Programación de sockets en Java

PID_00275867

Maria Isabel March Hermo

Tiempo mínimo de dedicación recomendado: 5 horas

Universitat Oberta de Catalunya

Maria Isabel March Hermo

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: Joan Manel Marquès Puig

Primera edición: septiembre 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Autoría: Maria Isabel March Hermo
Producción: FUOC

Índice

Introducción

La razón de ser de la existencia de Internet es la comunicación entre dispositivos de cualquier índole, conectados en todo el mundo. Actualmente hay una infinidad de aplicaciones en red, de temáticas muy diferentes: web, correo electrónico, navegación GPS, reproducción vía *streaming* de audio y vídeo, juegos multiusuarios, comercio electrónico, electrodomésticos inteligentes, videotutoriales, redes sociales...

Esta red estructurada, compleja y heterogénea, constantemente vehicula los intercambios de mensajes que se producen entre los procesos que ejecutan estas aplicaciones, de manera transparente al usuario final.

En este módulo aprenderemos a programar una de estas aplicaciones en red, mediante la interfaz de programación de aplicaciones (API) más popular para comunicar procesos remotos: los *sockets*.

Cuando estamos comunicando dos o más aplicaciones de dispositivos diferentes, necesitamos una serie de convenciones para que la red pueda redirigir los paquetes hacia una y otra parte. Los *sockets* son la interfaz software para crear este marco comunicativo, es decir, un conjunto de instrucciones que tiene que seguir la aplicación para comunicarse con el nivel de transporte y sus homólogos en el destino, y así poder permitir el intercambio de datos entre procesos locales o remotos.

Existen varias clasificaciones de *sockets*, pero se profundizará especialmente en la tipología según el servicio que ofrecen: orientados o no a la conexión o, lo que es lo mismo, basados en el protocolo de transporte TCP o UDP. Esta diferenciación es clave, puesto que define la naturaleza de los *sockets*, cómo actúan y cómo se comportan a lo largo de la comunicación. En consecuencia, la secuencia de operaciones que se tienen que realizar en uno u otro caso son diferentes.

Después de definir el marco conceptual, el módulo se adentra en las clases en Java que dan soporte a la programación de *sockets*. Se ven en detalle los constructores, los métodos principales y las excepciones que pueden lanzarse, por lo que puede usarse como manual de referencia en la programación de aplicaciones en red.

Si bien no hacen falta conocimientos avanzados de la programación en lenguaje Java, sí es recomendable saber cómo se lleva a cabo la compilación y ejecución de archivos en esta plataforma, así como los fundamentos de la pro-

gramación estructurada (clases, constructores, métodos *getters* y *setters*...) y la sintaxis de los tipos de datos primitivos más relevantes y las instrucciones básicas para manipularlos.

Objetivos

Los objetivos principales que el estudiante tiene que lograr en la comprensión de este módulo didáctico son:

1. Definir la interfaz de programación *sockets* y situarla dentro del marco conceptual del conjunto de protocolos TCP/IP, el paradigma cliente-servidor y los diferentes mecanismos de comunicación que existen.

2. Diferenciar los rasgos característicos de las comunicaciones orientadas y no orientadas a la conexión, para escoger la idónea según la funcionalidad final de la aplicación en red.

3. Dibujar el diagrama de flujo típico de un cliente y de un servidor orientado y no orientado a la conexión.

4. Conocer las clases Java relacionadas con la programación de *sockets*, orientados o no a la conexión. En particular, sus constructores, los métodos principales y sus parámetros.

5. Programar una aplicación en red, comunicando procesos remotos vía *sockets*, tanto en el rol de los clientes como en el rol de los servidores.

1. Historia

El término *connector* o *socket*, entendido como mecanismo de comunicación entre procesos remotos, surgió en 1971, en RFC 147 «The Definition of a Socket», escrito por J. M. Winett.

> «A socket is defined to be the unique identification to or from which information is transmitted in the network. The socket is specified as a 32 bit number with even sockets identifying receiving sockets and odd sockets identifying sending sockets. A socket is also identified by the host in which the sending or receiving processer is located.»

Este documento no era público, sino utilizado dentro de la red ARPANET, considerada como la precursora de las redes que se usan hoy en día. Inicialmente era una red de carácter militar creada por el Departamento de Defensa de los Estados Unidos a finales de los años sesenta del siglo pasado, y que acabó conectando muchas universidades e instalaciones gubernamentales utilizando líneas telefónicas convencionales. Más adelante, cuando se añadieron enlaces por satélite o radio, los sistemas empezaron a tener problemas para interactuar con estas nuevas redes. Se hizo patente que hacía falta una nueva arquitectura de referencia para poder conectar diferentes modelos de redes. Esta arquitectura se popularizó como el modelo de referencia TCP/IP (iniciales de sus dos principales protocolos), que es el modelo de Internet. Los *sockets* surgen a raíz de la necesidad de conectar los niveles superiores de este modelo TCP/IP y, por otra parte, comunicar aplicaciones que se estaban ejecutando en dispositivos heterogéneos de la red ARPANET.

No obstante, hoy en día, la definición de *sockets* dista mucho de la idea inicial definida en la RFC 147, a pesar de que su función es la misma. La mayoría de implementaciones de los *sockets* se basan en los *Berkeley sockets*, denominados así porque toman como referencia una distribución del sistema operativo Unix, en la variante que hizo la Universidad de Berkeley, versión 4.2 (UNIX BSD 4.2), en 1983. En esta distribución, se implementó una serie de llamadas a sistema que implementaban la interconectividad entre procesos vía *sockets*, aglutinados en una interfaz de programación de aplicaciones específica (*sockets API*). Estas aplicaciones empleaban el conjunto de protocolos TCP/IP y eran testadas en la red ARPANET y su sucesora ARPA.

Según el estándar de Berkeley, los *sockets* eran implementados como un tipo de descriptor de ficheros, siguiendo la filosofía Unix. Por eso, al trabajar con *sockets* encontraremos funciones similares a los clásicos `open()`, `read()`, `write()` o `close()`, más propias del ámbito de ficheros. Actualmente, la interfaz de programación vía *sockets* ha sido adaptada a los diferentes sistemas operativos. Por ejemplo, *WinSock* es otra implementación basada en los Berkeley *sockets*, usada por el sistema operativo Microsoft.

Hay otras implementaciones similares a la API de *sockets* que definen un conjunto de llamadas a sistema para comunicarse con los protocolos de transporte. Uno de ellos es la interfaz TLI (*Transport Layer Interface*), basado en flujos o *streams* de datos, distribuida en otra variante de UNIX llamada System V, en su versión 3 (AT&T UNE System V o SVR3), en 1987. La intefaz TLI está basada en el modelo de referencia OSI (*Open System Interconnection*), que quedó obsoleto frente al modelo TCP/IP.

2. ¿Qué es un *socket*?

En el apartado anterior hemos visto qué motivó la necesidad de definir una interfaz de programación vía *sockets*, pero, ¿qué son exactamente los *sockets*?

La analogía clásica que se usa para entender el concepto de *sockets* es el sistema telefónico como medio para comunicar personas. Los *sockets* serían los teléfonos, que permiten intercambiar información entre procesos, que serían las personas. Así pues, los *sockets* son puntos de comunicación entre agentes (procesos o personas respectivamente), mediante los que se puede enviar o recibir información.

Los *sockets* son mecanismos de comunicación entre procesos, locales o remotos, que permiten que un proceso se comunique bidireccionalmente con otro proceso.

Otra analogía la encontramos en el servicio postal. Cuando queremos mandar una carta a otra persona, primero tenemos que escribir el contenido (los datos) y ponerla dentro de un sobre. Después de cerrarlo, tendremos que escribir el nombre completo del destinatario, la dirección y el código postal, típicamente en la parte derecha inferior del sobre. Después pondremos un sello en la parte derecha superior; y finalmente, llevaremos la carta a un buzón o una oficina de mensajería. Del mismo modo que el servicio postal tiene este conjunto de reglas que tiene que seguir el emisor de la comunicación para asegurarse de que el destino recibe la carta, Internet recurre a las interfaces *sockets* para que el proceso que envía los datos lo haga según las normas establecidas.

Los *sockets* son una interfaz software formada por conjunto de instrucciones de código, que los programas que se comunican por la red tienen que seguir, para que Internet pueda entregar los datos.

2.1. Los *sockets* en el modelo de referencia TCP/IP

Los estándares de comunicaciones en red proporcionan la base para la transmisión de datos entre diferentes equipos, para la fabricación de equipos de red compatibles y para el diseño de las rutinas dentro de los sistemas operativos que faciliten las comunicaciones a distancia.

Hay dos modelos de comunicación principales que utilizan una estructuración basada en niveles: el modelo de referencia OSI (*Open System Interconnection*), que ha quedado relegado al ámbito teórico, y el modelo de protocolos TCP/

IP (*Transport Control Protocol/Internet Protocol*), ampliamente utilizado y, por eso, denominado también modelo de Internet o conjunto de protocolos de Internet.

Recordemos que el conjunto de protocolos de Internet se estructura en cinco niveles, tal y como se muestra en la tabla siguiente.

Tabla 1.

Aplicación	***Datos***	***DNS, HTTP, P2P, EMAIL, TELNET, FTP***
Transporte	***Segmentos/Datagramas***	***TCP, UDP***
Red	***paquetes***	***IP, ARP, ICMP***
Enlace	***Tramas***	***ETHERNET, 802.11, ATM, PPP***
Físico	***Bits***	***RS-232, RJ-45, DSL, 100BASE-TX***

Cada nivel tiene una función diferenciada, que resumiremos a continuación. Además, en cada uno de ellos podemos encontrar distintos protocolos que responden a unos servicios particulares y diferenciados del nivel donde se encuentran.

- **Físico**: procesa la transmisión de bits por el medio físico de transporte, como fibra óptica, coaxial, radio...

- **Enlace**: permite la transmisión de tramas entre máquinas conectadas directamente. La red más usual es Ethernet.

- **Red**: define el direccionamiento y el encaminamiento independiente de paquetes por la red, de forma que vayan del origen al destino siguiendo la mejor ruta. El protocolo más utilizado es IP. También está ARP (resolución de direcciones) o ICMP (mensajes de control).

- **Transporte**: se encarga de proporcionar servicios a la comunicación extremo a extremo, como la fiabilidad y la seguridad de que los datos lleguen al destino en el orden correcto. El protocolo orientado a la conexión TCP o no orientado a la conexión UDP se utiliza según las necesidades de las aplicaciones.

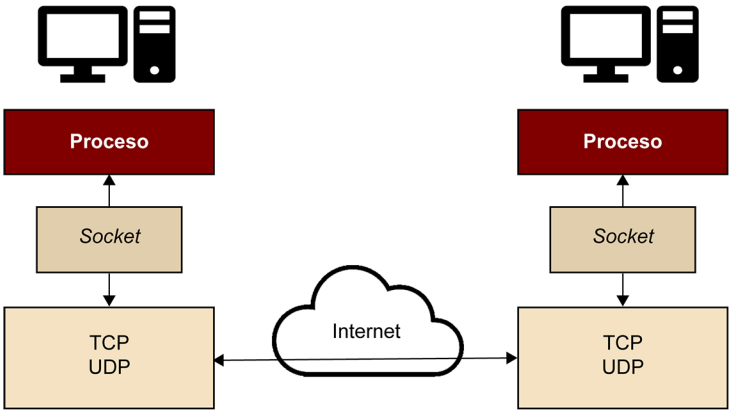
- **Aplicación**: es la más cercana al usuario. Comprende las aplicaciones y los procesos que utilizan las redes de comunicaciones. Los protocolos más populares son HTTP (*Hypertext Transfer Protocol*), SMTP (*Simple Mail Transfer Protocol*) o SSH (*Secure Shell*), entre otros.

La comunicación entre niveles tiene dos vertientes:

- Extremo a extremo de la comunicación respecto al mismo nivel.

- Nivel inmediatamente inferior y superior respecto al mismo dispositivo.

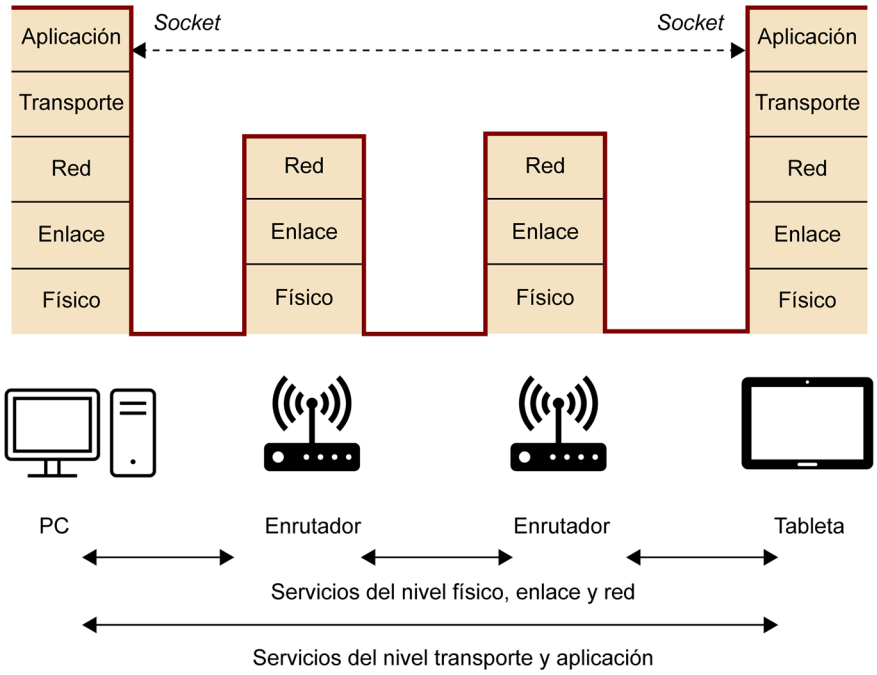
Figura 1.



