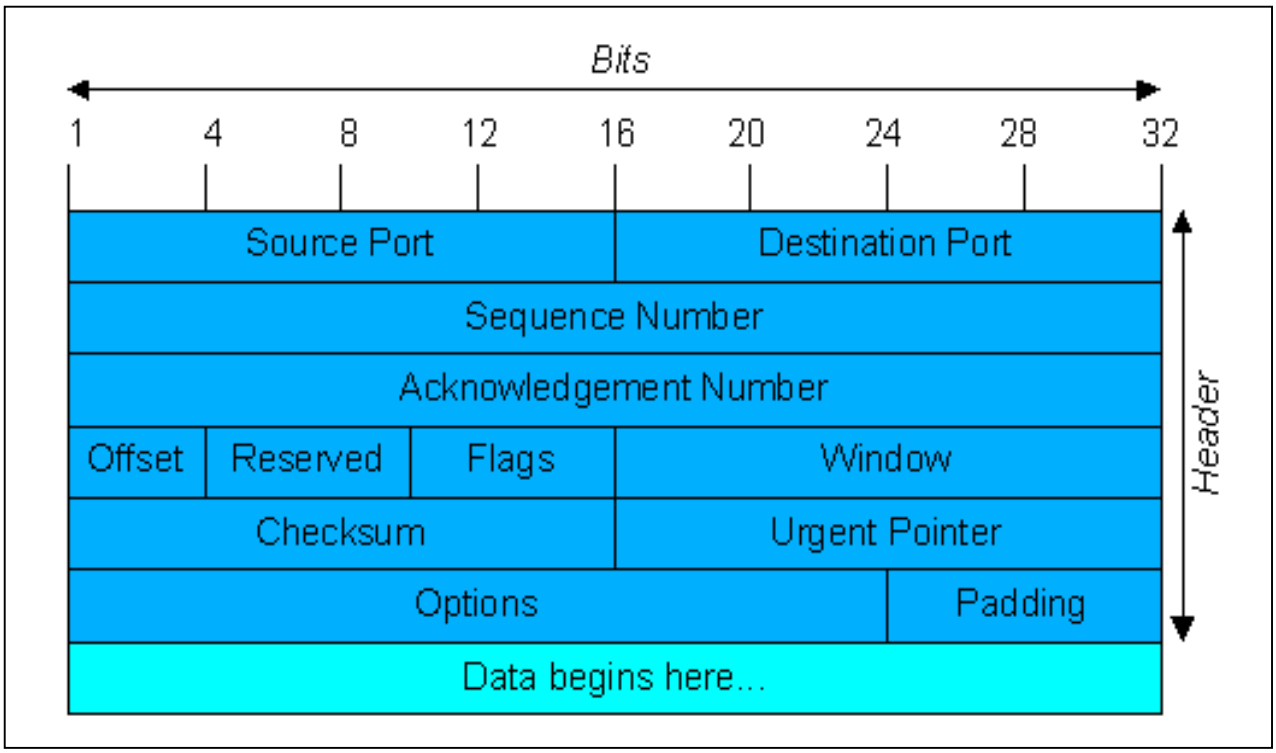

Figura 10. Esquema de un datagrama TCP. El campo inferior alberga los datos. En el proceso de envío de la información entre ordenadores, los datagramas TCP se encapsulan en datagramas IP

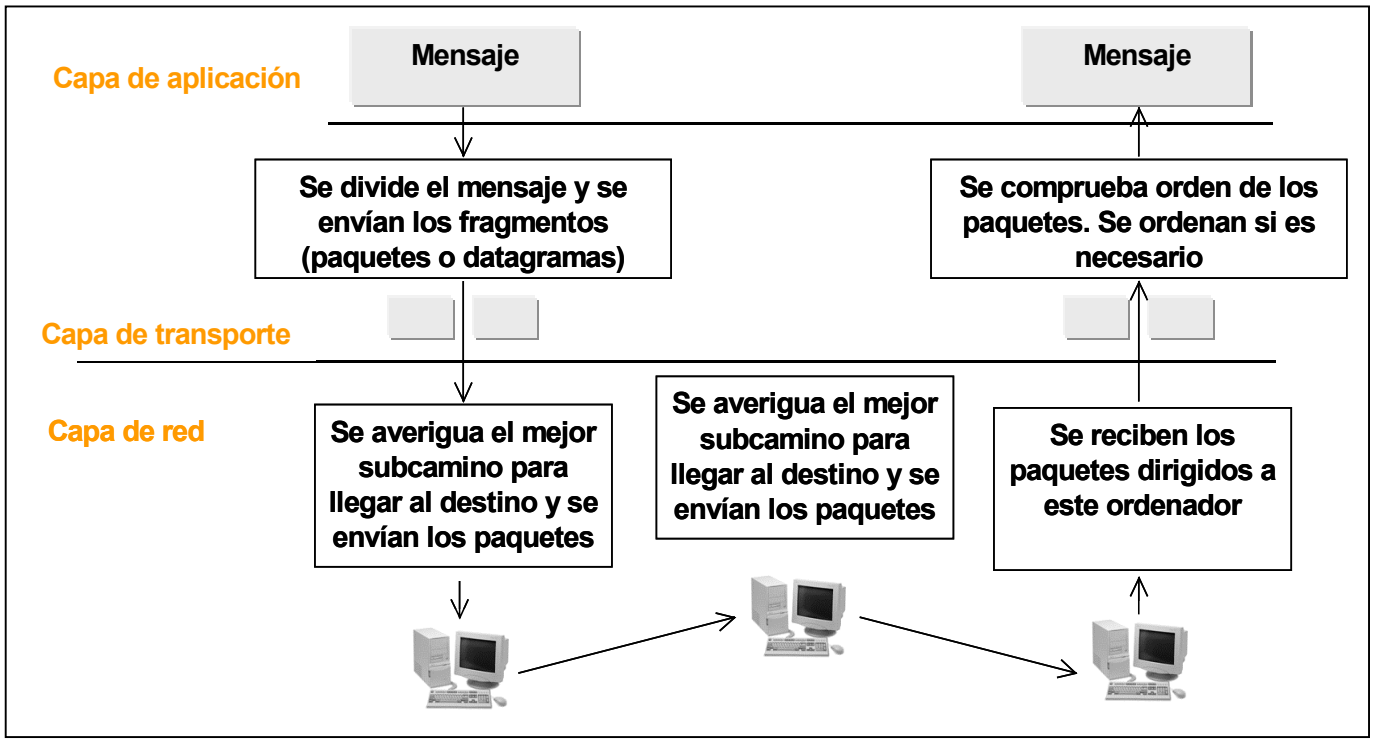

Figura 11. Esquema del funcionamiento del protocolo TCP

El protocolo **UDP** (*User Datagram Protocol*, protocolo de datagramas de usuario) es **un servicio de transporte sin conexión**: el mensaje del emisor se descompone en paquetes UDP (datagramas UDP); y cada datagrama se envía al receptor por el camino que se envíe oportuno, de forma independiente de los otros. Los datagramas pueden llegar en cualquier orden al receptor (incluso pueden no llegar), y el emisor ignora si han llegado correctamente o no. Por este motivo, UDP suele compararse con el servicio postal: se envían los datos sin que medie conexión (una carta puede enviarse sin que se sepa si existe la dirección de destino o si alguien vive allí).

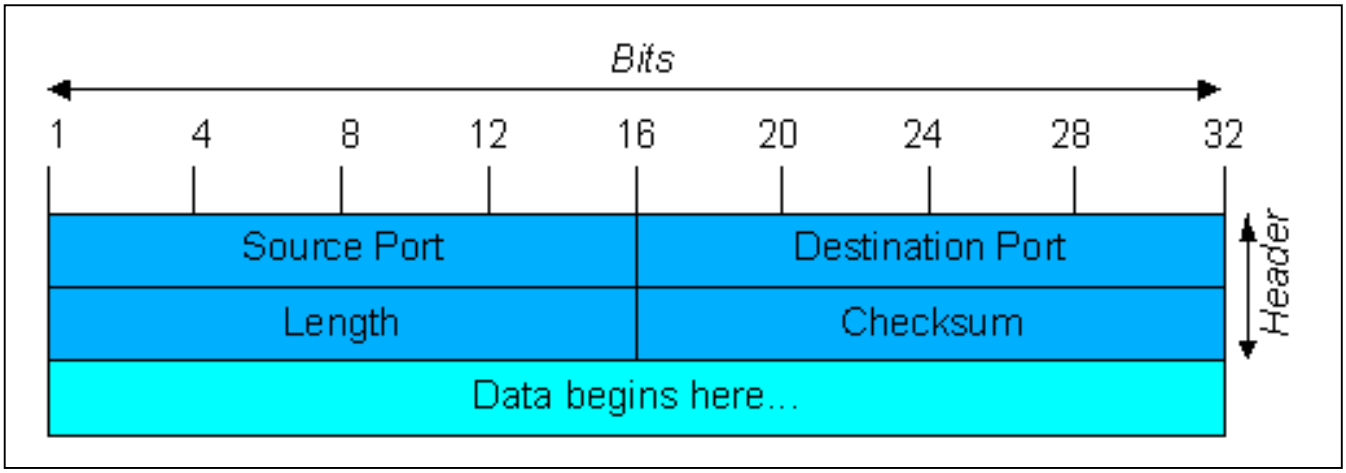

Figura 12. Esquema de un datagrama UDP. El campo inferior alberga los datos. En el proceso de envío de la información entre ordenadores, los datagramas UDP se encapsulan en datagramas IP

TCP y UDP usan ***sockets*** (literalmente, "conectores" o "enchufes"; el término procede de los paneles de conexiones que usaban antes las centralitas telefónicas, paneles donde la conmutación de las líneas telefónicas se hacía a mano) para comunicar programas entre sí en una arquitectura cliente-servidor. Un socket es un punto terminal o extremo en el enlace de comunicación entre dos aplicaciones (que, por lo general, se ejecutan en ordenadores distintos). Las aplicaciones se comunican mediante el envío y recepción de mensajes mediantes sockets. Un socket TCP viene a ser un teléfono; un socket UDP, un buzón de correos.

Además de clasificarse en sockets TCP y UDP, los sockets se encuadran dentro de uno de estos grupos:

- **Sockets activos:** pueden enviar y recibir datos a través de una conexión abierta.
- **Sockets pasivos:** esperan intentos de conexión. Cuando llega una conexión entrante, le asignan un socket activo. No sirven para enviar o recibir datos.

En *Thinking in Java 3rd Edition*, Bruce Eckel describe los sockets de una forma visual, con su chispeante estilo:

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

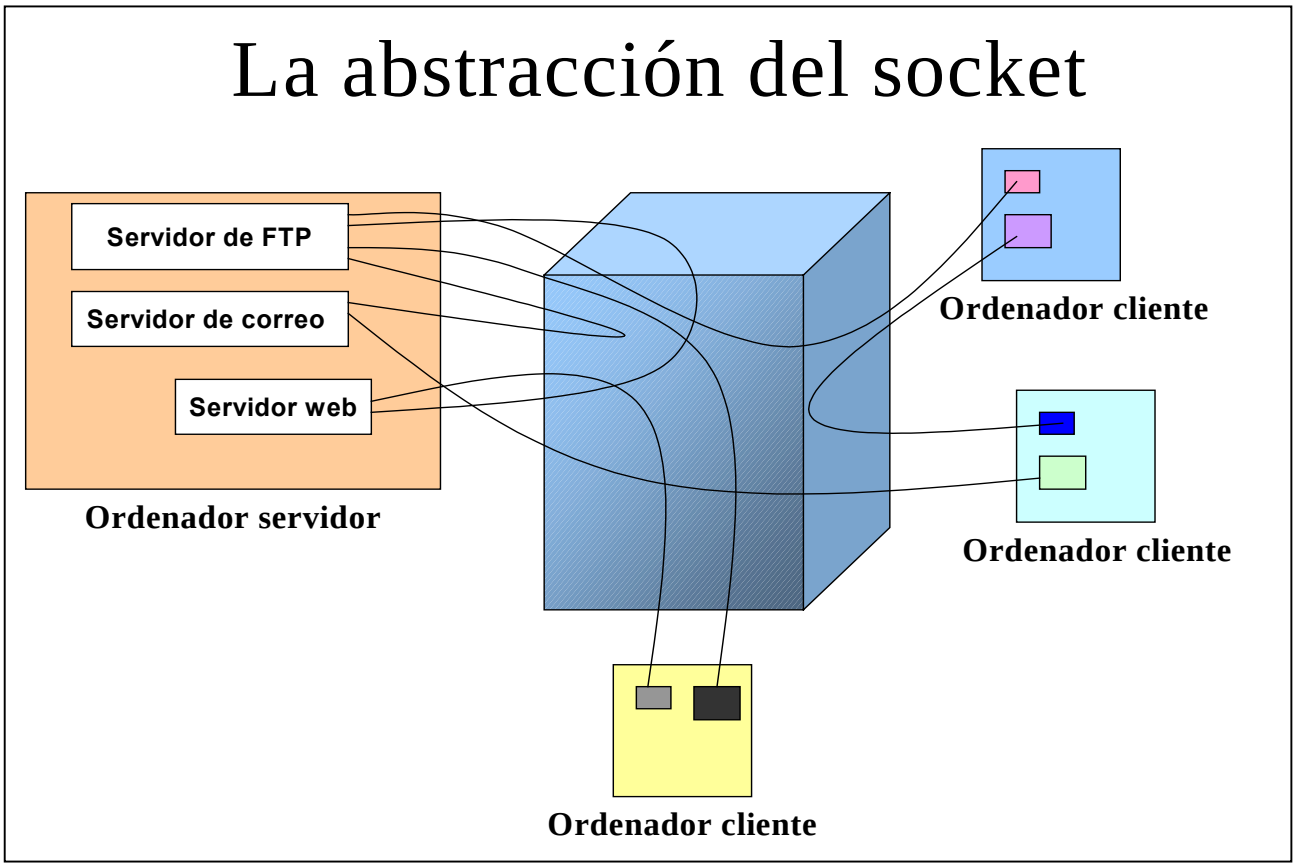

Figura 13. La abstracción del socket

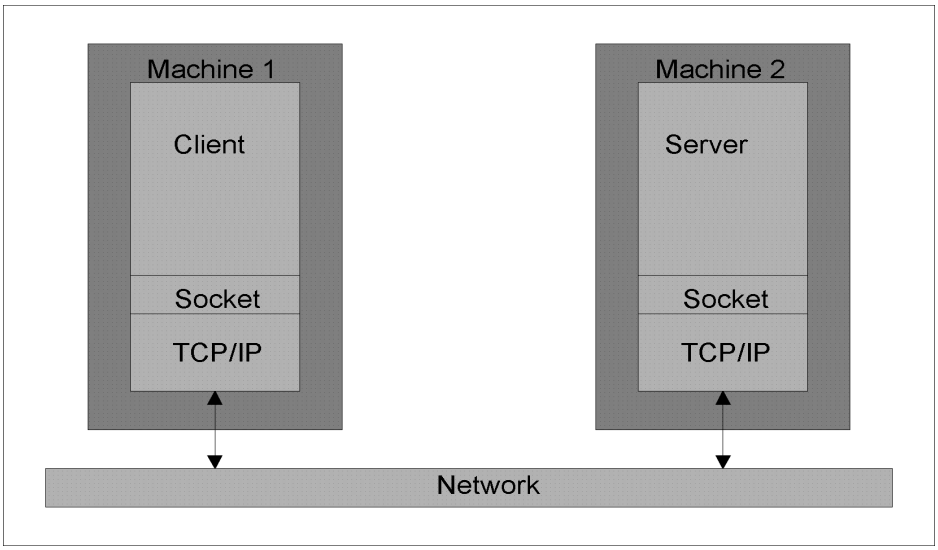

Figura 14. Ubicación conceptual de los sockets

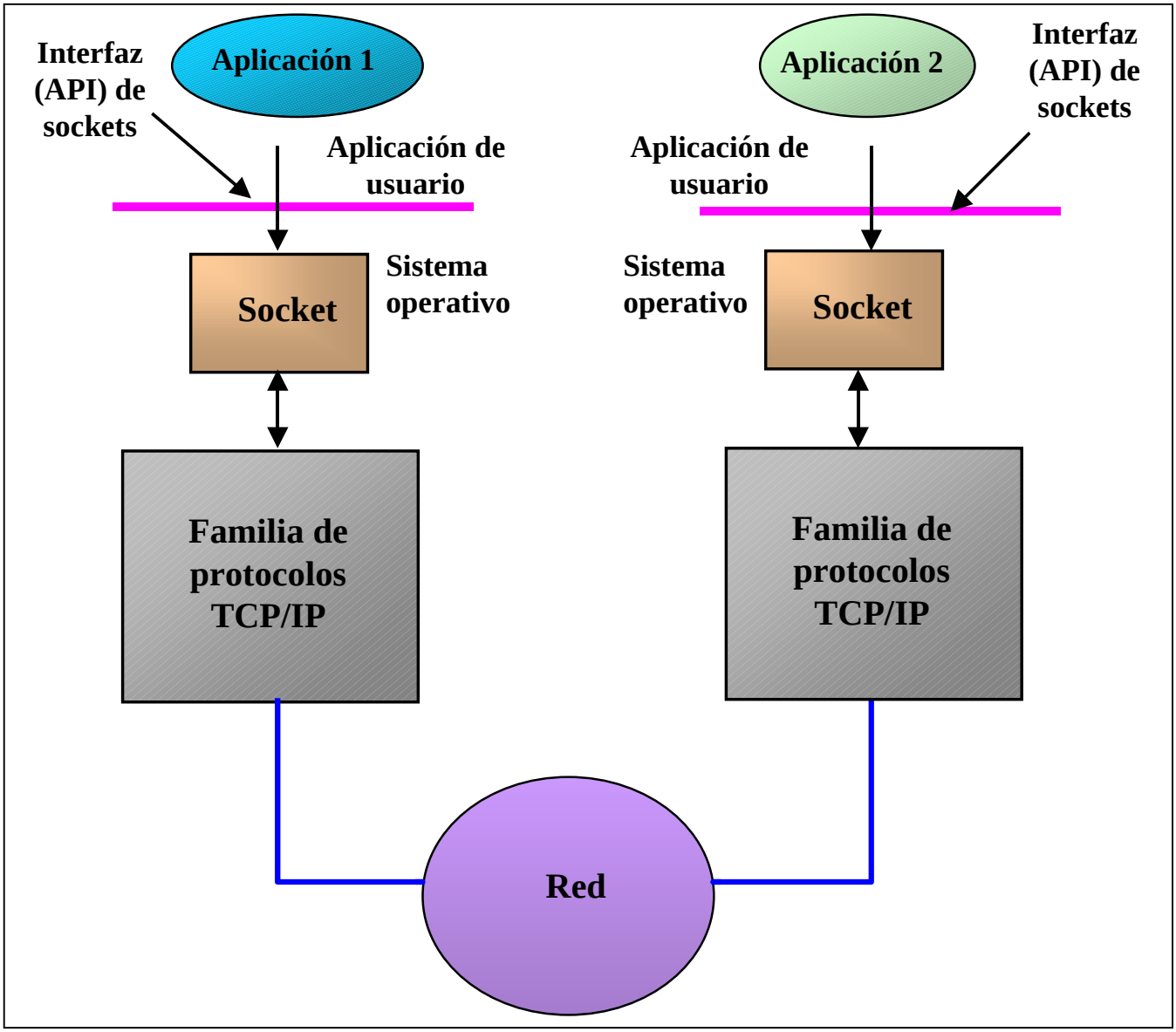

Figura 15. Comunicación mediante sockets

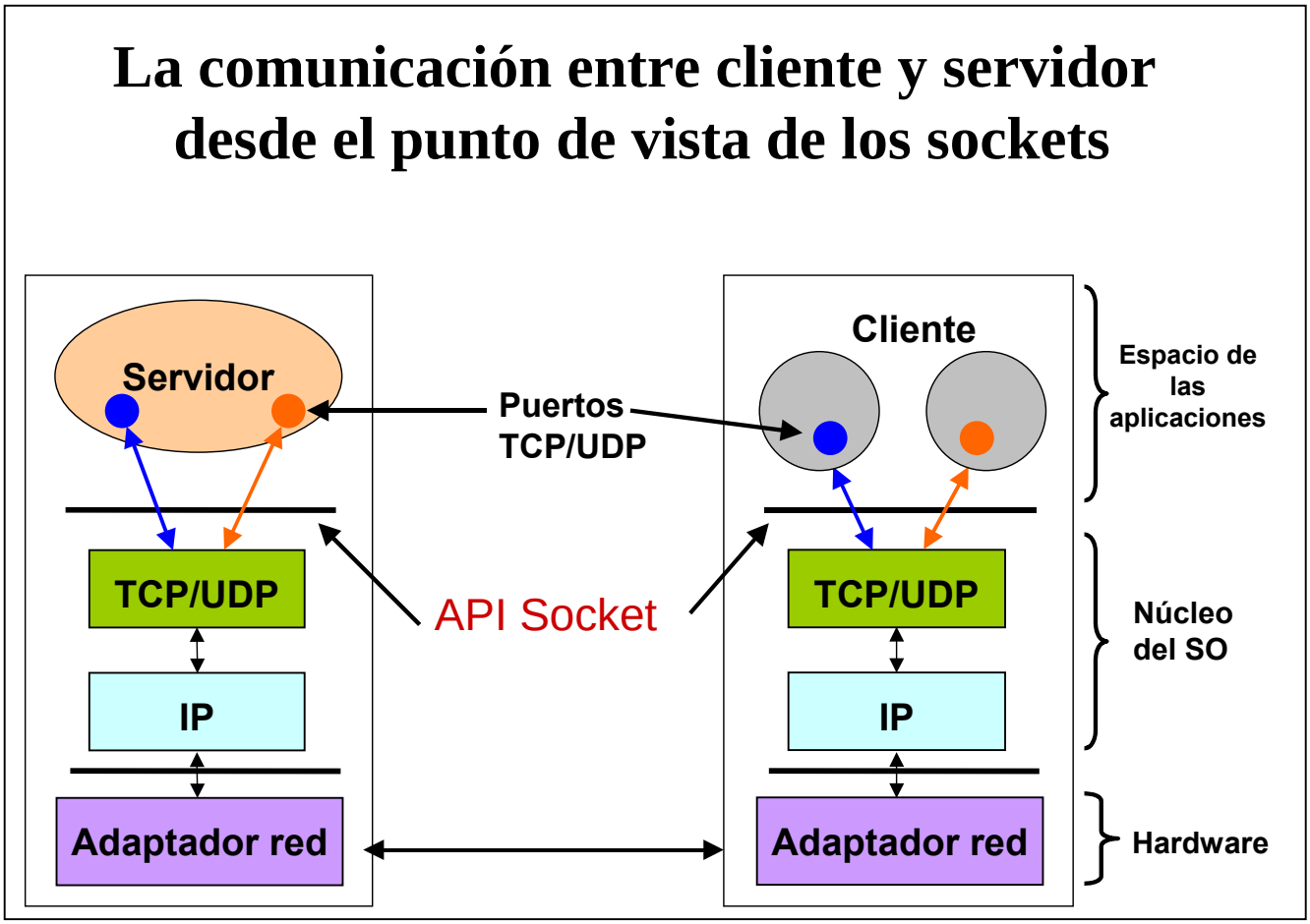

Figura 16. Comunicación mediante sockets incluyendo puertos

Los sockets son creados por el sistema operativo y ofrecen una interfaz de programación de aplicaciones (API) mediante la cual pueden las aplicaciones enviar mensajes a otras aplicaciones, ya sean locales o remotas. Las operaciones de los sockets (enviar, recibir, etc.) se implementan como llamadas al sistema operativo en todos los modernos SO; dicho de otra manera: los sockets forman parte del núcleo del SO. En lenguajes orientados a objetos como Java o C#, las clases de sockets se implementan sobre las funciones ofrecidas por el SO para sockets.

Con lo visto hasta el momento, podemos plantearnos la siguiente cuestión:

> Consideremos un servidor web (es decir, HTTP) que atiende simultáneamente a varios clientes por el puerto estándar (80). Los clientes usan el protocolo TCP (HTTP usa este protocolo por defecto). ¿Cómo puede el servidor saber a qué cliente corresponde cada petición?

La respuesta se representa en la figura 17.

Un socket tiene una dirección. En ella se almacena un número de puerto, una dirección IP y un tipo de protocolo.

La dirección del socket cliente almacena el número de puerto por el cual hace la petición, la dirección IP del ordenador y el tipo de protocolo (TCP, en este caso).

La dirección del socket servidor almacena el número de puerto por el cual espera peticiones (80, en este caso), la dirección IP del ordenador y el tipo de protocolo (TCP, en este caso).

Una conexión se caracteriza por dos direcciones de sockets: la del cliente y la del servidor.

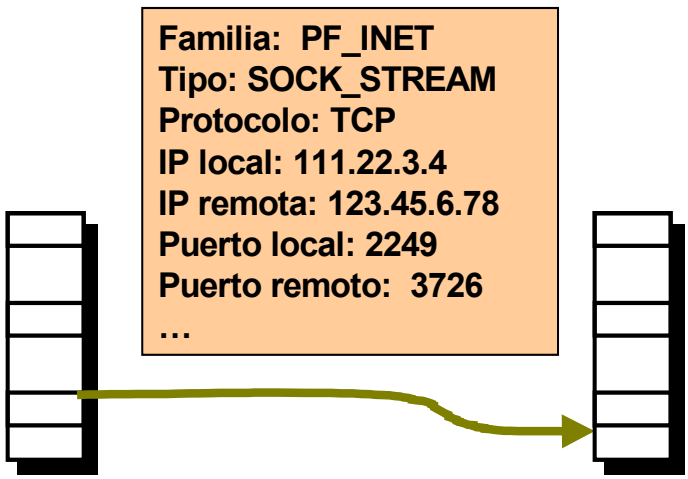

Figura 17. Estructura interna de una comunicación con sockets TCP

Por ejemplo, para el protocolo HTTP:

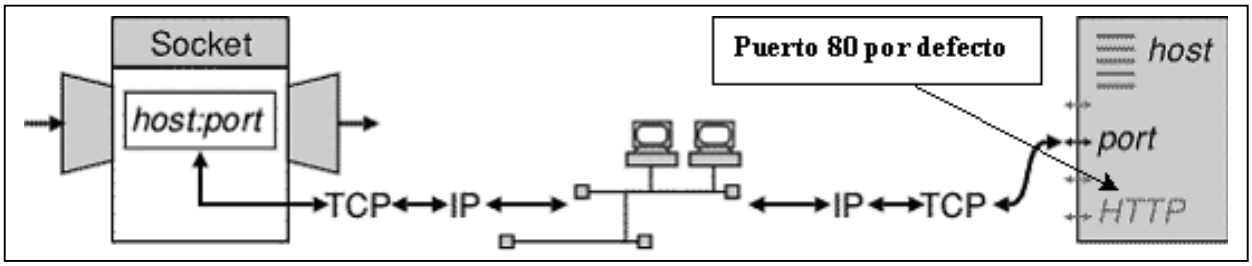

Figura 18. Una comunicación HTTP desde el punto de vista de los sockets

Como he apuntado antes, las diferencias entre las comunicaciones con sockets TCP y con sockets UDP resultan similares a las que existen entre **llamar por teléfono** y **enviar una carta**.

Cuando se llama por teléfono a alguien, el estado de la comunicación se conoce en todo momento: quien llama sabe si el teléfono está ocupado o no (en algunos casos, el destinatario de la llamada también sabe si tiene alguna llamada entrante); percibe cuándo puede comenzar la comunicación (cuando acaban los pitidos largos y suena una voz); sabe si se está dirigiendo a la persona deseada (salvo que lo engañen) y distingue cuándo la comunicación ha concluido o se ha interrumpido (cuando cesa la voz del interlocutor y se oyen unos pitidos breves).

Cuando se envía una carta, el remitente la introduce en un buzón y se olvida de ella durante un tiempo: ignora si llegará a su destino o si recibirá respuesta (la dirección puede no existir; el destinatario puede no vivir ya allí; la carta puede perderse en alguna oficina de correos con exceso de trabajo). No puede haber, en definitiva, un seguimiento continuo del estado de la comunicación. Si, al cabo de un tiempo, el remitente recibe una carta con el texto “La Agencia Tributaria ha recibido sus alegaciones, pero las ha desestimado. Puede usted reclamar por la vía civil, pero perderá tiempo y dinero. Ahórrese sufrimiento y pague ya”, sabrá dos cosas: a) que su carta fue recibida, y b) que no ha servido para nada. Si no recibe respuesta, no estará seguro de si la carta llegó a su destino; lo mismo ocurre en el protocolo UDP: un emisor de datagramas UDP desconoce si llegan o no a su destino.

Para que sea posible la comunicación cliente-servidor, el cliente debe establecer sockets y conectarlos a sockets del servidor. Los pasos que se siguen en una comunicación típica con sockets TCP son éstos:

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

Las cinco primeras etapas merecen algunos comentarios. En primer lugar, el servidor debe estar en ejecución antes de que los clientes intenten conectarse a él y debe mantener activo un socket pasivo que permanezca a la espera de peticiones entrantes.

En segundo, el cliente debe crear un socket activo en el cual se especifiquen la dirección IP y el número de puerto de la aplicación que proporciona el servicio deseado. Mediante este socket, el cliente comienza una conexión TCP con el servidor.

En tercer lugar, el intento de conexión TCP desencadena que el servidor cree un socket activo para el cliente (durante su tiempo de vida, no podrá ser asignado a otros clientes).

Finalmente, se establece la conexión TCP entre el socket del cliente y el socket activo del servidor. A partir de entonces, uno y otro pueden enviar datos o recibirlos a través de la conexión.

Para el programador de Java, un socket es la representación de una conexión para la transmisión de información entre dos aplicaciones, ya se ejecuten en un mismo ordenador o en varios. Esta abstracción de alto nivel permite despreocuparse de los detalles que yacen bajo ella (correspondientes a los protocolos subyacentes). Para implementar los sockets TCP, el paquete **java.net** de Java proporciona dos clases: **ServerSocket** y **Socket**. La primera representa un socket pasivo que espera conexiones de los clientes y que se ubica en el lado del servidor. La segunda representa un socket activo que puede enviar y recibir datos (puede ubicarse en el servidor y en el cliente).

En resumen, un socket permite conectarse a un equipo a través de un puerto, enviar o recibir datos y cerrar la conexión establecida. Todos los detalles de los protocolos de red, de la naturaleza física de las redes de telecomunicaciones y de los sistemas operativos involucrados se ocultan a los programadores. Si no fuera así, no habría muchos más programadores de aplicaciones para redes que estudiosos de la literatura en sánscrito.

3. EL PAQUETE java.net DE JAVA

3.1 Introducción

Gran parte de la popularidad de Java se debe a su orientación a Internet. Cabe añadir que esta aceptación resulta merecida: Java proporciona una interfaz de sockets orientada a objetos que simplifica muchísimo el trabajo en red.

Con este lenguaje, comunicarse con otras aplicaciones a través de Internet es muy similar a obtener la entrada del usuario a través de la consola o a leer archivos, en contraste con lo que sucede en lenguajes como C. Cronológicamente, Java fue el primer lenguaje de programación donde la manipulación de la entrada y salida de datos a través de la red se realizaba como la E/S con archivos.

El paquete **java.net**, que proporciona una interfaz orientada a objetos para crear y manejar sockets, conexiones HTTP, localizadores URL, etc., comprende clases que se pueden dividir en dos grandes grupos:

a) Clases que corresponden a las API (interfaces de programación de aplicaciones) de los sockets: **Socket**, **ServerSocket**, **DatagramSocket**, etc.

b) Clases correspondientes a herramientas para trabajar con URL: **URL**, **URLConnection**, **HttpURLConnection**, **URLEncoder**, etc.

El contenido completo de **java.net** en la J2SE 1.2 es el siguiente:

Clases:
Authenticator
ContentHandler
DatagramPacket
DatagramSocket
DatagramSocketImpl
HttpURLConnection
InetAddress
JarURLConnection
MulticastSocket
NetPermission
PasswordAuthentication
ServerSocket
Socket
SocketImpl
SocketPermission
URL
URLClassLoader
URLConnection
URLDecoder
URLEncoder
URLStreamHandler

Excepciones:
BindException
ConnectException

MalformedURLException
NoRouteToHostException
ProtocolException
SocketException
UnknownHostException
UnknownServiceException

Interfaces:
ContentHandlerFactory
FileNameMap
SocketImplFactory
SocketOptions
URLStreamHandlerFactory

La jerarquía de clases del paquete **java.net** se detalla en la siguiente figura:

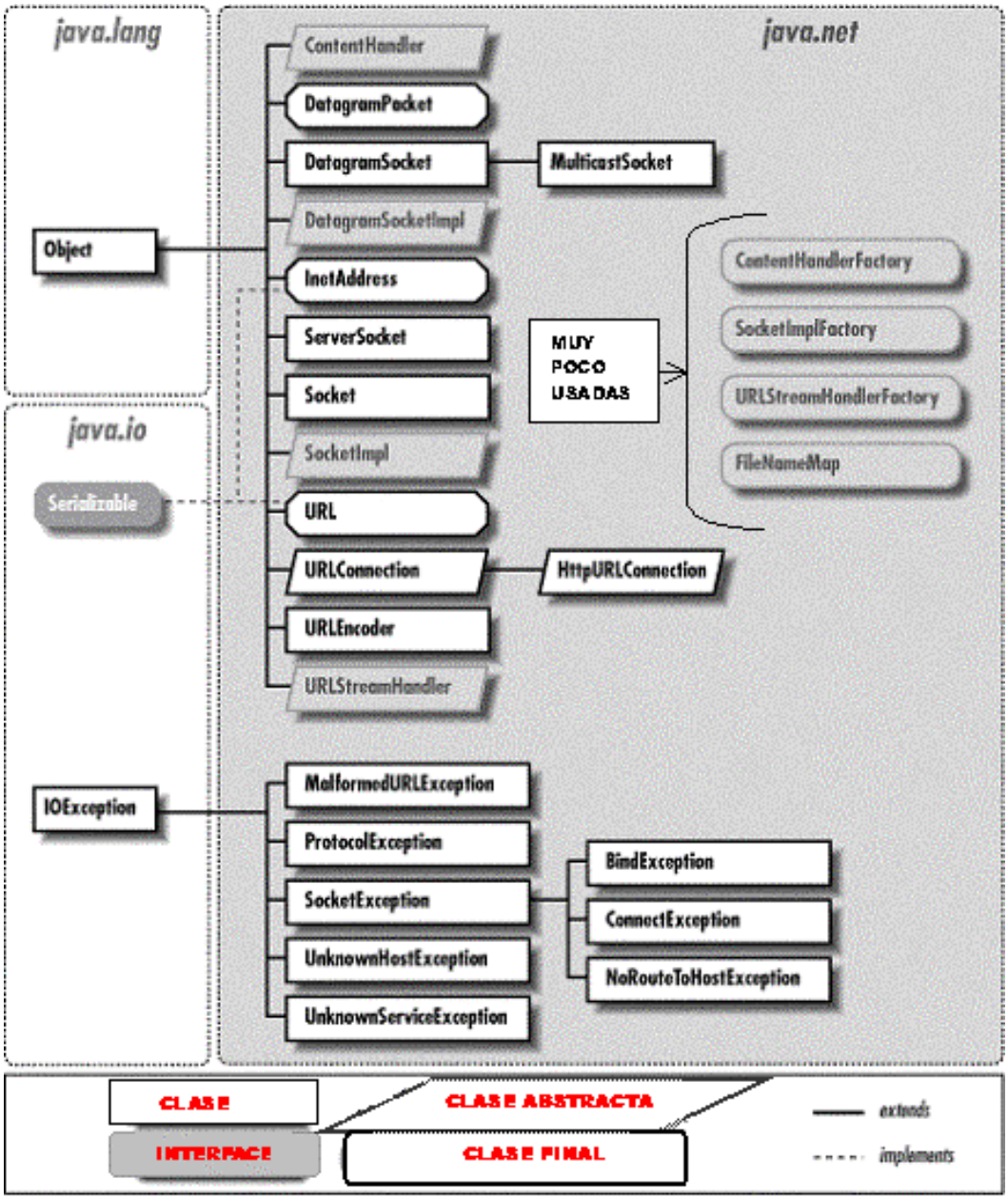

Figura 19. Esquema de clases del paquete java.net

El paquete **java.net** permite trabajar con los protocolos TCP y UDP. La clase **java.net.Socket** permite crear sockets TCP para el cliente; la clase **java.net.ServerSocket** hace lo mismo para el servidor. Para las comunicaciones UDP, Java ofrece la clase **java.net.DatagramSocket** para los dos lados de una comunicación UDP, y la clase **java.net.DatagramPacket** para crear datagramas UDP.