Como puede observarse, los *sockets* se encuentran entre el nivel de aplicación y el nivel de transporte. También se denominan API (*Application Programming Interface*) entre la aplicación y el resto de niveles que nos comunicarán por la red.

El nivel de aplicación, concretamente a partir de los procesos que son realmente los artífices activos de las comunicaciones, envía mediante *sockets* los datos a la capa de transporte, que después del análisis y gestión pertinente, los trasladará a los niveles inferiores. Los datos se enviarán por la red al *host* destino, donde finalmente, después del análisis y gestión por los niveles inferiores, llegarán al nivel de transporte, desde donde se redirigirán al proceso correspondiente mediante el *socket*. Después de tratar los datos, es habitual que se dé el proceso inverso. Este proceso se denomina **encapsulamiento y desencapsulamiento de paquetes** entre los niveles del modelo de Internet. Podemos observar este proceso como la línea marrón fuerte de la siguiente figura.

Figura 2.



Los *sockets* son interfaces de programación de aplicaciones (API) en red que se sitúan entre el nivel de transporte y el nivel de aplicación, en el modelo de referencia TCP/IP.

Los elementos intermedios de la red, como enrutadores (*routers* en inglés) y conmutadores (*switches* en inglés), no necesitan implementaciones de los *sockets*, puesto que ellos operan en el nivel de enlace (*switches*) o en el nivel de red (*routers*). No obstante, los cortafuegos (*firewalls* en inglés), los traductores de direcciones IP o los *proxies*, sí que hacen un seguimiento de los *sockets* activos. También en ciertas políticas de calidad de servicio (QoS) implementadas en los *routers* o ciertos protocolos de encaminamiento, se emplea información sobre los *sockets*.

2.2. Los *sockets* en el paradigma cliente-servidor

Hasta el momento, hemos hablado de cómo las aplicaciones se comunican entre sí por la red. Existen diversas maneras para posibilitar esta comunicación extremo a extremo a través de Internet. La gran mayoría de aplicaciones en red, que emplean los *sockets* para comunicarse, siguen el paradigma cliente-servidor.

La arquitectura cliente-servidor es un modelo de diseño de software basado en dos perfiles que se comunican:

- Un proveedor de recursos o servicios, los llamados **servidores**.
- Los demandantes de estos servicios, los llamados **clientes**.

Los programas clientes se ejecutan en *hosts* que típicamente interactúan con los usuarios. Los procesos creados como consecuencia de la ejecución de estos programas realizan peticiones a uno o varios servidores para obtener unos determinados servicios. Esta idea se puede aplicar de manera local, pero es más común y útil en sistemas distribuidos, a través de una red de computadoras, donde muchos usuarios se comunican con un servidor.

Los programas servidores son programas que se están ejecutando en un *host* remoto y ofrecen servicios a múltiples potenciales clientes, típicamente de manera ininterrumpida y continuada.

Por ejemplo, en las aplicaciones web, un navegador cliente intercambia mensajes con un dominio alojado en un servidor web. En un sistema P2P de compartición de archivos como BitTorrent, un fichero se transfiere entre dos *hosts*, el que ofrece el archivo actúa como servidor, y el que se descarga el archivo actúa como cliente.

También se pueden dar casos en que un determinado *host* actúe como cliente y servidor a la vez, para varias aplicaciones o incluso dentro de la misma. Por ejemplo, en BitTorrent también es común que desde un extremo se esté ofreciendo un fichero y a la vez descargándose archivos ofrecidos por otros.

En cualquiera de los dos roles, los *sockets* posibilitan la comunicación entre ambas partes.

- En el lado de los procesos **servidores**, los *sockets* sirven para anunciar en Internet los servicios que se ofrecen y posibilitar la comunicación con los clientes que los pidan.

- En el lado de los procesos **clientes**, los *sockets* sirven para comunicarse con los servidores, realizando peticiones de servicios y obteniendo sus respuestas.

2.3. Los *sockets* como mecanismo de comunicación

Hay diversas técnicas y paradigmas para realizar la comunicación entre procesos que se ejecutan de manera distribuida, paralela o concurrente. Los máximos exponentes en los que se basan la gran parte del software en red hoy en día son:

- El **paso de mensajes**: sincronizar dos procesos de forma que se pasen cooperativamente un mensaje, de tal manera que uno de ellos lo envía y el otro lo recibe.

- La **invocación remota**: ejecutar un procedimiento o método, típicamente en otro dispositivo, como si lo estuviéramos haciendo en la misma máquina.

De hecho, ambos son similares, ya que hay un proceso activo que envía el mensaje o llamada la ejecución de un método, respectivamente. No obstante, la invocación remota no exige que el receptor esté a la espera de leer ninguna petición, y, además, la respuesta es atómica.

Los *sockets* se situarían como una implementación del **paso de mensajes**, puesto que existen diversos participantes que se comunican bidireccionalmente de manera explícita mediante el paso de datos.

2.4. Identificación de los *sockets*

Recuperaremos un momento el ejemplo anterior del envío postal. Uno de los puntos imprescindibles si queremos mandar una carta es saber la dirección completa del destinatario. Sin estos datos, a la empresa de transporte le sería imposible realizar la entrega correctamente. También se suele poner el remitente de la carta, para saber quién lo ha enviado y así poder responder o comunicar algún error en la entrega. Estos datos se ponen en el anverso y reverso de una carta en un orden concreto:

- el nombre completo,
- la dirección con la calle y el número,
- el código postal y la población.

Del mismo modo, los procesos que se quieren comunicar por la red necesitan identificarse con unos datos básicos. En el argot de las redes, este identificador de los procesos que se comunican por Internet es el **nombre del *socket*** y está formado por:

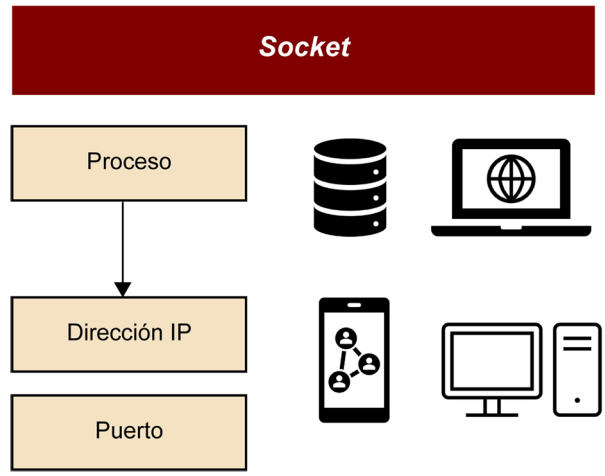
- **Dirección IP:** identifica cualquier terminal de cualquier red del mundo en Internet. Son 4 bytes que se suelen escribir en notación decimal: 4 números de 0..255 separados por puntos.

- **Puerto:** identifica un proceso concreto en una máquina. Son 2 bytes que se suelen escribir en notación decimal (un número). Por lo tanto, un puerto puede estar numerado del 0 al 655436.

No tenemos que confundir la dirección IP con la dirección física, que es la dirección que identifica unívocamente la tarjeta de red de la máquina. Por ejemplo, en las redes Ethernet es la dirección MAC, que se compone de 6 bytes que se suelen expresar en hexadecimal, como por ejemplo 54:D4:6F:AD:67:E4. A diferencia de la dirección IP y el puerto, el programador no conoce las direcciones MAC ni trabaja con ellas, a no ser que sea a bajo nivel para configurar algún aspecto específico relativo al nivel de enlace.

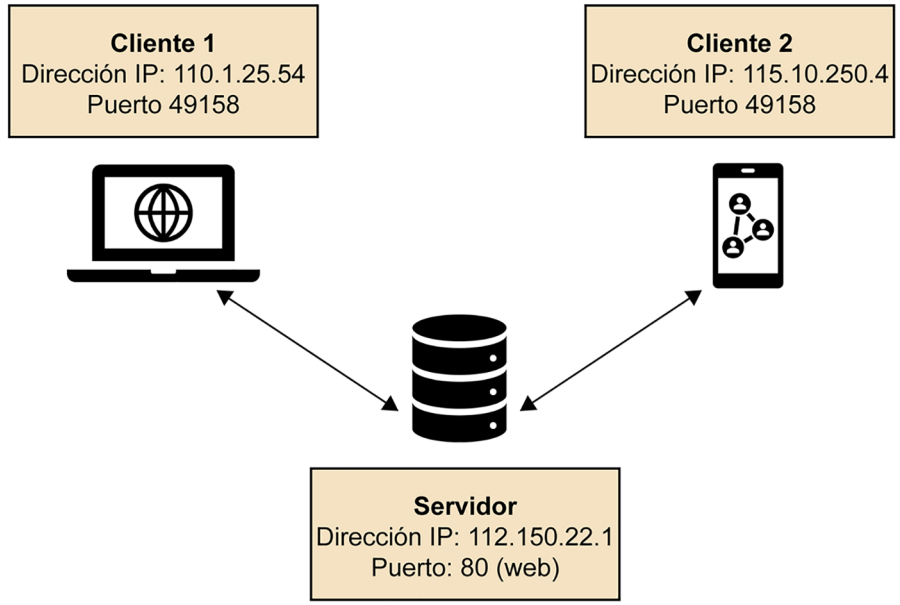
La identificación de cada *socket* por su dirección IP y puerto es única. La dirección IP identifica el dispositivo dentro de la red Internet y el puerto identifica el proceso dentro de este dispositivo, sea del tipo que sea.

Figura 3.



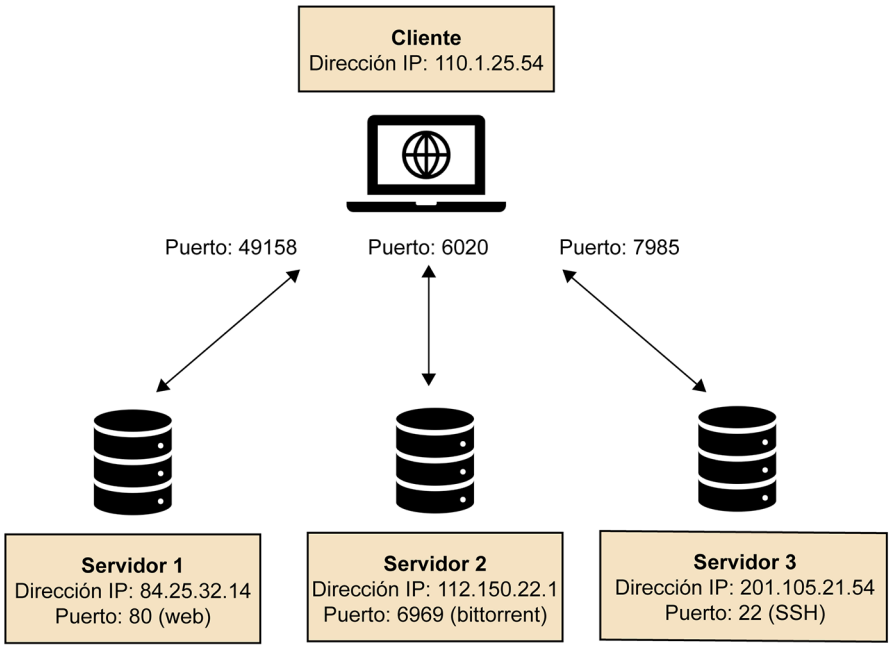
Para simplificar, nos pondremos en el caso de un ordenador con una única tarjeta de red. Este ordenador está identificado en la red por la IP 80.55.23.69. Todos los procesos que se estén ejecutando en el mismo ordenador tendrán esta misma IP. No obstante, no puede haber dos *sockets* abiertos en un mismo momento en la misma máquina, con el mismo número de puerto. Si no, el nivel de transporte no sería capaz de entregar los segmentos que le llegan a uno u otro proceso, puesto que no tendría manera de diferenciarlos. En cambio, sí puede haber dos procesos que se estén comunicando por la red con el mismo número de puerto, siempre y cuando pertenezcan a equipos diferentes. Es decir, puede haber dos procesos con el mismo número de puerto pero con direcciones IP diferentes. Así, el *socket* está identificado por esta dirección IP, puerto de manera unívoca igualmente, como se puede ver en la figura siguiente.

Figura 4.



Por ejemplo, en un mismo ordenador podemos tener abierto un navegador con diversas pestañas abiertas consultando webs, también nos estamos descargando unos archivos con el programa BitTorrent, o subiendo otros de forma segura por SSH. Cada una de estas aplicaciones habrá creado como mínimo un *socket* por donde se intercambian los datos con el servidor pertinente, como se puede ver en la figura siguiente:

Figura 5.



Una determinada máquina podrá tener un conjunto de servidores activos y conexiones con servidores remotos, gracias a la distinción de puertos.

A continuación, veremos con más detalle qué son las direcciones IP y los puertos utilizados para programar las aplicaciones en red.

Direcciones IP

En el RFC 1918 al 1996 se definen los conjuntos de direcciones IP destinadas a uso interno y a uso público:

- Las **direcciones privadas** son las direcciones IPv4 que se destinan a la creación de redes privadas, como puede ser la red interna de una empresa que no necesita que se pueda acceder a los equipos directamente desde Internet.

- Las **direcciones públicas** se ven en todo Internet y, en consecuencia, no puede haber dos máquinas con la misma dirección IP.

Tabla 2.

Clase	***Prefijo***	***Rango***
A	10.0.0.0/8	10.0.0.0 – 10.255.255.255
B	172.16.0.0/12	172.16.0.0 – 172.31.255.255
C	192.168.0.0/16	192.168.0.0 – 192.168.255.255
El resto de direcciones son públicas		

La asignación de direcciones IP a los *hosts* se suele realizar por nuestro ISP o proveedor de Internet. Como el número de direcciones IP es limitado, los ISP van rotando estas direcciones, por eso se denominan **direcciones IP dinámicas**.

Además, para no tener que recordar o anotar las direcciones IP concretas de los servidores donde nos queremos conectar, se idearon los **nombres de dominio** que utilizamos constantemente hoy en día, como *uoc.edu* o *google.com*. Mediante el protocolo DNS (*Domain Name Server*) se hace la traducción de nombre de dominio a dirección IP e inversa, de manera transparente al usuario. El programador también tiene a su alcance funciones de conversión para facilitar esta traducción a la hora de crear un *socket* o conectarse a un servidor.

Puertos

Por otra parte, la asignación de puertos a procesos tampoco es en absoluto arbitraria. La asociación IANA (*Internet Assigned Number Authority*) definió tres rangos de puertos:

- **Puertos conocidos** (0..1023): puertos fijos universales y conocidos, asignados por la IANA, asociados a servicios específicos que están estandarizados y regulados por un RFC.

- **Puertos registrados** (1024..49151): puertos usados por aplicaciones de usuario.

- **Puertos dinámicos o efímeros** (49151..65535): puertos usados de manera temporal para que un proceso cliente se pueda conectar con un proceso servidor.

Normalmente, si queremos acceder a un servidor conocido, tendremos que crear un *socket* cliente que realice peticiones a un *socket* servidor, indicando el nombre de dominio de este servidor y el puerto conocido según la tabla mostrada a continuación.

Tabla 3.

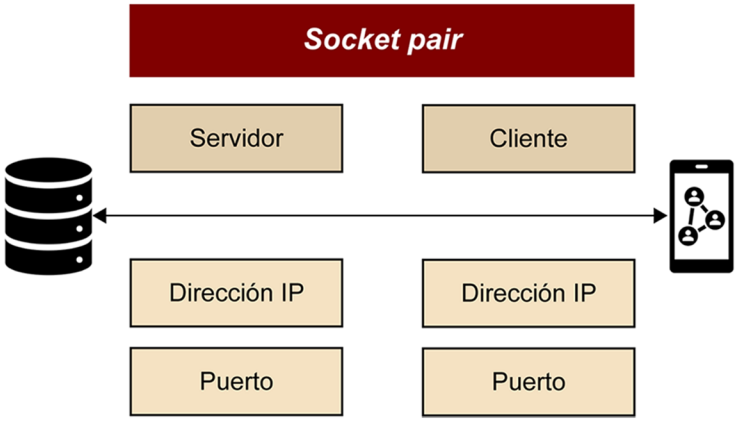
Servicio	***Puerto***	***TCP***	***UDP***
DayTime	13	√	√
FTP-Data	20	√	
FTP	21	√	
SSH	22	√	
Telnet	23	√	
SMTP	25	√	
DNS	53	√	√
HTTP	80	√	
POP3	110	√	
IMAP	143	√	
SNMP	161		√
HTTPS	443	√	
SIP	5060		√

2.5. Tipos de *sockets*

El enlace entre los dos *sockets*, el *socket* del proceso servidor y el *socket* del proceso cliente, permite una comunicación bidireccional, es decir, que los dos extremos de la comunicación pueden escribir y, así, leer a la vez. Es lo que

denominamos ***socket pair***, e identifica una conexión unívocamente. Como su nombre indica, está formado por los *sockets* correspondientes a los dos extremos de la comunicación.

Figura 6.



Esta es una característica propia de los *sockets* y que los diferencia de otros mecanismos de comunicación como las *pipes*.

Tabla 4.

Sockets	***Pipes***
Canales bidireccionales	Canales unidireccionales
Comunicación entre procesos remotos	Comunicación entre procesos locales
Filosofía cliente-servidor	Simple intercambio de datos

Un *socket* pair es un par de *sockets*, típicamente formado por el *socket* cliente y el *socket* servidor, que se comunican entre sí de manera bidireccional.

Si nos fijamos en el rol del *socket*, podemos distinguir entre:

- ***Sockets* activos**: los que inician la conexión con el otro extremo.
- ***Sockets* pasivos**: los que están a la espera de recibir conexiones.

Por ejemplo, en una comunicación cliente-servidor típica, el servidor estará a la espera de conexiones de potenciales clientes mediante un *socket* pasivo. En cambio, el cliente se intentará conectar al servidor mediante un *socket* activo.

Si nos fijamos en el tipo de aplicaciones que utilizan *sockets*, podemos tener:

- **Aplicaciones estándar**. Las que pretenden comunicarse mediante un protocolo estándar según las normas establecidas en un RFC. Por ejemplo, queremos programar un cliente que se comunique con un servidor web

cualquiera. La comunicación vía *sockets* de este cliente tendrá que seguir las normas establecidas en el RFC 2616 del protocolo HTTP (web), escribiendo peticiones e interpretando respuestas según dicta el protocolo. En caso contrario, el servidor web no entenderá lo que le dice el cliente y le devolverá error.

- **Aplicaciones propietarias**. Otra opción es desarrollar un cliente y un servidor propios, según unas normas que ideamos expresamente para tal comunicación, sin seguir ningún estándar. Esta opción es habitual si queremos ofrecer un servicio específico, que no está regulado todavía, como puede ser un juego en línea, por ejemplo.

No obstante, la clasificación más importante radica en los servicios de la comunicación ofrecidos por el *socket*, puesto que condiciona el tipo de intercambio que puede haber:

- *Sockets* orientados a la conexión.
- *Sockets* no orientados a la conexión.

Atendida su relevancia en la programación de comunicaciones vía sockets, la veremos en los siguientes apartados más a fondo.

3. Servicios de la comunicación

Siguiendo con la analogía del servicio postal, nos podemos fijar ahora en que todas las empresas repartidoras suelen ofrecer más de un servicio a sus clientes para la entrega de cartas o paquetes: la entrega exprés, una confirmación de recepción, seguimiento del envío realizado, etc. De manera similar, Internet proporciona múltiples servicios a sus aplicaciones en red.

En particular, cuando estamos desarrollando un programa que se comunicará por la red, tenemos que elegir qué protocolo de transporte nos interesa más utilizar, según los servicios que necesitaremos a la hora de realizar la supervisión de la transmisión de los datos extremo a extremo: TCP o UDP.

Esta elección nos condicionará la programación en uno u otro tipo de *sockets*, puesto que define la naturaleza del mismo. Por lo tanto, el tipo de comunicación que puede generarse entre clientes y servidores tiene que ser la misma, es decir, el par de *sockets* cliente-servidor tiene que ser TCP o UDP, puesto que el protocolo de transporte es único para cada comunicación.

Vamos a repasar brevemente los servicios diferenciados que ofrecen estos dos protocolos del nivel de transporte:

UDP (*User Datagram Protocol*) es un protocolo no orientado a la conexión:

- No existe ningún concepto de conexión establecida entre los dos extremos de la comunicación.
- Cada datagrama que se envía es independiente del anterior y el siguiente.
- No existe fragmentación de los datagramas.
- No hay ninguna garantía sobre el orden de recepción de los datagramas, ni siquiera si se han recibido correctamente. Estos errores los tiene que detectar la aplicación y retransmitir los datagramas si se considera necesario.

Estas características lo convierten en un protocolo muy rápido, muy utilizado en consultas de petición y respuesta puntuales a un servidor, y en aplicaciones en que la rapidez de la comunicación prima sobre la exactitud de los datos. Es muy utilizado en aplicaciones en tiempo real. Un ejemplo lo tenemos en la retransmisión de una carrera de bicicletas en línea. El usuario prefiere seguir la carrera aunque sea con unos breves cortes o sin tener la imagen con una alta calidad. También se emplea en protocolos rápidos de petición-respuesta, como DNS, empleado para resolver la dirección IP de un nombre de dominio.

TCP (*Transport Datagram Protocol*) es un protocolo orientado a la conexión que nos proporciona:

- Fiabilidad en la transmisión de los segmentos.
- Fragmentación de los segmentos.
- Mantenimiento del orden de los segmentos.
- Utiliza un checksum para detectar errores.
- Hay una negociación de establecimiento de la conexión inicial (handshake).

Estas características lo hacen un protocolo óptimo para aplicaciones como el web (HTTP) o SSH, donde necesitamos que los datos transmitidos por la red lleguen íntegramente al destinatario.

Según estemos implementando un servicio que requiera uno u otro protocolo de transporte, la serie de instrucciones de código que utilizaremos para programar los *sockets* será diferente.

Manteniendo presente esta dualidad según el protocolo de transporte, se definen principalmente dos tipos de *sockets*:

- ***Sockets* en UDP o *datagram sockets*** *(*datagramas): para comunicaciones en modo no conectado, con el envío de datagramas de tamaño limitado (tipo telegrama).

- ***Sockets* en TCP o *stream sockets*** (flujo de datos): para comunicaciones fiables en modo conectado, de dos vías y con tamaño variable de los mensajes de datos.

En *sockets* TCP o *stream sockets*, la interfaz de *sockets* distingue entre el rol cliente y el rol servidor claramente, que establecen una conexión permanente, bidireccional, unívoca y fiable mientras dure la comunicación. En cambio, en *sockets* UDP o *datagram sockets* no existe este concepto de conexión entre los dos extremos de la comunicación, ni hay ningún tipo de control de errores de los datagramas. Si se quiere ofrecer un servicio con un cierto grado de fiabilidad, se tendría que programar explícitamente.

También existen otros tipos de *sockets* más específicos, como los ***raw sockets***, que permiten el acceso a protocolos de más bajo nivel como el IP (nivel de red). Los *raw sockets* facilitan un acceso experto a las comunicaciones, y se emplean para configurar ciertos parámetros de los niveles inferiores del conjunto de protocolos TCP/IP, variando su funcionamiento estándar.

3.1. *Sockets* UDP o no orientados a la conexión

A continuación, veremos el funcionamiento de los *sockets* que utilizan UDP como protocolo de transporte. Recordemos que, al ser un protocolo no orientado a la conexión, no ofrece fiabilidad, y por tanto, no habrá un control del flujo de datos ni tampoco fragmentación. La capa de aplicación será la que se tiene que encargar de poner las instrucciones pertinentes, si se quiere cierto grado de fiabilidad.

Esta rapidez en las comunicaciones, sin que haya un establecimiento previo y continuo de la comunicación, implica que cada uno de los datagramas tenga que incorporar información de direccionamiento en su cabecera.

> En comunicaciones no orientadas a la conexión, cada datagrama tiene que incluir el *socket pair*, esto es, la dirección IP y puerto a quien va dirigido el datagrama; y la dirección IP y el puerto del remitente.

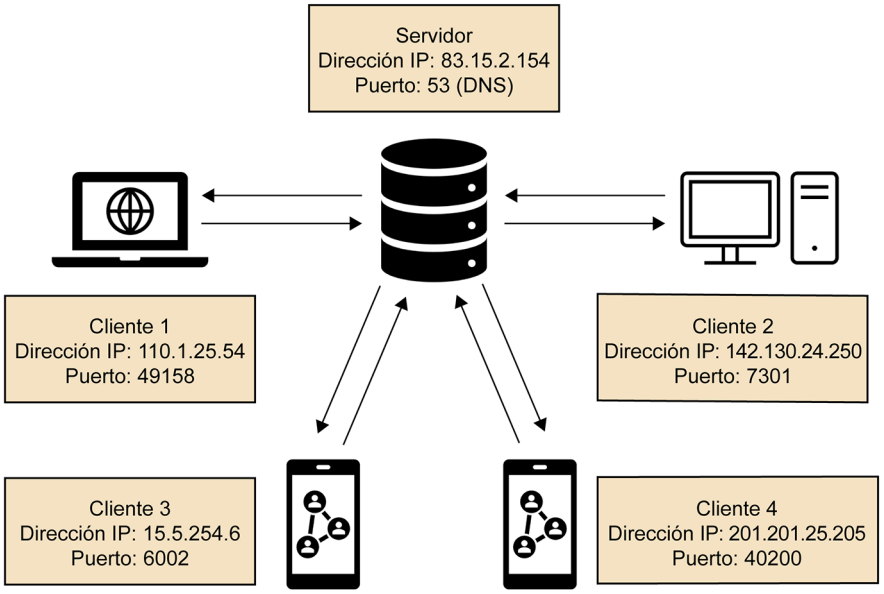
En caso de que no hubiera esta información, los extremos de la comunicación no sabrían quién se está comunicando con ellos, porque no existe una conexión establecida y mantenida por donde viajan los datos. Cada datagrama que se recibe, se trata individualmente.

De hecho, si recuperamos la analogía con el servicio postal, no sería más que los datos que ponemos acerca del destinatario y el remitente de una carta. Si una carta no tiene escrito el destinatario, el servicio postal no la puede entregar. Y si no tiene el remitente, quien recibe la carta no tendrá la dirección donde dirigir su respuesta y no la podrá contestar.

> Si nos fijamos en el paradigma cliente-servidor, los clientes y el servidor se guían por un mecanismo petición-respuesta.

Al no haber una conexión dedicada entre los dos extremos de la comunicación, el servidor tendrá que ir alternando la atención a los múltiples clientes que le hagan peticiones. Es decir, el servidor irá tratando las peticiones que le lleguen, independientemente del cliente que lo haga, según el orden de llegada. Las analizará y tratará y responderá al cliente destinatario que se ha indicado en la misma petición.

Figura 7.



En las comunicaciones no orientadas a la conexión, existe un único *socket* de nombre conocido en el servidor, que está a la espera de peticiones de nuevos clientes, y cuando las hay, las responde, escribiendo por el mismo *socket*.

Los procesos clientes también tendrán que crear un *socket* sin nombre conocido, que usarán para realizar peticiones con el servidor directamente, sin un establecimiento de la conexión previo.

El *socket pair* entre cliente y servidor no es, pues, dedicado. Va variando en el transcurso de la comunicación y se mantiene únicamente mientras dura este tratamiento de la petición y elaboración de la respuesta hacia un determinado cliente.

Veamos este proceso con más detalle:

- El servidor crea un *socket* de escucha, con nombre conocido (dirección IP y puerto), por medio del cual el servidor espera las peticiones de potenciales clientes.

- Cada cliente crea un *socket* sin nombre, para lanzar las peticiones al servidor.