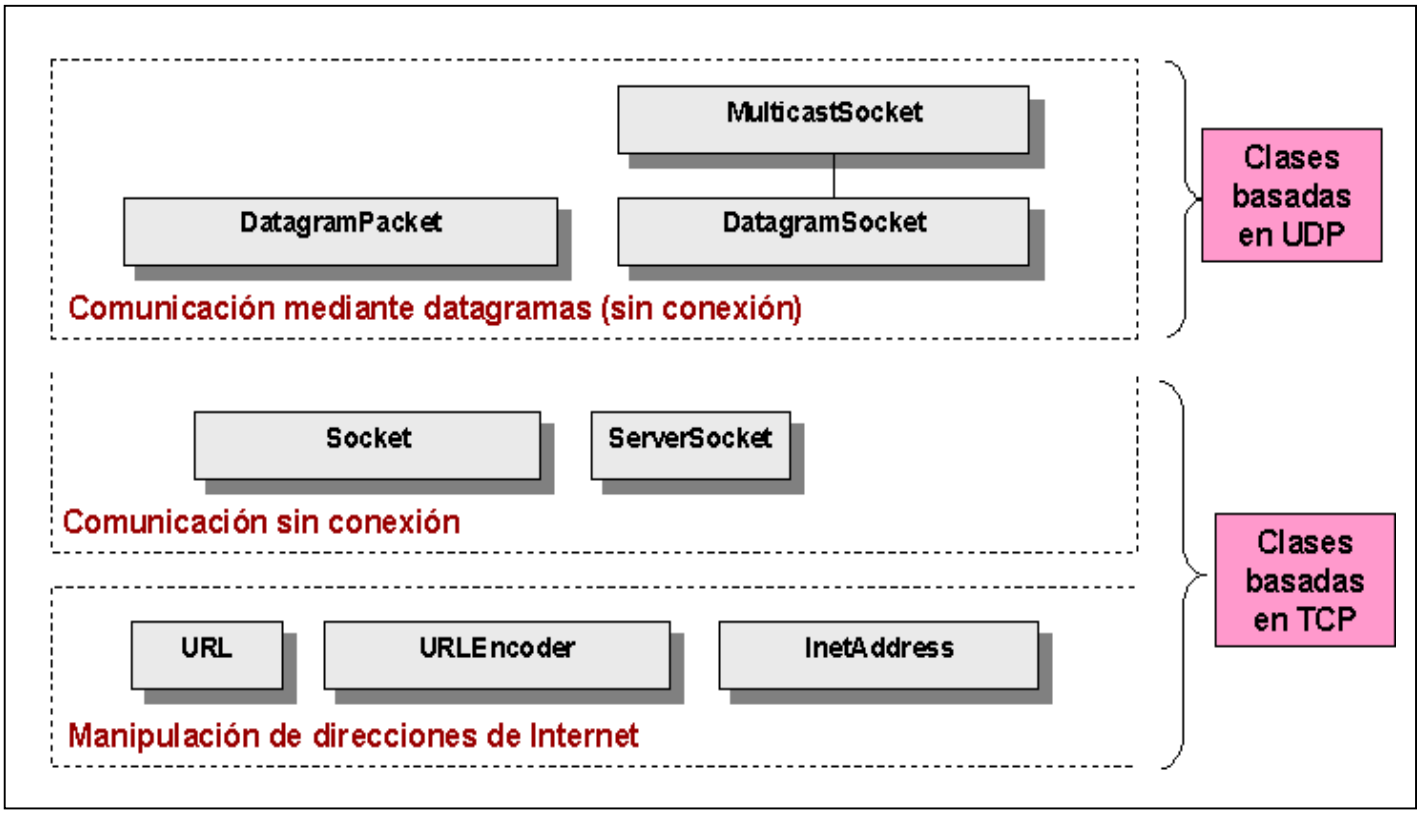

Figura 20. Esquema de las clases basadas en TCP y en UDP

El estudio detallado de cada una de las clases de **java.net** cae fuera del ámbito del tutorial: aquí se expondrán solamente las imprescindibles para el funcionamiento del *chat* del apartado 4 (correspondientes al protocolo TCP).

3.2 La clase java.net.ServerSocket

Esta clase es utilizada por el servidor para recibir conexiones de los clientes mediante el protocolo TCP. Toda aplicación que actúe como servidor creará una instancia de esta clase y llamará a su método *accept()*; la llamada hará que la aplicación se bloquee (esto es, que permanezca esperando) hasta que llegue una conexión por parte de algún cliente. Cuando suceda esto, el método *accept()* creará una instancia de la clase **java.net.Socket** que se usará para comunicarse con el cliente. Nada impide que un servidor tenga un único objeto *ServerSocket* y muchos objetos *Socket* asociados (cada uno a un cliente concreto).

Un objeto *ServerSocket* es siempre un socket pasivo. En la documentación de Sun y en muchos libros se considera que los *ServerSockets* son “sockets de servidor” (*server sockets*). A mi juicio, esta identificación resulta ambigua: un socket de servidor es un socket que reside en el servidor, ya sea pasivo (*ServerSocket*) o activo (*Socket*). No obstante mi opinión, usaré en lo que sigue “socket de servidor” para referirme a un objeto

ServerSocket; pues prefiero no discrepar, en un trabajo de divulgación, de la terminología aceptada.

Debo señalar también que Brucke Eckel considera equívoco el término *ServerSocket*, pues las clases **java.net.ServerSocket** y **java.net.Socket** no están relacionadas por herencia ni descienden de una superclase común.

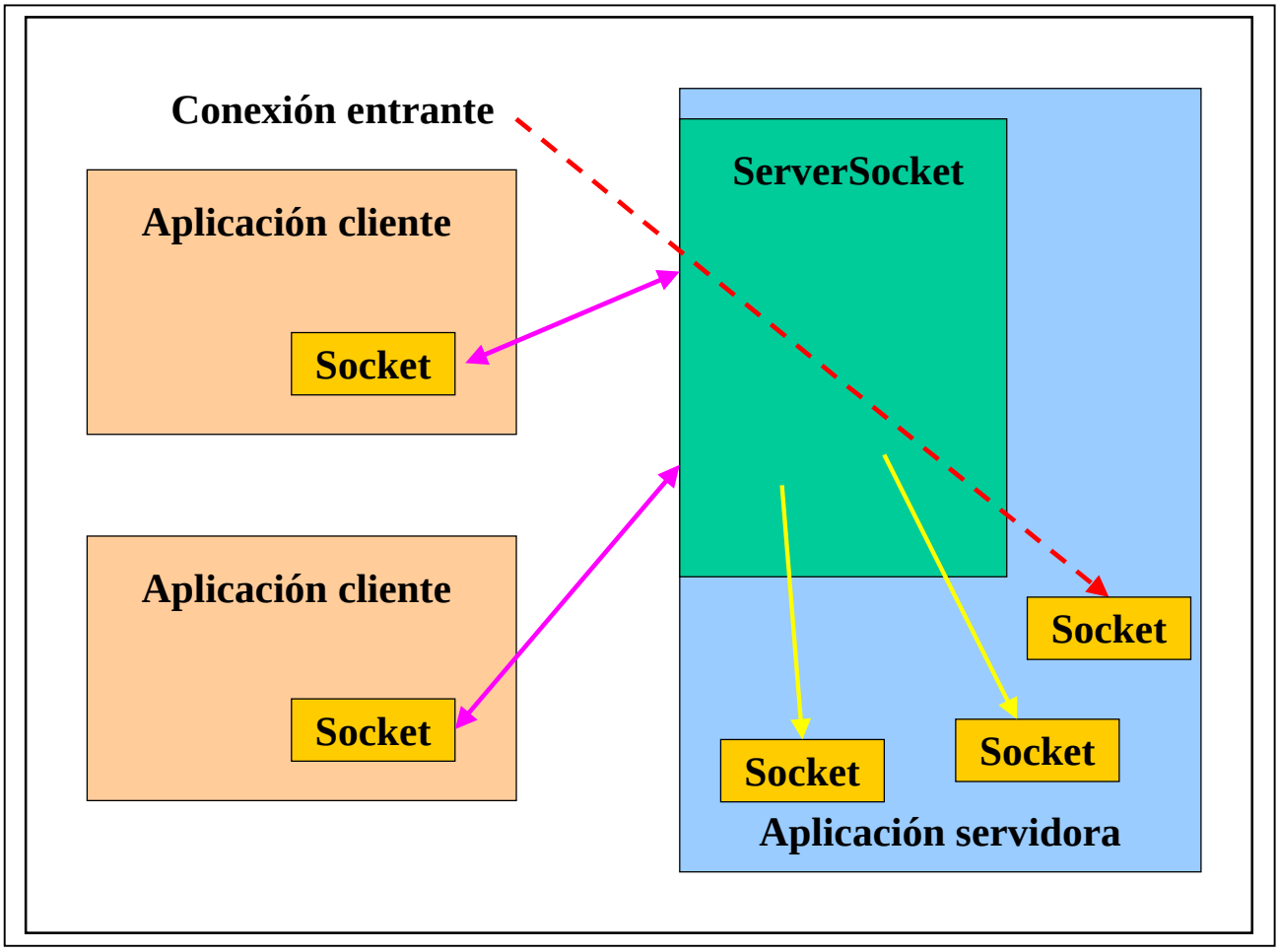

Figura 21. Las clases Socket y ServerSocket en funcionamiento

Los constructores más habituales de la clase **ServerSocket** son éstos:

```
public ServerSocket (int puerto) throws IOException
public ServerSocket (int puerto, int longitudCola) throws IOException
public ServerSocket (int puerto, int longitudCola, InetAddress dirInternet) throws IOException
```

El argumento *puerto* es el número del puerto utilizado por el socket del servidor para escuchar las peticiones TCP de los clientes; el entero *longitudCola* es la longitud máxima de la cola para conexiones entrantes; por último, *dirInternet* es un objeto *InetAddress*.

Resulta importante resaltar que la cola de conexiones entrantes es de tipo FIFO (*First In, First Out*: primero en entrar, primero en salir) y que su longitud máxima viene determinada por el sistema operativo. Es decir: si el argumento *longitudCola* supera el tamaño máximo de la cola establecida por el sistema operativo, se usará en realidad este último valor, sin que se genere ningún aviso para el programador. En el caso de que la cola

esté llena, el objeto *ServerSocket* rechazará nuevas conexiones hasta que surjan huecos en ella.

Las instancias de la clase **java.net.InetAddress** representan direcciones IP. Esta clase no tiene constructores públicos. Para obtener un objeto *InetAddress*, hay que llamar al método estático *byName()* con un argumento de tipo *String* que representa el nombre de un nodo ("www.uv.es", por ejemplo) o una dirección IP ("12.4.234.11", por ejemplo). **InetAddress** proporciona un método –*getHostName()*– para averiguar el nombre del ordenador local. Los dos siguientes ejemplos servirán para ilustrar el uso de esta clase (el primero requiere conexión a Internet):

InformarConexionUV.java

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

// Proporciona información sobre la conexión a la página web de la
// Universidad de Valencia.

public class InformarConexionUV {

   public static void main(String [] args) {
      Socket socket = null;
      try {

         InetAddress direccion = InetAddress.getByName("www.uv.es");
         System.out.println(direccion);
         socket = new Socket(direccion, 80);
         System.out.println ("Conectado a " + socket.getInetAddress());
         System.out.println ("por el puerto " + socket.getPort());
         System.out.println ("desde el puerto local " + socket.getLocalPort());
         System.out.println ("y desde la dirección local " + socket.getLocalAddress());
      }
      catch (UnknownHostException e1) {
         System.out.println ("No se pudo encontrar una máquina con ese nombre.");
      }
      catch (SocketException e2) {
         System.out.println ("No se pudo conectar con la máquina por el puerto establecido.");
      }
      catch (IOException e3) {
         System.out.println (e3);
      }
      finally {
         try {
            socket.close();
         }
         catch (IOException e4) {
            // No se hace nada: no se pudo cerrar el socket.
         }
      }
   }

}
```

DireccionLocal.java

```
import java.net.*;

public class DireccionLocal {

   public static void main (String args[]) {
      try {
         InetAddress direccion = InetAddress.getLocalHost();
         System.out.println(direccion);
      }
      catch (UnknownHostException e) {
         System.out.println("Error al intentar obtener el nombre de la máquina local.");
      }
   }

}
```

En mi equipo, éste es el resultado de ejecutar el anterior código:

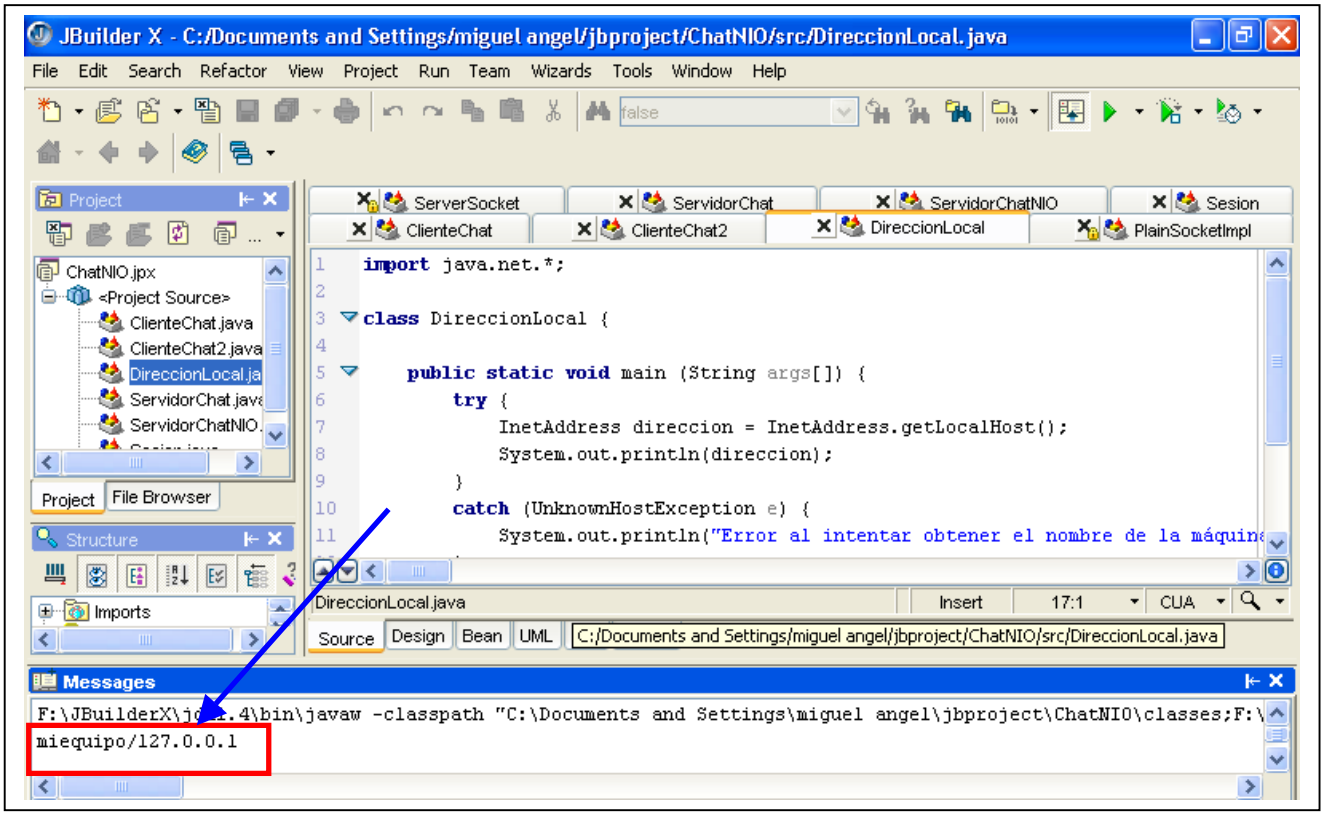

Figura 22. Ejemplo de uso de la clase InetAddress

La dirección IP 127.0.0.1 es una dirección especial que representa al ordenador local, esté o no conectado a Internet. Para que sea válida, el ordenador debe tener instalada y configurada una implementación de la pila de protocolos TCP/IP (los modernos sistemas operativos suelen ocuparse de esto; en Windows 3.1, por ejemplo, había que instalar por separado el software de TCP/IP).

Volviendo a la clase **ServerSocket**, la siguiente tabla indica los métodos más usados:

Método	Descripción
accept()	Escucha conexiones a este socket de servidor y las acepta
getInetAddress()	Devuelve la dirección IP local del socket de servidor
getLocalPort()	Devuelve el puerto por el que escucha el socket de servidor
close()	Cierra el socket de servidor

El método *public Socket **accept**() throws IOException* espera conexiones. Una vez recibida una, el objeto *ServerSocket* la acepta y crea un objeto *Socket*. En ausencia de conexiones, el *ServerSocket* permanece bloqueado. Al intentar crear el objeto *Socke*t, pueden producirse excepciones debidas a la configuración de seguridad del ordenador donde reside el *ServerSocket*.

El método *public InetAddress **getInetAddress**()* devuelve un objeto *InetAddress* que contiene la dirección IP del ordenador al cual está conectado el socket. Si el socket de servidor ya no está conectado, devuelve *null*.

El método *public int **getLocalPort**()* devuelve el puerto por el cual el socket del servidor permanece a la escucha de posibles peticiones de clientes.

El método *public void **close**() throws IOException* cierra el socket de servidor y libera los recursos asociados. Si la operación no es posible (por ejemplo, porque ya ha sido cerrado) se lanza una excepción *IOException*.

Curiosamente, no existen métodos en esta clase para extraer los flujos de datos de entrada y de salida: es necesario usar el método *accept()* y aplicar los métodos *getInputStream()* y *getOutputStream()* de la clase **Socket** –que se verá a continuación– sobre el socket devuelto.

Ejemplo:

```
// Se ha seleccionado el puerto 9000 para escuchar las
// peticiones de los clientes
ServerSocket socketservidor = new ServerSocket(9000);

// Se creará un socket cuando un cliente haga una petición
Socket socketcliente = socketservidor.accept();
```

En el caso de que un objeto *ServerSocket* intente escuchar peticiones de los clientes por un puerto ocupado por otro *ServerSocket*, se producirá una excepción **java.net.BindException**.

Nota: Un **flujo** (*stream*) es una secuencia de datos de longitud indeterminada.
En Java, un flujo está formado por una secuencia ordenada de bytes, que pueden representar caracteres, datos primitivos, objetos, etc. Los flujos de entrada (*input streams*) se usan para leer datos de una fuente de entrada, como un fichero, un periférico o la memoria. Los flujos de salida (*output streams*) se emplean para enviar datos a consumidores de datos; por ejemplo, un fichero o una conexión de red.

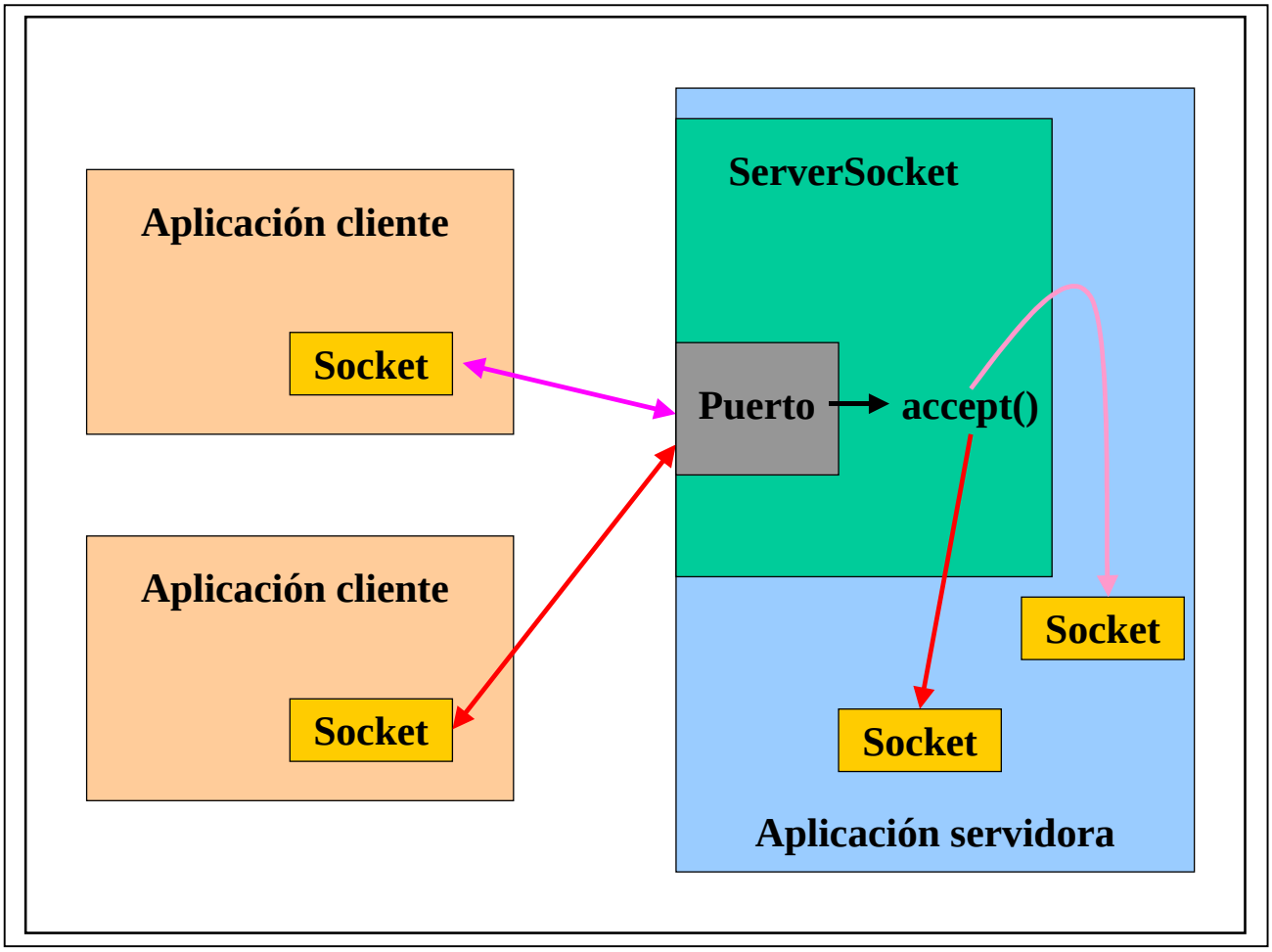

Figura 23. El método accept() en marcha

3.3 La clase java.net.Socket

Esta clase implementa sockets TCP activos (capaces de recibir y transmitir datos). Sus constructores son

protected Socket () throws SocketException
protected Socket (SocketImpl impl) throws SocketException
public Socket (String host, int puerto) throws UnknownHostException, IOException
public Socket (InetAddress direccion, int puerto) throws IOException
public Socket(String host, int puerto, InetAddress dirLocal, int puertoLocal) throws IOException
public Socket (InetAddress direccion, int puerto, InetAddress dirLocal, int puertoLocal) throws IOException
public Socket (String host, int puerto, boolean flujo) throws IOException
public Socket (InetAddress host, int puerto, boolean flujo) throws IOException

El constructor *public **Socket** (String host, int puerto) throws UnknownHostException, IOException* es uno de los más usados. Este método crea un objeto *Socket* y lo conecta con el puerto con número *puerto* del ordenador cuyo nombre es *host*. En el caso de que se haya instalado algún gestor de seguridad para la máquina virtual Java, se llama al método *public void **checkConnect**(String host, int puerto, Object contexto) throws SecurityException, NullPointerException* de la clase **java.lang.SecurityManager** para averiguar si la operación está permitida. Si no es así, se lanza una excepción **java.lang.SecurityException**. Si no se puede localizar el ordenador con nombre *host*, se lanza una excepción **java.net.UnknownHostException**.

No hay que olvidar que el argumento *puerto* corresponde a la máquina de destino. Generalmente, los programadores no especifican el puerto local vinculado al socket, que es asignado por la máquina virtual Java: la MVJ pregunta al sistema operativo cuál es el primer puerto libre y lo usa para el socket. De todos modos, hay dos constructores que permiten especificar el puerto local (*puertoLocal*). Si no estuviera libre, se lanzaría una excepción.

Los métodos más usados de esta clase aparecen en la siguiente tabla:

Método	Descripción
getInetAddress()	Devuelve la dirección *InetAddress* a la cual está conectado el socket
getPort()	Devuelve el número de puerto al que está conectado el socket
getLocalAddress()	Devuelve la dirección local a la cual está conectado el socket
getLocalPort()	Devuelve el número de puerto local al cual está conectado el socket
getInputStream()	Devuelve un flujo de entrada para el socket
getOutputStream()	Devuelve un flujo de salida para el socket
close()	Cierra el socket

El método *public InetAddress **getInetAddress**()* devuelve la dirección IP remota (en forma de un objeto *InetAddress*) a la que está conectado el socket.

El método *public int **getPort**()* devuelve el puerto remoto al cual está conectado el socket.

El método *public InetAddress **getLocalAddress**()* devuelve la dirección IP remota (en forma de un objeto *InetAddress*) a la que está conectado el socket.

El método *public int **getLocalPort**()* devuelve el puerto local al cual está conectado el socket.

El método *public void **close**() throws IOException* cierra el socket y libera los recursos asociados. Si la operación no es posible (por ejemplo, porque ya ha sido cerrado), se lanza una excepción *IOException*. Al cerrar un socket, también se cierran los flujos de entrada y de salida asociados.

El método *public InputStream **getInputStream**() throws IOException* devuelve un flujo de entrada para el socket. Con él, una aplicación puede recibir información procedente de la máquina de destino (es decir, del otro lado de la conexión).

El método *public OutputStream **getOutputStream**() throws IOException* devuelve un flujo de salida para el socket, que puede usarse para enviar información a la máquina de destino (es decir, al otro lado de la conexión).

Para un socket activo, los dos últimos métodos son imprescindibles: de algún modo debe el socket enviar y recibir datos a través de la conexión TCP. El paquete **java.net** recurre al paquete **java.io**, en concreto, a las clases **java.io.InputStream** y **java.io.OutputStream**. Estas dos clases tratan los datos como bytes y proporcionan métodos sencillos para leer y escribir bytes y arrays de bytes. Asimismo, sus subclases pueden "envolverse" en clases como **java.io.Reader** y **java.io.Writer**, que transmiten datos en forma de caracteres Unicode, no de bytes.

Cuando se trabaja con **java.net**, hay tres clases de **java.io** que suelen usarse como envoltorio para los objetos devueltos por *getOutputStream()* y *getOutputStream()*:

- **BufferedReader.** Los dos métodos más interesantes para nuestros propósitos son *read()* y *readLine()*. El primero devuelve el entero correspondiente al carácter leído (o –1 si se ha alcanzado el final del flujo); el segundo devuelve un *String* que corresponde a una línea de texto. Ambos provocan ***bloqueo***: no terminan de ejecutarse hasta que haya datos disponibles o hasta que se lance una excepción.
- **PrintStream.** Incluye los métodos *print()* y *println()*, que permiten enviar datos primitivos y objetos *String*. El método *write()* permite enviar bytes o arrays de bytes. Los tres métodos bloquean la E/S.
- **PrintWriter.** Es una clase similar a la anterior, e incluye también los métodos *print()* y *println()*. La principal diferencia es que permite enviar caracteres codificados mediante distintas codificaciones (ISO Latin, UTF-8...). Ambos métodos bloquean la E/S.

Ejemplos:

```
// 1 Se crea un socket con el nombre del nodo y el número de puerto 25
Socket socketCliente = new Socket("www.aidima.es", 25);


// 2 Se crea un socket con la dirección IP dada y el puerto 25.
Socket socketCliente = new Socket("26.56.78.140", 25);


// 3 Se crea un socket con la dirección IP dada y el puerto 1025.
Socket socketCliente = new Socket("26.56.78.140", 1025);

// Se obtiene el nombre del ordenador al que pertenece el socket.
System.out.println(socketCliente.getInetAddress());

// Se obtiene el número de puerto asociado al socket.
System.out.println(socketCliente.getPort());


// 4 Se crea un socket con la dirección IP dada y el puerto 25.
Socket socketCliente = new Socket("26.56.78.140", 25);

// Una vez establecida la conexión, se extraen los flujos de E/S asociados al socket.
InputStream entrada = socketCliente.getInputStream();
OutputStream salida = socketCliente.getOutputStream();

// Se lee del servidor un byte mediante el método read().
entrada.read();

// Se envía al servidor un byte mediante el método write().
salida.write(64);

// Se usa la clase PrintWriter como envoltorio de OutputStream para enviar una cadena de
// caracteres al flujo de salida. Luego, se cierra el socket.
PrintWriter pw = new PrintWriter(salida, true);
pw.println("Escribiendo en la salida");
socketCliente.close();


// 5 Se crea un socket con la dirección IP dada y el puerto 25.
Socket socketCliente = new Socket("26.56.78.140", 25);

// Una vez establecida la conexión, se extraen los flujos de E/S asociados al socket.
InputStream entrada = socketCliente.getInputStream();
OutputStream salida = socketCliente.getOutputStream();

// Se usa la clase BufferedReader como envoltorio de InputStream para leer líneas
// completas del flujo de entrada.
BufferedReader br = new BufferedReader(new
                                    InputStreamReader(socketCliente.getInputStream()));

// Se lee una línea completa y se cierra el socket.
br.readLine();
socketCliente.close();
```

La implementación de cualquier programa servidor en Java sigue esta secuencia de pasos:

1. Se crea un objeto *ServerSocket* para escuchar a las peticiones que llegan al puerto asociado al servicio.
2. Cuando se llama al método *accept()*, el socket de servidor permanece a la espera de peticiones de clientes por el puerto.
3. Al llegar una solicitud se siguen tres pasos:
   3.1. Se acepta la conexión, lo cual genera un objeto *Socket* asociado al cliente.
   3.2. Se asocian objetos de las clases contenidas en el paquete **java.io** a los flujos (*streams*) de entrada y salida del socket.
   3.3. Se lee de los flujos y se escribe en ellos; es decir, se leen y procesan los mensajes entrantes y se envían las respuestas a los clientes.
4. Se cierran los flujos.
5. Se cierra el socket vinculado al cliente.
6. El socket de servidor continúa a la espera de nuevas peticiones.

Igualmente, los programas cliente se implementan así:

1. Se crea un objeto *Socket*, que tiene asociado un nodo de destino y un puerto donde se ejecuta el servicio de interés.
2. Se asocian objetos de las clases contenidas en el paquete **java.io** a los flujos (*streams*) de entrada y salida del socket.
3. Se lee de los flujos o se escribe en ellos.
4. Se cierran los flujos.
5. Se cierra el socket.

El cierre de los sockets no debe pasarse por alto: por su naturaleza bidireccional consumen bastantes recursos, tanto de la máquina virtual Java como del sistema operativo.

Un socket se cierra cuando

- finaliza el programa que lo creó;
- se llama a su método *close()*;
- se cierra uno de los flujos de E/S asociados;
- es eliminado por el recolector de basura.

Nota: Si un socket no está en uso pero permanece abierto, no se podrá utilizar el puerto local asociado. Una vez cerrado, el sistema operativo podrá reutilizar el puerto para otro socket.

Confiar en el recolector de basura para cerrar los sockets o los flujos de E/S es práctica poco recomendable. Consideremos, por ejemplo, este método:

```
public void escribirArchivo() {
   try {
      File fichero = new File("C:\temporal.txt");
      BufferedOutputStream salida = new BufferedOutputStream(new FileOutputStream(fichero));
      salida.write( (int) 'A'); // Escribe el byte correspondiente al carácter 'A'
      salida.flush();
   }
   catch (IOException e) {
      e.printStackTrace();
   }
}
```

Como el objeto *salida* no se cierra explícitamente, quedará marcado como posible "presa" del recolector de basura cuando termine el método. Cuando el recolector de basura lo "cace", lo cerrará y liberará su memoria. Al cerrarse el objeto *salida*, también se cerrará el *FileOutputStream*. Perfecto. Es lo que se quiere, ¿verdad? ¿Para qué hay que preocuparse de cerrar los flujos? Hay varios motivos para rechazar el código anterior y mandarlo a un infierno mucho peor que el que sufren los objetos eliminados por el recolector de basura:

a) Hasta que el recolector de basura actúe, el archivo permanecerá abierto. Si se intentara borrarlo o volverlo a abrir, se arrojaría una excepción que indicaría que el archivo está abierto y en uso.

b) La tarea del recolector de basura consiste en liberar memoria de los objetos Java que no se usan, no liberar recursos del sistema operativo (estructuras de datos, etc.). La única manera "limpia" de liberar recursos del SO usados por los objetos Java es usar métodos –*close()*, *dispose()*– que se encarguen de llamar al código de limpieza escrito específicamente para cada plataforma (suele estar escrito en C o C++).

Algunos programadores se olvidan de cerrar los sockets y los flujos cuando terminan de usarlos y colocan el código de limpieza dentro del método *finalize()* (este método es llamado por el recolector de basura cuando se dispone a borrar algún objeto para el cual no existen referencias). Así, los recursos del SO se liberarán cuando el objeto sea eliminado por el recolector de basura. En general, esta práctica resulta nefasta.

Supongamos, por ejemplo, un programa que abre muchos archivos y no los cierra. Puede suceder que el límite de ficheros abiertos admitido por el SO se alcance sin que el recolector de basura haya actuado –y, por ende, sin que se haya llamado a *finalize()*, donde reside el código de limpieza–; pues el recolector sólo empieza a ponerse nervioso y a afilarse las uñas cuando queda poca memoria en la pila, no cuando quedan pocos recursos libres para el sistema operativo. En este caso, el programa se colgaría o sería incapaz de abrir más archivos.

Aparte, si se confía en el recolector de basura y en la limpieza de recursos dentro de *finalize()*, el comportamiento del programa será diferente en cada implementación de la MVJ, pues cada una tiene sus técnicas para el recolector de basura: en las que el recolector fuera muy activo, funcionaría bien; en las que no, podría provocar fallos inexplicables a simple vista. La peor situación se daría en las implementaciones sin recolector de basura (la especificación de la MVJ establece claramente que el recolector de basura es potestativo), típicas de los dispositivos pequeños.

Los problemas derivados de no cerrar los sockets empeoran los correspondientes a mantener abiertos los flujos de E/S asociados a archivos: los sockets mantienen una conexión bidireccional que atraviesa las redes que median entre las máquinas donde están. Para mantenerla, se usan muchos recursos del sistema operativo y se produce un constante envío de datagramas IP entre las máquinas. No cerrar los sockets cuando se ha terminado de usarlos constituye un claro desperdicio de recursos.

Cuando un socket no se cierra correctamente porque la aplicación ha fallado, el sistema operativo puede tardar minutos en conseguir cerrarlo. Dependiendo de la MVJ y del SO, la “desaparición” de un socket sin hacer antes un *close()* puede ser interpretada por el socket del otro extremo como una falta de envío de datos, y este último socket puede colgarse hasta que se cierre el programa.

Nota: De un *socket* cerrado puede obtenerse su *InetAddress*, su puerto, su dirección local y su puerto local.

Nota: Desde la versión 1.3 de la J2SE se puede desactivar el flujo de entrada de un socket sin desactivar el de salida, y viceversa. Los métodos *public void shutdownInput() throws IOException* y *public void shutdownOutput() throws IOException* se utilizan para ese propósito. Estos métodos rompen el carácter bidireccional de las comunicaciones TCP

Ambos se pueden usar en conexiones de tipo HTTP: el cliente hace una petición y llama a *shutdownOutput()*. El servidor interpreta esto como que el cliente ha acabado su petición, pero que puede recibir datos.

3.4 Un ejemplo de aplicación cliente-servidor con java.net

Para mostrar la implementación en Java de los pasos para escribir aplicaciones cliente-servidor, incluyo este ejemplo:

RegistroConexiones.java

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

/**
 * Esta clase envía un breve mensaje a los clientes que se conectan y cierra la conexión.
 * No puede atender a la vez a más de un cliente. Si hay algún error al intentar enviar el mensaje
 * al cliente (por ejemplo, porque se ha cerrado tras conectarse), la aplicación se cierra.
 */
public class RegistroConexiones {

   public static void main(String args[]) {
      ServerSocket socketServidor = null;
      Socket socketCliente = null;
      PrintWriter salida = null;

      // Se crea el socket de servidor en el puerto 4000
      try {
         socketServidor = new ServerSocket(4000);
      }
      catch (IOException e1) {
         System.out.println("No se ha podido arrancar el servidor.");

         // Se intenta cerrar el socket de servidor.
         if (socketServidor != null)
            try {
               socketServidor.close();
            }
            catch (IOException e2) {
               // No se hace nada
            }
         System.exit(-1);
      }

      while (true) { // bucle infinito
         try {
            // Se aceptan peticiones de los clientes.
            socketCliente = socketServidor.accept();

            // Se abre un flujo de salida.
            salida = new PrintWriter(socketCliente.getOutputStream());

            // Se muestra información sobre la conexión entrante y se envía un mensaje al cliente.
            System.out.println("Conexión del cliente con dirección " +
                  socketCliente.getInetAddress().getHostAddress() + " por el puerto " +
                  socketCliente.getPort());
            salida.println("Hola y adiós");
            salida.close();
```

```
            // Se cierra el socket.
            socketCliente.close();
        }
        catch (IOException e3) {
            if (salida != null) {
                salida.close();
            }
            if (socketCliente != null) {
                try {
                    socketCliente.close();
                }
                catch (IOException e4) {} // No se hace nada
            }
            if (socketServidor!= null) {
                try {
                    socketServidor.close();
                }
                catch (IOException e5) {} // No se hace nada
            }
            e3.printStackTrace();
            System.exit(-1); // Se sale del programa.
        }
    }
  }

}
```

¿Y el programa cliente? En este caso no se necesita escribirlo: basta usar un navegador web (Figura 24) o hacer un *telnet* (Figura 25).