- El cliente escribe los datos por el *socket* creado, indicando la petición realizada y el *socket pair*, esto es, la dirección IP y el puerto del servidor, por un lado, y la dirección IP y el puerto (del cliente), por el otro. Al poner los datos del emisor de la comunicación, el servidor sabrá dónde contestar.

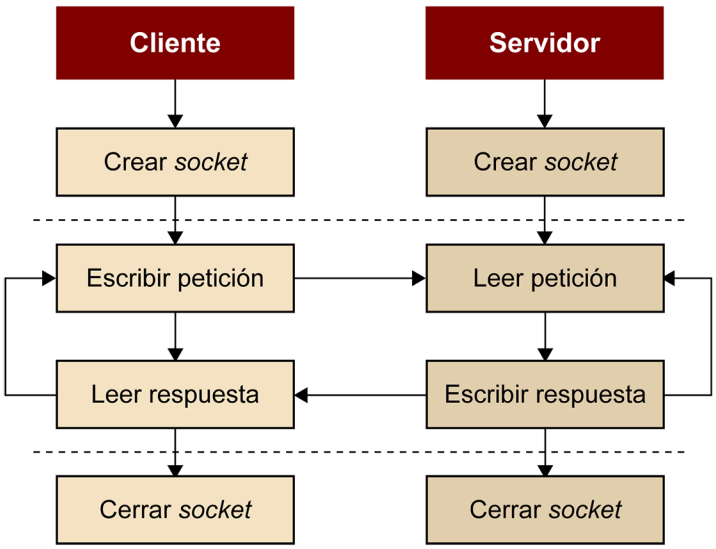
- El servidor recibe la petición del cliente, leyendo del *socket* de escucha. La analiza y la trata, escribiendo los datos relativos a la respuesta en el mismo *socket*. Esta respuesta también lleva las direcciones y puertos del *socket pair*.

- Cuando se finaliza la comunicación, el cliente cierra su *socket* para evitar el consumo innecesario de recursos.

- El servidor seguirá atendiendo peticiones de clientes que le lleguen.

- El servidor cerrará su *socket*, cuando se pare su ejecución, es decir, cuando ya no quiera actuar más como tal.

Figura 8.



En comunicaciones no orientadas a la conexión también se puede emplear concurrencia (hilos o *threads*) en el servidor. No obstante, al ser un protocolo orientado a tratamiento de peticiones y envío de respuestas rápidas, no es tan imprescindible.

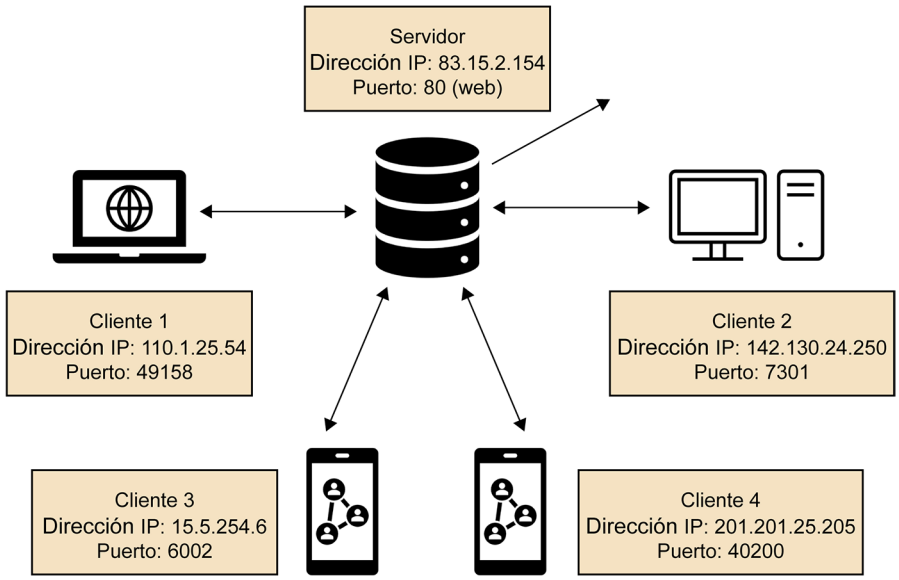
3.2. *Sockets* TCP u orientados a la conexión

Vamos a concretar qué ocurre durante la comunicación entre procesos mediante *sockets* orientados a la conexión. Recordemos que este tipo de *sockets* usan TCP como protocolo de transporte, y es este protocolo el que se encarga de la entrega de los segmentos de manera fiable. Esta fiabilidad se traduce en que el flujo de bytes que se genera en el proceso origen se entrega sin errores al proceso destino. Además, el protocolo fragmenta el flujo de datos procedente de la capa de aplicación en mensajes más pequeños.

Siguiendo el paradigma cliente-servidor, un proceso actuará de proceso servidor creando un *socket* de nombre conocido y ofreciendo un servicio. Este *socket* se denomina *socket* de escucha. Así, los potenciales clientes podrán conectarse para obtener los servicios ofrecidos.

Por otro lado, los procesos clientes también tienen que crear un *socket* sin nombre conocido, que usarán para comunicarse con el servidor al que se conectan. Este *socket* establecerá una conexión con el servidor. Como consecuencia, se creará un nuevo *socket* sin nombre en el servidor que conectará únicamente con el cliente conectado.

Figura 9.



En comunicaciones orientadas a la conexión, tendremos:

- Un *socket* de escucha en el servidor a la espera de clientes potenciales.
- Un *socket* en el servidor por cada cliente que se haya conectado.
- Un *socket* en el cliente que lo conecta con el servidor en cuestión.

Estos dos últimos *sockets* son los que definen el *socket pair*, que vehiculará el intercambio de mensajes entre clientes y servidor.

Puede haber tantos *sockets pairs* en los servidores como clientes estén conectados. En cambio, normalmente solo hay un *socket* de escucha que espera conexiones de potenciales clientes.

Ahora que tenemos una idea general de ello, veremos el funcionamiento particular:

- El servidor crea un *socket* de escucha, con nombre conocido (dirección IP y puerto), por medio del cual el servidor espera las conexiones de potenciales clientes.

- Cada cliente crea un *socket* sin nombre, para lanzar las peticiones de conexión al servidor.

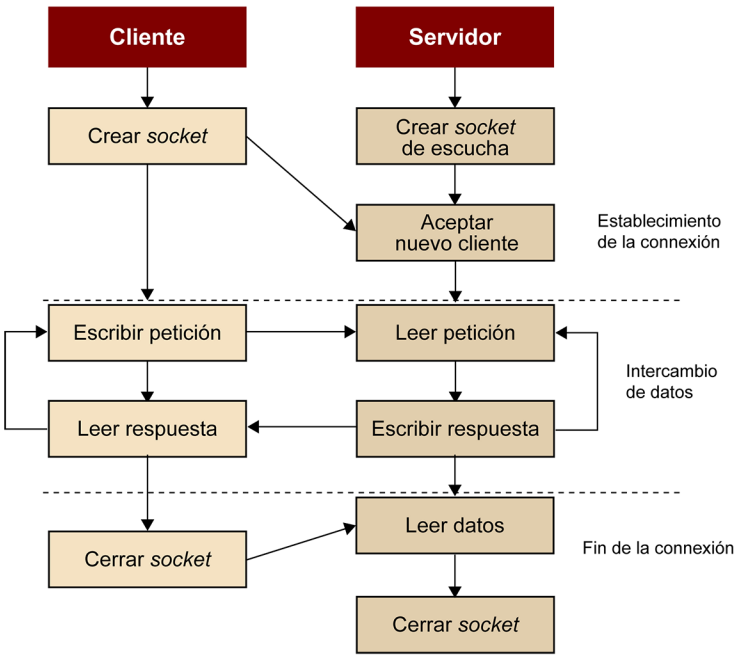
- El proceso servidor acepta esta conexión creando un nuevo *socket* que será el que realmente mantenga una comunicación bidireccional con el cliente, mediante el cual intercambiarán peticiones y respuestas. El *socket* con nombre creado inicialmente se suele mantener a la espera de nuevas conexiones de otros clientes.

- Una vez establecida la conexión, se trata como un flujo (*stream*) típico de entrada y salida, escribiendo datos y leyéndolos, tanto en el cliente como en el servidor.

- Cuando han acabado de comunicarse, tanto el *socket* cliente como el *socket* servidor, ambos sin nombre conocido, tienen que cerrarse para dar por finalizada la comunicación y así evitar el consumo de recursos de sistema innecesarios.

- El servidor seguirá a la espera de peticiones de conexión de nuevos clientes mediante el *socket* de escucha.

Figura 10.



Este es el proceso típico seguido en un servidor secuencial. No obstante, en las comunicaciones en red, es muy común emplear concurrencia (hilos o *threads*) en el servidor, es decir, que un proceso *padre* cree otros procesos *hijos* para

atender en paralelo a los clientes que se van conectando. Así, un cliente no tiene que esperar que el servidor acabe de atender al cliente que se ha conectado anteriormente, para comunicarse, evitando los consecuentes tiempos de espera. El resultado de la concurrencia son múltiples comunicaciones bidireccionales y simultáneas, que se dan en paralelo entre los hijos del servidor y los clientes. Este hecho implica que hay múltiples *sockets* creados en un mismo servidor, uno de escucha y uno por cada cliente que se conecta.

4. Programación en Java

4.1. Conceptos básicos

Ahora que ya hemos finalizado la exposición del marco conceptual, vamos a llevarlo a la práctica y crear nuestra primera aplicación en red.

Java es un lenguaje de programación creado en 1995 por la empresa Sun Microsystems. El gran rasgo diferencial de esta plataforma gratuita es que es multiplataforma, es decir, se diseñó con la idea de que los programadores escribieran el programa con independencia del sistema operativo o dispositivo en el que se ejecutara. Este hecho es posible gracias a la máquina virtual de java (JVM), que se encarga de esta portabilidad, haciéndola transparente a programadores y usuarios finales. La rapidez, robustez, seguridad y fiabilidad hacen que sea ideal para escribir fácilmente aplicaciones en red. Además, cuenta con una comunidad en la red muy grande, para consultar temas básicos o específicos de cualquier aspecto de programación. Como apoyo, los desarrolladores de Java pueden consultar la documentación en línea oficial de sus API, también llamada *Javadoc*.

Como cualquier otro lenguaje de programación, Java tiene sus propias reglas y sintaxis, derivadas del C y de la programación orientada a objetos (OOP). Java se estructura en proyectos, que contienen clases. Cada clase tiene una serie de métodos y variables, entre otros, que dictan su funcionamiento. Cada uno de estos archivos se denominan con la extensión *.java*. Después de la compilación, obtendremos los archivos *.class* correspondientes, que ya pueden ser ejecutados por la máquina virtual de java (JVM).

Para programar cualquier aplicación, antes que nada, nos tendremos que instalar un kit de desarrollo en Java (JDK), que contiene, entre otros, el compilador y las bibliotecas de clases de utilidad general. No hay que confundirlo con el JRE (*Java Runtime Environment*), que incluye componentes necesarios para la ejecución de programas en Java.

Normalmente, para programar aplicaciones en el lenguaje de programación Java, se utiliza un IDE, esto es, una interfaz gráfica que nos facilita la programación, detección de errores, la compilación y la ejecución de las aplicaciones. Hay muchas, pero probablemente *Eclipse* y *Netbeans* sean las más populares.

Java se basa en una **programación estructurada**. Típicamente tendremos una clase que coincidirá con el nombre del fichero que contenga el código del programa. La ejecución del código se iniciará en el método `main()` de esta clase. Esta función se considera el punto de entrada de la aplicación, donde se

inicia el proceso. En consecuencia, las instrucciones que contenga el método `main()` se ejecutarán en orden secuencial y siguiendo los saltos de código, según haya indicado el programador.

A continuación, se muestra un esqueleto básico de lo que sería un programa escrito en este lenguaje de programación. En este caso, se tendría que guardar en un archivo llamado *prueba.java*.

```
import java.io.*; //librerías
public class prueba {  //clase prueba
    public static void main(String argv[]) {
        //primer método que se ejecutará
        //código del programa
    }
}
```

4.2. Excepciones en Java

Java tiene su propio sistema de gestión de excepciones o errores, basado en un mecanismo de programación por eventos.

Un proceso va ejecutando las instrucciones de código según dicta el programa que ejecuta. Si el programador no gestiona las excepciones y en cualquiera de estas instrucciones se produce un error, el proceso finaliza por error inesperado. En cambio, si el programador gestiona las excepciones, el proceso continúa su ejecución y se redirige hacia una serie de instrucciones según se hayan programado.

La gestión de excepciones es muy habitual y necesaria cuando programamos aplicaciones en red, puesto que los errores son comunes y no tienen que implicar que el programa aborte de manera inesperada. Por ejemplo, imaginemos que un servidor está caído o que no nos puede atender porque hay sobrecarga de clientes. El cliente programado podría buscar un servidor alternativo o esperar a que la situación se resuelva y volverlo a intentar, comunicándolo al usuario con un mensaje informativo. Un buen programador no solo tiene que escribir las instrucciones para que el proceso se ejecute cuando no hay errores; más bien al contrario, tiene que contemplar todas las posibilidades y plantear caminos alternativos de código, cuando las funciones ejecutadas no respondan según se espera.

Las palabras reservadas para capturar y tratar excepciones en Java son:

- ***Try***: dentro de este bloque pondremos el código del programa que queremos someter a «prueba», es decir, se capturarán las excepciones del código que esté dentro del bloque `try`.

- ***Catch***: este bloque recoge las intruccions de código que se ejecutarán cuando se produzca un error en alguna parte del bloque anterior, esto es, contiene el tratamiento de errores.

- ***Finally***: esta parte contiene un bloque de instrucciones de código que se ejecutan siempre, tanto si hay error como si no.

Veamos un ejemplo sencillo. Antes de continuar, intentad entender el código y pensad cuál sería el resultado de este programa.

```
import java.io.*;
public class prueba {
    public static void main (String [] args) {
        try {
            System.out.println("Prueba1”);
            int num = Integer.parseInt("E");
            System.out.println("Prueba2”);
        } catch (Exception e) {
            System.out.println("Prueba3");
        } finally {
            System.out.println("Prueba4");
        }
    }
}
```

Fijaos en que la línea de código referente a la función `parseInt()` produce error, porque le estamos pasando como parámetro un carácter cuando espera un entero. El resultado de ejecutar el código anterior sería:

```
Prueba1
Prueba3
Prueba4
```

4.3. Las clases Java sobre *sockets*

Después de exponer unas pinceladas básicas de Java, vamos a adentrarnos en la programación de aplicaciones en red mediante *sockets*. Dentro del paquete *java.net*, Java proporciona las siguientes clases para facilitar la programación en *sockets*, según la naturaleza del mismo:

- ***DatagramSocket***: Clase encargada de realizar comunicaciones no fiables, no orientadas a la conexión (protocolo UDP). La unidad de envío es un datagrama, simbolizado en una instancia del objeto *DatagramPacket*.

- ***Socket***: Objeto básico en una comunicación por red. Representa un *socket*, esto es, uno de los extremos de la conexión o del *socket pair*, en comunicaciones fiables orientadas a la conexión (que usan TCP como protocolo de

transporte). Los clientes se conectarán a un determinado servidor mediante este objeto *Socket*. También, tanto clientes como servidores escribirán y leerán los datos que se intercambien.

- ***ServerSocket***: Clase encargada de implementar el servidor de la conexión en comunicaciones fiables orientadas a la conexión (protocolo TCP). Representa el *socket* de escucha, mediante el cual el servidor está a la espera de peticiones de conexión de potenciales clientes. Al aceptar la petición de conexión de un cliente, devolverá una instancia de la clase *Socket*, por donde realmente se llevará a cabo la comunicación con el cliente.

- ***SocketImpl***: Clase abstracta para poder definir cualquier tipo de comunicación, es decir, para poder configurar los *sockets* sin tenernos que ceñir a las propiedades estándares de los *sockets* TCP o UDP descritos anteriormente. Si instanciamos una subclase de *SocketImpl*, podemos redefinir sus servicios para tener un control pleno de la comunicación. Por ejemplo, si quisiéramos implementar un *firewall*, tendríamos que redefinir el método `accept()` añadiendo los controles de seguridad necesarios. De hecho, todas las clases enumeradas son instancias de la clase *SocketImpl*.

Otra clase empleada a menudo a la hora de programar aplicaciones en red es:

- ***InetAddress***: Clase encargada de implementar una dirección IP.

4.4. Programación de *sockets* no orientados a la conexión

Como ya hemos visto en el apartado «*Sockets* UDP o no orientados a la conexión», UDP es un protocolo no fiable, que permite el envío de datagramas a través de una red, sin que se haya establecido previamente una conexión, puesto que la cabecera del datagrama mismo que se intercambian los extremos de la comunicación incorpora suficiente información de direccionamiento como para poder saber el remitente y el destinatario.

La secuencia de instrucciones de programación típica de un **servidor secuencial** que atiende a múltiples clientes es la siguiente:

- Crear un *socket* no orientado a la conexión que esté a la espera de peticiones de clientes en el puerto configurado.
- Leer el datagrama relativo a la petición que nos envía un cliente a través de este *socket* de escucha.
- Responder esta petición, escribiendo el datagrama de respuesta por el mismo *socket*.
- Repetir los pasos 2 y 3 tantas veces como sea necesario.
- Cerrar el *socket* para liberar los recursos de sistema.

En comunicaciones no orientadas a la conexión, recordemos que los datagramas que se intercambian clientes y servidores contienen el *socket pair*, esto es, las direcciones IP y puertos de los dos extremos de la comunicación.

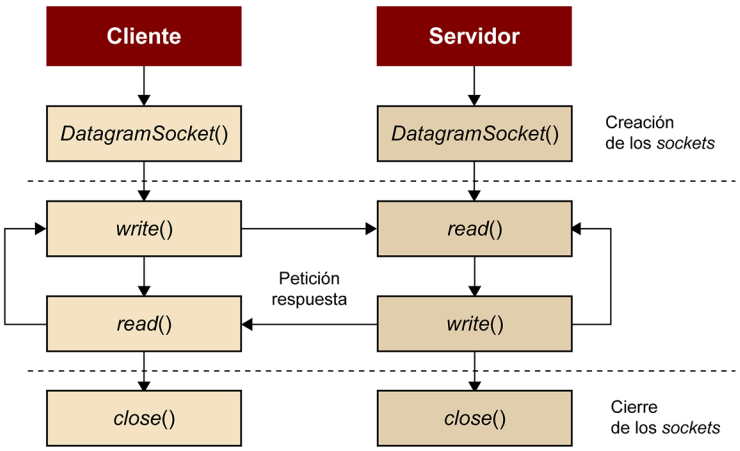
Otro punto a resaltar en este tipo de comunicaciones es que el servidor puede recibir peticiones de cualquier cliente. Al no haber un establecimiento de la comunicación, ni un canal bidireccional único entre servidor y un cliente, solo existe un *socket* por donde se reciben las peticiones y se responden. Este hecho implica que podemos intercalar peticiones de diversos clientes a la vez, puesto que no existe una conexión dedicada.

Por otro lado, el **cliente** empleará un mecanismo similar, pero siendo la parte activa de la comunicación:

- Crear un *socket* no orientado a la conexión por donde comunicarse con el servidor.
- Enviarle la petición de servicio al servidor, escribiendo el datagrama en el *socket* creado, indicando el *socket* (la dirección IP y puerto) del servidor a quien va dirigida.
- Leer el datagrama relativo a la respuesta que nos envíe el servidor.
- Repetir los pasos 2 y 3 tantas veces como sea necesario.
- Cerrar el *socket* para liberar los recursos de sistema.

En comunicaciones no orientadas a la conexión, tanto cliente como servidor utilizan las mismas clases con sus métodos, por lo que las explicaremos genéricamente y después veremos un ejemplo concreto para cada uno de estos dos roles. El gráfico genérico de los métodos implicados en el proceso anterior es:

Figura 11.



4.4.1. Crear un *socket*

Para crear un nuevo *socket* UDP, bien sea servidor o cliente, lo primero que tenemos que hacer es crear un nuevo objeto de la clase `DatagramSocket()`, por donde enviar y recibir datagramas, utilizando uno de los siguientes constructores:

- `DatagramSocket()`: crea un *socket* UDP, asignándole un número puerto disponible de la máquina local.
- `DatagramSocket(int port)`: crea un *socket* y lo asigna al puerto pasado como parámetro.
- `DatagramSocket(SocketAddress binaddr)`: crea un *socket* UDP, asignándole la dirección pasada como parámetro.
- `DatagramSocket(int port, InetAddress laddr)`: crea un *socket* y lo asigna a la dirección y puerto pasados como parámetro.

Las excepciones que pueden produirse a raíz de la ejecución de estas llamadas son:

- `SocketException`: si el *socket* no se puede abrir.
- `SecurityException`: si el gestor de seguridad del dispositivo no permite crear el *socket*.

Normalmente, en el caso de los clientes, utilizaremos el primero de los constructores enumerados, dejando que el sistema sea quien elija crear el *socket* con la dirección IP de la máquina y uno de los puertos libres. En caso de los servidores, es habitual usar el constructor descrito como segundo punto de la enumeración, puesto que se necesita crear un *socket* con nombre conocido y necesitamos conocer a priori el puerto por donde estará ofreciendo servicios para que los clientes le puedan pedir. Por ejemplo, el siguiente servidor no orientado a la conexión escucha peticiones de clientes por el puerto 6000.

```
try {
    DatagramSocket se = new DatagramSocket (6000);
} catch (SocketException ex) {
    System.err.println(ex);
}
```

4.4.2. El datagrama

En el transcurso de la comunicación entre clientes y servidores no orientados a la conexión, la unidad del paquete enviado es el datagrama. En Java, la clase que representa el datagrama es `DatagramPacket()`. Como hemos comentado, aparte de los datos en sí, tiene que contener la dirección del destinatario para poder dirigir el paquete. Los constructores más relevantes de esta clase son:

- `DatagramPacket(byte[] buf, int len, InetAddress address, int port)`
- `DatagramPacket(byte[] buf, int len, SocketAddress address)`

En ambos casos, se construye un paquete de tipo datagrama, que contiene:

- Los datos pasados como primer parámetro, de longitud `len`.
- El *socket* destinatario identificado por una dirección IP y puerto.

Dado un objeto de tipo datagrama siempre podemos consultar o configurar los atributos anteriores, con los métodos:

- `InetAddress getAddress()`
- `byte[] getData()`
- `int getLength()`
- `int getPort()`
- `SocketAddress getSocketAddress()`
- `setAddress(InetAddress iaddr)`
- `setData(byte[] buf)`
- `setLength(int length)`
- `setPort(int iport)`
- `setSocketAddress(SocketAddress address)`

4.4.3. Enviar datos

Como hemos visto, las comunicaciones orientadas a la conexión están basadas en un protocolo petición-respuesta. Tanto para enviar la petición de cliente a servidor como para enviar la respuesta de servidor a cliente, la llamada por excelencia de la clase `DatagramSocket()` es `send()`:

- `send(DatagramPacket p)` : envía un datagrama por el *socket*. Este datagrama pasado como parámetro contiene los datos que se quieren enviar, la dirección IP y el puerto del destinatario, tal y como se ha especificado en el apartado anterior.

El método `send()` puede provocar las siguientes excepciones principalmente:

- `IOException`: si ocurre un error de entrada salida.
- `SecurityException`: si el gestor de seguridad del dispositivo no permite el envío de datagramas por el *socket*.
- `PortUnreachableException`: si no se puede encontrar el *socket* destino. No hay certeza de que esta excepción siempre se lance.

El siguiente ejemplo construye un datagrama que lleva como destinatario un servidor local que se está ejecutando en el puerto 6000 y le manda un mensaje.

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

```
import java.io.*;
public class clientetcp {
    public static void main(String argv[]) {
        InetAddress adr;
        String mensaje = "";
        byte[] mensaje_bytes = new byte[256];
        DatagramPacket paquete;
        try {
            DatagramSocket socket = new DatagramSocket (6001);
            adr = InetAddress.getByName("localhost");
            mensaje = "Petición a enviar al servidor: ";
            mensaje_bytes = mensaje.getBytes();
            paquete = new DatagramPacket(mensaje_bytes, mensaje.length(), adr, 6000);
            socket.send(paquete);
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

4.4.4. Leer datos

La lectura de los datagramas que nos llegan por un *socket* no orientado a la conexión puede proceder, bien de las peticiones de los clientes en el servidor, bien de las respuestas de los servidores en los clientes. El método para leer del *socket* en cualquiera de los dos casos es `receive()`:

- `receive(DatagramPacket p)`: leer un datagrama por el *socket*. Este datagrama contendrá datos, la dirección IP y el puerto del remitente, según se ha especificado en el apartado 4.4.2.

Podemos destacar las siguientes excepciones de este método:

- `IOException`: si ocurre un error de entrada salida.
- `PortUnreachableException`: si no se puede encontrar el *socket* destino. No hay certeza de que esta excepción siempre se lance.
- `SocketTimeoutException`: si se agota el tiempo de espera que se ha configurado para la recepción del datagrama.

El método `receive()` es bloqueante, es decir, el proceso esperará en esta instrucción hasta que se reciba un datagrama. Si se quiere configurar este tiempo de espera, podemos emplear las siguientes funciones:

- `setSoTimeout(int timeout)`: activa o desactiva SO_TIMEOUT con el tiempo especificado como parámetro, en milisegundos. Si es 0, la llamada `receive` será bloqueante hasta que se reciba un datagrama.

- `int getSoTimeout()`: para obtener el valor actual de SO_TIMEOUT.

En caso de agotarse el tiempo de espera mientras el proceso está esperando en la llamada `receive()`, se lanza la excepción *SocketTimeoutException*, que podremos tratar con las instrucciones que se crean convenientes, en la sección de tratamiento de errores del código.

A continuación, veremos un sencillo ejemplo de lectura de un datagrama. Una vez recibido, se muestra por pantalla su contenido.

```
import java.net.*;
import java.io.*;
public class pruebaudp {
    public static void main(String argv[]) {
        try {
            DatagramSocket socket= new DatagramSocket (6000);
            byte[] mensaje_bytes = new byte[256];
            DatagramPacket paquete = new DatagramPacket(mensaje_bytes, 256);
            socket.receive(paquete);
            System.out.println("Dirección IP del remitente: " + paquete.getAddress());
            System.out.println("Puerto del remitente: " + paquete.getPort());
            String mensaje = new String(paquete.getData());
            System.out.println("Datos recibidos: " + mensaje);
            System.out.println("Longitud de los datos recibidos: " +
            paquete.getLength());
            socket.close();
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

4.4.5. Cerrar una conexión

Para cerrar un *socket*, se tiene que invocar al método `close()` sobre la instancia de *socket* que nos comunica explícitamente con el cliente:

- `close()`: cierra un *socket*.

La excepción que puede lanzar esta llamada es:

- `IOException`: si ocurre un error de entrada salida.

Un ejemplo sería:

```
sc.close();
```

4.4.6. Ejemplo de un cliente y un servidor

A continuación, se muestra un ejemplo completo y funcional de un cliente y un servidor no orientados a la conexión. El funcionamiento general es el siguiente:

- El cliente envía al servidor todo lo que el usuario le introduce por teclado, hasta que recibe la palabra *end*.
- El servidor lee todo lo que envían los clientes y lo muestra por pantalla.