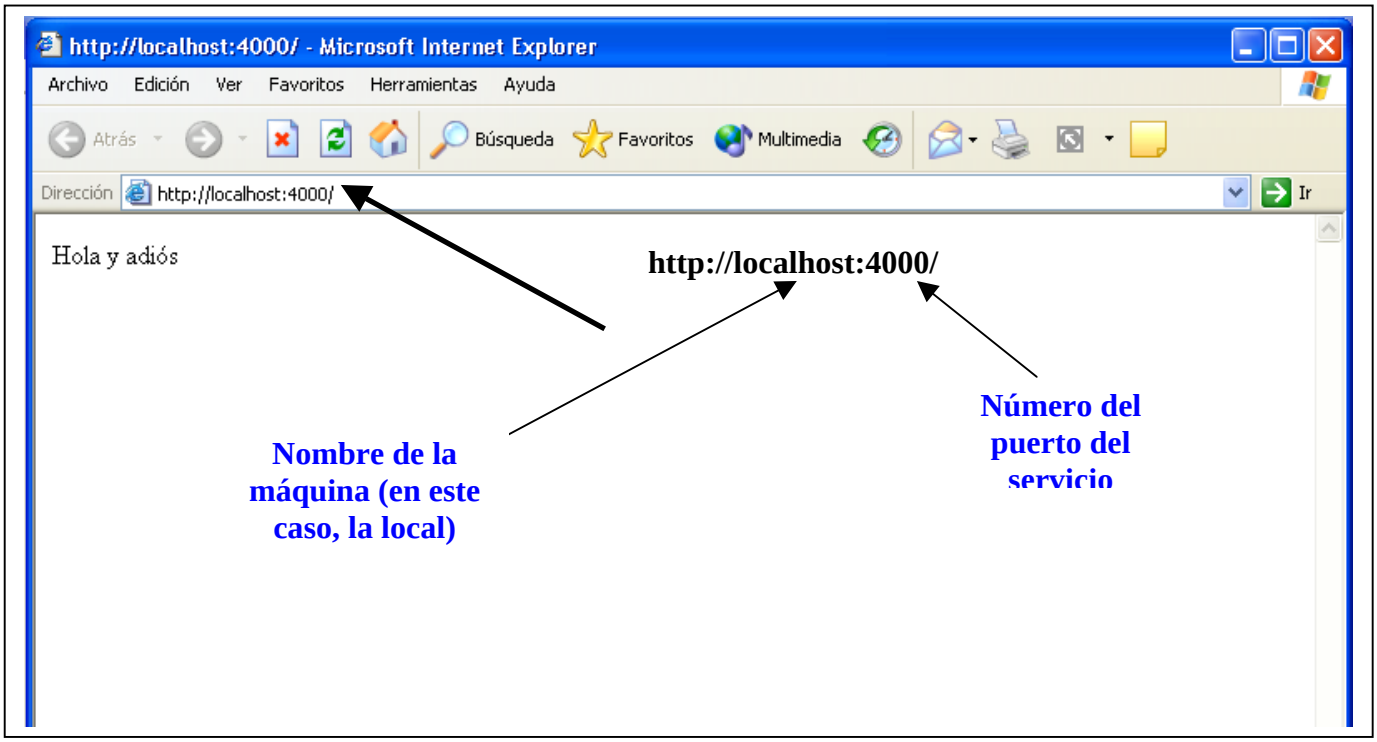

Figura 24. Un cliente del servicio de registro de conexiones

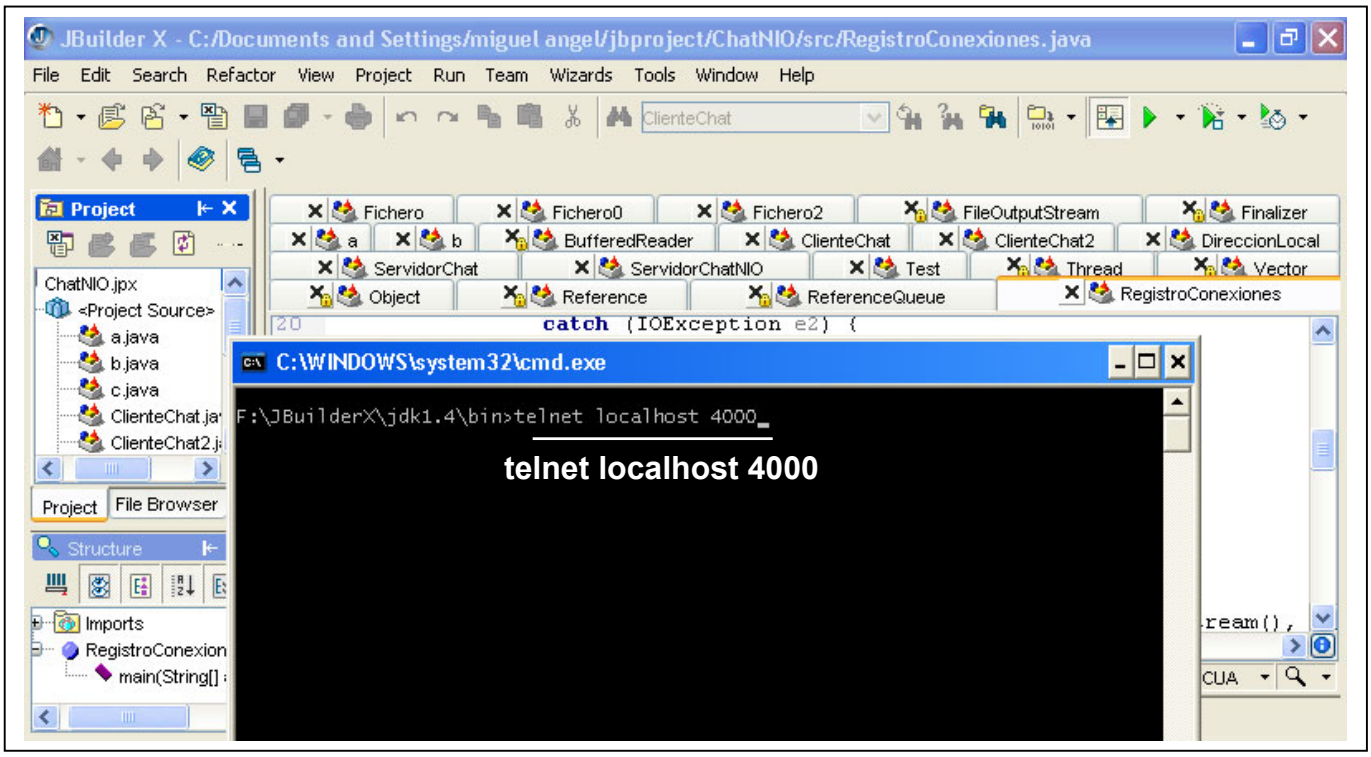

Figura 25. Otro cliente del servicio de registro de conexiones

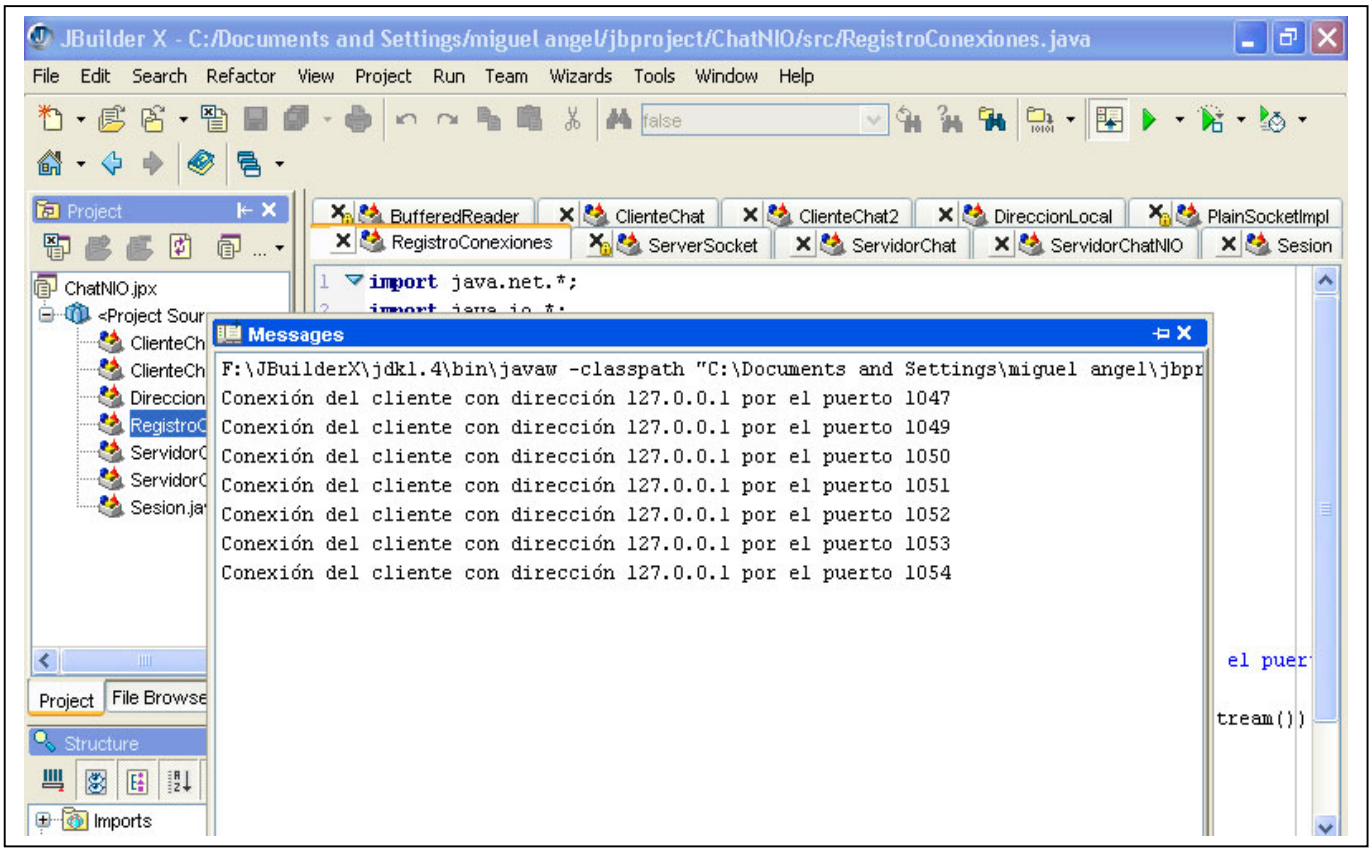

Figura 26. El servicio de conexiones en funcionamiento

El ejemplo de la clase **RegistroConexiones** no es aceptable en ninguna aplicación profesional: sólo es capaz de atender simultáneamente a un cliente. Mientras no acabe con uno, no podrá comenzar con el siguiente.

La solución más sencilla al problema es utilizar hilos (*threads*). Un hilo no es más que un flujo de control dentro de una aplicación. En el programa anterior, si se asigna un hilo a cada cliente que se conecte, el programa principal podrá seguir aceptando conexiones. Veámoslo con código:

RegistroConexionesHilos.java

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

/**
 * Versión con hilos de RegistroConexiones.
 * Esta clase envía un breve mensaje a los clientes que se conectan y cierra la conexión.
 * Puede atender simultáneamente a varios clientes. Si hay algún error al intentar enviar el mensaje
 * al cliente (por ejemplo, porque se ha cerrado tras conectarse), se cierra el hilo correspondiente a
 * ese cliente, pero no la aplicación.
 */
public class RegistroConexionesHilos {

    public static void main(String args[]) {
        ServerSocket socketServidor = null;
        Socket socketCliente = null;
        PrintWriter salida = null;

        // Se crea el socket de servidor en el puerto 4000
        try {
            socketServidor = new ServerSocket(4000);
        }
        catch (IOException e1) {
            System.out.println("No se ha podido arrancar el servidor.");

            // Se intenta cerrar el socket de servidor.
            if (socketServidor != null)
                try {
                    socketServidor.close();
                }
                catch (IOException e2) {
                    // No se hace nada
                }
            System.exit(-1);
        }

        while (true) { // bucle infinito
            try {
                // Se aceptan peticiones de los clientes.
                socketCliente = socketServidor.accept();
                new ThreadCliente(socketCliente);
            }
            catch (IOException e3) {
```

```
                if (socketCliente != null) {
                    try {
                        socketCliente.close();
                    }
                    catch (IOException e4) {} // No se hace nada
                }
            }
        } // fin while
    }

}


class ThreadCliente extends Thread {

    private Socket socketCliente;

    public ThreadCliente(Socket socket) {
        socketCliente = socket;
        start(); // Se arranca el hilo.
    }

    public void run() {
        PrintWriter salida = null;
        try {
            // Se abre un flujo de salida.
            salida = new PrintWriter(socketCliente.getOutputStream());

            // Se muestra información sobre la conexión entrante y se envía un mensaje al cliente.
            System.out.println("Conexión del cliente con dirección " +
                               socketCliente.getInetAddress().getHostAddress() +
                               " por el puerto " +
                               socketCliente.getPort());
            salida.println("Hola y adiós");
            salida.close();

            // Se cierra el socket.
            socketCliente.close();
        }
        catch (IOException e1) {
            if (salida != null) {
                salida.close();
            }
            if (socketCliente != null) {
                try {
                    socketCliente.close();
                }
                catch (IOException e2) {} // No se hace nada
            }
            e1.printStackTrace();
        }
    }

}
```

4. UN CHAT POR CONSOLA

En este apartado se presenta el código necesario para crear una aplicación de charla electrónica mediante consola, en el que se han utilizado los conceptos, clases y métodos expuestos antes. Esta aplicación corresponde con exactitud a una arquitectura cliente-servidor: los usuarios del *chat* (los clientes) se conectan al servicio de *chat* (el servidor) y, luego, intercambian mensajes entre sí. El protocolo utilizado es TCP.

Para centrar la atención en las clases y métodos utilizados, he evitado en el código de este apartado el uso de una interfaz gráfica. La aplicación de charla electrónica permite el intercambio de mensajes entre los clientes a través de la consola (véase la figura 27). *ServidorChat.java* actúa como servidor, mientras que *ClienteChat.java* actúa como cliente. Como es de suponer, *ServidorChat.java* debe estar en ejecución antes de arrancar *ClienteChat.java*.

El programa servidor genera un archivo de texto plano a modo de historial en el directorio raíz del ordenador servidor (*historial.txt*), archivo donde figura la fecha y hora de cada conexión, así como el nombre del nodo cliente y el puerto asociado, y la fecha y hora del cierre de cada conexión (véase la figura 28). Si el archivo de historial no se puede generar o aparecen problemas al acceder a él (por ejemplo, por cuestiones de asignación de permisos), se lanzará un mensaje de aviso para el usuario (véase la figura 29), pero se podrá usar el programa.

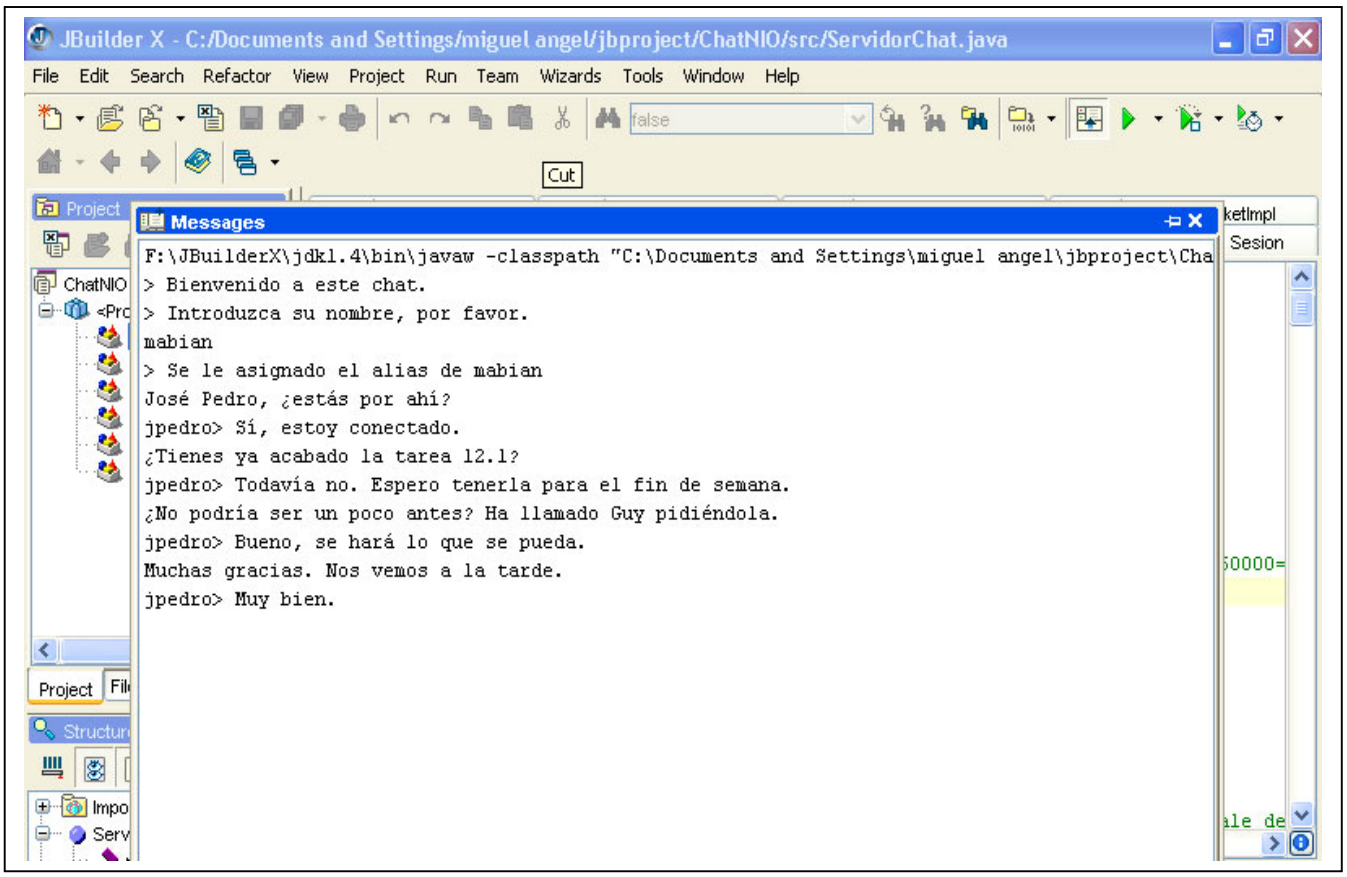

Figura 27. Ejemplo de conversación por el chat

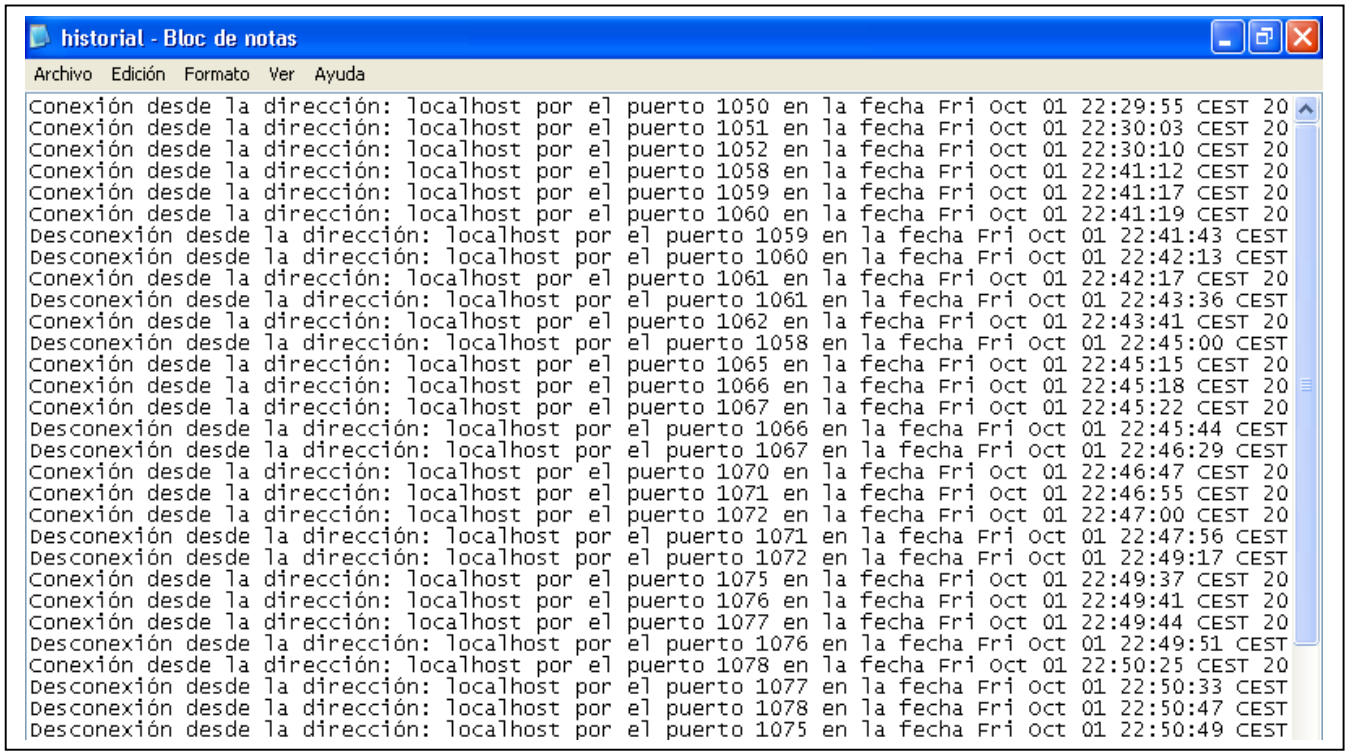

Figura 28. Ejemplo del archivo historial.txt, correspondiente a la depuración del programa en modo local

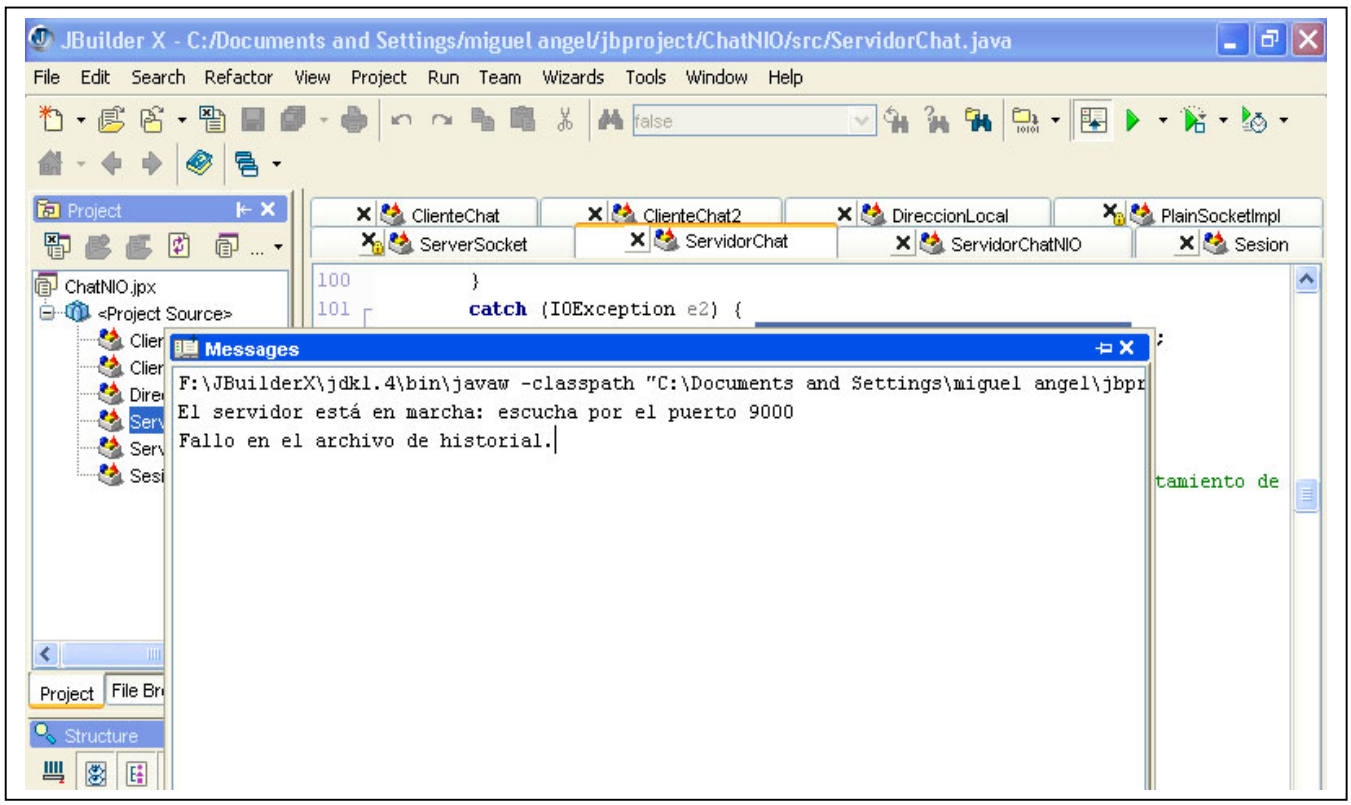

Figura 29. Mensaje de fallo al escribir en el archivo de historial o al acceder a él.

Cuando un usuario se conecta para charlar, puede introducir un nombre para identificarse. Si está repetido, se toma como identificador el nombre introducido, concatenado con el número de puerto que se le asigna en el servidor. Si no introduce ningún nombre, se le asigna "Invitado" como identificador.

Los usuarios pueden usar la orden *DESCONECTAR* para cerrar la conexión y la orden *LISTAR* para saber qué personas están en línea (véase la figura 30).

Por omisión, se ha asignado un tiempo máximo de inactividad de diez minutos. Si el programa servidor no recibe de un usuario ningún texto en diez minutos, envía un mensaje de cierre de conexión y la cierra (véase la figura 31). Inmediatamente después, el programa cliente se cerrará.

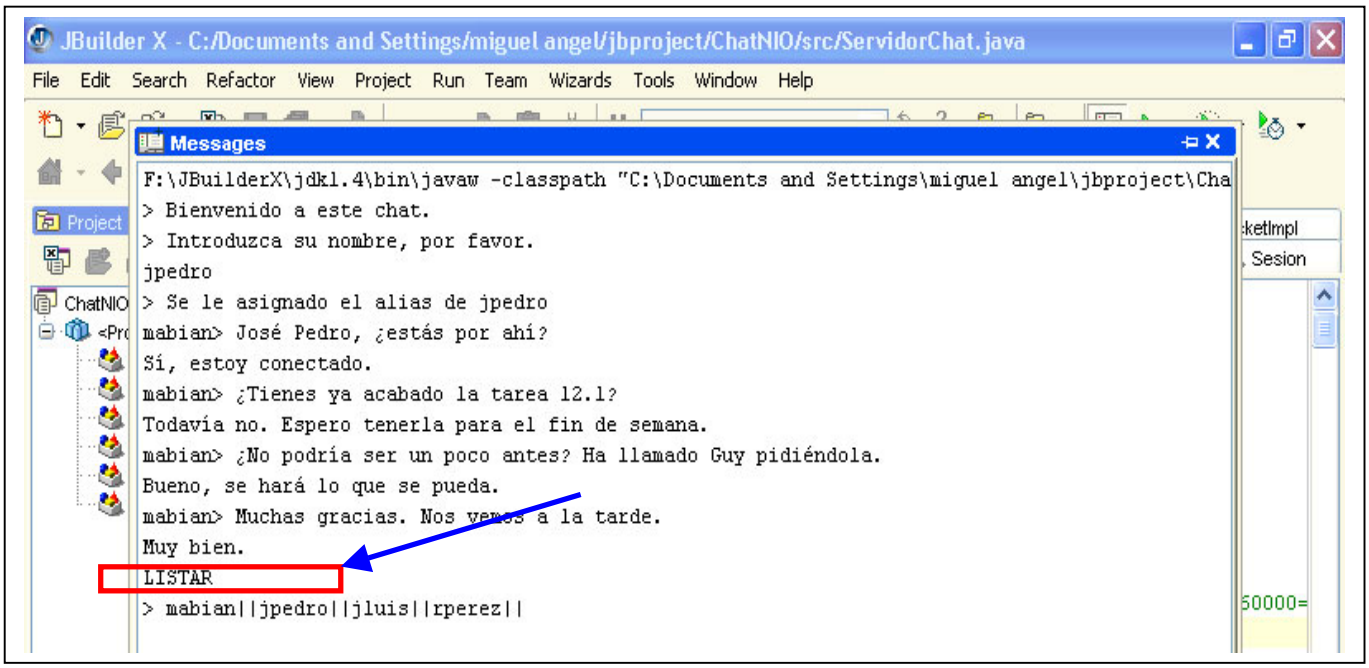

Figura 30. Ejemplo de uso de la orden LISTAR

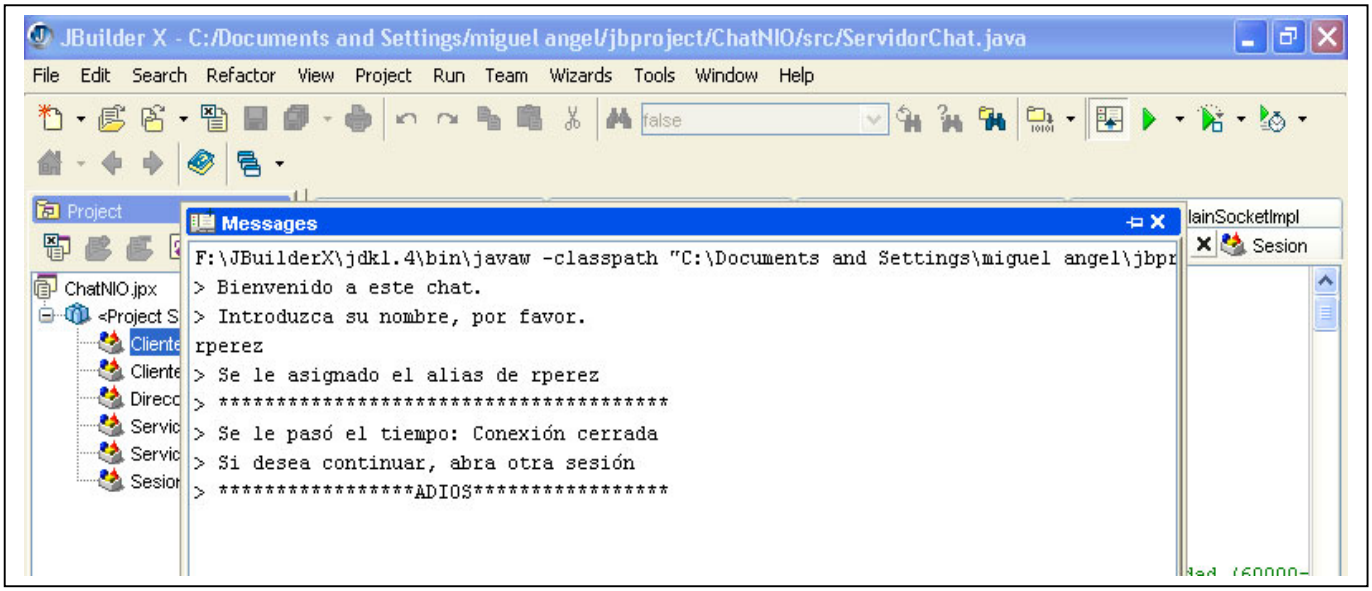

Figura 31. Ejemplo de desconexión por inactividad

Visto el funcionamiento de la aplicación de charla, llega ya el momento de dar el código.

> **Nota:** El lector que desee recibir las clases Java del *chat* puede remitirme un correo electrónico a la dirección **mabian ARROBA aidima PUNTO es** con el asunto "Código chat".

ServidorChat.java

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


/**
 * Esta clase actúa como servidor para el chat. El puerto que se usa para escuchar peticiones de
 * los clientes es PUERTO. Asigna un hilo a cada cliente que se conecta.
 * Cada hilo hace lo siguiente:
 * <p>  1) espera datos de entrada; </p>
 * <p>  2) cuando los recibe, los envía a todos los clientes conectados (excepto a él mismo); </p>
 * <p>  3) vuelve a esperar datos. </p>
 *
 * Los clientes pueden enviar el texto DESCONECTAR para abandonar el chat, y pueden saber
 * los alias de los clientes conectados enviando LISTAR.
 * Si el servidor no recibe ningún mensaje de un cliente en
 * TIEMPO_DESCONEXION_AUTOMATICA milisegundos, lo desconectará automáticamente.
 */
public class ServidorChat {

   private static final int PUERTO= 9000; // Puerto por omisión: no conviene que sea < 1024.

   // Tiempo de desconexión automática si el cliente no tiene actividad (600000=10 minutos).
   protected static final int TIEMPO_DESCONEXION_AUTOMATICA = 600000;

   private ServerSocket socketServidor; // socket de servidor (pasivo)

   /** Punto de entrada a la aplicación. En este método se arranca el servidor
    * de chat y se comienza el procesado de las conexiones y mensajes de los clientes.
    *
    * @param args
    */
   public static void main(String[] args) {
      new ServidorChat();
   }

   /** Constructor
    */
   public ServidorChat() {
      System.out.println("Arrancando el servidor por el puerto " + PUERTO);
```

```
        arrancarServidor();
        procesarClientes();
    }

    /**
     * Arranca el servidor: crea el ServerSocket y lo vincula al número de puerto
     * PUERTO.
     */
    private void arrancarServidor() {
        // Se intenta crear un socket de servidor en el número de puerto PUERTO.
        // Cualquier excepción en la creación del socket de servidor es fatal para el programa. Para dar
        // la mayor información posible al usuario se tratan los tres tipos posibles de excepciones:
        //    1) Excepción al intentar ligar el socket de servidor al número de puerto PUERTO.
        //    2) Excepción debida a restricciones de seguridad en el ordenador servidor.
        //    3) Excepción general de Entrada/Salida.

        try {
            socketServidor = new ServerSocket(PUERTO);
            System.out.println("El servidor está en marcha: escucha por el puerto " + PUERTO);
        }
        catch (java.net.BindException e1) {
            // No se puede arrancar el servidor. Error irrecuperable: se sale del programa.
            // No se limpia el socket de servidor porque no ha llegado a crearse.
            String mensaje = "No puede arrancarse el servidor por el puerto " + PUERTO +
                             ". Seguramente, el puerto está ocupado.";
            errorFatal(e1, mensaje);
        }
        catch (java.lang.SecurityException e2) {
            // No se puede arrancar el servidor. Error irrecuperable: se sale del programa.
            // No se limpia el socket de servidor porque no ha llegado a crearse.
            String mensaje = "No puede arrancarse el servidor por el puerto " + PUERTO +
                             ". Seguramente, hay restricciones de seguridad.";
            errorFatal(e2, mensaje);
        }
        catch (IOException e3) {
            // No se puede arrancar el servidor. Error irrecuperable: se sale del programa.
            // No se limpia el socket de servidor porque no ha llegado a crearse.
            String mensaje = "No puede arrancarse el servidor por el puerto " + PUERTO;
            errorFatal(e3, mensaje);
        } // fin del try-catch
    }

    /** Procesa las conexiones entrantes de los clientes. A cada cliente que se conecta le
     * asigna un hilo ThreadServidor, que se encargará de gestionar el envío y la recepción
     * de mensajes.
     */
    private void procesarClientes() {
        Socket socketCliente = null; // socket (activo) para el cliente que se conecta

        // Se entra en un bucle infinito, que sólo se romperá si se producen excepciones
        // debidas a restricciones de seguridad en el ordenador servidor.
        while (true) {
            try {
                socketCliente = socketServidor.accept(); // Se asigna un socket a cada petición entrante.
                try {
                    new ThreadServidor(socketCliente);    // Se crea un hilo para cada conexión entrante.
```

```
            }
            catch (IOException e1) {
               // Excepción en el constructor de ThreadServidor: seguramente se debe a que el
               // cliente cerró la comunicación nada más hecha la conexión con el servidor.
               // Se intenta cerrar el socket de cliente.
               // Si no se puede cerrrar, no se hace nada. Posiblemente, el socket fue cerrado
               // por el cliente.  Se volverán a esperar conexiones.

               if (socketCliente != null) {
                  try {
                     socketCliente.close();
                  }
                  catch (IOException e2) {} // No se hace nada.
               }
            }
         } // fin del try-catch  interno
         catch (java.lang.SecurityException e3) {
            // Excepción debida a restricciones de seguridad. Error irrecuperable: se sale del
            // programa. Si el socket de servidor no es nulo, se intenta cerrarlo antes de salir.

            if (socketServidor != null) {
               try {
                  socketServidor.close();
               }
               catch (IOException e4) {} // No se hace nada
            }

            String mensaje = "Con su configuración de seguridad, los clientes no pueden " +
                             "conectarse por el puerto  " + PUERTO;
            errorFatal(e3, mensaje);
         }
         catch (IOException e5) {
            // No se hace nada: el socket  de cliente no llegó a crearse en accept(). Se volverán a
            // esperar conexiones.
         } // fin del try-catch externo
      } // fin del while
   }

   /**
    * Informa al usuario de la excepción arrojada y sale de la aplicación.
    *
    * @param excepcion  excepción cuya información se va a mostrar.
    * @param mensajeError  mensaje orientativo sobre la excepción.
    */
   private static void errorFatal(Exception excepcion, String mensajeError) {
      excepcion.printStackTrace();
      JOptionPane.showMessageDialog(null, "Error fatal."+ System.getProperty("line.separator") +
         mensajeError, "Información para el usuario", JOptionPane.WARNING_MESSAGE);
      System.exit(-1);
   }

}
```

```
/**
 * Esta clase representa el hilo que se asigna a cada cliente que se conecta
 * a la aplicación de chat.
 */
class ThreadServidor extends Thread {

    private String nombreCliente; // alias para el usuario

    // El String historial da el camino del archivo donde se guardaran las conexiones y
    // desconexiones.
    // En este caso, la estructura de directorios corresponde a un sistema Windows.
    private static String historial = "C:" + File.separatorChar + "historial.txt";

    // Lista de clientes activos. ArrayList es una colección NO sincronizada.
    private static List clientesActivos = new ArrayList(); // contiene objetos ThreadServidor

    private Socket socket;

    // flujos de E/S
    private BufferedReader entrada;
    private PrintWriter salida;

    /**
     * Constructor. Este método crea un hilo a partir de un socket
     * pasado por la clase ServidorChat. Cualquier error al tratar
     * de obtener los flujos de E/S asociados al socket provocará
     * una excepción.
     *
     * @param socket socket para el cliente
     * @throws IOException
     */
    public ThreadServidor(Socket socket) throws IOException {
        this.socket = socket;

        PrintWriter salidaArchivo = null;

        // Se crea un PrintWriter asociado al flujo de salida orientada
        // a bytes del socket. El PrintWriter convierte los caracteres
        // leídos en bytes, siguiendo el juego de caracteres por defecto
        // de la plataforma. Se establece el flush() automático.
        salida = new PrintWriter(socket.getOutputStream(), true);

        // Se crea un BufferedReader asociado al flujo de entrada
        // orientada a bytes del socket. El InputStreamReader convierte en caracteres
        // los bytes leídos del socket, siguiendo el juego de caracteres por omisión
        // de la plataforma.
        // La clase BufferedReader proporciona el método readLine().
        entrada = new BufferedReader(new InputStreamReader(socket.getInputStream()));

        escribirHistorial("Conexión desde la dirección");

        // Se lanza el hilo. A partir de este punto, el tratamiento de las
        // excepciones queda en mano de run().
        start();
    }
```

```
/**
 * Se encarga de leer los mensajes de los usuarios y de reenviarlos. Cualquier excepción acaba
 * provocando la muerte del hilo.
 */
public void run() {
    String textoUsuario = "";

    try {
        salida.println("> Bienvenido a este chat.");
        salida.println("> Introduzca su nombre, por favor.");

        // Si el cliente no introduce su nombre, se le asigna "Invitado".
        // Si el nombre esta repetido, se le añade al nombre el número del puerto.
        nombreCliente = (entrada.readLine()).trim();
        if ( (nombreCliente.equals("")) || (nombreCliente == null) ) {
            nombreCliente = "Invitado";
        }
        Iterator it = clientesActivos.iterator();
        while (it.hasNext()) {
            if (nombreCliente.equals(( (ThreadServidor) it.next()).nombreCliente)) {
                nombreCliente = nombreCliente + socket.getPort();
                break;
            }
        }

        // Se añade la conexión a la lista de clientes activos.
        anyadirConexion(this);

        // Se informa al usuario de su alias.
        salida.println("> Se le asignado el alias de " + nombreCliente);

        // Se establece el límite de tiempo sin actividad para la desconexión.
        // Transcurridos TIEMPO_DESCONEXION_AUTOMATICA milisegundos sin
        // ninguna llamada al método readLine(), se lanzará una excepción.
        socket.setSoTimeout(ServidorChat.TIEMPO_DESCONEXION_AUTOMATICA);

        // Se lee la entrada y se comprueba si corresponde a las órdenes "DESCONECTAR" o
        // "LISTAR".
        while ( (textoUsuario=entrada.readLine()) != null ) {
            if ((textoUsuario.equals("DESCONECTAR"))) {
                // Se envía mensaje al usuario, se registra la desconexión y se sale del bucle while.
                salida.println("> **********HASTA LA VISTA**********");
                escribirHistorial("Desconexión voluntaria desde la dirección:");
                break;
            }
            else if ((textoUsuario.equals("LISTAR"))) {
                // Se escribe la lista de usuarios activos
                escribirCliente(this,"> " + listarClientesActivos());
            }
            else {
                // Se considera que estamos ante un mensaje normal y se envía
                // a todos los usuarios.
                escribirATodos(nombreCliente+"> "+ textoUsuario);
            }
        } // fin del while
```

```
        } // fin del try

        catch (java.io.InterruptedIOException e1) {
            // Las excepciones porque se ha alcanzado el tiempo máximo permitido sin
            // actividad se tratan enviando un mensaje de conexión cerrada al cliente.
            escribirCliente(this, "> "+ "****************************************");
            escribirCliente(this, "> "+ "Se le pasó el tiempo: Conexión cerrada");
            escribirCliente(this, "> "+ "Si desea continuar, abra otra sesión");
            escribirCliente(this, "> "+ "*****************ADIOS*****************");

            // Se registra la desconexión por inactividad.
            escribirHistorial("Desconexión por fin de tiempo desde la dirección:");
        }
        catch (IOException e2) {
            escribirHistorial("Desconexión involuntaria desde la dirección:");
        }

        // Ocurra lo que ocurra (excepción por final de tiempo o de otro tipo), deben
        // cerrarse los sockets y los flujos (o, al menos, intentarlo).
        finally {
            eliminarConexion(this);
            limpiar();
        } // fin try-catch-finally
    }

    /**
     * Se encarga de cerrar el socket y los flujos de E/S asociados.
     */
    private void limpiar() {
        // Si se llega a usar este método es porque ha habido una excepción (ya sea por un verdadero
        // error o porque el cliente ha terminado la conexión o porque han pasado
        // TIEMPO_DESCONEXION_AUTOMATICA milisegundos sin comunicación del cliente).

        // Nótese que el código de cierre es riguroso: una versión más abreviada podría limitarse
        // a intentar cerrar el socket (los flujos de E/S se cierran al cerrarse éste); el único problema
        // que podría aparecer en esa versión del código es que podrían quedarse abiertos los flujos si
        // no se pudiera cerrar el socket (esta posibilidad es inhabitual: normalmente, si el socket no
        // se puede cerrar es porque ya se había cerrado antes).
        // El igualar entrada, salida y socket a null es una buena práctica cuando se puede llamar
        // varias veces a limpiar(); así se evita intentar cerrar dos veces un objeto.

        if ( entrada != null ) {
            try {
                entrada.close();
            }
            catch (IOException e1) {}
            entrada = null;
        }
        if ( salida != null ) {
                salida.close();
                salida = null;
        }
        if ( socket != null ) {
            try {
                socket.close();
            }
```

```
            catch (IOException e2) {}
            socket = null;
        }
    }

    /**
     * Método sincronizado.
     * Elimina la conexión representada por el argumento threadServidor de la lista de
     * clientes activos.
     *
     * @param threadServidor hilo que se va a eliminar de la lista de clientes activos.
     */
    private static synchronized void eliminarConexion(ThreadServidor threadServidor) {
        clientesActivos.remove(threadServidor);
    }

    /**
     * Método sincronizado.
     * Añade la conexión representada por el argumento threadServidor a la lista de
     * clientes activos.
     *
     * @param threadServidor hilo (correspondiente a una nueva conexión) que
     * se añade a la lista de clientes activos
     */
    private static synchronized void anyadirConexion(ThreadServidor threadServidor) {
        clientesActivos.add(threadServidor);
    }

    /**
     * Método sincronizado.
     * Escribe la cadena de texto textoUsuario en todas las conexiones de la lista de clientes
     * activos, exceptuando el cliente activo que envía el mensaje.
     *
     * @param textoUsuario texto que se envía a los clientes.
     */
    private synchronized void escribirATodos(String textoUsuario) {
        Iterator it = clientesActivos.iterator();

        // Se envía el texto a todos los usuarios, excepto al que lo ha escrito.
        while (it.hasNext()) {
            ThreadServidor tmp = (ThreadServidor) it.next();
            if ( !(tmp.equals(this)) )
                escribirCliente(tmp, textoUsuario);
        }
    }

    /**
     * Método sincronizado.
     * Escribe la cadena de texto textoUsuario en la conexión representada por el hilo threadServidor.
     *
     * @param threadServidor hilo (cliente) al que se envía el mensaje.
     * @param textoUsuario texto para enviar.
     */
    private synchronized void escribirCliente(ThreadServidor threadServidor, String textoUsuario) {
        (threadServidor.salida).println(textoUsuario);
    }
```

```
    /**
     * Método sincronizado.
     * Escribe los alias de todos los clientes activos en la conexión representada por el hilo
     * threadServidor que los solicita.
     *
     * @return StringBuffer lista de alias de los clientes activos.
     */
    private static synchronized StringBuffer listarClientesActivos() {
        // Se usa StringBuffer por eficacia.
        StringBuffer cadena = new StringBuffer();

            for (int i = 0; i < clientesActivos.size(); i++) {
                ThreadServidor tmp = (ThreadServidor) (clientesActivos.get(i));
                cadena.append(
                    (((ThreadServidor) clientesActivos.get(i)).nombreCliente)).append("||") ;
            }
        return cadena;
    }

    /**
     * Método sincronizado.
     * Escribe en el archivo de historial el texto mensaje, asociado a conexiones y
     * desconexiones de los clientes.
     *
     * @param mensaje texto que se escribe en el archivo de historial.
     */
    private synchronized void escribirHistorial(String mensaje ) {
        PrintWriter salidaArchivo = null;
        try {
            salidaArchivo = new PrintWriter(new BufferedWriter
                 (new FileWriter(historial, true))); // true = autoflush
            salidaArchivo.println(mensaje + " " +
                  socket.getInetAddress().getHostName() +
                  " por el puerto " + socket.getPort() +
                  " en la fecha " + new Date());
        }
        catch (IOException e1) {
            System.out.println( "Fallo en el archivo de historial.");
        }
        finally {
            // limpieza de salidaArchivo
            if (salidaArchivo != null) {
                salidaArchivo.close();
                salidaArchivo = null;
            }
        }
    }

}
```