Es aconsejable que copiéis el código del servidor en un archivo llamado *servidorudp.java* y el código del cliente en un archivo llamado *clienteudp.java*. Los podéis compilar y ejecutar para probar la interacción entre cliente y servidor. Su ejecución sería similar a la siguiente pantalla:

```
escribiendo
al
servidor
end
END
```

```
/** SERVIDOR NO ORIENTADO A LA CONEXIÓN **/
/** Librerías habituales cuando trabajamos con sockets **/
import java.net.*;
import java.io.*;
/** Se crea una nueva classe Java llamada Servidor UDP y una nueva función main que gobierna su
funcionamiento. En primer lugar se declaran dos variables, una de tipo socket Servidor
UDP y otro mensaje para gestionar los datos que intercambiaremos con los potenciales clientes **/
public class servidorudp {
    public static void main(String argv[]) {
        DatagramSocket socket = null;
        String mensaje = "";
/** Se crea un socket servidor en UDP llamado socket por el puerto 6000 que espera conexiones
    de potenciales clientes. **/
        try {
            socket = new DatagramSocket(6000);
/** Se declaran y se inicializan las variables que leerán los datos que nos envía el cliente
    mediante el socket que hemos creado con él. **/
            byte[] mensaje_bytes = new byte[256];
            DatagramPacket paquete = new DatagramPacket(mensaje_bytes, 256);
/*Se inicia el intercambio de mensajes mediante un bucle que lee lo que el client envía y
    lo muestra por pantalla hasta que el cliente envía la palabra "END" **/
            do {
                socket.receive(paquete);
                mensaje = new String(mensaje_bytes);
                System.out.println(mensaje);
            } while (!mensaje.startsWith("END"));
/** Cerramos el socket que nos permitía comunicarnos con el cliente **/
```

```
            socket.close();
        } catch(IOException e1){
         e1.printStackTrace();
            System.err.println(e1);
        }
    }
}
```

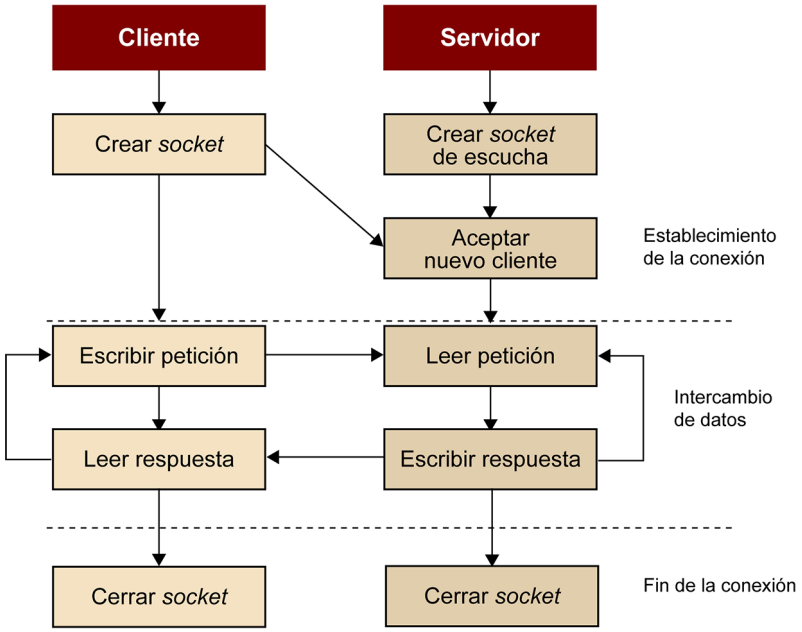
```
/** CLIENTE NO ORIENTADO A LA CONEXIÓN **/
/** Librerías habituales cuando trabajamos con sockets **/
import java.net.*;
import java.io.*;
/** Se crea una nueva clase Java llamada Cliente UDP y una nueva función main que gobierna
su funcionamiento. En primer lugar se declaran las variables, una de tipo socket que representará
un socket cliente UDP; la otra llamada adr para gestionar la dirección del servidor a la cual nos
queremos conectar; finalmente la variable mensaje se utiliza para gestionar los datos que
intercambiaremos con el servidor **/
public class clienteudp {
    public static void main(String argv[]) {
        DatagramSocket socket = null;
        InetAddress adr;
        String mensaje = "";
        byte[] mensaje_bytes = new byte[256];
        DatagramPacket paquete;
        BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));
        mensaje_bytes = mensaje.getBytes();
/** Se crea un socket cliente llamado socket que se conectará al servidor UDP. También en la
    variable adr se pone la dirección del servidor que hemos pasado como parámetro del programa **/
        try {
            socket = new DatagramSocket();
            adr = InetAddress.getByName(argv[0]);
/** Se inicia el intercambio de mensajes mediante un bucle que lee lo que el usuario introduce
    por teclado y lo envía al servidor, hasta que escribimos la palabra “END” **/
            do {
                mensaje = entrada.readLine();
                mensaje_bytes = mensaje.getBytes();
                paquete = new DatagramPacket(mensaje_bytes, mensaje.length(), adr, 6000);
                socket.send(paquete);
            } while (!mensaje.startsWith("end"));
/** Cerramos el socket cliente que nos permitía comunicarnos con el servidor **/
            socket.close();
        } catch(IOException e1){
         e1.printStackTrace();
            System.err.println(e1);
        }
    }
```

```
    }
```

5. Programación de *sockets* orientados a la conexión

TCP es un protocolo orientado a la conexión, que permite la entrega de segmentos de manera fiable, tal y como se ha explicado en el subapartado 3.2. Antes de poder transmitir ningún dato, es necesario establecer una conexión entre los dos extremos que se quieren comunicar, por donde se intercambian los mensajes. A continuación, veremos el diagrama de flujo genérico de cliente y servidor.

Figura 12.



5.1. El servidor TCP

En comunicaciones orientadas a la conexión, la secuencia de operaciones habitual que un **servidor secuencial** realiza en el transcurso de la comunicación con los clientes es la siguiente:

- Crear un *socket* servidor y asignarle un determinado número de puerto conocido para que los potenciales clientes se puedan conectar a él.
- Escuchar nuevas conexiones de clientes y aceptarlas. Como resultado, se crea un nuevo *socket* que comunicará al cliente y servidor. El *socket* de escucha se mantendrá a la espera de nuevas conexiones.
- Leer datos que nos envía el cliente, mediante el *socket pair* establecido con él.
- Enviar datos al cliente mediante el *socket pair*.
- Repetir los pasos 3 y 4 tantas veces como sea necesario.
- Cerrar la conexión con el cliente.

- Repetir los pasos 2 en adelante.

A continuación veremos la clase de Java *ServerSocket* del paquete *java.net* en detalle, para ver los métodos que implementan estos pasos de un servidor secuencial típico. El intercambio de datos entre cliente y servidor lo veremos más adelante, puesto que las llamadas son los mismos para los dos extremos de la comunicación.

5.1.1. Crear un *socket*

Para crear un nuevo *socket* servidor que se ponga a escuchar nuevas peticiones de conexiones por parte de los clientes, lo primero que tenemos que hacer es crear un nuevo objeto de la clase `ServerSocket`, utilizando uno de los siguientes constructores:

- `ServerSocket(int port)`: crea un *socket* servidor vinculado al puerto pasado como parámetro. Si el puerto es 0, se asigna un puerto aleatorio, típicamente un número dentro del rango de puertos efímeros. El máximo número de clientes que se quieren conectar al servidor y están esperando en la cola es 50. A partir de este número, los nuevos clientes se descartan.

- `ServerSocket(int port, int backlog)`: crea un *socket* servidor vinculado al puerto pasado como parámetro. El máximo número de conexiones en la cola es el especificado en el parámetro `backlog`.

- `ServerSocket(int port, int backlog, InetAddress bindAddr)`: crea un *socket* servidor vinculado al puerto y la dirección IP pasados como parámetro, con el máximo número de conexiones en la cola que especifica el parámetro `backlog`. Especificar la dirección IP es útil cuando tenemos diversas interfaces de red en la máquina donde se ejecuta el servidor: wifi, ethernet... También se puede emplear cuando queremos que el servidor solo acepte clientes de una determina dirección.

De las opciones enumeradas, la más utilizada es la primera, por su facilidad de uso. Fijaos en que, como nos encontramos en el servidor, siempre tenemos que indicar un número de puerto que será conocido por el resto de potenciales clientes que se quieran conectar a él. Sin la dirección IP y el puerto de este servidor, que son los que identifican el *socket*, sería imposible que los clientes supieran dónde tienen que conectarse.

Las llamadas anteriores pueden lanzar las siguientes excepciones:

- `IOException`: si ocurre algún error cuando abramos el *socket*.

- `SecurityException`: si existe un gestor de seguridad, como un cortafuegos, que no permite la operación realizada.

- `IllegalArgumentException`: si los parámetros no son correctos. Por ejemplo, si ponemos un número de puerto que está fuera del rango de puertos válidos, de 0 a 65535 incluido.

Por ejemplo, el siguiente código crea un nuevo *socket* servidor en la variable `se`, y muestra por pantalla un mensaje en caso de error :

```
try {
    ServerSocket se = new ServerSocket(80);
} catch (IOException ex) {
    System.err.println(ex);
}
```

5.1.2. Aceptar una conexión

Una vez hemos creado una instancia de objeto `ServerSocket`, el siguiente paso es empezar a escuchar peticiones de nuevos clientes que se quieren conectar con el servidor y aceptarlas, empleando el método `accept()`. Este método bloquea el proceso en curso hasta que se realiza una conexión con el cliente. Es decir, no pasaremos a la siguiente instrucción de código hasta que un nuevo cliente se conecte al servidor.

- `Socket accept()`: se mantiene el *socket* a la espera, escuchando una nueva petición de un cliente y, cuando llega, lo acepta. Devuelve el *socket* creado, que formará parte del *socket pair* que lo vinculará con el extremo cliente y por donde se vehiculará la comunicación bidireccional.

Las llamadas anteriores pueden lanzar las siguientes excepciones:

- `IOException`: si ocurre algún error cuando estamos esperando una nueva conexión de un cliente.

- `SecurityException`: si existe un gestor de seguridad, como un cortafuegos, que no permite la operación realizada.

- `IllegalBlockingModeException`: si un *socket* tiene un canal asociado, el canal se encuentra en modo no bloqueante y no hay una conexión preparada ya para ser aceptada.

Continuando con el ejemplo del apartado anterior, tendríamos:

```
try {
    ServerSocket se = new ServerSocket(80);
    Socket sc = se.accept();
} catch (IOException ex) {
    System.err.println(ex);
```

```
}
```

Fijaos en que el método `accept` devuelve un nuevo objeto `socket (sc)`, que se ha creado para comunicarse explícitamente con el cliente, al cual se le ha aceptado la petición. Es este en el que leeremos las peticiones de los clientes y por el que enviaremos las respuestas del servidor. Así pues, el `socket sc` del servidor, junto con el *socket* creado en la parte cliente, forman el *socket pair* mediante el cual se dará la comunicación bidireccional entre cliente y servidor. En cambio, el `socket se` es un *socket* que se mantiene en modo espera a la escucha de nuevas peticiones de clientes.

Por otro lado, existe la posibilidad de limitar el tiempo que el método `accept()` está bloqueado, esperando un nuevo cliente:

- `setSoTimeout(int timeout)`: configura el tiempo en milisegundos que el servidor está esperando nuevas conexiones de clientes, es decir, el tiempo que el proceso está bloqueado en la llamada `accept()`. Cuando pase el tiempo especificado, se lanza una excepción `SocketTimeoutException`, que tendremos que tratar según convenga.

- `int getSoTimeout()`: relacionado con la llamada anterior, devuelve el tiempo configurado en SO_TIMEOUT.

5.1.3. Cerrar una conexión

Para cerrar un *socket*, se ha de invocar al método `close()` sobre la instancia de *socket* que nos comunica explícitamente con el cliente o que se encuentra escuchando peticiones de nuevos clientes:

- `close()`: cierra un *socket*.

La llamada anterior puede lanzar la excepción:

- `IOException`: si ocurre un error de entrada o salida cuando se realiza el cierre del *socket*.

Cuando hayamos finalizado la comunicación con un determinado cliente, podemos cerrar el *socket* que era extremo local del servidor en el *socket pair*. Por otra parte, un servidor siempre tiene que estar activo, esperando conexiones de nuevos clientes a los cuales servir. En el supuesto de que lo queramos parar por cualquier motivo, podemos emplear el método *close*() anterior sobre el *socket* de escucha.

Esta instrucción es especialmente relevante a la hora de programar y hacer pruebas, puesto que, en caso contrario, se deja el proceso en un estado inconsistente y ocurren problemas para reusar la misma conexión (mismo número de puerto en el mismo equipo).

El siguiente código acepta solo la conexión de un cliente y finaliza cerrando los *sockets* creados.

```
try {
    ServerSocket se = new ServerSocket(8000);
    Socket sc = se.accept();
    sc.close();
    se.close();
} catch (IOException ex) {
    System.err.println(ex);
}
```

5.2. El cliente TCP

Ahora que ya hemos visto las principales clases y sus métodos en el extremo del servidor, veremos en detalle la parte del cliente. La secuencia de operaciones es la siguiente:

- Crear un *socket* cliente, indicándole la dirección IP y el puerto del servidor al cual se quiere conectar.
- Enviar datos al servidor, mediante el *socket* pair establecido con él.
- Leer datos que nos envía el servidor, mediante el *socket* pair establecido.
- Repetir los pasos 2 y 3 tantas veces como se quiera.
- Cerrar la conexión con el servidor.

A continuación veremos la clase *Socket* del paquete *java.net* en detalle, que es la clase que implementa un *socket* cliente y, recordad, también el *socket* servidor que se comunica con el cliente, formando el *socket pair*.

5.2.1. Establecer una conexión

Para conectarse a un servidor, tenemos que crear un nuevo objeto `Socket` empleando uno de los siguientes constructores:

- `Socket(InetAddress address, int port)`: crea un *socket* cliente que se conecta al servidor que tiene la dirección IP y el puerto indicados como parámetros.

- `Socket(String host, int port)`: crea un *socket* cliente que se conecta al servidor, que tiene el nombre de dominio y el puerto indicados como parámetros.