ClienteChat.java

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

/**
 * Esta clase actúa como cliente para el chat. Recibe los mensajes de otros clientes y los muestra
 * por consola (la Entrada/Salida estándar).
 *
 * El programa puede ejecutarse sin argumentos (se considera que el programa servidor
 * ServidorChat.java se ejecuta en la máquina local y por el número de puerto 9000) o con dos
 * argumentos (nombre del ordenador servidor y número de puerto). Si no se introducen dos
 * argumentos, se considera que se está en el primer caso.
 * Ejemplos de uso: java ClienteChat; java ClienteChat www.mimaquina.com 1200
 *
 * Si tiene el programa servidor en una máquina remota e intenta usar este programa cliente desde
 * una red con cortafuegos o un proxy, ejecútelo con código como éste: <p>
 * java -DproxySet=true -DproxyHost=PROXY -DproxyPort=PUERTO ClienteChat www.mimaquina.com
 * 1200 </p>
 * (PROXY es el nombre de la máquina que actúa como proxy o cortafuegos y PUERTO es el
 * número de puerto por el que escucha esta máquina).
 *
 */
public class ClienteChat {
   private static final int PUERTO = 9000; // puerto por omisión

   private Socket socketCliente; // socket para enlazar con el socket de servidor

   /** Punto de entrada a la aplicación. En este método se arranca el cliente
    * de chat y se comienza el procesado de los mensajes de los clientes.
    *
    * @param args nombre de la máquina donde se ejecuta el servidor y puerto
    * por el que escucha.
    */
   public static void main(String args[]) {
      new ClienteChat(args);
   }

   /** Constructor
    *
    * @param args nombre de la máquina donde se ejecuta el servidor y puerto
    * por el que escucha.
    */
   public ClienteChat(String args[]) {
     System.out.println("Arrancando el cliente.");

     arrancarCliente(args);
     procesarMensajes();
   }
```

```
/** Arranca el cliente: lo intenta conectar al servidor por el núm. de puerto PUERTO o por
 * el especificado como argumento.
 *
 * @param args nombre de la máquina donde se ejecuta el servidor y puerto
 * por el que escucha.
 */
private void arrancarCliente(String[] args) {
   // Si se introduce como argumento un nombre de máquina  y un número de puerto, se
   // considera que el servidor está ubicado allí y que escucha por ese número de puerto; en caso
   // contrario, se considera que está en la máquina local (localhost) y que escucha por el
   // puerto PUERTO.

   // Cualquier excepción en la creación del socket es fatal para el programa. Para dar la
   // mayor información posible al usuario se tratan los cuatro tipos posibles de
   // excepciones:
   //    1) Excepción porque no se ha introducido un número puerto válido (es decir, entero).
   //    2) Excepción porque no se encuentra ningún ordenador con el nombre introducido por el
   //         usuario.
   //    3) Excepción debida a restricciones de seguridad en el ordenador servidor.
   //    4) Excepción general de Entrada/Salida.
   try {
      if (args.length == 2)
         socketCliente = new Socket (args[0], Integer.parseInt(args[1]));
      else
         socketCliente = new Socket("localhost", PUERTO);  // puerto del servidor por omisión

         System.out.println("Arrancado el cliente.");
   }
   catch (java.lang.NumberFormatException e1) {
      // No se puede arrancar el cliente porque se introdujo un número de puerto que no es entero.
      // Error irrecuperable: se sale del programa. No hace falta limpiar el socket, pues no llegó a
      // crearse.
      errorFatal(e1, "Número de puerto inválido.");
   }
   catch (java.net.UnknownHostException e2) {
      // No se puede arrancar el cliente. Error irrecuperable: se sale del programa.
      // No hace falta limpiar el socket, pues no llegó a crearse.
      errorFatal(e2, "No se localiza el ordenador servidor con ese nombre.");
   }
   catch (java.lang.SecurityException e3) {
      // No se puede arrancar el cliente. Error irrecuperable: se sale del programa.
      // No hace falta limpiar el socket, pues no llegó a crearse.
      String mensaje ="Hay restricciones de seguridad en el servidor para conectarse por el " +
                      "puerto " + PUERTO;
      errorFatal(e3, mensaje);
   }
   catch (IOException e4) {
      // No se puede arrancar el cliente. Error irrecuperable: se sale del programa.
      // No hace falta limpiar el socket, pues no llegó a crearse.
      String mensaje = "No se puede conectar con el puerto " + PUERTO + " de la máquina " +
                       "servidora. Asegúrese de que el servidor está en marcha.";
      errorFatal(e4, mensaje);
   }
}
```

```
    /** Crea los flujos de entrada y salida asociados al socket que ha llevado a cabo con éxito su
     * conexión al servidor y asocia un hilo ThreadCliente al flujo de entrada del socket. Así
     * el usuario podrá a la vez escribir y recibir mensajes.
     */
    private void procesarMensajes() {

        // flujos de Entrada/Salida
        BufferedReader entrada=null;
        PrintWriter salida=null;

       // Se crean los flujos de E/S: dos para el socket y uno para la entrada por consola.
        try {
            entrada= new BufferedReader(new
                 InputStreamReader(socketCliente.getInputStream()));
            salida = new PrintWriter(socketCliente.getOutputStream(), true);
            BufferedReader entradaConsola = new BufferedReader(new
                 InputStreamReader(System.in));

            // Se crea un hilo que se encarga de recibir y mostrar por consola los mensajes del
            // servidor (procedentes de otros clientes).
            new ThreadCliente(entrada);

            // Se entra en un bucle infinito para leer la entrada del usuario por la consola y enviarla al
            // servidor de chat.
            while (true)
                salida.println(entradaConsola.readLine());
        }
        catch (IOException e) {
            e.printStackTrace();

            // Limpieza del socket y de los flujos asociados para que el cierre sea "limpio".
            if ( entrada != null) {
                try {
                    entrada.close();
                }
                catch (Exception e1) {
                    entrada = null;
                }
            }
            if ( salida != null) {
                try {
                    salida.close();
                }
                catch (Exception e1) {
                    salida = null;
                }
            }
            if ( socketCliente != null) {
                try {
                    socketCliente.close();
                }
                catch (Exception e1) {
                    socketCliente = null;
                }
            }
```

```
            // Aviso al usuario y cierre.
            String mensaje = "Se ha perdido la comunicación con el servidor. Seguramente se debe a "+
                             " que se ha cerrado el servidor o a errores de transmisión";
            errorFatal(e, mensaje);
        }
    }

    /**
     * Informa al usuario de la excepción arrojada y sale de la aplicación.
     *
     * @param excepcion  excepción cuya información se va a mostrar.
     * @param mensajeError  mensaje orientativo sobre la excepción .
     */
    private static void errorFatal(Exception excepcion, String mensajeError) {
        excepcion.printStackTrace();
        JOptionPane.showMessageDialog(null, "Error fatal."+ System.getProperty("line.separator") +
            mensajeError, "Información para el usuario", JOptionPane.WARNING_MESSAGE);
        System.exit(-1);
    }

}


/**
 * Esta clase representa el hilo que gestiona la entrada de mensajes procedentes del
 * servidor de chat.
 */
class ThreadCliente extends Thread {

    private BufferedReader entrada;

    /**
     * Constructor.
     *
     * @param entrada BufferedReader bufferedReader con la entrada que llega al socket de
     * ClienteChat
     * @throws IOException
     */
    public ThreadCliente (BufferedReader entrada) throws IOException {
        this.entrada=entrada;

        start(); // Se arranca el hilo.
    }

    /**
     * Muestra por consola los mensajes enviados por el servidor.
     */
    public void run() {
        // Última línea del aviso de desconexión por falta de actividad.
        String fin1 =  "> **********ADIOS**********";

         // Última línea del aviso de desconexión por uso de la orden DESCONECTAR.
        String fin2 = "> **********HASTA LA VISTA**********";

        String linea = null;
```

```
        try {
            while( ( linea=entrada.readLine() ) != null ) {
                System.out.println(linea);
                // Si se produce una desconexión por que se ha terminado el tiempo permitido de
                // inactividad o porque se ha dado previamente la orden DESCONECTAR, se sale del
                // bucle.
                if ( linea.equals(fin1) || linea.equals(fin2) )
                    break;
            }
        }
        catch (IOException e1) {
            e1.printStackTrace();
        }
        finally {
            if (entrada !=null) {
                try {
                    entrada.close();
                }
                catch (IOException e2) {} // No se hace nada con la excepción
            }

            System.exit(-1);
        }
    }  // fin try-catch-finally

}
```

Como ya se adelantó, esta aplicación de *chat* por consola consta de dos partes claramente separadas: una actúa como servidor (*ServidorChat.java*) y otra como cliente (*ClienteChat.java*).

La clase **ServidorChat** contiene un método *arrancarServidor()* que crea un socket de servidor unido a un puerto determinado (en este ejemplo el puerto 9000: un homenaje personal a **HAL 9000**, una IA general con demasiada I). A continuación, el programa entra en un bucle infinito –dentro del *método procesarClientes*()–, en el cual espera conexiones de los clientes mediante llamadas al método *accept()*. Cuando *accept()* devuelve un objeto *Socket*, el servidor asigna un hilo a la conexión y vuelve a esperar nuevas conexiones.

El socket de servidor se crea con la clase *ServerSocket*:

```
        try {
            if (args.length == 2)
                socketCliente = new Socket (args[0], Integer.parseInt(args[1]));
            else
                socketCliente = new Socket("localhost", PUERTO);  // puerto del servidor por omisión

                System.out.println("Arrancado el cliente.");
        }
```

La espera de las conexiones se implementa con una llamada a *accept()*:

```
scliente = sservidor.accept();
```

Cuando un cliente se conecta, se instancia un socket activo para el cliente:

```
Socket scliente = null;
...
scliente = sservidor.accept();
```

A continuación, se crea un hilo asociado al objeto *Socket* devuelto por *accept()*:

```
new ThreadServidor(scliente);
```

En el constructor de la clase **ThreadServidor** (nótese que no es pública) se asocian flujos de entrada y salida a cada socket que procede de la clase **ServidorChat**:

```
public ThreadServidor(Socket socket) throws IOException {
    this.socket = socket;

    PrintWriter salidaArchivo = null;

    // ...
    salida = new PrintWriter(socket.getOutputStream(), true);

    // ...
    entrada = new BufferedReader(new InputStreamReader(socket.getInputStream()));

    escribirHistorial("Conexión desde la dirección");

    // Se lanza el hilo. A partir de este punto, el tratamiento de las
    // excepciones queda en mano de run().
    start();
}
```

El flujo de entrada es un *BufferedReader*. Así se puede emplear su método *readLine()*, muy conveniente para aplicaciones en las que el fin de los mensajes se marca mediante caracteres de salto de línea o de nueva línea.

El flujo de salida es un *PrintWriter*, que proporciona el método *println()*. En el constructor de *PrintWriter* se ha usado –no por casualidad– un argumento de tipo booleano. Para entender para qué sirve este argumento potestativo, debe saberse que Java emplea *buffers* (memorias de almacenamiento temporal) para muchas clases de entrada y salida. Éstos se usan por eficacia: en vez de leer o escribir carácter a carácter (o byte a byte), resulta mucho más rápido leer de un buffer o escribir en él, pues se puede trabajar a la vez con decenas o cientos de caracteres.

En determinadas aplicaciones, el uso automático de buffers no es recomendable. Por ejemplo, en la aplicación de charla se desea que, cuando el usuario apriete *Enter*, se envíe su mensaje a todos los demás usuarios. Usar memorias de almacenamiento temporal resulta incompatible con este comportamiento: el mensaje se enviará cuando el buffer de lectura esté lleno, no cuando lo desee el usuario. En consecuencia, el usuario podría necesitar escribir varios mensajes para llenar el buffer; mientras, los otros no recibirían nada. Además, cuando llegaran se recibirían todos de golpe.

En el siguiente código, por ejemplo, no hay ninguna garantía de que se envíe la cadena “> Hola” cuando se llame al método *println()*:

```
salida = new PrintWriter(socketCliente.getOutputStream()); // Nótese que no hay
                                                           // true en el constructor
salida.println("> Hola");
```

Para solucionar este inconveniente sin perder las ventajas de los buffers o sin tener que estar abriendo y cerrando flujos (al cerrar un flujo se envía todo lo que hay en el buffer), es posible usar el método *flush()*, cuyas llamadas se traducen como "Envíe a la salida todos los caracteres que tenga almacenados, aunque el buffer aún no esté lleno. Sí, ya sé que se pierde rapidez, pero así lo quiere el cliente". En el siguiente ejemplo, la cadena "> Hola" se enviará cuando se llama al método *flus*h(), esté o no lleno el buffer interno de almacenamiento.

```
salida = new PrintWriter(socketCliente.getOutputStream()); // Nótese que no hay
                                                           // true en el constructor
salida.println("> Hola");
salida.flush();
```

Si se utiliza con el valor *true* el constructor de **PrintWriter** que usa un argumento booleano, se llamará al método *flush()* cada vez que se use *println()*. Por consiguiente, en la aplicación de *chat*, no hace falta preocuparse por usar *flush()*.

Varios de los métodos dentro de la clase **ThreadServidor** están marcados con la palabra reservada *synchronized*. Los hilos que llaman a un método sincronizado de un objeto, sólo pueden llamarlo de uno en uno. Cuando un hilo llama a un método sincronizado de un objeto, éste es bloqueado de manera que no se puede llamar a otro método sincronizado de ese objeto hasta que no termine la primera llamada.

La falta de sincronización puede provocar situaciones extrañas e inconsistencias en los datos. Imagine, por ejemplo, un método que abra un archivo y escriba algo en él. Si fuera accedido a un tiempo por varios hilos, podrían interrumpirse entre sí: uno podría comenzar a escribir cuando otro no había acabado; un tercero podría comenzar a escribir antes de que los dos primeros hubieran terminado, y podría ser interrumpido por un cuarto; un quinto podría descubrir que los otros han cerrado el archivo mientras él estaba escribiendo... Finalmente, lo que se grabara en el archivo parecería la lista de la compra de un pingüino disléxico con algunas copas de más.

En el mundo real, estamos acostumbrados a que los procesos estén sincronizados: los jugadores de un equipo de fútbol no intentan tirar todos juntos los penaltis, un jugador de ajedrez no mueve una pieza hasta que llega su turno, todos los inspectores de Hacienda no solicitan a la vez las cuentas a una empresa, los jugadores de un partido de baloncesto no intentan encestar todos a la vez en la misma canasta, los conductores no intentan apretar todos los pedales a la vez... En cambio, en Java, la sincronización de hilos debe declararse de forma explícita.

En el caso de la aplicación de charla, he aquí un ejemplo de método sincronizado:

```
private synchronized void escribirATodos(String textoUsuario) {
    Iterator it = clientesActivos.iterator();

    // Se envía el texto a todos los usuarios, excepto al que lo ha escrito.
    while (it.hasNext()) {
        ThreadServidor tmp = (ThreadServidor) it.next();
        if ( !(tmp.equals(this)) )
            escribirCliente(tmp, textoUsuario);
    }
}
```

Si el método no estuviera sincronizado, un hilo podría comenzar a escribir un mensaje a un cliente antes de que otro hubiera acabado. En ese caso, el cliente recibiría los dos mensajes mezclados.

Si un método como el siguiente no estuviera sincronizado, las consecuencias aún serían más graves:

```
private static synchronized void eliminarConexion(ThreadServidor ths) {
   clientesActivos.remove(ths);
}
```

Así, un hilo podría intentar eliminar una conexión de cliente (representada por un hilo) que otro hilo acaba de eliminar. Y las cosas podrían empeorar con mucha facilidad: nada impediría que un hilo eliminara a un hilo que, a su vez, estuviera borrando a otro. ¿En qué estado quedaría este último si al segundo no le diera tiempo a completar la operación de borrado antes de ser eliminado? Como siempre sucede, las cosas podrían seguir empeorando, pero me detengo aquí para no caer en trabalenguas no sincronizados.

Debe tenerse en cuenta que los métodos sincronizados consumen recursos y disminuyen la eficacia del programa. Por ejemplo, un método sincronizado tarda en ejecutarse entre diez y veinte veces más que su versión no sincronizada.

El tratamiento de las excepciones del *chat* se ha diseñado con cuidado, pues las aplicaciones en red son muy propensas a excepciones y muertes súbitas:

- Si se produce alguna excepción al intenta crear el *ServerSocket* en el método *arrancarServidor()*, se lanza un mensaje de error y se cierra el programa servidor. Generalmente, estos errores tempranos se deben a restricciones de seguridad (no se permiten enlazar sockets a ese puerto) o a que el puerto especificado (por defecto, 9000) está ocupado por otro proceso.
- Si se produce alguna excepción cuando se intenta crear *socketCliente* en el método *procesarClientes()*, se comprueba si es una *SecurityException*. En caso afirmativo, existe alguna restricción de seguridad que impide que los sockets del cliente se conecten al puerto por defecto. Por ello, lo mejor es cerrar el programa servidor: el problema no se solucionará hasta que se modifiquen los permisos de seguridad. En caso negativo, se considera que es un error temporal de conexión y se permite que continúe la aplicación.
- Si se arroja alguna excepción al intentar crear un hilo (*new ThreadServidor(socketCliente)*), resulta probable que se deba a que el cliente ha cerrado la conexión nada más conectarse. Estas desconexiones tempranas no se registran en el archivo de historial: sólo se comprueba si se ha creado un socket para el cliente (si así es, se intenta cerrarlo).
- Si el hilo se crea correctamente, el control de las excepciones queda encomendado al método *run()* de la clase **ThreadServidor**. Cualquier excepción en este método provocará que se salga del bucle *while* de lectura, que se borre el hilo de la lista de clientes activos y que, a la postre, se salga de *run()*.

Dentro del método *run()* de la clase **ThreadServidor**, hay unas líneas que precisan un comentario:

```
...
        // Se establece el límite de tiempo sin actividad para la desconexión.
        // Transcurridos TIEMPO_DESCONEXION_AUTOMATICA milisegundos sin
        // ninguna llamada al método readLine(), se lanzará una excepción.
        socket.setSoTimeout(ServidorChat.TIEMPO_DESCONEXION_AUTOMATICA);