- `Socket(InetAddress address, int port, InetAddress localAddr, int localPort)`: crea un *socket* cliente a partir de los dos últimos parámetros, que se conecta al servidor indicado en los dos primeros parámetros.

- `Socket(String host, int port, InetAddress interface, int localPort)`: lo mismo que el anterior, pero le indicamos explícitamente la interfaz de red sobre la cual trabajaremos. Se emplea en equipos con dos o más conexiones a Internet, por ejemplo Wifi y Ethernet.

La opción más utilizada es la segunda por su facilidad de uso. Por ejemplo, la siguiente línea de código intenta conectarse al puerto 80 del dominio *uoc.edu*, creando un nuevo *socket* cliente en la variable *c*:

```
Socket c = new Socket("uoc.edu", 80);
```

El puerto del cliente lo asigna automáticamente el sistema operativo dentro del rango de puertos efímeros (de 49151 a 65535). La aplicación por el lado del cliente asigna el número de puerto de manera automática y transparente, mientras que en el lado servidor el *socket* de escucha tiene que ser conocido.

Estos constructores pueden lanzar las siguientes excepciones principalmente:

- `IOException`: si ocurre algún error de entrada o salida cuando creemos el *socket*.

- `UnknownHostException`: si la dirección IP del dispositivo no se puede resolver (en la llamada donde pasamos un nombre de dominio).

- `SecurityException`: si existe un gestor de seguridad, como un cortafuegos, que no permite la operación realizada.

- `IllegalArgumentException`: si los parámetros no son correctos. Por ejemplo, si ponemos un número de puerto que está fuera del rango de puertos válidos, de 0 a 65535 incluido.

Tal y como hemos visto antes, podemos gestionar estas excepciones cuando creamos una instancia de un *socket*:

```
try {
    Socket c = new Socket("uoc.edu", 80);
} catch(IOException e1){
    e1.printStackTrace();
    System.err.println(e1);
}
```

5.2.2. Cerrar una conexión

Para cerrar un *socket*, se ha de invocar al método `close()` sobre la instancia de *socket* que nos comunica explícitamente con el cliente. El funcionamiento es el mismo que ya vimos en el servidor:

- `close()`: cierra un *socket*.

La llamada anterior puede lanzar la excepción:

- `IOException`: si ocurre un error de entrada o salida cuando se realiza el cierre del *socket*.

El siguiente código es un ejemplo muy sencillo de un cliente que se conecta a un servidor y cierra la conexión:

```
try {
    Socket c = new Socket("uoc.edu", 80);
    c.close();
} catch (IOException ex) {
    System.err.println(ex);
}
```

5.3. Los flujos de comunicación en TCP

Cuando estamos programando *sockets* en Java, aparte de utilizar las clases ofrecidas por el paquete *java.net*, es habitual trabajar conjuntamente con el paquete *java.io*, que contiene un conjunto de canales de entrada y salida que podemos utilizar para leer y escribir datos fácilmente. Concretamente, se utilizan abstracciones de las operaciones de lectura y escritura de los llamados flujos de datos o *streams*. Esta abstracción permite usar el mismo modelo con independencia del tipo de datos intercambiados y del dispositivo que realiza estas operaciones. Es decir, se utilizan las mismas clases y métodos para gestionar ficheros, la pantalla de un dispositivo o los *sockets*. Este hecho ofrece una gran flexibilidad en la programación.

Un flujo o *stream* es una secuencia de datos que se intercambian entre un origen y un destino de la comunicación. Java define principalmente dos tipos de flujos: flujos de *bytes* (*byte streams*) y flujos de caracteres (*character streams*).

Los **flujos o *byte streams*** gestionan canales de entrada y salida de bytes, por ejemplo, al leer y escribir datos binarios, siendo especialmente útiles cuando trabajamos con archivos. Las siguientes clases trabajan sobre flujos de bytes (*byte streams*):

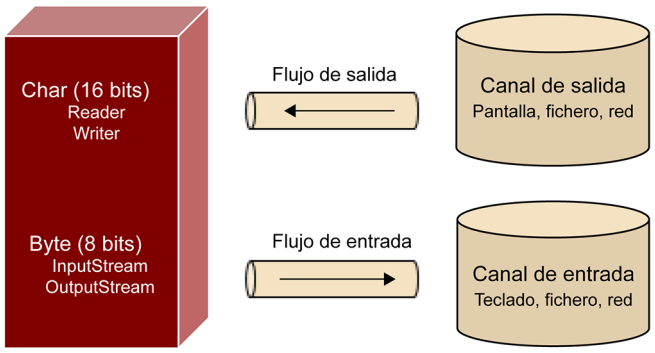
- *InputStream*: clase para la lectura.
- *OutputStream*: clase para la escritura.

Otro tipo son los **flujos o *streams* de caracteres**, para gestionar la entrada y salida de texto Unicode y, por lo tanto, pueden ser internacionalizados:

- *Reader*: clase para la lectura.
- *Writer*: clase para la escritura.

Los flujos de *bytes* y caracteres **se pueden combinar** sobre el canal de entrada y salida en cuestión según nos convenga, pudiendo emplear los dos a la vez, con la instancia correspondiente. Por ejemplo, podemos tener un protocolo de comunicaciones que utilice una cabecera como conjunto de caracteres y el contenido de un fichero binario. En tal caso, emplearíamos un flujo de caracteres para la cabecera y un flujo de tipo binario para el fichero. Podríamos intercambiarnos datos empleando los métodos que cada tipo de flujo nos ofrece, empleando el mismo canal.

Figura 13.

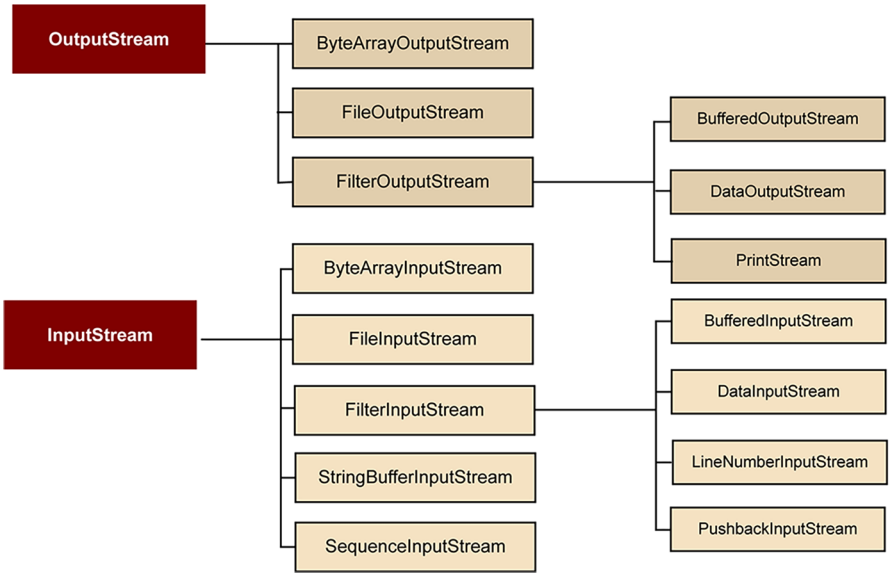


También se puede **traducir** o pasar de un flujo de bytes a uno de caracteres Unicode, codificando los datos, y a la inversa, utilizando las siguientes clases:

- *InputStreamReader*: lee bytes y los decodifica como caracteres.
- *OutputStreamWriter*: recibe caracteres y los codifica en bytes.

Tanto las clases expuestas para trabajar con un flujo de tipo binario como las que trabajan con un flujo de caracteres no se pueden usar directamente, puesto que son clases abstractas. Estas clases tendrán que ser extendidas por otras sobre las que sí podemos trabajar y, así, realizar las operaciones de entrada y salida que correspondan sobre objetos. Este **recubrimiento o *wrapper*** es habitual en Java, proporcionando un conjunto de clases que ofrecen métodos para la manipulación de los objetos de las clases abstractas. La jerarquía de subclases de *InputStream* y *OutputStream* que implementan tipos específicos de canales de entrada y salida es extensa, así que solo destacaremos los siguientes, por ser de uso popular cuando programamos *sockets*.

Figura 14.



Si son **flujos binarios** es habitual trabajar leyendo y escribiendo datos, mediante las clases:

- *DataInputStream*: para el canal de entrada (lectura).
- *DataOutputStream*: para el canal de salida (escritura).

Si estamos trabajando con **flujos de caracteres**, no existe esta opción directamente, pero sí la clase *PrintWriter* para escribir datos formateados.

También es habitual emplear la técnica de los ***buffers*** por su eficiencia a la hora de leer y escribir. Utilizamos un *buffer* interno donde se van almacenando los caracteres y así podemos elegir el ritmo de lectura si no es constante, optimizando el rendimiento del proceso. Las clases que dotan de esta funcionalidad son:

- *BufferedReader*: para el canal de entrada (lectura).
- *BufferedWriter*: para el canal de salida (escritura).

A continuación, vamos a trabajar estos conceptos con un ejemplo práctico: el teclado. La clase Java *System* representa el canal de entrada y salida estándar. El teclado se representa como *System.in*, que es de tipo *InputStream*, esto es, flujo de bytes. Como normalmente al trabajar con el teclado utilizamos caracteres y no bytes, crearemos un objeto *InputStreamReader* a partir de *System.in*. Ahora que ya tenemos el canal de entrada, lo podemos recubrir con otro para dotarlo de las funcionalidades que nos convengan, como por ejemplo, el *BufferedReader*. ´

Figura 15.

Una vez definida la clase que representa el canal de entrada o salida de bytes o caracteres, ya se pueden emplear las funciones habituales para leer y escribir datos, basadas en el típico `read()` y `write()`. Cada clase ofrece un conjunto de funciones propias, que difieren ligeramente entre sí, según los matices que introducen. Por ejemplo, leer o escribir bytes, caracteres, líneas, etc.

```
import java.io.*;
public class eco {
    public static void main (String[] args){
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String msg;
        System.out.println("Introduce una frase: ");
        msg = br.readLine();
        System.out.println("La frase es: " + msg);
    }
}
```

5.3.1. Enviar datos

El mecanismo de lectura y escritura para el cliente y servidor son idénticos. Una vez se ha establecido el *socket* entre cliente y servidor, podemos gestionar los flujos de bytes utilizando las superclases `InputStream` y `OutputStream`.

Veremos primero la escritura. Para enviar datos binarios a bajo nivel (array de bytes), se emplea una instancia de la clase `OutputStream`. Las funciones principales son:

- `OutputStream()`: constructor de la clase.
- `write(int b)`: escribe el byte indicado en el flujo de salida (output stream).
- `write(byte[] b)`: escribe todos los bytes pasados como parámetro.

- `write(byte[] b, int off, int len)`: escribe *len* bytes del parámetro b, empezando desde la posición *off* (offset). El primer elemento es *b[off]* y el último *b[off+len-1]*.
- `flush()`: fuerza la escritura de cualquier byte pendiente en el *buffer*, es decir, inmediatamente se escribe en el flujo de salida todo lo que se había indicado.
- `close()`: cierra el flujo de salida y libera los recursos de sistema asociados.

La excepción más relevante es:

- `IOException`: si ocurre cualquier error de entrada salida.

Un ejemplo de código sería:

```
OutputStream salida = socket.getOutputStream();
byte[] dades = {0x5b, 0x42, 0x40, 0x34};
salida.write(dades);
```

Si queremos enviar datos en formato texto podemos convertir el flujo de salida en otras clases, que heredarán el comportamiento esencial pero añadirán funcionalidades particulares. Una de estas clases envolventes es `DataOutputStream`, la cual permite escribir tipos de datos primitivos de Java. Aparte de los métodos explicados antes, podemos destacar:

- `DataOutputStream(OutputStream out)`: constructor de la clase.
- `writeByte(int v)`: escribe un byte en el flujo de salida.
- `writeChar(int v)`: escribe un carácter en el flujo de salida, esto es, dos bytes, el byte de más peso primero.
- `writeBytes(String s)`: escribe una secuencia de bytes, carácter a carácter, como un solo byte.
- `writeChars(String s)`: lo mismo que el anterior, pero cada carácter se escribe como dos bytes.
- `writeUTF(String s)`: escribe la cadena de caracteres utilizando la codificación UTF-8, creando independencia de la máquina.
- `writeInt(int v)`: escribe un entero en el flujo de salida, esto es, cuatro bytes, el byte de más peso primero.
- `writeLong`, `writeFloat`, `writeDouble`, ...

Como vemos, empleando la clase **`DataOutputStream`**, la riqueza de métodos ofrecidos es mayor, facilitando la tarea del programador. Un ejemplo para escribir un carácter o una cadena de caracteres sería:

```
DataOutputStream salida = new DataOutputStream(socket.getOutputStream());
salida.write('a');
salida.writeUTF("Este es un mensaje para el servidor.");
```

Otra de las clases que nos puede ser de utilidad a la hora de escribir vía *sockets* y que también es un envolvente de la superclase `OutputStream` es la clase **`PrintWriter`**, encargada de imprimir las representaciones formateadas de objetos hacia un flujo de caracteres. Los métodos más destacados son:

- `PrintWriter(OutputStream out)`: constructor de la clase. Implícitamente convertirá caracteres a bytes antes de escribirlo.

- `PrintWriter(Writer out)`: constructor de la clase. Tanto en esta opción como en la anterior, como segundo parámetro le podemos pasar un booleano para configurar el autoflush, es decir, forzar la escritura de todo lo que haya en el *buffer*. En caso contrario, no se hace.

- `print(boolean b)`, `print(char c)`, `print (int i)`, etc. Los caracteres son convertidos a bytes según el sistema de codificación por defecto, y se escriben por el flujo de salida correspondiente. Las mismas variantes están para la llamada `println()` que añade la línea actual escribiendo un final de línea.