        // Se lee la entrada y se comprueba si corresponde a las órdenes "DESCONECTAR" o
        // "LISTAR".
        while ( (textoUsuario=entrada.readLine()) != null ) {
           if ((textoUsuario.equals("DESCONECTAR"))) {
              // Se envía mensaje al usuario, se registra la desconexión y se sale del bucle
              salida.println("> **********HASTA LA VISTA**********");
              escribirHistorial("Desconexión voluntaria desde la dirección:");
              break;
           }
           else if ((textoUsuario.equals("LISTAR"))) {
              // Se escribe la lista de usuarios activos
              escribirCliente(this,"> " + listarClientesActivos());
           }
           else {
              // Se considera que estamos ante un mensaje normal y se envía
              // a todos los usuarios.
              escribir (nombreCliente+"> "+ textoUsuario, this);
           }
        } // fin del while
     } // fin del try

     catch (java.io.InterruptedIOException e1) {
        // Las excepciones porque se ha alcanzado el tiempo máximo permitido sin
        // actividad se tratan enviando un mensaje de conexión cerrada al cliente.
        escribirCliente(this, "> "+ "****************************************");
        escribirCliente(this, "> "+ "Se le pasó el tiempo: Conexión cerrada");
        escribirCliente(this, "> "+ "Si desea continuar, abra otra sesión");
        escribirCliente(this, "> "+ "**********ADIOS**********");

        // Se registra la desconexión por inactividad.
        escribirHistorial("Desconexión por fin de tiempo desde la dirección:");
     }
     catch (IOException e2) {
        escribirHistorial("Desconexión involuntaria desde la dirección:");
     }

     // Ocurra lo que ocurra (excepción por final de tiempo o de otro tipo), deben
     // cerrarse los sockets y los flujos (o, al menos, intentarlo).
     finally {
        eliminarConexion(this);
        limpiar();
     } // fin try-catch-finally
...
```

El método *public void setSoTimeout(int tiempoLimite) throws SocketException* de la clase **Socket** produce que una llamada a *read()* en el *InputStream* asociado con el socket

se bloquee durante *tiempoLimite* milesegundos. Tras ese lapso, se lanzará una excepción **java.lang.InterruptedException**, aunque el socket continuará siendo válido.

> **Nota:** Desde la versión 1.4 de la J2SE, la excepción que lanza *public void setSoTimeout(int tiempoLimite) throws SocketException* cuando acaba el *tiempoLimite* es de tipo **java.net.SocketTimeoutException**, una subclase de **java.lang.InterruptedException**. Tal y como está el código del chat, funcionará en todas las versiones de la J2SE. Si se usa **java.net.SocketTimeoutException** en el bloque *catch* que atrapa las excepciones por finalización de tiempo, el código sólo servirá para la versión 1.4 y posteriores.

El código anterior nos indica que cualquier conexión de la que no se reciba ningún mensaje en diez minutos (y que, por tanto, lance una *java.lang.InterruptedException*) será eliminada de la lista de clientes activos.

La captura de las excepciones *java.lang.InterruptedException* no sólo sirve para descartar a los usuarios perezosos, sino también para eliminar las conexiones inactivas. Puede ocurrir que una conexión se corte por parte del cliente y que el aviso de cierre no llegue al servidor (por un fallo de las redes, por un cierre brusco del ordenador cliente...); en este caso, el programa la descartará a los diez minutos de no recibir ningún mensaje de ella.

Para hacer pasar las conexiones defectuosas que no dan errores al sueño de los justos y de los injustos, se pueden usar otras muchas soluciones. Una bastante frecuente consiste en que cada hilo envíe cada cierto tiempo un mensaje de tipo “¿Hay alguien al otro lado?” al cliente, que debe enviar un mensaje de tipo “Recibido” cuando lo reciba. Si el hilo no recibe, pasado un tiempo, el mensaje de confirmación, sale del bucle *while* y muere.

La clase **ClienteChat** contiene un método *main()* que comienza creando, mediante el método *arrancarCliente()*, un *Socket* unido a un ordenador y a un puerto determinados Luego, el método *procesarMensajes()* crea un hilo para mostrar por consola los mensajes enviados por los clientes (y reenviados por el servidor), hilo en el cual hay un *BufferedReader* asociado al socket del cliente.

Dentro del método *run()* de la clase **ThreadCliente**, un bucle *while()* se encarga de leer de manera continua los mensajes, enviados por el servidor, de los otros clientes y de mostrarlos por consola. Si el cliente recibe del servidor los mensajes *fin1* o *fin2*, sale del bucle *while* y se cierra. La cadena *fin1* corresponde a la última línea que envía el servidor cuando se produce una excepción por fin del tiempo de inactividad; *fin2* es la última línea que envía el servidor cuando el cliente ha solicitado antes la desconexión.

A continuación, se muestran algunos mensajes de error de la aplicación de charla electrónica

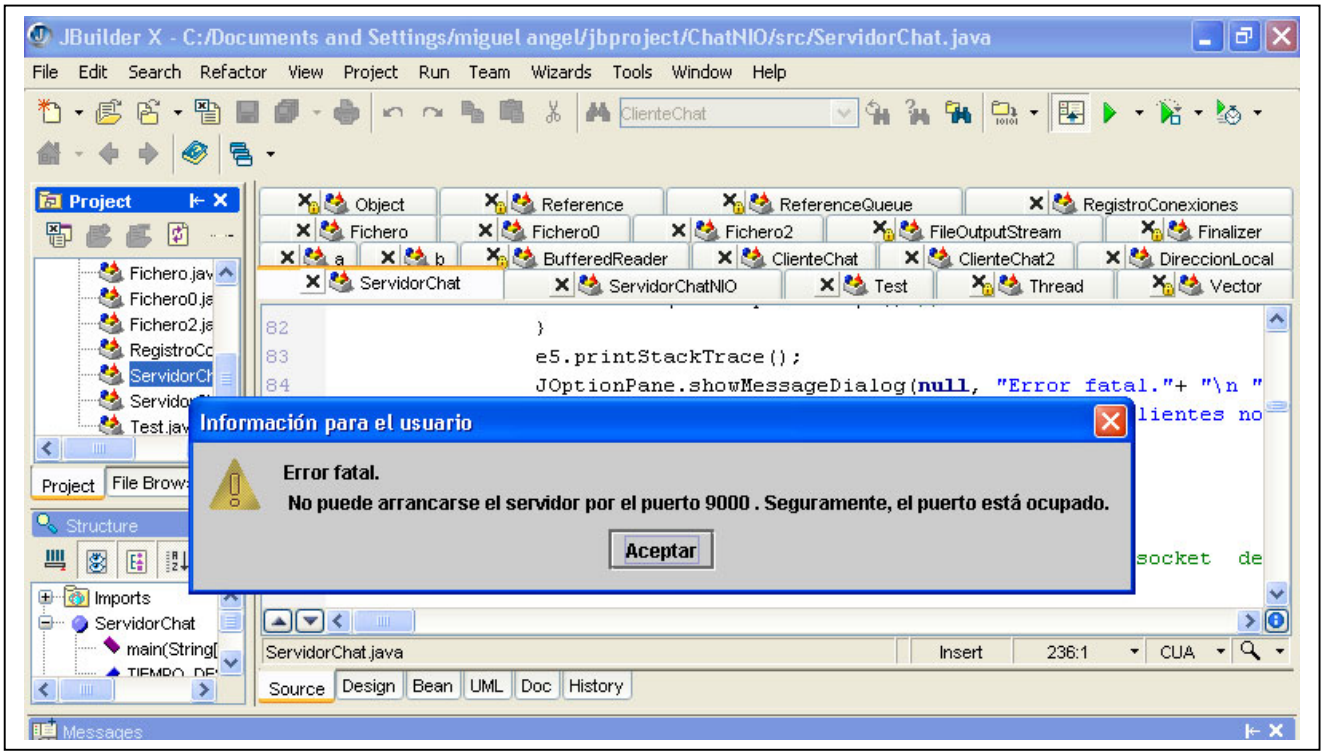

Figura 32. Ejemplo de mensaje de error en el servidor

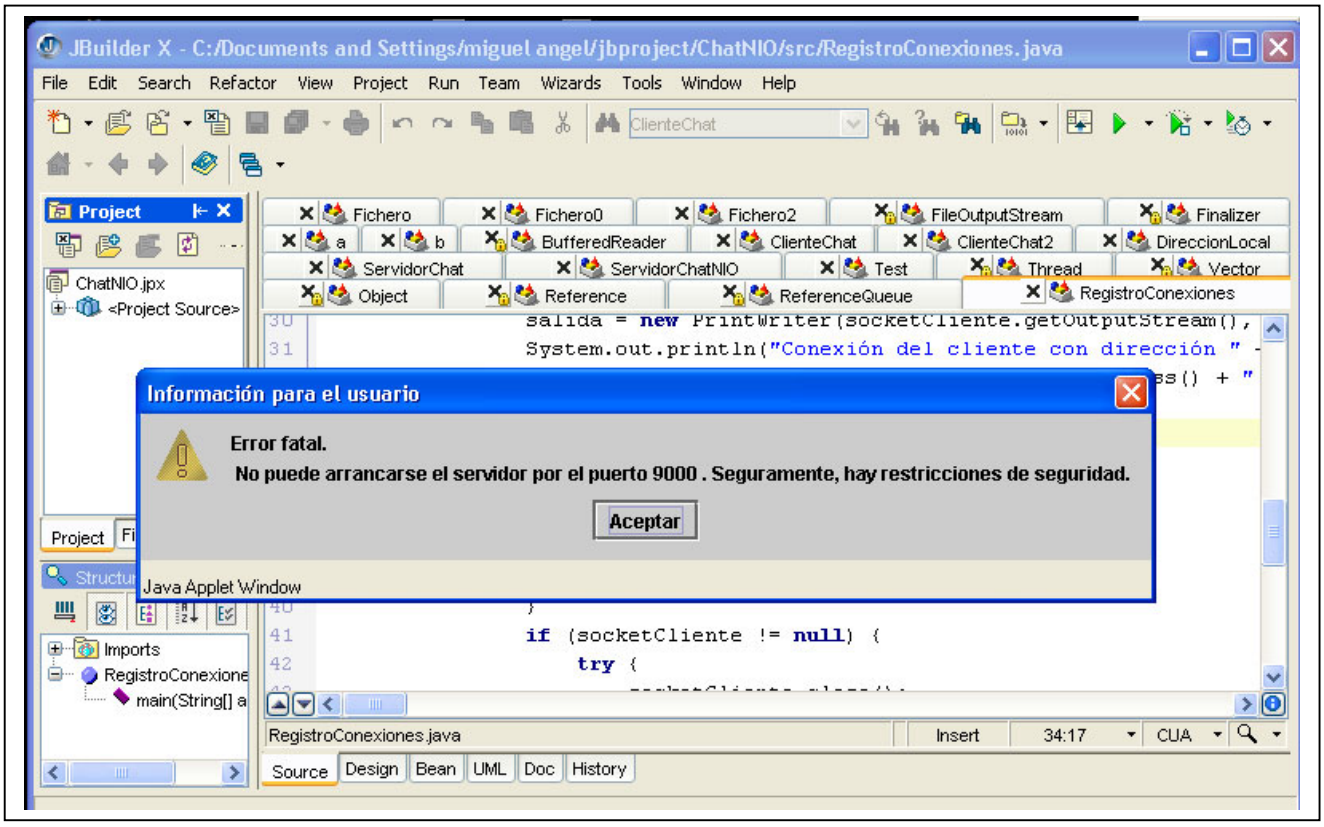

Figura 33. Ejemplo de mensaje de error en el servidor

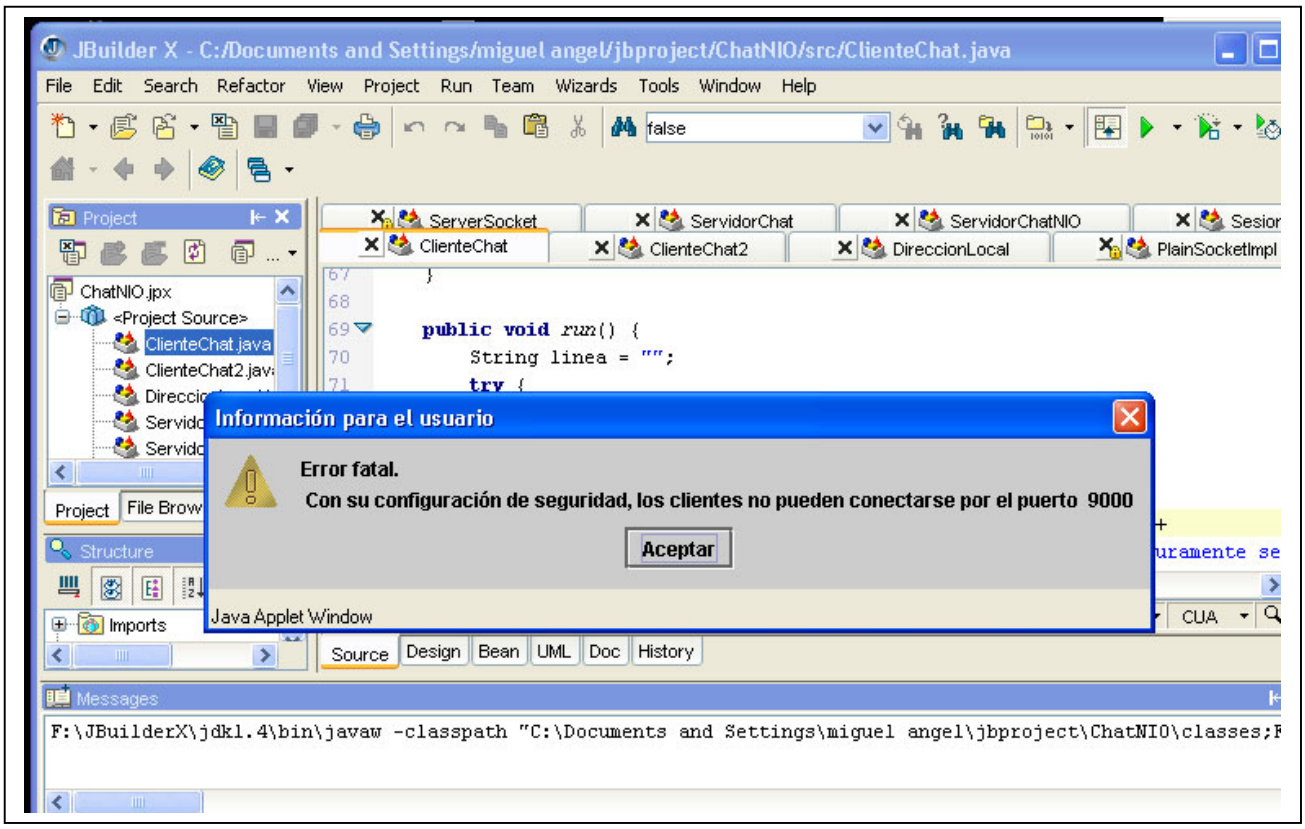

Figura 34. Ejemplo de mensaje de error en el servidor

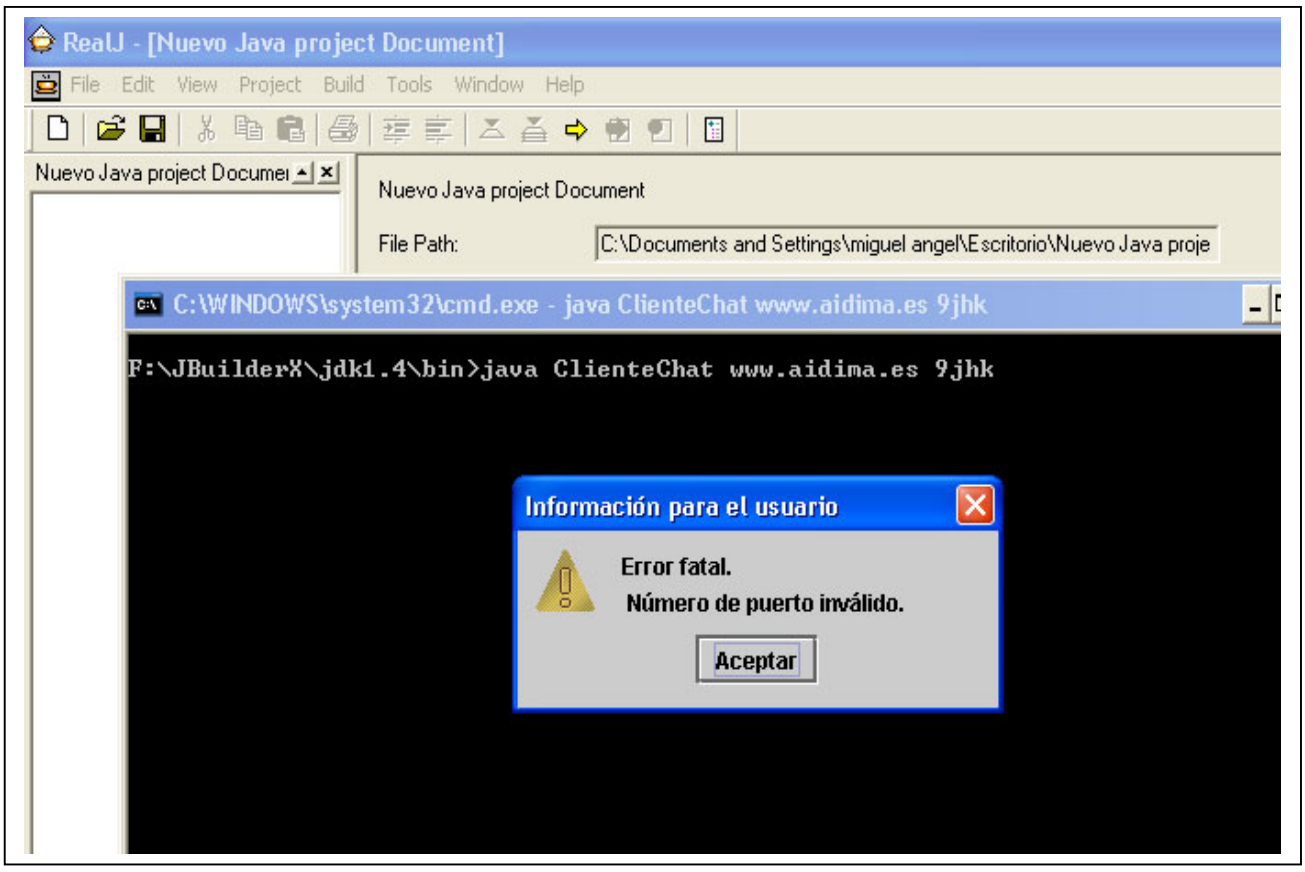

Figura 35. Ejemplo de mensaje de error en el servidor: el puerto no es un entero

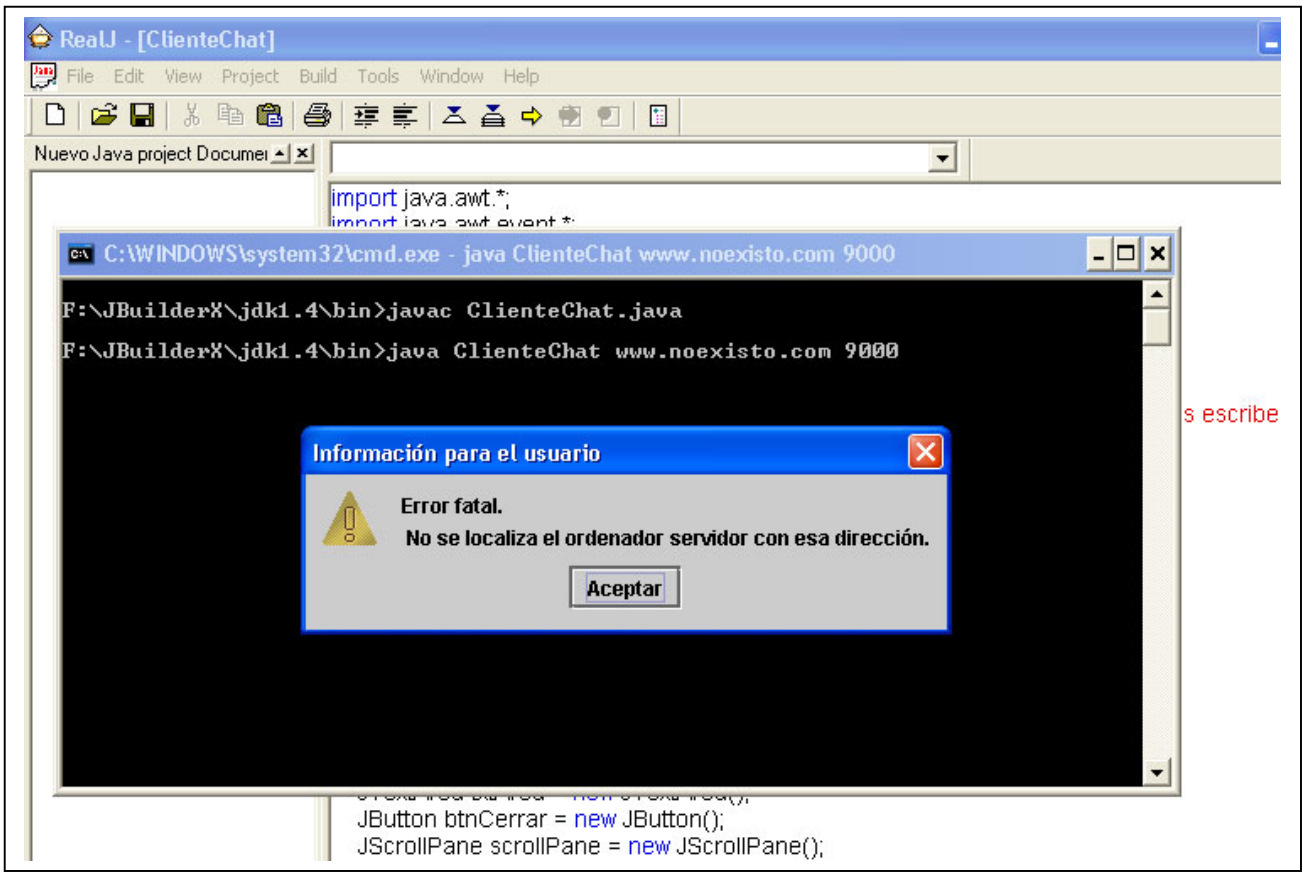

Figura 36. Ejemplo de mensaje de error en el servidor: el ordenador servidor no existe

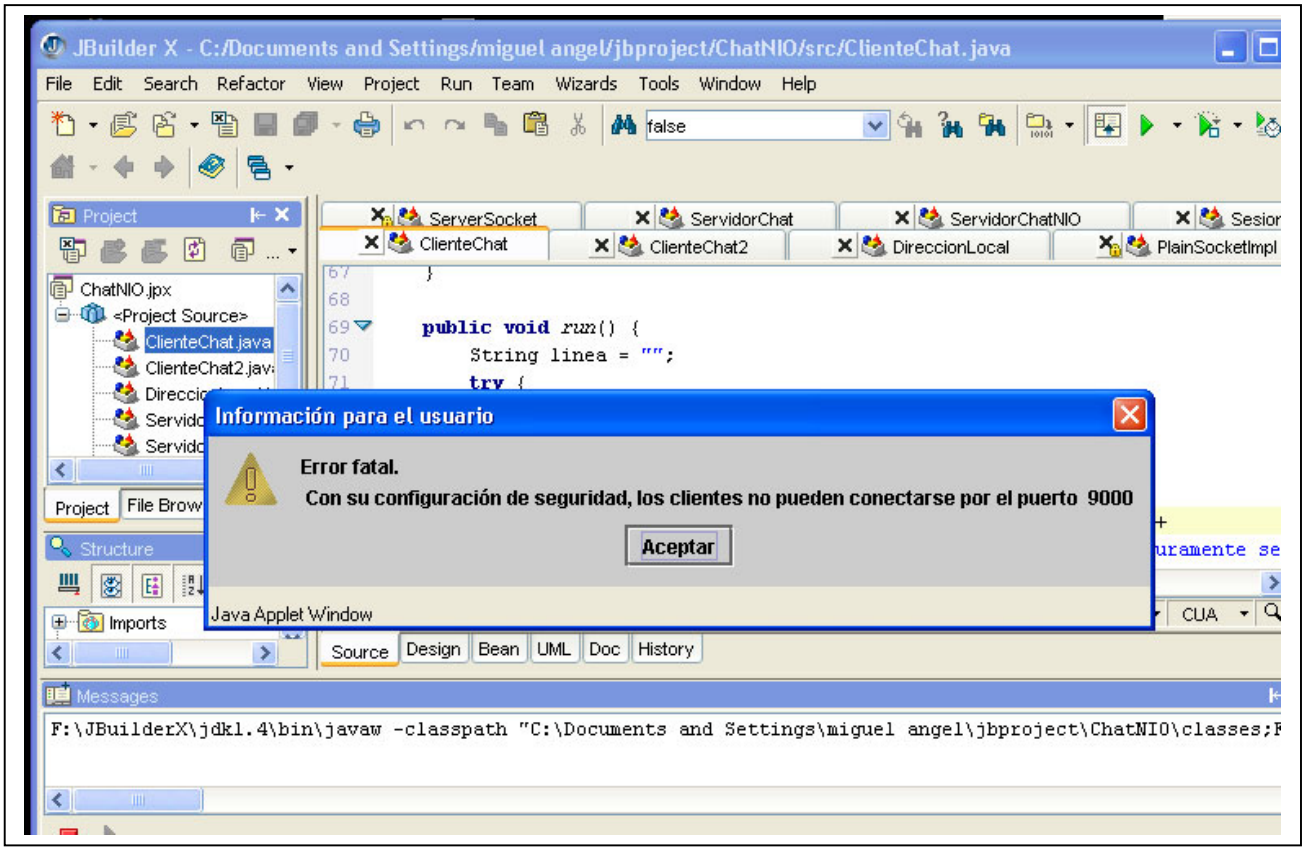

Figura 37. Ejemplo de mensaje de error en el cliente

5. UN CHAT CON INTERFAZ GRÁFICA

En este apartado se presenta una aplicación de charla con las mismas funciones que la del apartado 4, si bien añade una interfaz gráfica de este estilo:

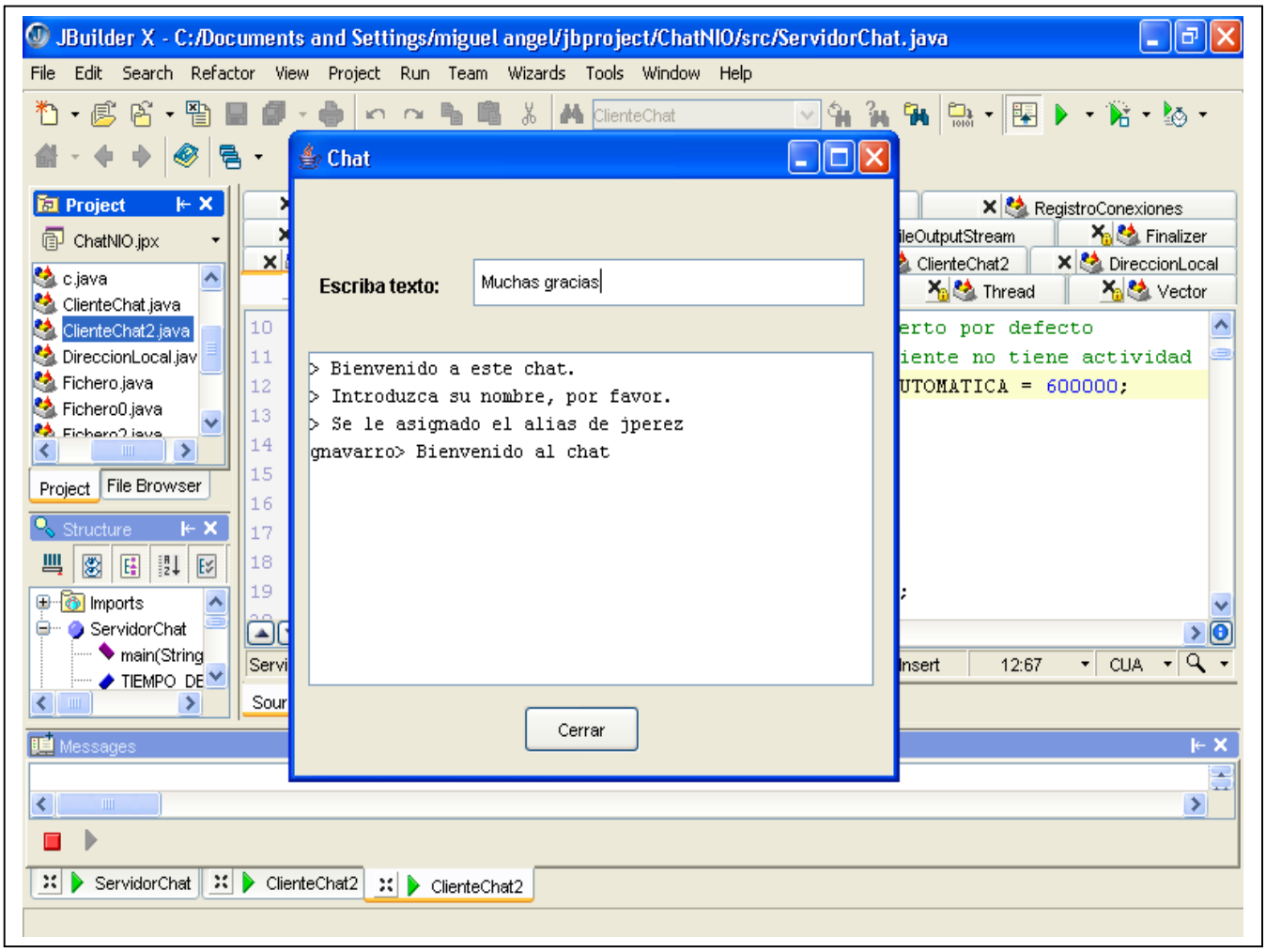

Figura 38. Captura de pantalla de la aplicación de chat con interfaz gráfica.

Esta aplicación de charla mediante interfaz gráfica consta de dos archivos: *ServidorChat.java* (es el mismo del apartado anterior y se ejecuta en el servidor) y *ClienteGraficoChat.java* (se ejecuta en el cliente y se encarga de la interfaz gráfica).

En el lado del cliente, debido a la interfaz gráfica, se hace preciso relacionar las acciones que puede realizar el usuario (apretar el botón *Cerrar*, por ejemplo) con el código relacionado con las comunicaciones de red. Por ejemplo, si el usuario introduce un texto en el cuadro de texto (*txtMensaje*) y aprieta la tecla *Enter*, el programa lee la cadena de texto introducida y la envía al servidor mediante el flujo de salida del socket.

Cuando el cliente recibe un mensaje del servidor, lo lee del flujo de entrada del socket y lo escribe en el área de texto (*txtArea*), manteniendo los mensajes anteriores.

El código se presenta a continuación:

ClienteGraficoChat.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;


/**
 * Esta clase actúa como cliente para el chat. El usuario recibe los mensajes y los escribe por
 * medio de una interfaz gráfica muy simple.
 *
 * El programa considera que el programa servidor ServidorChat.java se ejecuta en la
 * ordenador local y por el número de puerto PUERTO.
 * En caso contrario, deberá cambiar en el constructor la máquina remota donde se ejecuta el
 * programa servidor y el número de puerto por el que escucha éste. Si además intenta usar este
 * programa cliente desde una red con cortafuegos o un proxy, ejecútelo con código como
 * éste: <p>
 * java -DproxySet=true -DproxyHost=PROXY -DproxyPort=PUERTO ClienteGraficoChat </p>
 * (PROXY es el nombre de la máquina que actúa como proxy o cortafuegos y PUERTO es el
 * número de puerto por el que escucha esta máquina).
 *
 */
public class ClienteGraficoChat extends JFrame {

    private static final int PUERTO= 9000; // Puerto por omisión:
    protected Socket socketCliente;
    private Sesion sesion;

    // flujos de E/S
    protected PrintStream salida;
    protected BufferedReader entrada;

    // Componentes gráficos Swing
    private JPanel contentPane;
    private JTextField txtMensaje = new JTextField();
    protected JTextArea txtArea = new JTextArea(); //Sesión necesita acceso a este objeto.
    private JButton btnCerrar = new JButton();
    private JScrollPane scrollPane = new JScrollPane();
    private JLabel lblTexto = new JLabel();

    /**
     * Constructor.
     */
    public ClienteGraficoChat() {
        // Por omisión, usa la máquina local y el núm. de puerto PUERTO. Incluyo el control de la
        // excepción java.net.UnknownHostException por si el lector quiere probar con
        // algún servidor remoto.

        // Creación de la interfaz gráfica.
        enableEvents(AWTEvent.WINDOW_EVENT_MASK);
```

```
        iniciar();

        // Creación del socket y de los flujos E/S asociados.
        try {
            socketCliente = new Socket("localhost", PUERTO);  // Máquina por defecto: la local
                                                              // Puerto por omisión del servidor: PUERTO
            salida = new PrintStream(socketCliente.getOutputStream());
            entrada = new BufferedReader(new InputStreamReader(socketCliente.getInputStream()));
            sesion = new Sesion(this);

            System.out.println("Arrancado el cliente");
        }
        catch (java.net.UnknownHostException e1) {
            // No se puede arrancar el cliente. Error irrecuperable: se sale del programa.
            // No hace falta limpiar el socket, pues no llegó a crearse.
            String mensaje ="No se localiza el ordenador servidor con esa dirección.";
            errorFatal(e1, mensaje);
        }
        catch (IOException e2) {
            // No se puede arrancar el cliente. Error irrecuperable: se sale del programa.

            // cierre del socket
            if (socketCliente != null) {
                try {
                    socketCliente.close();
                }
                catch (IOException e3) {} // No se hace nada con la excepción.
            }

            errorFatal(e2, "Fallo al intentar conectar con el servidor.");
        }
    }

    /** Punto de entrada a la aplicación. En este método se arranca el cliente
     * de chat.
     *
     * @param args
     * @throws IOException
     * @throws ConnectException
     */
    public static void main(String[] args) throws IOException, ConnectException {
        // Se toma la apariencia y sensación del sistema.

        System.out.println("Arrancando el cliente");

        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
            ClienteGraficoChat f = new ClienteGraficoChat();

            // Se centra el frame en la pantalla.
            Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();
            Dimension frameSize = f.getSize();
            if (frameSize.height > screenSize.height) {
                frameSize.height = screenSize.height;
            }
            if (frameSize.width > screenSize.width) {
```

```
            frameSize.width = screenSize.width;
        }
        f.setLocation((screenSize.width - frameSize.width) / 2,
            (screenSize.height - frameSize.height) / 2);

        // Se muestra el frame.
        f.show();
    }
    catch (Exception e) {
        e.printStackTrace();
        errorFatal(e, "Error en el gestor de apariencia y estilo");
    } // fin try-catch
}

/**
 * Inicia los componentes gráficos.
 */
private void iniciar()   {
    contentPane = (JPanel) this.getContentPane();
    txtMensaje.setBounds(new Rectangle(117, 49, 256, 29));

    txtMensaje.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
            txtMensaje_actionPerformed(e);
        }
    }
    ); // fin clase interna anónima
    contentPane.setLayout(null);
    this.setSize(new Dimension(400, 400));
    this.setTitle("Chat");
    btnCerrar.setText("Cerrar");

    btnCerrar.setBounds(new Rectangle(150, 323, 76, 29));
    btnCerrar.addActionListener(new java.awt.event.ActionListener() {
        public void actionPerformed(ActionEvent e) {
            btnCerrar_actionPerformed(e);
        }
    }
    ); // fin clase interna anónima

    scrollPane.setBounds(new Rectangle(9, 106, 370, 205));

    lblTexto.setBackground(Color.red);
    lblTexto.setFont(new java.awt.Font("Dialog", 1, 12));
    lblTexto.setText("Escriba texto:");
    lblTexto.setBounds(new Rectangle(17, 49, 87, 32));

    contentPane.add(txtMensaje, null);
    contentPane.add(btnCerrar, null);
    contentPane.add(lblTexto, null);
    contentPane.add(scrollPane, null);

    scrollPane.getViewport().add(txtArea, null);
}
```

```
    /**
     * Procesa el cierre del frame.
     *
     * @param e suceso
     */
    protected void processWindowEvent(WindowEvent e) {
        super.processWindowEvent(e);

        // Si se intenta cerrar el frame, envía la orden DESCONECTAR al servidor.
        if (e.getID() == WindowEvent.WINDOW_CLOSING) {
            salida.println("DESCONECTAR");
        }
    }

    /**
     * Procesa los eventos del cuadro de texto txtMensaje.
     *
     * @param e suceso
     */
    void txtMensaje_actionPerformed(ActionEvent e) {
        // Envía al servidor el mensaje escrito en txtMensaje.
        salida.println(txtMensaje.getText());

        // Vacía el cuadro de texto txtMensaje.
        txtMensaje.setText("");
    }

    /**
     * Procesa los eventos del botón btnCerrar.
     *
     * @param e suceso
     */
    void btnCerrar_actionPerformed(ActionEvent e) {
        salida.println("DESCONECTAR");
    }

    /**
     * Informa al usuario de la excepción arrojada y sale de la aplicación.
     *
     * @param excepcion  excepción cuya información se va a mostrar.
     * @param mensajeError  mensaje orientativo sobre la excepción .
     */
    private static void errorFatal(Exception excepcion, String mensajeError) {
        excepcion.printStackTrace();
        JOptionPane.showMessageDialog(null, "Error fatal."+ System.getProperty("line.separator") +
            mensajeError, "Información para el usuario", JOptionPane.WARNING_MESSAGE);
        System.exit(-1);
    }

}
```

```
/**
 * Esta clase representa el hilo que gestiona la entrada de mensajes procedentes del
 * servidor de chat.
 */
class Sesion extends Thread {

    private ClienteGraficoChat clienteChat;

    /**
     * Constructor.
     *
     * @param clienteChat ClienteGraficoChat
     * @throws IOException
     */
    public Sesion(ClienteGraficoChat clienteChat) throws IOException {

        // Mantiene una  referencia al objeto ClienteGraficoChat que lo genera.
        this.clienteChat = clienteChat;

        start(); // Se arranca el hilo
    }

    /**
     * Muestra por la interfaz gráfica los mensajes enviados por el servidor.
     */
    public void run(){
        // Última línea del aviso de desconexión por falta de actividad.
        String fin1 =  "> **********ADIOS**********";

         // Última línea del aviso de desconexión por uso de la orden DESCONECTAR.
        String fin2 = "> **********HASTA LA VISTA**********";

        String linea = null;

        try {
          while ( (linea=clienteChat.entrada.readLine()) != null ) {
              // Si se ha leído correctamente una línea, la muestra por el área de texto de la interfaz.
              clienteChat.txtArea.setText(clienteChat.txtArea.getText() + linea +
                      System.getProperty("line.separator"));

              // Si se produce una desconexión por que se ha terminado el tiempo permitido de
              // inactividad o porque se ha dado previamente la orden DESCONECTAR, se sale del
              // bucle tras una pausa de 5 segundos para que el usuario pueda leer el mensaje final de
              // despedida que envía el servidor.
              if ( linea.equals(fin1) || linea.equals(fin2) ) {
                  try {
                      this.sleep(5000); // El hilo "duerme" durante 5 segundos.
                      }
                   catch (java.lang.InterruptedException e1) {} // No se hace nada.
                  break;
              }
          }
        }
        catch (IOException e2) {
            e2.printStackTrace();
        }
```

```
        finally {
            // Se cierran el frame, el socket y los flujos de E/S del objeto ClienteGraficoChat asociado.

            try {
                clienteChat.dispose();
                clienteChat.entrada.close();
                clienteChat.salida.close();
                clienteChat.socketCliente.close();
            }
            catch (IOException e3) {} // No se hace nada con la excepción.

            System.exit(-1);  // salida
        }
    }

}
```

Por omisión, se considera que el programa cliente y el servidor se ejecutan en el ordenador local y que este último escucha peticiones por el puerto 9000. Si el servidor se ejecuta en un ordenador llamado *www.mimaquina.es* y escucha por un número de puerto *puerto*, habrá que igualar la constante *PUERTO* a *puerto* y escribir

```
socketCliente = new Socket("www.mimaquina.es", PUERTO);
```

en el constructor de la clase **ClienteGraficoChat**.

Al igual que sucedía con la clase **ClienteChat**, el uso de hilos en **ClienteGraficoChat** resulta imprescindible: por un lado, debe recogerse la entrada del usuario; por otro, deben mostrarse en la interfaz gráfica los mensajes de los otros usuarios.

La clase **Sesion** se encarga de leer los mensajes del servidor y de mostrarlos por la interfaz gráfica. Para ello, contiene un objeto *ClienteGraficoChat*. Si se produce una excepción, se envía la orden *DESCONECTAR* al servidor. Si está en marcha y no hay problemas de comunicación, responderá a la orden *DESCONECTAR* devolviendo al cliente un mensaje de cierre de conexión, lo borrará de la lista de clientes activos y registrará la desconexión en el archivo de historial. Por último, el cliente mostrará al usuario el mensaje de cierre de conexión durante cinco segundos, saldrá del bucle *while*, intentará cerrar los objetos que consumen recursos del SO y se cerrará.

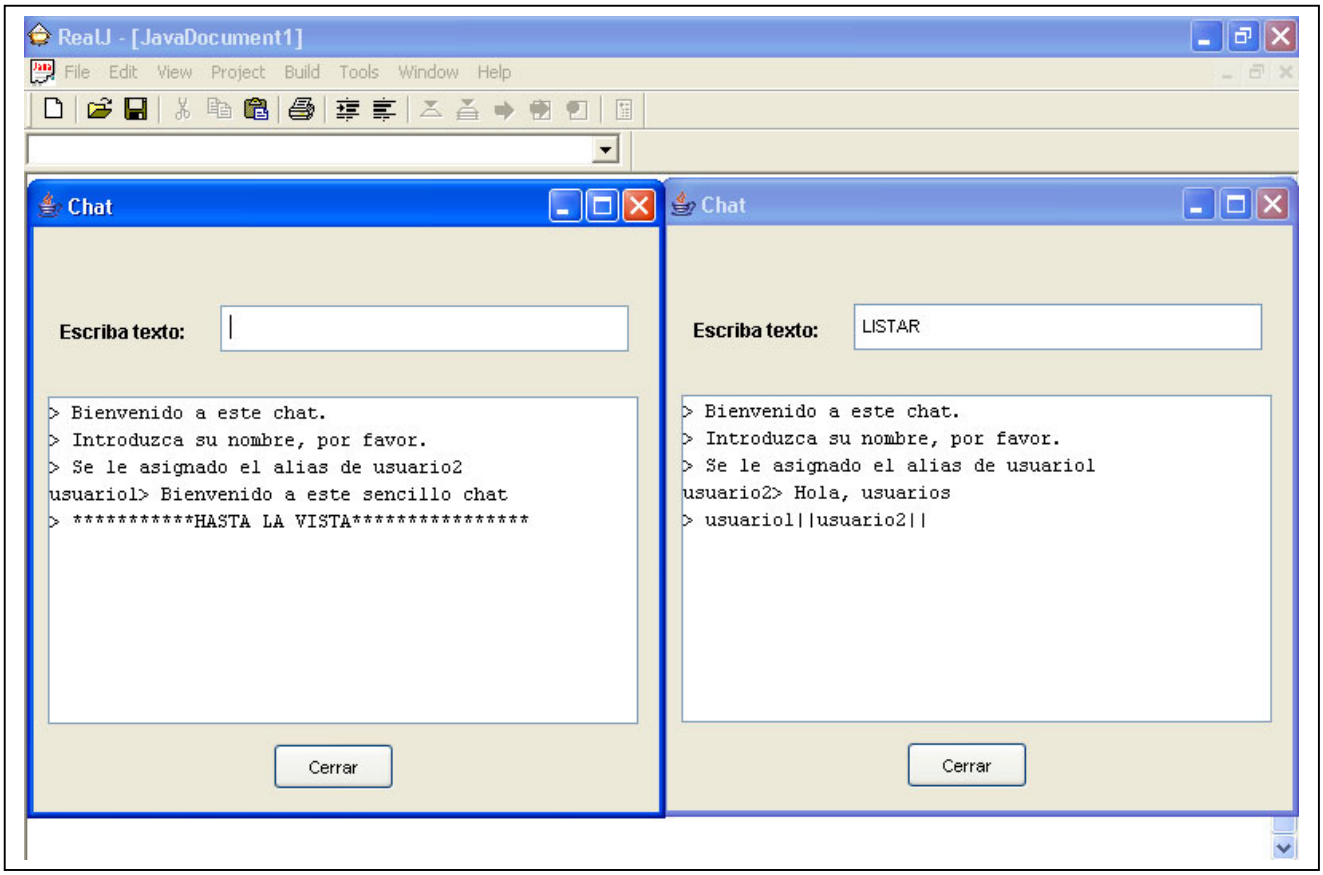

Figura 39: Ejemplo de uso de la aplicación de chat con interfaz gráfica. La ventana de la izquierda desaparecerá cinco segundos después de que aparezca la última línea.

6. NIO: UNA NUEVA API DE ENTRADA Y SALIDA PARA JAVA

6.1. Introducción

La versión 1.4 de Java (oficialmente, J2SE 2 versión 1.4) incorporó nuevas características al lenguaje, mantenidas en la versión 5.0 (antes conocida como 1.5 o *Tiger*). Las dos más importantes son las aserciones y el paquete **java.nio**. Este paquete ofrece una nueva API (interfaz de programación de aplicaciones) de entrada y salida (E/S), conocida como **NIO** (*New Input/Output*).

En lo que sigue, usaré *buffer* para referirme a las instancias de las clases que heredan de la superclase abstracta **java.nio.Buffer**, no a las memorias de almacenamiento temporal que usan algunas clases de **java.io** (véase el apartado 4).

La nueva API puede dividirse en tres grandes paquetes:

- **java.nio** define buffers, que se usan para almacenar secuencias de bytes o de otros valores primitivos (*int*, *float*, *char*, *double*, etc.).
- **java.nio.channels** define canales, esto es, abstracciones mediante los cuales pueden transferirse datos entre los buffers y las fuentes o los consumidores de datos (un socket, un fichero, un dispositivo de hardware). Además, proporciona clases que permiten E/S sin bloqueo.
- **java.nio.charset** contiene clases que convierten de manera muy eficaz buffers de bytes en bytes de caracteres y que permiten el uso de expresiones regulares.

NIO proporciona grandes ventajas sobre la E/S estándar de java (**java.io**). Si bien insistiré en ello más adelante, conviene dejar claro ya que NIO es un sustituto para las clases e interfaces de **java.io** ni una implementación mejorada de **java.io**: sus objetivos son muy distintos. Sun añadió mejoras en la implementación de **java.io** para la versión 1.4, pero el programador no precisa conocer NIO para aprovecharlas.

El porqué decidió Sun esperar hasta la versión 1.4 para incorporar NIO es un misterio (¿quizá por qué C# ha incorporado desde el comienzo sockets sin bloqueo?). El problema de la ineficacia de la E/S en Java ha estado presente desde las primeras versiones de Java. Resultaba contradictorio encontrarse ante un lenguaje con muchísimas características enfocadas hacia Internet y, al mismo tiempo, con un rendimiento tan pobre en cuanto a transferencias masivas de datos. Una de cal y una de arena, dirán algunos. Pese a que las discusiones sobre eficacia casi siempre se han orientado hacia la velocidad de ejecución del *bytecode* en la máquina virtual de Java, la recepción y envío de grandes cantidades de datos siempre ha sido el verdadero cuello de botella o eslabón débil en cuanto a rendimiento.

A continuación se exponen las principales características de NIO, en contraposición con la E/S estándar de Java:

- Incorpora buffers (contenedores de datos), canales (encargados de la transferencia de datos entre los buffers y las peticiones de E/S) y selectores (encargados de proporcionar el estado de los canales: si están preparados para leer, escribir, recibir conexiones).
- Permite la transferencia eficaz de grandes cantidades de datos. **java.io** emplea flujos de bytes; **java.nio**, en cambio, usa bloques de datos.

- Los sockets de NIO permiten trabajar sin bloqueo (también permiten bloqueo). Sin bloqueo, un solo hilo puede encargarse de controlar tantos canales como se precise. No se necesita destinar un hilo a cada canal, lo cual aumenta el rendimiento de la E/S. Asimismo, al no necesitarse un hilo por socket, la complejidad del código se reduce notablemente. Más trabajo para el sistema operativo y la máquina virtual Java, menos para el programador.
- Aprovecha las prestaciones del sistema operativo donde se ejecuta la MVJ.

En suma: NIO es más eficiente que la API de **java.io** para leer y escribir cantidades importantes de datos, pero no para manipularlos: **java.io** se sigue necesitando para el tratamiento de datos.

Los programadores no tienen por qué abandonar la API tradicional de E/S si no quieren; pero pueden considerar el uso de la nueva API de E/S si trabajan en aplicaciones que necesitan transferir con rapidez grandes bloques de datos, si quieren eliminar los cuellos de botella que pueden provocar las clases de **java.io** o si desean usar sockets sin bloqueo.

Para el lector que no haya tenido ningún contacto con **java.nio**, incluyo esta versión del famoso **HolaMundo**:

SaludosConNIO.java

```
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class SaludosConNIO {

   private String saludo = "Bienvenido a java.nio" + System.getProperty("line.separator");

   public SaludosConNIO() throws IOException {
      ByteBuffer buffer = ByteBuffer.allocate(23);
      buffer.put(saludo.getBytes());
      buffer.rewind();
      WritableByteChannel canal = Channels.newChannel(System.out);
      canal.write(buffer);
      canal.close();
   }

   public static void main(String args[]) throws IOException {
      new SaludosConNIO();
   }

}
```

6.2. La clase java.nio.Buffer

La clase **java.nio.Buffer** es una clase abstracta que tiene, por ahora, siete implementaciones:

- **java.nio.ByteBuffer**
- **java.nio.CharBuffer**
- **java.nio.IntBuffer**
- **java.nio.LongBuffer**
- **java.nio.ShortBuffer**
- **java.nio.FloatBuffer**
- **java.nio.DoubleBuffer**

Cada una de estas clases es un contenedor de datos primitivos (*byte*, *char*, *int*, etc.) con una capacidad limitada. En cada una, los datos se almacenan de manera lineal y secuencial. En comparación con las colecciones de Java, los buffers evitan la sobrecarga de la ubicación de los objetos en la pila y de la recolección de basura.

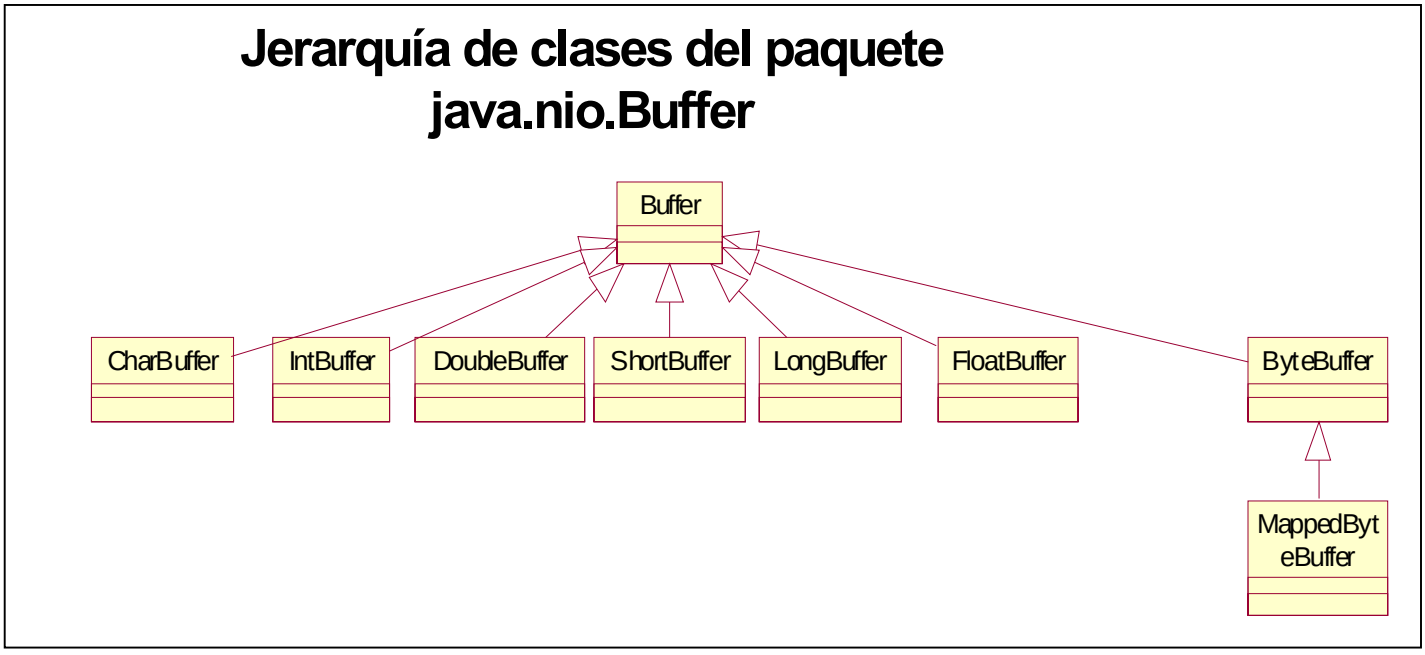

Figura 40. Jerarquía de clases de java.nio.Buffer

Las dos maneras más frecuentes de crear un buffer son mediante los métodos estáticos *allocate(int capacidad)* y *allocateDirect(int capacidad)*. El segundo permite crear buffers directos, esto es, buffers en los que la máquina virtual de Java intentará no usar memorias de almacenamiento temporal entre la MVJ y el sistema operativo en cada llamada a alguna operación específica del SO.

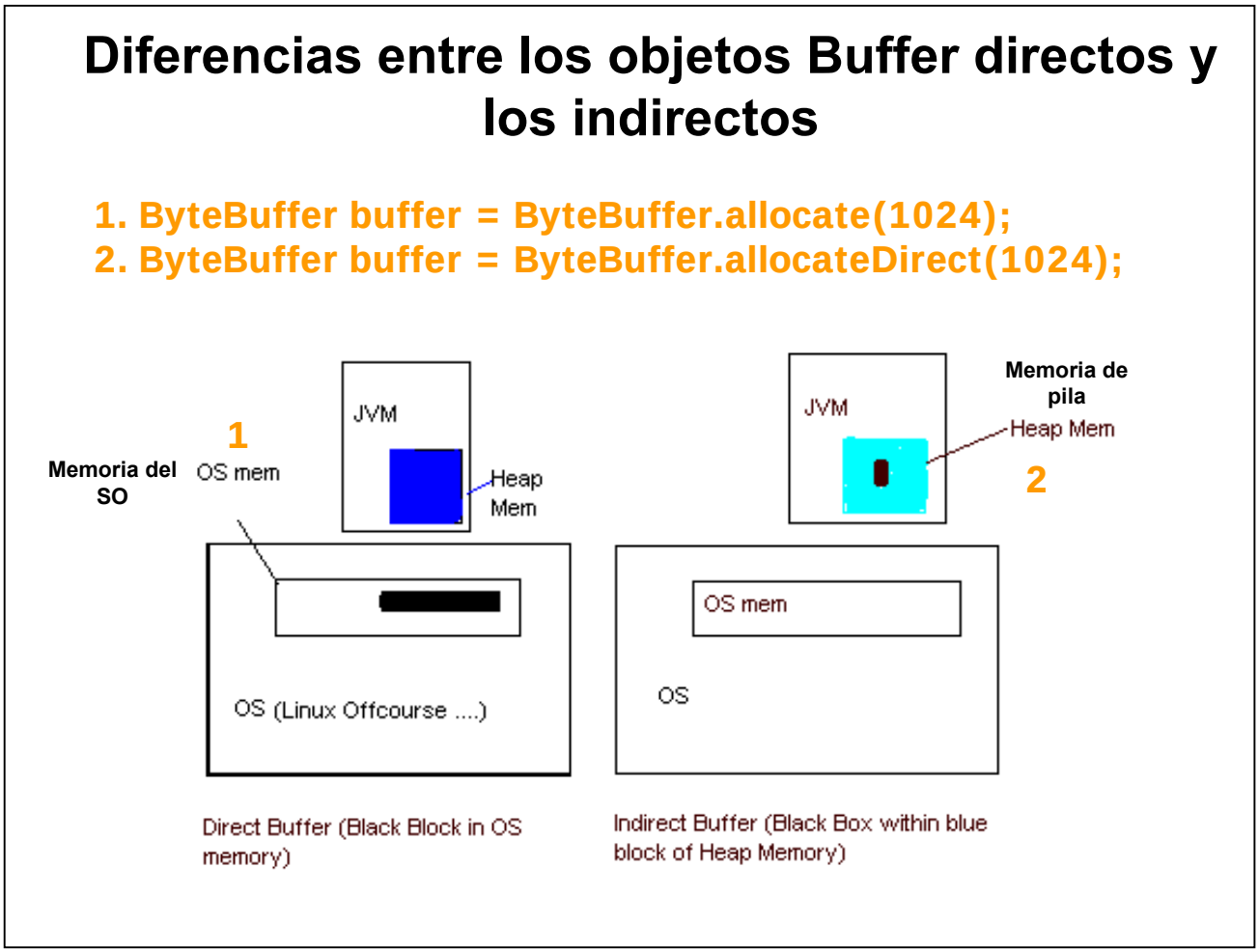

Figura 41. Ejemplo de creación de un buffer directo y de uno indirecto

Veamos varios ejemplos de creación de buffers:

```
ByteBuffer bb1 = ByteBuffer.allocate(256);
ByteBuffer bb2 = ByteBuffer.allocateDirect(256);
IntBuffer ib = IntBuffer.allocate(124);
DoubleBuffer db = DoubleBuffer.allocate(1024);
```

> **Advertencia:** No todos los sistemas operativos admiten la posibilidad de utilizar buffers directos. Si encuentra que el código que contiene llamadas a *allocateDirect()* genera errores, cámbielas por llamadas a *allocate()*.

En un buffer, la **capacidad** es el número de elementos que puede contener; el **límite**, el índice del primer elemento que no puede leerse o escribirse (porque no hay nada más en esa posición ni en las siguientes); la **posición**, el índice del próximo elemento que va a leerse o escribirse. La posición y el límite son menores o iguales a la capacidad, y la posición no puede ser mayor que el límite.

La capacidad especifica cuántos datos pueden ponerse en el buffer; el límite, cuántos se han puesto realmente en él. La capacidad de un buffer es inmodificable: una vez creado, ésta no puede cambiarse. Cuando se crea uno, la posición se iguala a cero y el límite toma el valor de la capacidad.

Hay tres métodos que deben conocerse para realizar operaciones de lectura y escritura con buffers:

1) *public final Buffer* ***clear()*** limpia el buffer. La posición se iguala a cero, y el límite a la capacidad.
2) *public final Buffer* ***flip()*** "da la vuelta" al buffer. El límite se establece en la posición actual y la posición se iguala a cero. Tras una llamada a este método, el buffer queda preparado para que los canales puedan leer sus datos.
3) *public final Buffer* ***rewind()*** rebobina el buffer: la posición se iguala a cero. Tras una llamada a este método, el buffer queda listo para que sus datos puedan escribirse en cualquier canal.

Su funcionamiento se muestra en las figuras 42 y 43:

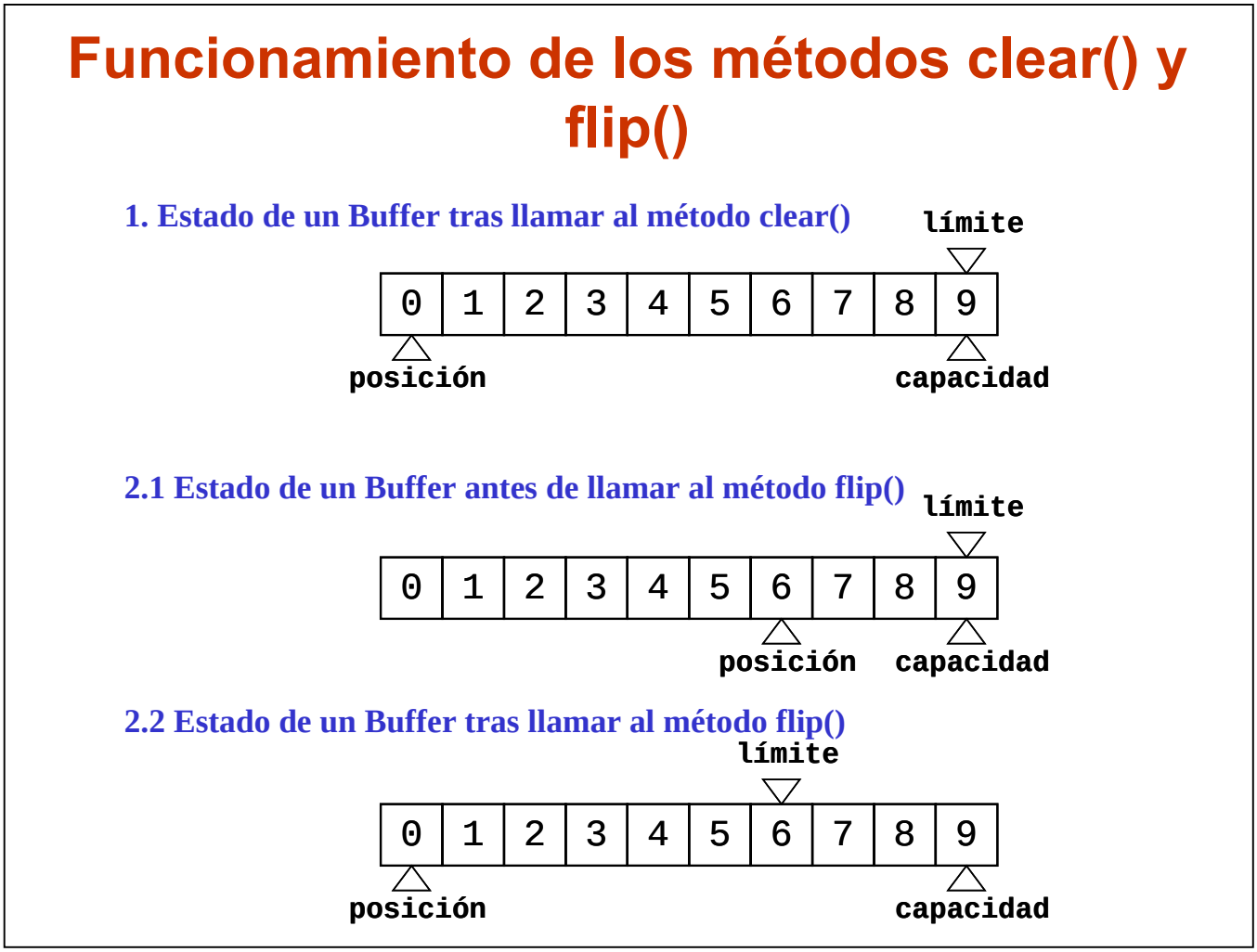

Figura 42. Los métodos clear() y flip() en acción. La capacidad del buffer es nueve (en la novena posición no se puede leer ni escribir)

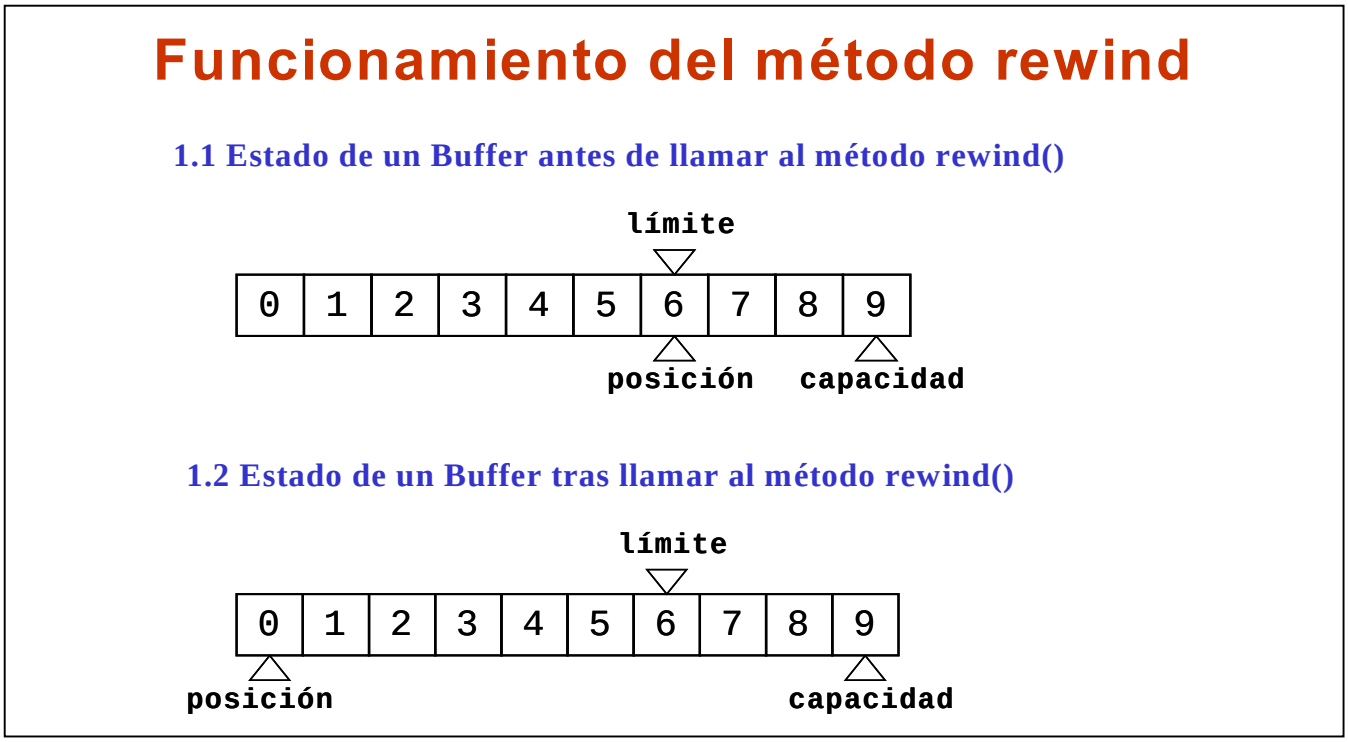

Figura 43. El método rewind() en acción. La capacidad del buffer es nueve (en la novena posición no se puede leer ni escribir)

Cada clase **Buffer** cuenta con un método *public final int* ***remaining()***, que devuelve el número de elementos que quedan por leer o por escribir, es decir, entre la posición actual y el límite; *public final boolean* ***hasRemaining()*** nos permite saber si existen elementos entre la posición actual y el límite.

Asimismo, cada una de estas clases tiene sus métodos *put()* y *get()*, que permiten leer o escribir un dato del tipo correspondiente (*char*, *int*, etc.). En la clase **ByteBuffer**, estos métodos tienen la forma:

public abstract byte get() throws BufferUnderflowException
public abstract byte get(int indice) throws IndexOutOfBoundsException
public ByteBuffer get(byte[] dst) throws BufferUnderflowException
public ByteBuffer get(byte [] dst, int offset, int longitud) throws BufferUnderflowException, IndexOutOfBoundsException
public abstract ByteBuffer put(byte b) throws BufferOverflowException, ReadOnlyBufferException
public abstract ByteBuffer put(int indice, byte b) throws IndexOutOfBoundsException, ReadOnlyBufferException
public final ByteBuffer put(byte[] src) throws BufferOverflowException, ReadOnlyBufferException
public ByteBuffer put(byte [] src, int offset, int longitud) throws BufferOverflowException, IndexOutOfBoundsException, ReadOnlyBufferException
public ByteBuffer put(ByteBuffer src) throws BufferOverflowException, IllegalArgumentException, ReadOnlyBufferException

Veamos dos ejemplos de uso de estos métodos:

```
// 1 Creación del buffer (buffer indirecto)
DoubleBuffer db = DoubleBuffer.allocate(256);

// Se rellena el buffer
for (int i=0; i < db.capacity(); i++)
    db.put(i * 1.0);

// Se deja el buffer preparado para lecturas.
db.flip();

// Se lee del buffer.
int i= 0;
while (db.hasRemaining()) {
    double tmp = db.get();
    System.out.println("El elemento " + i + " del buffer es " + tmp);
    i++;
}

// Se muestra el primer elemento del buffer (hay que rebobinar antes).
db.rewind();
double tmp1 = db.get();
System.out.println("El primer elemento del buffer es " + tmp1);

// Se limpia el buffer.
db.clear();


// 2 Creación del buffer (buffer directo)
ByteBuffer bb = ByteBuffer.allocateDirect(1024);

// Escritura en el buffer (putDouble() introduce doubles)
bb.putDouble(2.718);
bb.putDouble(3.1415);

// Se deja el buffer preparado para lecturas.
bb.flip();

// Se lee del buffer.
System.out.println(bb.getDouble()); // muestra 2.718
System.out.println(bb.getDouble()); // muestra 3.1415

// Se limpia el buffer.
bb.clear();
```

En el primer ejemplo, la versión de *get()* usada obtiene el *double* cuyo índice coincide con la posición y aumenta una unidad la posición (siempre que no se haya alcanzado el límite: el método nunca puede leer del límite o de más allá).

El método *put()* coloca su argumento (un *double*) en el elemento del buffer cuyo índice es igual a la posición y aumenta ésta en una unidad (siempre que no se haya colocado en el límite: el método nunca puede escribir en el límite o más allá).

6.3. Los canales y la clase java.nio.channels.ServerSocketChannel

Según la documentación oficial de Java, un canal (*channel*)

> [...] representa una conexión abierta a una entidad como un dispositivo de hardware, un archivo, un socket de red o un componente de software que es capaz de realizar una o más operaciones distintas de E/S; por ejemplo, leer o escribir. Un canal está abierto tras su creación, y una vez cerrado permanece cerrado. Una vez que un canal está cerrado, cualquier intento de llamar a una operación de E/S sobre él causará que se arroje una *ClosedChannelException.*

Dicho de otra manera: un canal es una conexión entre un buffer y una fuente o un consumidor de datos (un socket, un archivo, etc.). Los datos pueden leerse de los canales mediante buffers, y los datos de un buffer pueden escribirse en los canales. Es decir, los buffers pueden escribirse en canales o leerse de éstos.