Aparte, como en el caso anterior, tenemos la llamada`write()` sobre enteros, caracteres y cadenas de caracteres *(string)*, `flush()`, `close()`...

```
PrintWriter pw = new PrintWriter(socket.getOutputStream(),true);
pw.println("Este es el mensaje que enviamos al servidor.");
```

5.3.2. Leer datos

De manera paralela, vamos a resumir brevemente las principales clases y métodos involucrados en la lectura de datos, tanto en clientes como en servidores, en comunicaciones orientadas a la conexión. Para leer datos binarios a bajo nivel (*array* de *bytes*), se emplea una instancia de la superclase `InputStream`. Las funciones principales son:

- `InputStream()`: constructor de la clase.

- `int read()`: lee el siguiente byte el flujo de datos (0..255). Si no hay dispobible ningún byte porque nos encontramos al final del flujo, devuelve el valor -1. Este método es bloqueante, es decir, el proceso se quedará en esta instrucción esperando un nuevo dato.

- `int read(byte[] b)`: lee un número de bytes determinado por la longitud de la variable que se pasa como parámetro. Se almacenan en la posición *b[0]* hasta la *b[len-1]*. Devuelve el número de bytes que se han leído.

- `int available()`: devuelve el número de bytes que están disponibles para ser leídos en el flujo de entrada.

- `close()`: cierra el flujo de entrada y libera los recursos de sistema asociados.

La excepción más relevante es:

- `IOException`: si ocurre cualquier error de entrada salida.

Un ejemplo de código sería:

```
InputStream entrada = socket.getInputStream();
byte[] dades = new byte[10];
int l = entrada.read(dades);
```

Si queremos leer datos a más alto nivel (caracteres o strings) podemos utilizar un envolvente de la superclase `InputStream`, convertirlo en un objeto **`DataInputStream`**, de manera similar a como hemos explicado en la escritura. Esta clase permite leer tipos de datos primitivos de un flujo de entrada, hereda de la clase `DataInput`, que representa cadenas de caracteres codificadas en un formato Unicode que es una ligera modificación del UTF-8. Aparte de los métodos de lectura comentados en la superclase anterior, podemos destacar:

- `DataInputStream(InputStream out)`: constructor de la clase.
- `byte readChar()`: lee 2 bytes del flujo de entrada y los convierte en un carácter.
- `String readUTF()`: lee en una cadena de caracteres Unicode, codificada según el formato UTF-8 modificado.
- `int readInt()`: lee 4 bytes del flujo de entrada y los convierte en un entero.
- `readDouble`, `readFloat`, `readLong` ...

Otra clase para realizar la lectura es **`InputStreamReader`**, interesante porque convierte de manera transparente al usuario los bytes leídos en el flujo de entrada a caracteres, utilizando la codificación que se especifique o la que esté configurada por defecto en la máquina. Los métodos más relevantes son:

- `InputStreamReader(InputStream out)`: constructor de la clase.

- `InputStreamReader(InputStream out, String charsetName)`: constructor de la clase, donde le especificamos la codificación que se usará en el proceso de lectura de bytes desde el flujo de entrada.

- `int read()`: lectura de un solo carácter.

- `int read(char[] cbuf, int offset, int len)`: lectura de una cadena de caracteres de longitud *len*, almacenados a partir de la posición *offset* del primer parámetro.

- `close()`: cierra el flujo de entrada y libera los recursos de sistema asociados.

Un ejemplo para leer un único carácter sería:

```
InputStream entrada = socket.getInputStream();
InputStreamReader lector = new InputStreamReader(entrada);
int c = lector.read();
System.out.println("Datos recibidos " + (char)c);
```

Si empleamos la clase `InputStreamReader` y leemos byte a byte, el proceso de conversión haría nuestra aplicación poco eficiente. Por eso, es habitual emplear un nuevo envolvente como la clase **`BufferedReader`**, que utiliza *buffers* de lectura para ir almacenando lo que está disponible e ir consumiéndolo según se pida, mediante la técnica FIFO. Aparte de los métodos de lectura descritos en la clase anterior tenemos:

- `BufferedReader(Reader in)` : constructor de la clase. Como segundo parámetro se puede indicar el tamaño del *buffer*.

- `String readLine()` : lee una línea de texto, considerando como final de línea \n, \r o \r\n.

Un ejemplo sería el siguiente:

```
InputStream entrada = socket.getInputStream();
BufferedReader lector = new BufferedReader(new InputStreamReader(entrada));
String linia = lector.readLine();
System.out.println("Datos recibidos " + línea);
```

5.4. Ejemplo de un cliente y un servidor

A continuación, se muestra un ejemplo completo y funcional de un cliente y un servidor orientados a la conexión. El funcionamiento general es el siguiente:

- El cliente envía al servidor todo lo que el usuario le introduce por teclado, hasta que recibe la palabra *END*.

- El servidor lee todo lo que envía el primer cliente conectado y lo muestra por pantalla. Cuando recibe el mensaje END de este cliente, finaliza la ejecución.

Es aconsejable que copiéis el código del servidor en un archivo *servidortcp.java* y el código del cliente en un archivo *clienttcp.java*. Los podéis compilar y ejecutar para probar la interacción entre cliente y servidor. Una ejecución tanto en cliente como en servidor sería:

```
escribiendo
al
servidor
end
END
```

```
/** SERVIDOR ORIENTADO A LA CONEXIÓN **/
/** Librerías habituales cuando trabajamos con sockets **/
import java.net.*;
import java.io.*;
/** Se crea una nueva clase Java llamada Servidor TCP y una nueva función main que gobierna
    su funcionamiento. En primer lugar se declaran dos variables, una de tipo socket Servidor
    TCP y otra mensaje para gestionar los datos que intercambiamos con los potenciales
    clientes **/
public class servidortcp {
    public static void main(String argv[]) throws IOException {
        ServerSocket socket = null;
        String mensaje;
/** Se crea un socket servidor en TCP llamado socket por el puerto 6001 que espera conexiones de
    potenciales clientes. Cuando un cliente concreto se conecta al servidor, se acepta creándose
    un nuevo socket llamado socket_cli, mediante el cual se establecerá la comunicación
    bidireccional entre servidor y cliente **/
        socket = new ServerSocket(6001);
        Socket socket_cli = socket.accept();
/** Se declara y se inicializa una variable llamada entrada por donde se leen los datos que
nos envía el cliente mediante el socket que hemos creado con él. **/
DataInputStream entrada = new DataInputStream(socket_cli.getInputStream());


/** Se inicia el intercambio de mensajes mediante un bucle que lee lo que el cliente envía y
    lo muestra por pantalla hasta que el cliente envía la palabra "END" **/
        do {
            mensaje = entrada.readUTF();
            System.out.println(mensaje);
        } while (!mensaje.startsWith("END"));
/** Cerramos el socket que nos permitía comunicarnos con el cliente y el socket que estaba a
    la espera de nuevas conexiones de otros clientes **/
        socket_cli.close();
        socket.close();
    }
```

```
}
```

```
/** CLIENTE ORIENTADO A LA CONEXIÓN **/
/** Librerías habituales cuando trabajamos con sockets **/
import java.net.*;
import java.io.*;
/** Se crea una nueva clase Java llamada Cliente TCP y una nueva función main que gobierna
    su funcionamiento. En primer lugar es declaran las variables, una de tipo socket que
    representará un socket cliente TCP; la otra llamada adr para gestionar la dirección del
    servidor a la cual nos queremos conectar; finalmente la variable mensaje se utiliza para
    gestionar los datos que intercambiaremos con el servidor **/
public class clienttcp {
    public static void main(String argv[]) throws IOException {
        Socket socket = null;
        InetAddress adr;
        String mensaje = "";
/** Se crea un socket cliente llamado socket que se conectará al servidor que le pasemos
    como parámetro del programa (argv[0]) y puerto 6001. **/
        adr = InetAddress.getByName(argv[0]);
        socket = new Socket(adr, 6001);
/** Se declara y se inicializa una variable llamada entrada que leen lo que el usuario le
    escriba por teclado y la variable salida que será la encargada de escribir los mensajes
  en el servidor. **/
        BufferedReader entrada = new BufferedReader(new InputStreamReader(System.in));
        DataOutputStream salida = new DataOutputStream(socket.getOutputStream());
/** Se inicia el intercambio de mensajes mediante un bucle que lee lo que el usuario introduce
    por teclado y le envía al servidor, hasta que escribimos la palabra “END” **/
        do {
            mensaje = entrada.readLine();
            salida.writeUTF(mensaje);
        } while (!mensaje.startsWith("END"));
/** Cerramos el socket cliente que nos permitía comunicarnos con el servidor **/
        socket.close();
    }
}
```

6. Otras operaciones

A continuación se listan brevemente otras operaciones auxiliares para obtener información del *socket*. Algunas son exclusivas de los servidores, otras pueden ser empleadas tanto en cliente como en servidor.

Tanto en la clase *Socket* como en la clase *ServerSocket*, destacamos:

- `InetAddress getInetAddress()`: devuelve la dirección local del *socket*.
- `int getLocalPort()`: devuelve el número de puerto por el cual el *socket* está escuchando.
- `SocketAddress getLocalSocketAddress()`: devuelve la dirección del *socket* local.
- `setReuseAddress(boolean on)`: activar o desactivar la opción SO_REUSEADDR. Si no está activada, cuando una conexión TCP se cierra, la conexión se encuentra en un estado de *timeout* durante cierto tiempo (conocido como el estado TIME_WAIT), haciendo que no pueda reutilizarse la misma dirección y puerto. De hecho, para aplicaciones que utilizan direcciones o puertos conocidos, probablemente no será posible esta reutilización.
- `setReceiveBufferSize(int size)`: especifica el tamaño del *buffer* de clientes esperando a ser aceptados por el servidor, según la opción SO_RCVBUF.
- `int getReceiveBufferSize()`: obtiene el tiempo anterior.
- `boolean isClosed()`: devuelve el estado del *socket*, si está o no cerrado.

Exclusivas de la clase *Socket* (y en consecuencia, no se pueden usar en el *socket* de escucha del servidor), destacamos las siguientes funciones:

- `int getPort()`: devuelve el número de puerto remoto con el que nos estamos comunicando.
- `SocketAddress getRemoteSocketAddress()`: devuelve la dirección del *socket* remoto.

A continuación, presentamos un ejemplo con algunas de ellas. Básicamente, se crea un *socket* cliente que se intenta conectar al puerto 80 (web) del servidor pasado como parámetro y muestra por pantalla todos los datos de la conexión:

```
import java.io.*;
import java.net.*;
public class informaciónSocket {
    public static void main(String[] args) {
        try {
            Socket sc = new Socket(args[0], 80);
            System.out.println("Conectado al host " + args[0]
                + " con dirección IP " + sc.getInetAddress().getHostAddress()
                + " y puerto " + sc.getPort()
                + ", desde mi socket con dirección " + sc.getLocalAddress()
                + " y puerto " + sc.getLocalPort());
        }
        catch (UnknownHostException ex) {
            System.err.println("Error host desconocido " + args[0]);
        } catch (SocketException ex) {
            System.err.println("Error al conectar en el host " + args[0]);
        } catch (IOException ex) {
            System.err.println(ex);
        }
    }
}
```

El resultado esperado, si no se lanza ninguna excepción durante la creación del *socket*, es:

```
Conectado al host uoc.edu con dirección IP 213.73.40.242 y puerto
80, desde mi socket con dirección /192.168.1.137 y puerto 55102
```

Resumen

Los *sockets* son una interfaz software formada por un conjunto de instrucciones de código, que los programas que se comunican por la red tienen que seguir, para que Internet pueda entregar los datos. A continuación, se exponen resumidamente las principales clases y métodos para programar una aplicación en red en el lenguaje de programación Java.

En comunicaciones no orientadas a la conexión, se crea un único *socket* en el servidor secuencial y un *socket* en cada uno de los clientes. Ambos son del mismo tipo, se emplea la clase `DatagramSocket` para crearlos. La unidad de intercambio de datos es el datagrama y se representa por la clase `DatagramPacket`. Para enviar y recibir datos por el *socket* cliente y servidor se usan los métodos `send()` y `receive()`, indicando siempre los datos del *socket pair*.

En comunicaciones orientadas a la conexión, el servidor secuencial creará dos tipos de *sockets*. En primer lugar, un *socket* de escucha, con nombre conocido, representado por la clase *ServerSocket*, por donde aceptará clientes, usando el método `accept()`. Por otra parte, en el servidor también se creará un socket para atender a cada cliente que se conecte, sin nombre conocido, representado por la clase *Socket*. En el cliente se crea el mismo tipo de *socket* de la clase *Socket*. Por el canal de comunicación bidireccional y unívoco que forma el *socket pair* entre servidor y cada cliente, y que se mantiene activo mientras dure la comunicación, se intercambian flujos de datos. Para enviar y recibir datos, usaremos los métodos `read()` y `write()` o similares de las superclases `InputStream` y `OutputStream`. Para facilitar la programación y el tratamiento de los datos, es habitual emplear clases envolventes o *wrappers* y sus métodos, como `DataInputStream`, `BufferedReader` o `DataOutputStreamPrintWriter`.

En cualquiera de los tipos de *sockets*, siempre es aconsejable cerrar el *socket* empleando el método `close()`, cuando ya no se tenga que usar. De este modo, liberamos recursos innecesarios del sistema y podemos reutilizar las conexiones.

Bibliografía

Kurose, J., y Ross, K. (2000). *Computer Networking: A Top-Down Approach*. Pearson.

Webgrafía

<https://www.ibm.com/support/knowledgecenter/ssw_ibm_i_71/rzab6/howdosockets.htm>.

<https://docs.oracle.com/javase/tutorial/networking/overview/networking.html>.

<http://web.mit.edu/6.031/www/sp19/classes/23-sockets-networking/>.

<https://ioc.xtec.cat/materials/FP/Materials/2201_SMX/SMX_2201_M05/web/html/index.html>.

<https://www3.uji.es/~belfern/libroJava.pdf>.