Los canales pueden operar con bloqueo o sin él. Una operación de E/S con bloqueo no retorna hasta que se produce una de estas situaciones: a) se completa la operación; b) se produce una interrupción debida al sistema operativo; c) se lanza una excepción. Todos los métodos *read()* y *write()* de **java.io** producen bloqueos hasta que se produce a), b) o c).

Una operación de E/S sin bloqueo retorna al instante, devolviendo algún valor de retorno que indica si la operación se realizó correctamente o no. Un programa o un hilo que ejecute una operación sin bloqueo no se quedará "dormido" esperando datos, una interrupción o una excepción: su curso de ejecución continuará.

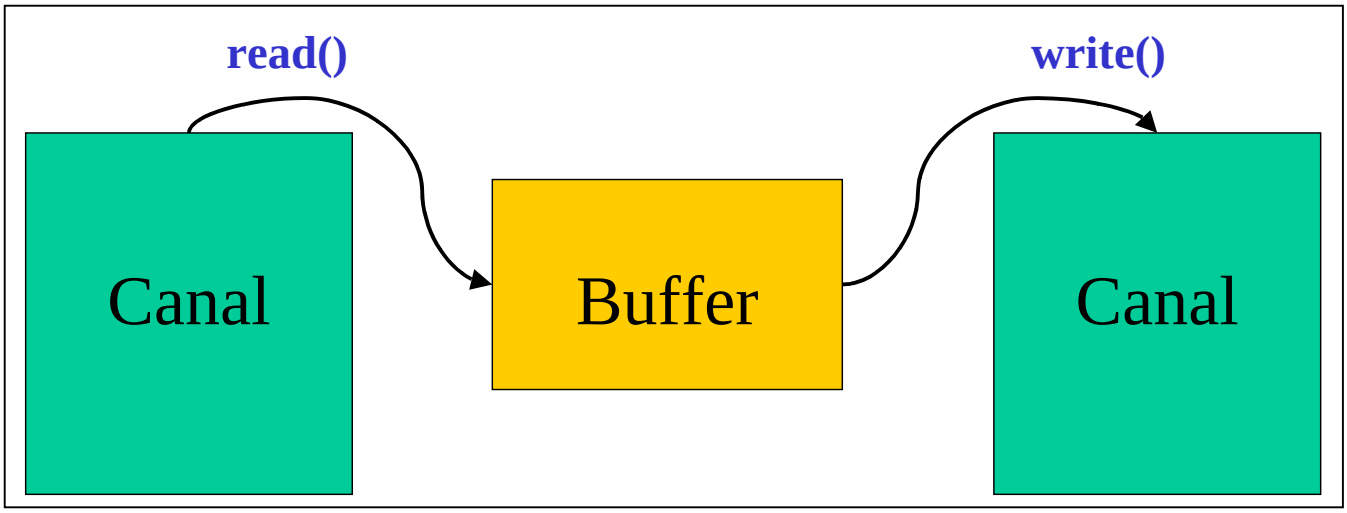

Figura 44. Los buffers son los intermediarios entre los canales. No es posible pasar directamente datos de un canal a otro. Para leer datos de un canal y escribirlos en otro, hay que leer del primero a un buffer y, luego, escribir del buffer al segundo canal.

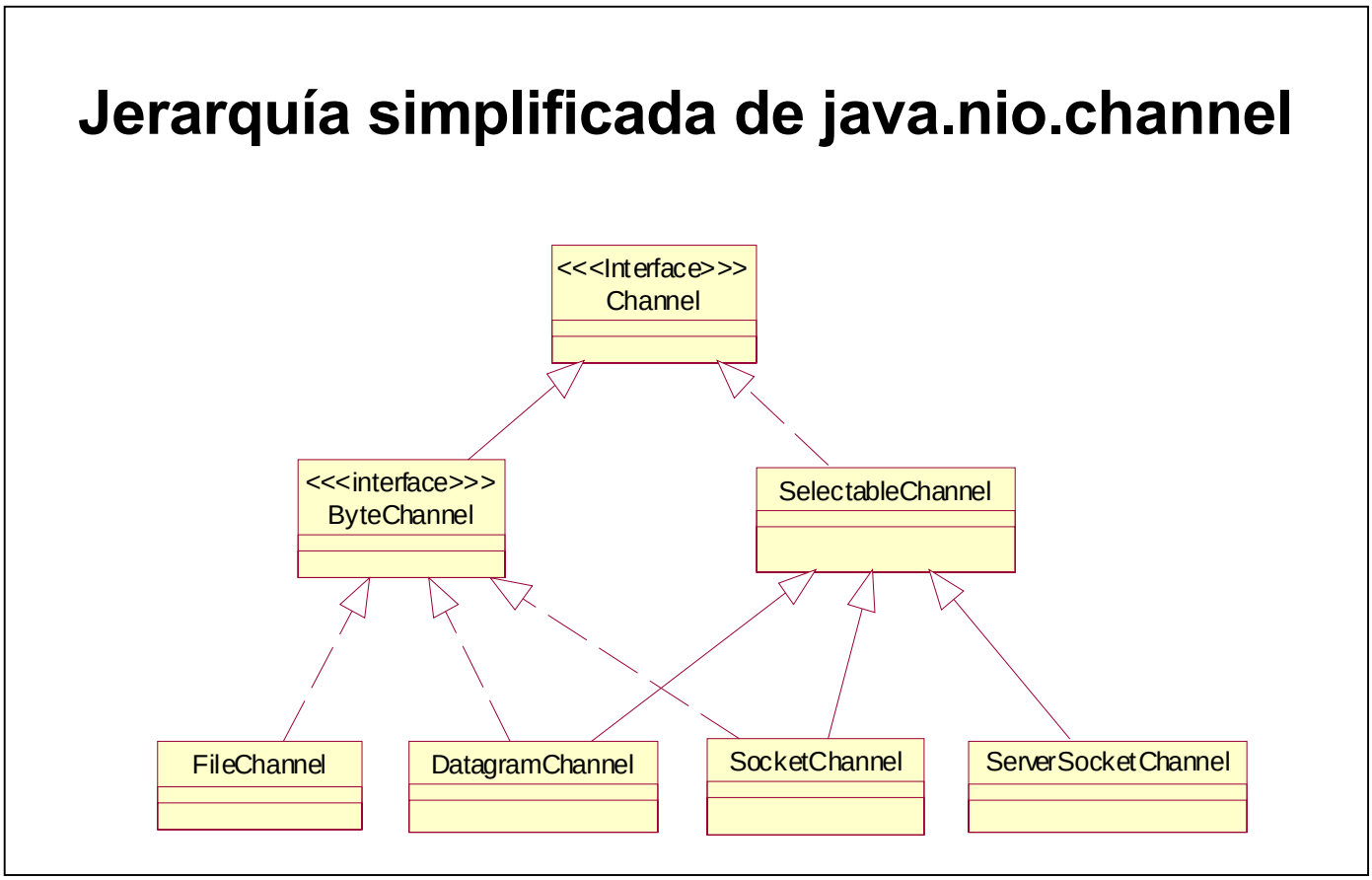

Figura 45. Esquema simplificado de la jerarquía de java.nio.channel

El lector que haya revisado la documentación de los paquetes **java.io** y **java.net** de la versión 1.4 de Java o de la 5.0 habrá notado que hay diferencias respecto a las versiones anteriores: aparte de las clases nuevas, algunas de las antiguas tienen un método *getChannel()* del que carecían antes:

- **java.io.FileInputStream**
- **java.io.FileOutputStream**
- **java.io.RandomAccessFile**
- **java.net.ServerSocket**
- **java.net.Socket**
- **java.net.DatagramSocket**
- **java.net.MulticastSocket**
- **java.net.SocketInputStream**
- **java.net.SocketOutputStream**

Por ejemplo, el método *getChannel()* de la clase **java.net.ServerSocket** se declara como *public ServerSocketChannel* ***getChannel()***.

Los canales que pueden obtenerse con *getChannel()* vienen a ser objetos paralelos a los tradicionales. Las nueve clases anteriores mantienen las funciones que tenían en las versiones de Java anteriores a la 1.4; los canales vienen a ser como “envoltorios” que proporcionan las funciones propias de la nueva API de E/S.

Con NIO, los programadores pueden trabajar con dichas clases usando flujos, como siempre se había hecho hasta J2SE 1.4, o pueden extraer de ellas canales y trabajar con buffers para leer y escribir. En el primer caso, se trabaja con bytes; en el segundo, con bloques de datos. La decisión dependerá de las características particulares de la aplicación que se desee. Si interesa mover muchos datos sin procesarlos, NIO es lo más recomendable. Si se necesita procesarlos (hacer comprobaciones sobre ellos, cambiar unos bytes por otros, etc.), la API estándar de E/S resultará ineludible.

Las clases **java.nio.channels.DatagramChannel**, **java.nio.channels.FileChannel**, **java.nio.channels.Pipe.SinkChannel** y **java.nio.channels.SocketChannel** implementan el método *public int* ***write****(ByteBuffer bb) throws IOException, NonWritableChannelException, ClosedChannelException, AsynchronousCloseException, ClosedByInterruptException* de la interfaz **java.nio.channels.WritableByteChannel**. El entero devuelto por el método corresponde al número de bytes escritos en el canal (o –1 si el canal ha alcanzado el fin del flujo de datos). Asimismo, *write()* actualiza la posición actual del buffer *bb*.

Las clases **java.nio.channels.DatagramChannel**, **java.nio.channels.FileChannel**, **java.nio.channels.Pipe.SourceChannel** y **java.nio.channels.SocketChannel** implementan el método *public int* ***read****(ByteBuffer bb) throws IOException, NonReadableChannelException, ClosedChannelException, AsynchronousCloseException, ClosedByInterruptException* de la interfaz **java.nio.channels.ReadableByteChannel**. El entero que devuelve el método es el número de bytes leídos del canal. Este valor viene limitado por la posición actual del buffer *bb* y su capacidad, ya que un buffer podrá leer, como mucho, *remaining()* bytes. Igualmente, *read()* actualiza la posición del buffer *bb*.

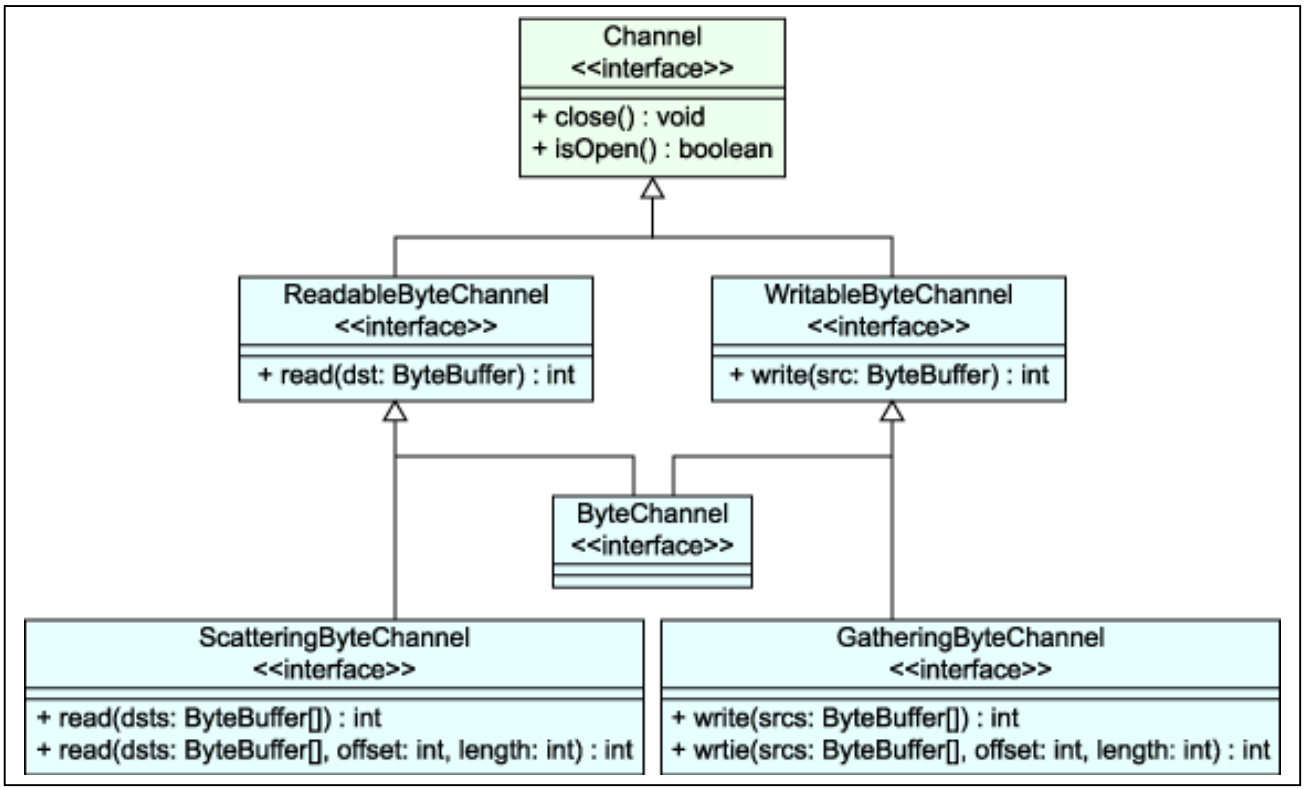

Figura 46. Esquema del inicio de la jerarquía de java.nio.channel

La manera habitual de transferir datos a un canal se muestra aquí:

```
// ByteBuffer para almacenar los datos y pasarlos de un canal a otro
// Es decir, de canalFuente a canalDestino.
ByteBuffer buffer = ByteBuffer.allocateDirect(BUF_SIZE) ;

// El bucle se ejecutará hasta que se alcance el
// final de los datos.
while (canalFuente.read(buffer) != -1) {
   buffer.flip() ;    // Se prepara el buffer para ser escrito.

   // Este bucle es necesario porque puede ser que no se escriban todos los datos
   // en una sola llamada a write().
   while(buffer.hasRemaining()) {
      canalDestino.write(buffer) ;
   }

   buffer.clear() ;    // Se vacía el buffer. Queda listo para leer más datos.
}
```

La clase **java.nio.channels.ServerSocketChannel** es un canal seleccionable para sockets TCP pasivos (más adelante veremos el significado del término *seleccionable*). Por decirlo de manera llana, un objeto *ServerSocketChannel* viene a ser un envoltorio para un objeto *ServerSocket*, al cual asocia un canal. Para crear un objeto *ServerSocketChannel* se usa el método *public static ServerSocketChannel **open**() throws IOException*.

Un *ServerSocketChannel* recién creado no está ligado a una dirección ni a un puerto. La ligadura se consigue mediante uno de los métodos *bind()* de la clase **java.net.ServerSocket**, vista en el apartado 3.

El método *public abstract ServerSocket **socket**()* devuelve un socket de servidor asociado con el canal.

El método *abstract SocketChannel **accept**() throws IOException* acepta una conexión hecha al socket del canal. Si el *ServerSocketChannel* está en modo no bloqueado, devolverá *null* si no hay conexiones pendientes; en caso contrario, se bloqueará indefinidamente hasta que llegue una conexión o se produzca una excepción de E/S.

El método *public abstract SelectableChannel **configureBlocking**(boolean bloqueo) throws IOException, ClosedChannelException, IllegalBlockingModeException* procede de la clase **java.nio.channels.SelectableChannel** y establece si el canal podrá bloquearse o no. Es decir, determina si las operaciones de E/S sobre el canal, como *write()* o *read()*, lo bloquearán o no.

A diferencia de los otros canales, un *ServerSocketChannel* sólo puede establecer conexiones. La transmisión de datos se realiza mediante los canales asociados a los sockets que se obtienen de él cuando se llama al método *accept()*.

Si se configura sin bloqueo un *ServerSocketChannel*, el método *accept()* no bloqueará el programa en ejecución hasta que reciba una conexión. Consideremos el siguiente código:

```
ServerSocketChannel canalSocketServidor = ServerSocketChannel.open();

canalSocketServidor.socket().bind(new InetSocketAddress ("localhost", 9000));
canalSocketServidor.configureBlocking (false);

while (true) {
   SocketChannel canalSocket = canalSocketServidor.accept();

   if (canalSocket == null) {
      // Código 1: la llamada a accept() devuelve null si no hay ninguna llamada entrante.
   }
   else {
      // Código 2: aquí se manejan las conexiones.
   }
}
```

Mientras no se reciban conexiones, se ejecutará el código 1; situación imposible en el caso de que se use la clase **java.net.ServerSocket**, pues las llamadas a su método *accept()* bloquerían la ejecución del programa hasta que hubiera alguna conexión. De ahí la necesidad de crear hilos cuando se usa **ServerSocket**.

> **Nota:** Crear un canal de sockets crea implícitamente un objeto socket del paquete **java.net**. Lo contrario no es cierto (prueba de ello es todo el apartado 3).

Crear un objeto *ServerSocketChannel* y ligarlo a una dirección de red resulta sencillo:

```
// Se crea un canal para el socket de servidor (ServerSocketChannel)
ServerSocketChannel canalSocketServidor = ServerSocketChannel.open();

// Se le asigna el modo de no bloqueo
canalSocketServidor.configureBlocking(false);

// Se liga el canal con una dirección de red (nombre del ordenador, puerto)
// mediante un socket de servidor (ServerSocket)
ServerSocket socketServidor = canalSocketServidor.socket();
socketServidor.bind(new java.net.InetSocketAddress("localhost", 9000));
```

La clase **java.net.**InetSocketAddress, introducida en la J2SE 1.4, implementa una dirección IP de socket mediante un par (dirección IP, número de puerto) o (nombre del ordenador, número de puerto).

Si un socket se configura sin bloqueo, se cumple lo siguiente:

a) Una llamada a *read()* transferirá los bytes disponibles en el momento de la llamada. Si no hay datos disponibles, devolverá 0.
b) Una llamada a *write()* transferirá a un socket los datos disponibles en ese mismo momento. Si no hay datos disponibles, devolverá 0.
c) Una llamada a *accept()* devolverá *null* si en ese momento no hay ningún cliente que intente conectarse.

6.4 La clase java.nio.channels.SocketChannel

La clase **java.nio.channels.SocketChannel** es un canal seleccionable para sockets TCP activos (más adelante veremos el significado del término *seleccionable*). Un objeto *SocketChannel* viene a ser un envoltorio para un objeto *Socket*, que permite asociarle un canal. Las operaciones de enlazado con una dirección IP y un puerto, de cierre, etc., se realizan mediante el socket asociado.

Los objetos de la clase **SocketChannel** se crean mediante llamadas a los métodos estáticos *open()* de la clase. El método *public abstract boolean **connect**(SocketAddress direccionRemota) throws IOException* conecta un canal de socket con un objeto *InetSocketAddress*. El método *public abstract Socket **socket**()* devuelve el socket asociado al canal.

Al igual que sucede con los *ServerSocketChannels*, los *SocketChannels* también pueden configurarse sin bloqueo, de forma que las operaciones de E/S no bloqueen el programa en ejecución. Veamos un ejemplo:

```
// Se crea un objeto SocketChannel
SocketChannel canalSocket = SocketChannel.open();

// Se conecta con un objeto InetSocketAddress (es decir, con un ordenador y un puerto)
canalSocket.connect(new java.net.InetSocketAddress("localhost", 9000));

// Se configura sin bloqueo. Las llamadas a read(), write() o connect() no bloquearán la
ejecución del programa.
canalSocket.configureBlocking(false);

// Se crea un ByterBuffer y se lee del canal.
ByteBuffer buffer = ByteBuffer.allocate(1024);
buffer.clear();
canalSocket.read(buffer);
```

Nota: En el caso de las llamadas al método *connect()* siempre existe un corto bloqueo, se haya configurado el *socket* como con bloqueo o sin él. El bloqueo se debe a la naturaleza del protocolo TCP: se precisa un cierto tiempo para que el cliente conecte con el servidor y para que éste envíe una respuesta que indique su disponibilidad para aceptar peticiones.

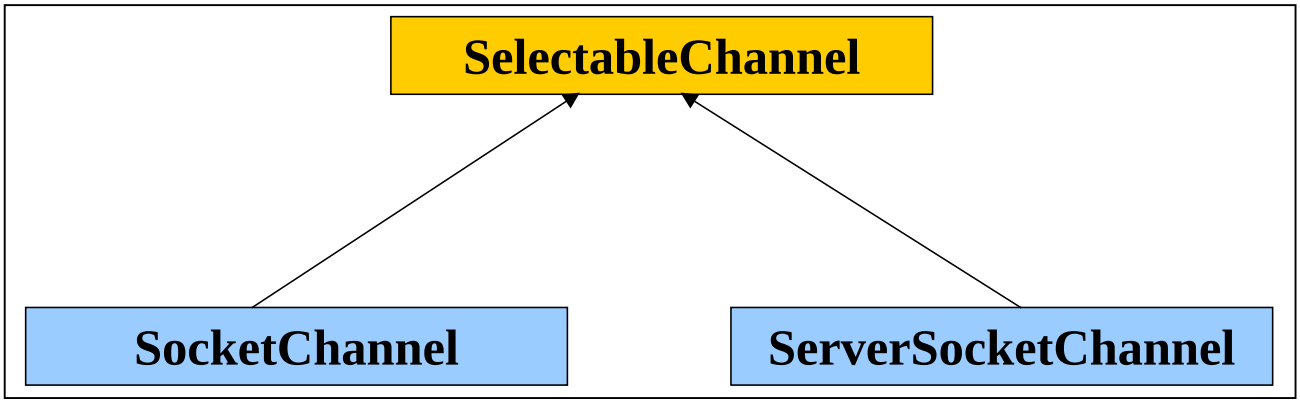

Figura 47. SocketChannel y ServerSocketChannel descienden de la superclase SelectableChannel

6.5 Las clases java.nio.channels.Selector y java.nio.channels.SelectionKey

La clase **java.nio.channels.Selector** es una de las principales de la API NIO. Un objeto *Selector* controla una serie de canales y lanza un aviso cuando uno de ellos lanza un suceso de E/S. La clase **Selector** informa a la aplicación de las operaciones de E/S que ocurren en los canales que ésta mantiene activos.

La información sobre las operaciones de E/S se registra en un conjunto de claves, que son instancias de la clase **java.nio.channels.SelectionKey**. Cada clave almacena información sobre el canal que desencadena la operación y el tipo de ella (lectura, escritura, conexión entrante, conexión aceptada).

En la clase **Selector**, las instancias se crean con el método estático *public static Selector **open**() throws IOException*.

El método *public abstract int **select**() throws IOException* bloquea el programa hasta que algún canal recibe datos. Los ingenieros de Sun no han querido disimular sus orígenes: tiene el nombre de una llamada del sistema operativo UNIX que permite a los programas escritos en C trabajar a la vez con varias fuentes de datos. Rebuscando un poco en la documentación de varios sistemas UNIX, he podido encontrar un primitivo servidor web escrito en C para la plataforma Solaris. La estrategia que usa para atender simultáneamente a varios clientes es casi idéntica a la que se emplearía con Java para implementar un servidor multiusuario con NIO (eso sí, el código en C es mucho más *verboso*).

Este método supone para el programador de Java una importante herramienta para aumentar la claridad y eficacia de su código. El método *select()* se bloquea hasta que uno o más canales reciban algún suceso de E/S. No es necesario, por tanto, comprobar proceso a proceso –como había que hacer antes de la J2SE 1.4– si se han recibido, por ejemplo, entradas o conexiones de los clientes. Con una llamada a *select()* se espera simultáneamente a todas las entradas de los clientes.

Para saber qué sucesos E/S de los que estamos interesados se producen en un determinado canal es necesario registrar el canal con el selector y especificar el tipo o los tipos de sucesos de interés. Esto se realiza mediante el método *public final SelectionKey **register**(Selector sel, int ops) throws ClosedChannelException* del canal. Veamos un ejemplo:

```
// Se obtiene una dirección de socket proporcionando el nombre y el puerto.
java.net.InetSocketAddress direcSocket= new java.net.InetSocketAddress("localhost", 9000);

// Se crea un objeto SocketChannel y se conecta al objeto InetSocketAddress.
// Se configura sin bloqueo.
ServerSocketChannel canalSocketServidor = ServerSocketChannel.open();
canalSocketServidor.configureBlocking(false);
canalSocketServidor.socket().bind(direcSocket);

// Se crea un objeto Selector.
Selector selector = Selector.open();

// Se registra el canal con selector para que “esté al tanto” de operaciones
// de conexiones, de lectura y de escritura.
canalSocketServidor.register(selector, SelectionKey.OP_CONNECT |
                                       SelectionKey.OP_READ |SelectionKey.OP_WRITE);
```

Se dice que una clase es *seleccionable* si puede registrarse con un selector. Todos los canales que descienden de la clase **java.nio.channels.SelectableChannel** son seleccionables. Un canal seleccionable es un canal que puede multiplexarse.

> **Nota:** El concepto de **multiplexión** es muy común en telecomunicaciones. En general, el término se refiere a la combinación de varias señales o cadenas de datos en un mismo canal de transmisión. La multiplexión puede ser física (varias señales eléctricas pueden compartir un mismo cable, varias señales electromagnéticas pueden viajar por una misma guía de ondas) o lógica (como en el caso de los canales seleccionables: el canal no existe físicamente).

Los sucesos de E/S para los que se puede registrar un canal mediante un selector se especifican con las siguientes constantes enteras:

- **SelectionKey.OP_READ**. Registra el canal para sucesos de lectura.
- **SelectionKey.OP_WRITE**. Lo registra para sucesos de escritura.
- **SelectionKey.OP_ACCEPT**. Lo registra para las peticiones entrantes de los clientes (se usa en servidores).
- **SelectionKey.OP_CONNECT**. Lo registra para las conexiones aceptadas por los servidores (se usa en clientes).

Distintos canales pueden registrarse, para diversos sucesos, en un mismo selector; de ahí que se diga que un conjunto de canales se multiplexa en un selector. He aquí un ejemplo:

```
// canal1, canal2 y canal3 son canales seleccionables sin bloqueo. Mientras permanezcan
// registrados en el selector deben permanecer en ese estado.
Selector selector = Selector.open();
canal1.register (selector, SelectionKey.OP_READ) ;
canal2.register (selector, SelectionKey.OP_WRITE) ;
canal3.register (selector, SelectionKey.OP_READ | SelectionKey.OP_WRITE) ;
```

Un canal *ServerSocketChannel* puede registrarse con *OP_ACCEPT*, *OP_READ* y *OP_WRITE*. Asimismo, uno *SocketChannel* puede registrarse con *OP_CONNECT*, *OP_READ* y *OP_WRITE*.

Cada objeto *SelectionKey* representa la clave o identificador de un canal que ha lanzado un suceso de E/S. Por ello, el conjunto de *SelectionKeys* que devuelve una llamada al método **selectedKeys**() es equivalente a un conjunto de canales.

Un *SelectionKey* es válido hasta que se cierra el canal, el selector o se llama al método *cancel()*.

Para eliminar un canal de un selector (desregistrarlo), se usa el método *public abstract void **cancel**()* del objeto *SelectionKey* asociado. Si se cancela una clave, su canal se desregistrará del selector durante la próxima operación con *select()*. Si un canal se cierra, se desregistra de todos los selectores donde esté registrado.

El método *public abstract void **close**() throws IOException* cierra el selector. Al cerrarse, hace lo siguiente: a) invalida todas las claves asociadas con él; b) desregistra

todos los canales asocidos; y c) libera los recursos del sistema operativo que tenía ocupados.

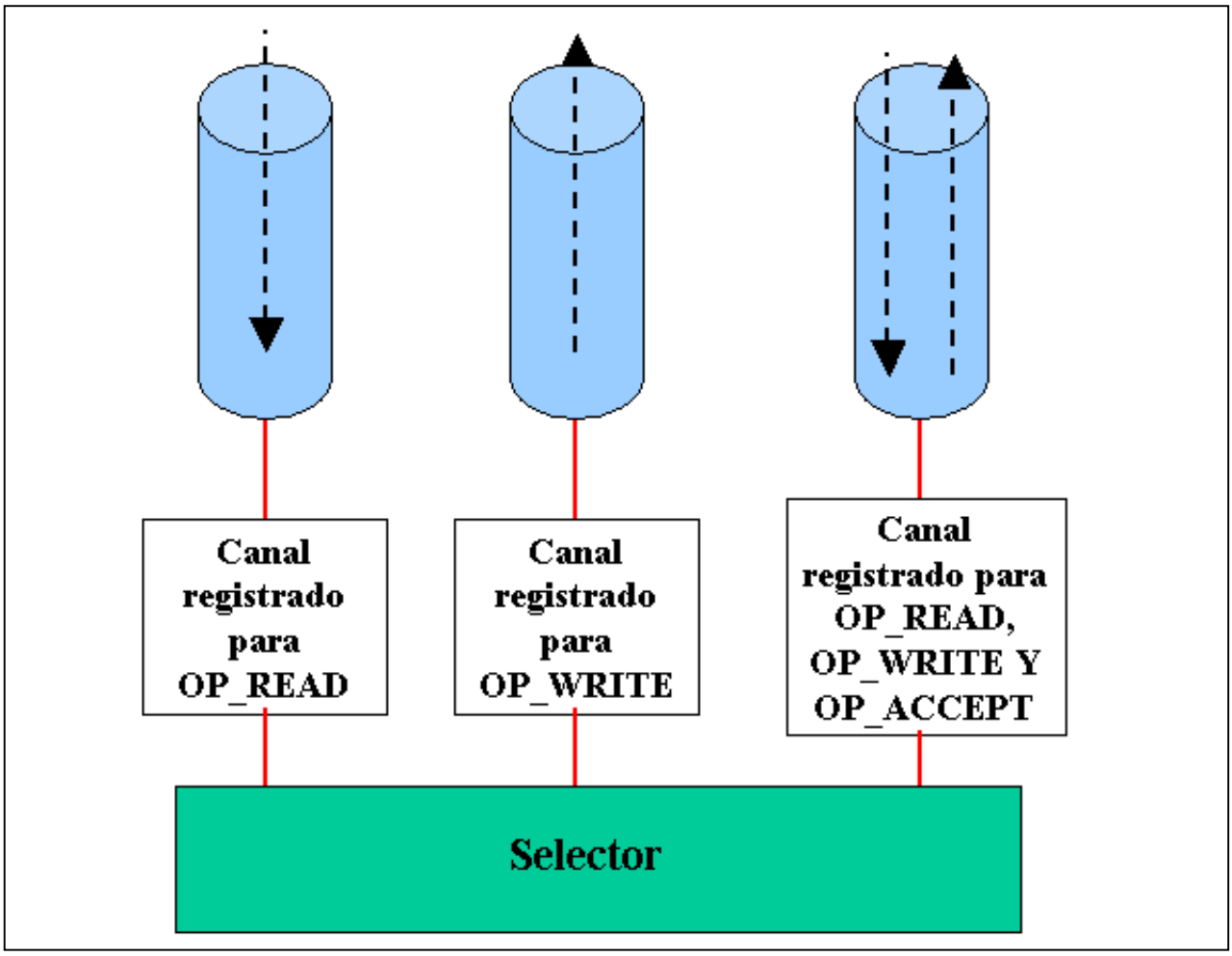

Figura 48. Varios canales pueden registrarse para distintas operaciones en un mismo selector

El método *public abstract Set* ***selectedKeys****() throws ClosedSelectorException* devuelve un conjunto (*Set*) de claves correspondientes a los canales que han recibido operaciones de E/S.

Cuando uno o más sucesos de E/S han sido lanzados por cualquiera de los canales registrados, puede accederse así al conjunto de claves de los canales con sucesos:

```
Set claves = selector.selectedKeys();
```

Con el siguiente código, se puede recorrer el conjunto *claves* y procesar cada suceso:

```
Iterator it = claves.iterator();
while (it.hasNext()){
   SelectionKey clave = (SelectionKey) it.next();
   if ( clave.isAcceptable() ) {
      // Código de aceptación de la conexión entrante.
   }
   if ( clave.isConnectable() ) {
```

```
            // Código de procesado de la conexión.
        }
        if ( clave.isReadable() ) {
            // Código de lectura
        }
        if ( clave.isWritable() ) {
            // Código de escritura.
        }

        // Nótese que remove quita la clave del Set asociado al
        // iterador, no solo de éste.
        it.remove(clave);
    }
```

Los métodos de la clase **SelectionKey** *public final boolean **isAcceptable**() throws CancelledKeyException*, *public final boolean **isConnectable**() throws CancelledKeyException*, *public final boolean **isReadable**() throws CancelledKeyException* y *public final boolean **isWritable**() throws CancelledKeyException* comprueban el tipo de suceso producido en el canal seleccionado (conexión entrante, conexión aceptada, recepción de datos o envío de datos).

Es necesario usar el método *remove()* tras procesar un canal; si no, el suceso o los sucesos ya procesados volverían a lanzarse –y a tratarse– cuando se produjera cualquier otro nuevo suceso.

Advertencia: Bajo ninguna circunstancia debe escribir código como éste:

```
        if (clave.isReadable() ) {
            // Se extrae el canal de la clave.
            SocketChannel sc = (SocketChannel) clave.channel();

            // Se extrae el socket del canal
            Socket socket = sc.socket();

            // Se escribe en el socket.
            salida = new PrintWriter(socket.getOutputStream(), true);
            salida.println("Buenos días");
        }
```

El método *println()* de la clase **PrintWriter** provoca el bloqueo del programa hasta que se lleve a cabo la escritura del mensaje (lo mismo sucedería con *read()* o *readLine()*). Anula, por tanto, las ventajas de NIO. Para aprovecharlas, deben usarse buffers y canales para la lectura y escritura de datos, y no las clases de **java.io**.

Más que perderse en una maraña de nuevos métodos y clases, me interesa que el lector o la lectora se quede con las siguientes ideas acerca del funcionamiento de la clase **Selector**:

- En un selector se registran varios canales, cada uno para ciertos tipos de sucesos de E/S. Si un canal no se registra para un suceso, éste no será comprobado por el selector.
- El selector detecta los sucesos que se producen en cada canal y permite que la aplicación decida qué hacer en respuesta a cada suceso.

6.6 La programación cliente-servidor con NIO

De modo general, un programa servidor se puede implementar con NIO tomando como base este pseudocódigo:

```
se crea un ServerSocketChannel para el servidor;
se crea un Selector.
se registra el ServerSocketChannel en el selector (para conexiones entrantes)
while (true) {
   Se esperan peticiones en el Selector (a cada una le corresponderá una clave)
   for cada clave en el Selector {
      if (clave es OP_ACCEPT) {
         se crea un SocketChannel para el nuevo cliente;
         se registra el SocketChannel con el Selector (para operaciones de lectura y escritura);
      }
      else if {clave es OP_READ) {
         se obtiene el SocketChannel de la clave;
         se lee del SocketChannel;
         se procesan los datos leídos;
      }
      else if {clave es OP_WRITE) {
         se obtiene el SocketChannel de la clave;
         se escribe en el SocketChannel;
      }
   }
}
```

El pseudocódigo de un cliente sería muy similar, pero habría que sustituir el *ServerSocketChannel* por un *SocketChannel* y tener en cuenta que un *SocketChannel* sólo puede registrarse para los sucesos representados por *OP_CONNECT*, *OP_READ* y *OP_WRITE*.

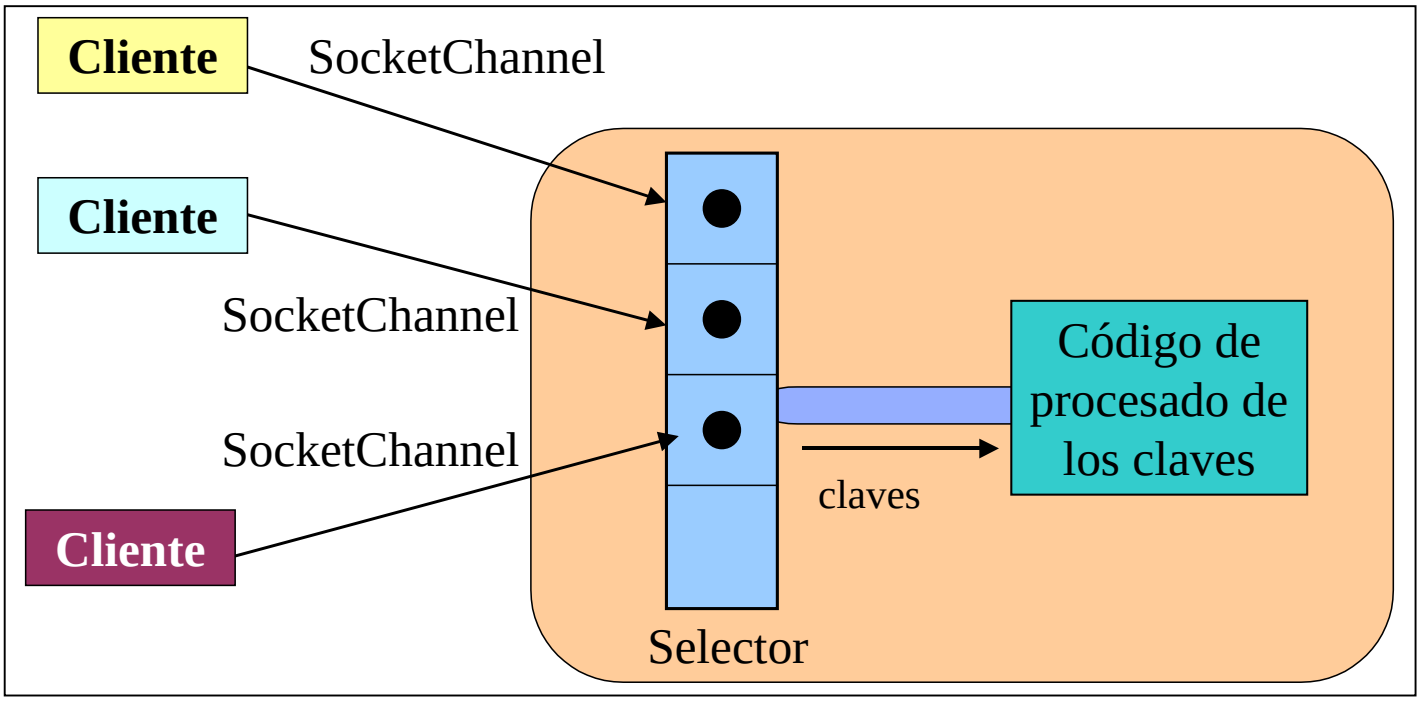

Figura 49. Esquema general de una aplicación cliente-servidor con canales y selectores

Como ejemplo de aplicación cliente-servidor con NIO, incluyo las clases **ServidorFechaNIO** y **ClienteFechaNIO**. Cada cliente que se conecta al servidor obtiene la fecha y hora del ordenador donde se ejecuta el servidor.

ServidorFechaNIO.java

```
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*; // Necesaria para usar la colección Set de java.

/**
 * Esta clase devuelve la fecha a cada cliente que se conecta y cierra la conexión con él.
 * No se controlan las excepciones. Código sólo válido en J2SE 1.4 o posterior.
 */
public class ServidorFechaNIO {

   public static final int PUERTO = 9000; // puerto por defecto

   public static void main(String[] args) throws java.io.IOException {
      // Se crea un canal para el ServerSocket y se configura sin bloqueo.
      ServerSocketChannel canalServidor = ServerSocketChannel.open();
      canalServidor.configureBlocking(false);

      // Se extrae el socket de servidor asociado. y se liga a una dirección (nombre ordenador, puerto).
      ServerSocket socketServidor = canalServidor.socket();
      InetSocketAddress dirSocket = new InetSocketAddress("localhost", PUERTO);
      socketServidor.bind(dirSocket);

      // Se crea un selector.
      Selector selector = Selector.open();

      // El canal se registra a sí mismo en el selector para sucesos de conexiones entrantes.
      canalServidor.register(selector, SelectionKey.OP_ACCEPT );

      System.out.println( "El servidor está escuchando por el puerto "+ PUERTO );

      // Se crea un ByterBuffer y, a continuación, se limpia.
      ByteBuffer buffer = ByteBuffer.allocate(31); // Puede tomarse un tamaño mayor, pero no
                                                   // menor: no cabria la fecha
      buffer.clear();

      while (true) { // bucle infinito
         // select() espera hasta que 1 o más canales registrados tengan sucesos u operaciones de E/S.
         selector.select();

         // selectedKeys() devuelve un Set que contiene SelectionKeys. Cada SelectionKey
         // representa un canal que tiene algún suceso de E/S y puede ser usada para aceder al canal.
         Set claves = selector.selectedKeys();

         // Se crea un iterador para recorrer el Set.
         Iterator it = claves.iterator();
```

```
            // Se recorre el iterador.
            while (it.hasNext()) {
               SelectionKey clave = (SelectionKey) it.next(); // Se obtiene cada SelectionKey del Set.

               // Se elimina clave del Set. Si no se hace esto, continuará en el Set y parecerá que
               // el suceso que representa ha vuelto a ocurrir.
               it.remove();

               if (clave.isAcceptable()) {
                  // Estamos ante una conexión entrante: un cliente solicita una conexión

                  // Se obtiene el canal de socket a partir de clave.
                  SocketChannel canalCliente = (SocketChannel) canalServidor.accept();

                  // Se configura el canal sin bloqueo.
                  canalCliente.configureBlocking(false);

                  // Se registra a sí mismo en selector para los sucesos de escritura
                  canalCliente.register(selector, SelectionKey.OP_WRITE);
               }
               if (clave.isWritable()) {
                  // El servidor está listo para escribir: un cliente solicita datos.

                  // Se obtiene el canal de socket de la clave.
                   SocketChannel canalCliente = (SocketChannel) clave.channel();

                  // Se configura el canal sin bloqueo.
                  canalCliente.configureBlocking(false);

                  // Se crea un String con la fecha del sistema y se almacenan los bytes
                  // a los que corresponde el String fecha en el buffer.
                  String fecha = (new java.util.Date()).toString() + System.getProperty("line.separator");
                  buffer.put(fecha.getBytes());

                  // Se establece el límite del buffer en la posición del último byte y
                  // la posición a cero. El buffer queda preparado para escribirse en un canal.
                  buffer.flip();

                  // Se escribe el buffer en el canal del cliente y se limpia el buffer.
                  canalCliente.write(buffer);
                  buffer.clear();

                  // Se cierra el canal. Al cerrarse, se desregistra del selector.
                  canalCliente.close();

                  // se cierra el socket asociado.
                  canalCliente.socket().close();
               }
            }
         }
      }

}
```

ClienteFechaNIO.java

```
import java.net.*;
import java.nio.*;
import java.nio.channels.*;

/**
 * Esta clase actúa como cliente para la clase ServidorFechaNIO.
 * Por sencillez, se considera que el servidor se ejecuta en la máquina local y por el número de
 * puerto 9000. Si no es así, hay que escribir en la constante MAQUINA el nombre
 * de la maquina, y en la constante PUERTO el número de puerto.
 * No se controlan las excepciones. Código sólo válido en J2SE 1.4 o posterior.
 */
public class ClienteFechaNIO {

    public static final int PUERTO = 9000; // puerto por defecto
    public static final String MAQUINA= "localhost"; // máquina por defecto: la local

    public static void main(String args[]) throws java.io.IOException {
        // Se crea un SocketChannel y se conecta a (MAQUINA, PUERTO);
        SocketChannel canalCliente = SocketChannel.open();
        canalCliente.connect(new InetSocketAddress(MAQUINA, PUERTO));

        // Se crea un canal de salida, con destino la salida estándar.
        WritableByteChannel canalSalida = Channels.newChannel(System.out);

        // Se crea un ByterBuffer y se limpia.
        ByteBuffer buffer = ByteBuffer.allocate(31);
        buffer.clear();

        // Se lee del canal. En la parte del servidor, esta operación se considera que es Writable (el
        // servidor puede escribir en el cliente).
        canalCliente.read(buffer);

        // Se establece el límite del buffer en la posición del último byte y
        // la posición a cero. El buffer queda preparado para escribirse en un canal.
        buffer.flip();

        // Se escribe en el canal asociado a la salidad estándar. Como el mensaje es muy
        // corto no hace falta comprobar con hasRemaining() si se ha leído todo o no.
        canalSalida.write(buffer);

        // Se limpia el buffer.
        buffer.clear();

        // Se cierran los canales y el socket.
        canalSalida.close();
        canalCliente.close();
        canalCliente.socket().close();
    }

}
```

7. UN CHAT CON JAVA: ACTUALIZACIÓN A J2SE 1.4 y 5.0

En el programa **ServidorChat.java** del apartado 3, a cada cliente se le asigna un hilo. Conforme vaya aumentando el número de usuarios del *chat*, irá aumentando de modo no lineal –debido a la naturaleza de los hilos- el tiempo que se tarda en enviar los mensajes a los usuarios. Además, como cada hilo permanece bloqueado mientras se espera la entrada del usuario con el método *readLine()* dentro del método *run()* del hilo, la aplicación no será escalable.

En las primeras versiones de Java, un par de decenas de hilos en una aplicación de charla bastaban para sobrecargar el ordenador donde se ejecutaban (en aquellos tiempos, los equipos tenían muchísima menos RAM que ahora). Hoy, una máquina virtual de Java puede trabajar con más hilos (siete u ocho decenas) sin que haya problemas, aunque la eficacia es paupérrima (muchos ciclos de CPU se desperdician esperando mensajes de los usuarios). El rendimiento de un sistema que maneje bastantes conexiones se puede mejorar un poco reutilizando hilos (*pooling*), pero esta solución no evita el problema fundamental: **las MVJ no están pensadas para mantener cientos o miles de hilos**. (Hay que tener en cuenta que no sólo cuesta crear cada hilo: la MVJ debe gestionarlos constantemente; lo que implica ceder el control a uno, luego quitárselo y dárselo a otro, vigilar las sincronizaciones...)

NIO permite manejar muchas conexiones (cientos o miles) de una manera mucho más escalable y eficaz. Con esta API se evita usar hilos; se consigue, por tanto, desacoplar los sockets de los hilos y se vuelve innecesario sincronizar métodos. El uso de selectores también implica consumo de ciclos de CPU y de recursos; pero ese consumo corresponde al sistema operativo, no a la MVJ. Por consiguiente, la MVJ dispone de mucha más memoria y recursos de CPU que cuando tiene que trabajar con hilos. La principal dificultad de usar NIO radica en su **abstracción**: el código se vuelve más complicado que cuando se usan hilos, si bien es más breve y eficaz. Al final, se cumple la máxima de que “se necesita pensar mucho para trabajar poco”. Es usted quien debe elegir qué prefiere.

En este apartado incluyo el código correspondiente a un servidor de *chat* desarrollado con la nueva API NIO en lugar de con hilos. En esencia, utilizo en él todo lo explicado en el apartado anterior. Dejo el código del cliente como ejercicio para el lector (de todas formas, el servidor se puede probar con los clientes de *chat* de los apartados 3 y 4: **no hace falta un cliente específico hecho con NIO**). Por supuesto, revisaré o corregiré cualquier propuesta de solución para el cliente que me envíen los lectores. Probando con distintos números de clientes, el lector podrá comprobar que la solución con NIO admite muchas más conexiones que la solución con hilos y carga menos el sistema donde se ejecuta.

Aprovechando las novedades de J2SE 1.4, el programa **ServidorChat.java** quedaría así (por no alargar demasido el código, he suprimido la identificación de los usarios y las órdenes *DESCONECTAR* y *LISTAR*):

ServidorChatNIO.java

```
import java.net.*;
import java.io.*;
import java.util.*;
import javax.swing.*;
import java.nio.*;
import java.nio.channels.*;

/**
 * Esta clase actúa como servidor para el chat. Utiliza la nueva API de entrada NIO, añadida
 * en la versión 1.4 de Java. No utiliza hilos ni necesita métodos sincronizados.
 * Para recibir y enviar datos se utilizan canales y buffers, no las clases del paquete
 * java.io.
 *
 * Sólo funciona con J2SE 1.4 o posterior.
 */
public class ServidorChatNIO {
   // La lógica de la clase es la siguiente:
   //    1. Se crea un ServerSocketChannel y se configura sin bloqueo.
   //    2. Se liga el ServerSocket asociado al número de puerto PUERTO.
   //    3. Se crea un selector.
   //    3. Se registra el ServerSocketChannel en el selector para los sucesos de petición de conexiones.
   //    4. Se entra en un bucle infinito en el que se van procesando los sucesos de los
   //       que informa el selector (conexiones, lectura de datos).

   // Lista de clientes activos. Contiene objetos Socket.
   private List clientesActivos = new ArrayList();

   // Puerto por omisión.
   private static final int PUERTO= 9000;

   // historial es el camino del archivo donde se registran las conexiones y desconexiones.
   // En este ejemplo se trata de un camino para Windows.
   private static final String historial="C:" + File.separatorChar + "historial.txt";

   // Tamaño por omisión del buffer de almacenamiento.
   private static final int TAMANYO_BUFFER = 1024;

   // Socket de servidor y su canal asociado.
   private ServerSocket socketServidor;
   private ServerSocketChannel canalSocketServidor;

   private Selector selector;
```

```
/** Punto de entrada a la aplicación. En este método se arranca el servidor
 * de chat.
 *
 * @param args String[]
 */
public static void main (String args[]) {
    new ServidorChatNIO();
}

/**
 * Constructor.
 */
public ServidorChatNIO() {
    System.out.println("Arrancando el servidor por el puerto " + PUERTO);

    arrancarServidor();
    procesarClientes();
}

/**
 * Arranca el servidor por el número de puerto PUERTO.
 * Cualquier error en esta etapa es fatal para la aplicación.
 */
private void arrancarServidor() {
    // La implementación de este método se basa por completo en la API NIO.

    try {
        // Se crea un canal ServerSocketChannel y se configura sin bloqueo.
        canalSocketServidor = ServerSocketChannel.open();
        canalSocketServidor.configureBlocking(false);

        // Se extrae el socket de servidor asociado al canal y se liga con la máquina local y el
        // núm. de puerto PUERTO.
        socketServidor = canalSocketServidor.socket();
        socketServidor.bind(new InetSocketAddress(PUERTO));

        // Se crea un selector y se registra el canal en él para las conexiones entrantes.
        selector = Selector.open();
        canalSocketServidor.register(selector, SelectionKey.OP_ACCEPT);
    }
    catch (java.net.BindException e1) {
        // No se puede arrancar el servidor. Error irrecuperable: se sale del programa.
        // No se limpia el socket de servidor porque no ha llegado a crearse.
        String mensaje =  "No puede arrancarse el servidor por el puerto " + PUERTO +
                          ". Seguramente, el puerto está ocupado.";
        errorFatal(e1, mensaje);
    }
    catch (java.lang.SecurityException e2) {
        // No se puede arrancar el servidor. Error irrecuperable: se sale del programa.
        // No se limpia el socket de servidor porque no ha llegado a crearse.
        String mensaje = "No puede arrancarse el servidor por el puerto " + PUERTO +
                         ". Seguramente, hay restricciones de seguridad.";
        errorFatal(e2, mensaje);
    }
    catch (IOException e3) {
```

```
            // No se puede arrancar el servidor. Error irrecuperable: se sale del programa.
            // Se intenta cerrar el socket de servidor (el canal se cerrará con él).
            if (socketServidor!= null) {
                try {
                    socketServidor.close();
                }
                catch (IOException e4) {} // No se hace nada.
            }

            String mensaje = "No puede arrancarse el servidor por el puerto " + PUERTO;
            errorFatal(e3, mensaje);
        } // fin del try-catch

        System.out.println("Arrancado el servidor por el puerto " + PUERTO);
    }


    /**
     * Gestiona las peticiones de los clientes (conexiones o envíos de datos).
     */
    private void procesarClientes() {
        ByteBuffer bb = ByteBuffer.allocate(TAMANYO_BUFFER);

        int n = 0; // número de claves

        while (true) { // bucle infinito donde se se procesan los sucesos de E/S
            try {
                n = selector.select();
            }
            catch(IOException e) {
                errorFatal(e, "Error de E/S al llamar al método select() del selector.");
            }

            if (n > 0) { // Ha ocurrido algún suceso de E/S.
                // Se recogen en un Set las claves de los canales que han registrado
                // sucesos.
                Set claves = selector.selectedKeys();

                // Se recorren los canales que han registrado sucesos.
                Iterator it = claves.iterator();
                while (it.hasNext()) {
                    SelectionKey clave = (SelectionKey) it.next();

                    if (clave.isAcceptable() ) {
                        // Estamos ante una conexión entrante de un cliente.
                        aceptarConexion(clave);
                    }
                    else if (clave.isReadable()) {
                        // Estamos ante un envío de datos de un cliente.
                        bb.clear();
                        bb = leerDatos(clave);
                        if (bb.hasRemaining()) {
                            escribirDatosATodos(bb, clave);
                        }
                    }
```

```
                // Se elimina la clave: ya ha sido procesada.
                it.remove();

            } // fin while interno
        } // fin if interno
    } // fin while externo
}


/**
 * Acepta conexiones de los clientes. Se registra el canal del socket del cliente para
 * operaciones de lectura, se envía un mensaje de bienvenida y se registra la conexión.
 *
 * @param clave objeto SelectionKey asociado con un canal que tiene una conexión entrante.
 * Si se produjera una excepción al mandar el mensaje de bienvenida, se cerraría el socket del
 * cliente y, por tanto, el canal asociado. En la próxima llamada a select(), el canal se
 * desregistraría del selector.
 */
private void aceptarConexion(SelectionKey clave) {
    canalSocketServidor = (ServerSocketChannel) clave.channel();
    SocketChannel sc = null;

    ByteBuffer bb = ByteBuffer.allocate(TAMANYO_BUFFER);

    // Tratamiento de la conexión y envío mensaje bienvenida.
    try {
        // Se acepta la petición del cliente y se obtiene un canal asociado al socket del cliente
        sc = canalSocketServidor.accept();

        if (sc == null) { // podría ocurrir
            return;
        }

        // Se configura sin bloqueo el socket del cliente y se registra
        // en selector para los sucesos de lectura
        sc.configureBlocking(false);
        sc.register(selector, SelectionKey.OP_READ);

         // Se escribe en el buffer un saludo de bienvenida
         String saludo = "****Bienvenido al chat****" + System.getProperty("line.separator");
         bb.put(saludo.getBytes());
         bb.flip(); // prepara el buffer para ser escrito.

         // Se escribe el mensaje de bienvenida en el canal asociado con el socket del cliente.
         sc.write(bb);

         // Se añade a la lista de clientes activos.
         clientesActivos.add(sc.socket());

         // Se registra la conexión.
         escribirHistorial(sc, true);
    }
    catch (java.lang.SecurityException e1) {
        // Excepción debida a restricciones de seguridad. Error irrecuperable: se sale del
        // programa. Si el socket de servidor no es nulo, se intenta cerrarlo antes de salir.
```

```
            if (socketServidor != null) {
                try {
                    socketServidor.close();
                }
                catch (IOException e2) {} // No se hace nada.
            }

            String mensaje = "Con su configuración de seguridad, los clientes no pueden " +
                             "conectarse por el puerto  " + PUERTO;
            errorFatal(e1, mensaje);
        }
        catch (IOException e3) {
            // Seguramente el cliente cerró la conexión en cuanto la hizo.
            // Estas desconexiones tempranas no se registran.

            // Se intenta cerrar el socket asociado al canal del cliente si no es nulo.
            // Si se cierra, se cerrará el canal.
            if (sc.socket() != null) {
                try {
                    sc.socket().close();
                }
                catch (IOException e4) {} // No se hace nada.
            }
        }
    }

    /** Lee datos de un canal que ha tenido un suceso de lectura (es decir, de uno que tiene
     * datos listos para leer).
     * Si se produce algún error al leer, devuelve un buffer vacío y listo para lecturas.
     *
     * @param clave SelectionKey que representa al canal con un suceso de lectura.
     *      Si el canal devuelve una excepción, una condición de fin de datos (-1) o no contiene datos,
     *      se cierra. Cerrar un canal vuelve inválidas todas sus claves y hace que el canal sea
     *      desregistrado del selector en la próxima llamada a select().
     *
     @return buffer en condición de lectura leído del canal representado por clave.
     */
    public ByteBuffer leerDatos (SelectionKey clave) {
        // Nótese que cierro explícitamente el canal si no hay datos o
        // hay una excepción. En realidad, basta cerrar el socket de cliente
        // asociado al canal para cerrar el canal; pero así evito la
        // posibilidad (remota) de que no se pudiera cerrar el socket y
        // quedara el canal abierto.

        // Se obtiene el canal asociado a la clave.
        SocketChannel sc = (SocketChannel) clave.channel();

        ByteBuffer bb = ByteBuffer.allocate(TAMANYO_BUFFER);

        try {
            // Se lee del canal correspondiente a la clave.
            sc.read(bb);
            bb.flip();
            if ( !(bb.hasRemaining()) || (bb.get(0) == -1) ) { // buffer sin datos o con EOF
                bb.clear();
                bb.flip();
```

```
            sc.close(); // cierre del canal

            // Se elimina el socket de la lista de clientes activo y se registra la desconexión.
            clientesActivos.remove(sc.socket());
            escribirHistorial(sc, false);

            // Se intenta cerrar el socket.
            try {
               Socket s = sc.socket();
               if (s != null) {
                  s.close();
                  s = null;
               }
            }
            catch (IOException e1) {} // No se hace nada.
         }
      }
      catch (IOException e2) {
         // Error en la conexión del cliente: se ha desconectado o hay un problema
         // con las redes intermedias.

         bb.clear();
         bb.flip();

         sc.close(); // cierre del canal;

         // Se elimina el socket de la lista de clientes activo y se registra la desconexión.
         clientesActivos.remove(sc.socket());
         escribirHistorial(sc, false);

         // Se intenta cerrar el socket.
         try {
            Socket s = sc.socket();
            if (s != null) {
               s.close();
               s = null;
            }
         }
         catch (IOException e3) {} // No se hace nada.
      }
      finally {
         return bb;
      }
   }

   /** Envía los datos de un buffer a todos los canales (excepto al canal del cual proceden éstos),
    * añadiendo el signo "<"
    *
    * @param buffer ByteBuffer obtenido de un canal que ha tenido un suceso de lectura y cuyos
    *      datos van a ser escritos en el resto de canales aún activos.
    * @param clave SelectionKey que representa al canal con un suceso de lectura.
    *      Si se produce una excepción al intentar escribir en un canal, se intenta cerrarlo, se registra
    *      la desconexión y se intenta cerrar el socket. Cerrar un canal vuelve inválidas todas sus
    *      claves y hace que el canal sea desregistrado del selector en la próxima llamada a select().
    */
```

```
    public void escribirDatosATodos (ByteBuffer buffer, SelectionKey clave)  {
        // Se envía el mensaje dentro de buffer a todos los canales menos al que ha recibido datos.

        ByteBuffer temp = ByteBuffer.allocate(1); // buffer temporal que almacena ">"
        temp.put(">".getBytes());
        temp.flip();
        SocketChannel canalOrigen = (SocketChannel) clave.channel(); // canal de socket
                                                                     // con suceso de lectura.
        for (int i=0; i < clientesActivos.size(); i++) {
            Socket socket = null;

            buffer.rewind(); // se rebobina
            temp.rewind();  // se rebobina

            socket = (Socket) clientesActivos.get(i);
            SocketChannel canalDestino = socket.getChannel();

            if ( !(canalDestino.equals(canalOrigen)) ){ // Sólo envío a los otros canales
                if (canalDestino.isConnected()) { // canal sigue conectado
                    try {
                        canalDestino.write(temp);
                        while (buffer.remaining() > 0) {
                            canalDestino.write(buffer);
                        }
                    }
                    catch (IOException e1) {
                        // No se ha podido escribir en el canal de destino.

                        // Cierre del canal. Basta cerrar el socket de cliente
                        // asociado al canal para cerrar el canal; pero así se evita la
                        // posibilidad (remota) de que no se pudiera cerrar el socket y
                        // quedara el canal abierto.
                        try {
                            canalDestino.close();
                        }
                        catch (IOException e2) {}

                        // Se elimina el socket de la lista de clientes activo y se registra la desconexión.
                        clientesActivos.remove(socket);
                        escribirHistorial(canalOrigen, false);

                        // Se intenta cerrar el socket.
                        try {
                            if (socket != null) {
                                socket.close();
                                socket = null;
                            }
                        }
                        catch (IOException e3) {} // No se hace nada.
                    } //fin try-catch externo
                } // fin if interno
            } // fin if externo
        } // fin for
    }
```

```
/**
 * Registra en el archivo descrito por la cadena historial las conexiones y
 * desconexiones de los clientes.
 *
 * @param sc SocketChannel que representa el canal del cual se va a
 *       registrar información. .
 * @param bool indica si se está ante una conexión (true) o una desconexión
 *       (false). En cada caso se utiliza un mensaje.
 */
public void escribirHistorial(SocketChannel sc, boolean bool) {

   if (bool) { //Si bool es true estamos ante una conexión
      try {
         PrintWriter salidaArchivo = new PrintWriter(new BufferedWriter
               (new FileWriter(historial, true))); // true indica autoflush

         salidaArchivo.println("Conexión desde la dirección: "+
         sc.socket().getInetAddress().getHostName() +
            " por el puerto "+ sc.socket().getPort()+
            " en la fecha " + new Date());

         salidaArchivo.close();
      }
      catch (IOException e1){
         JOptionPane.showMessageDialog(null, "Fallo en el archivo de historial",
               "Información para el usuario", JOptionPane.WARNING_MESSAGE);
      }
   }
   else if (!bool) { //Si bool es false estamos ante una desconexión
      try {
         PrintWriter salidaArchivo = new PrintWriter(new BufferedWriter
               (new FileWriter(historial, true))); // true indica autoflush

         salidaArchivo.println("Desconexión desde la dirección: "+
         sc.socket().getInetAddress().getHostName() +
               " por el puerto "+ sc.socket().getPort()+
               " en la fecha " + new Date());

         salidaArchivo.close();
      }
      catch (Exception e2){
         JOptionPane.showMessageDialog(null, "Fallo en el archivo de historial",
               "Información para el usuario", JOptionPane.WARNING_MESSAGE);
      }
   }
}

/**
 * Informa al usuario de la excepción arrojada y sale de la aplicación.
 *
 * @param excepcion  excepción cuya información se va a mostrar.
 * @param mensajeError  mensaje orientativo sobre la excepción .
 */
private void errorFatal(Exception excepcion, String mensajeError) {
   excepcion.printStackTrace();
   JOptionPane.showMessageDialog(null, "Error fatal."+ System.getProperty("line.separator") +
```

```
            mensajeError, "Información para el usuario", JOptionPane.WARNING_MESSAGE);

        System.exit(-1);
    }

}
```

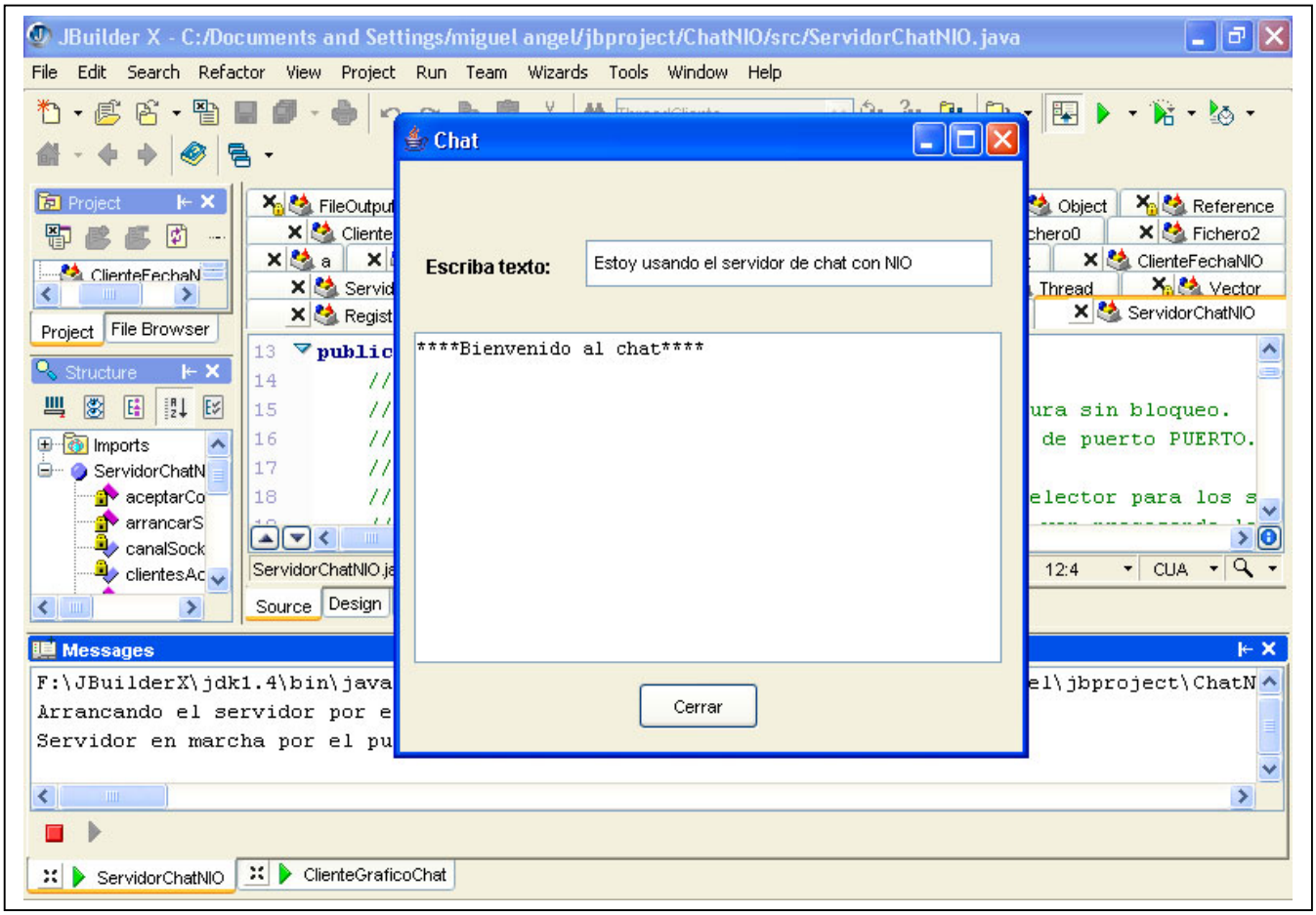

Figura 50. Utilización conjunta del servidor de chat con NIO y del cliente de chat con interfaz gráfica del apartado 4. ¿Por qué el botón Cerrar no funciona como uno espera?

Tal y como se explicó en el apartado anterior, la estructura del programa servidor de *chat* con NIO (y, en general, de cualquier programa servidor) sigue los siguientes pasos:

a) Se crea un objeto *ServerSocketChannel* que se usará para escuchar las peticiones de los clientes.

b) Se registra el anterior objeto *ServerSocketChannel* en un *Selector*.

c) El método *select()* del *Selector* aguarda hasta que se producen sucesos de E/S en algunos de los canales registrados.

d) Se usa el método *SelectedKeys()* del *Selector* para averiguar qué canales han tenido sucesos de E/S. Este método devuelve un *Set* (conjunto) de *SelectedKeys* que pueden ser inspeccionados para acceder al canal de cada uno.

e) El método *accept()* del *ServerSocketChannel* acepta las conexiones entrantes y genera objetos del tipo *SocketChannel* (cada uno tiene un *Socket* asociado).

f) Se registran los *SocketChanel*s que provienen del método *accept()* con el *Selector*, tal como se hizo en b) con *ServerSocketChannel*. Cada *SocketChannel* generado se encarga del intercambio de datos con un cliente, mientras que el *ServerSocketChannel* continúa esperando nuevas conexiones.

g) Se intercambian datos (lectura/escritura) con los clientes mediante *SocketChannels* y las clases del paquete **java.nio**.

Nota biográfica del Autor: Miguel Ángel Abián es licenciado en Ciencias Físicas por la U. de Valencia y obtuvo la suficiencia investigadora dentro del Dpto. Física Aplicada de la U.V con una tesina acerca de relatividad general y electromagnetismo. Además, ha realizado diversos cursos de postgrado sobre bases de datos, lenguajes de programación Web, sistemas Unix, comercio electrónico, firma electrónica, UML y Java. Ha colaborado en diversos programas de investigación TIC relacionados con el estudio de fibras ópticas y cristales fotónicos, ha obtenido becas de investigación del IMPIVA y de la Universidad Politécnica de Valencia y ha publicado artículos en el *IEEE Transactions on Microwave Theory and Techniques* y en el *European Congress on Computational Methods in Applied Sciences and Engineering*, relacionados con el análisis de guías de onda inhomogéneas y guías de onda elípticas.

En el ámbito laboral ha trabajado como gestor de carteras y asesor fiscal para una agencia de bolsa y actualmente trabaja en el Laboratorio del Mueble Acabado de AIDIMA (Instituto Tecnológico del Mueble y Afines), ubicado en Paterna (Valencia), en tareas de normalización y certificación, traducción e interpretación y asesoramiento técnico. En dicho centro se están desarrollando proyectos europeos de comercio electrónico B2B para la industria del mueble basados en Java y XML (más información en www.aidima.es). Ha impartido formación en calidad, normalización y programación para ELKEDE (Grecia), CETEBA (Brasil) y CETIBA (Túnez), entre otros.

Últimamente, aparte de asesorar técnica y financieramente a diversas empresas de la Comunidad Valenciana, es investigador en las Redes de Excelencia **INTEROP** y **ATHENA** del Sexto Programa Marco de la Comisión Europea, que pretenden marcar las pautas para tecnologías de la información en la próxima década y asegurar el liderazgo de Europa en las tecnologías de la sociedad del conocimiento. Ambos proyectos tienen como fin la interoperabilidad del software (estudian tecnologías como J2EE, .Net y CORBA, servicios web, tecnologías orientadas a aspectos, ontologías, etc.), y en ellos participan empresas como IBM U.K., COMPUTAS, SIEMENS, FIAT, TXT, GRAISOFT, SAP, EADS, además de numerosas universidades europeas y centros de investigación en ingeniería del software.

Sus intereses actuales son el diseño asistido por ordenador de guías de ondas y cristales fotónicos, la evolución de la programación orientada a objetos, Java, UEML, el intercambio electrónico de datos, el surrealismo y París, siempre París.