

3

Programación de comunicaciones en red

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

3.1 CONCEPTOS BÁSICOS: COMUNICACIÓN ENTRE APLICACIONES

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

COMPUTACIÓN DISTRIBUIDA

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

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

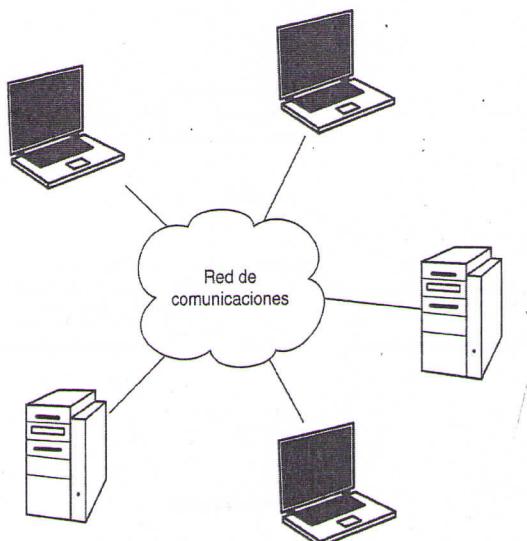


Figura 3.1. Ejemplo de un sistema distribuido



¿SABÍAS QUE...?

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

COMUNICACIÓN ENTRE APLICACIONES

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



¿SABÍAS QUE...?

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

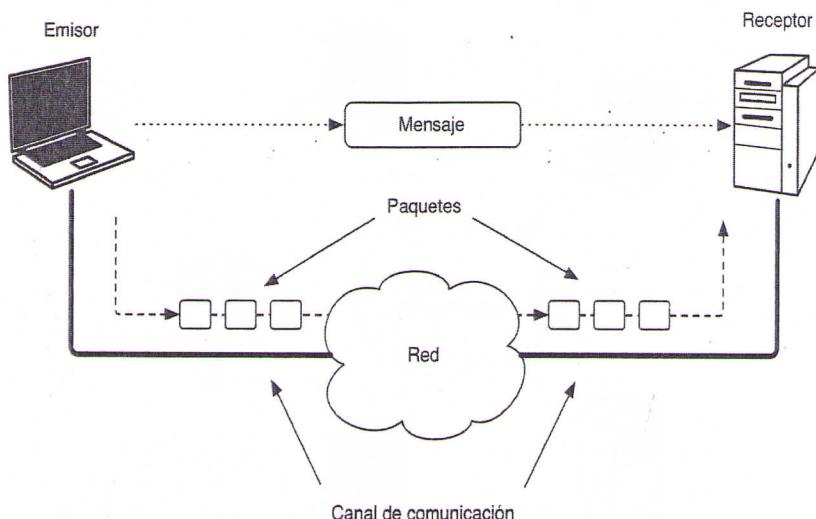


Figura 3.2. Elementos de la comunicación entre aplicaciones

3.1.2.1 Mensaje

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

3.1.2.2 Emisor

Es la aplicación que envía el mensaje.

3.1.2.3 Receptor

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

3.1.2.4 Paquete

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

3.1.2.5 Canal de comunicación

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

3.1.2.6 Protocolo de comunicaciones

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

ACTIVIDADES 3.1

Busca información sobre la historia de la computación distribuida. Averigua qué era ARPANET y cuál es su relación con la red Internet que conocemos actualmente.

Otro paradigma de computación relacionado con la computación distribuida es la computación paralela. Busca información sobre esta e identifica las diferencias fundamentales entre ambas.

¿Qué tipos de redes de comunicaciones conoces? ¿Cuál es la diferencia entre una red de área local (LAN) y una de área extendida (WAN)?



PROTOCOLOS DE COMUNICACIONES: IP, TCP, UDP

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

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

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

¿SABÍAS QUE...?

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

¿SABÍAS QUE...?

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

PILA DE PROTOCOLOS IP

Como se ha mencionado anteriormente, el protocolo IP¹ es el protocolo estándar de comunicaciones en Internet y en la mayoría de sistemas distribuidos. Define además una pila de protocolos completa con los siguientes niveles, por orden ascendente: **nivel de red, nivel de Internet, nivel de transporte y nivel de aplicación**.



Figura 3.3. Pila de protocolos IP

3.2.1.1 Nivel de red

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

3.2.1.2 Nivel de Internet

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

3.2.1.3 Nivel de transporte

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

1 Las siglas en inglés de *Internet Protocol*.

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

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



3.2.1.4 Nivel de aplicación

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

3.2.1.5 Funcionamiento de la pila de protocolos

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

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

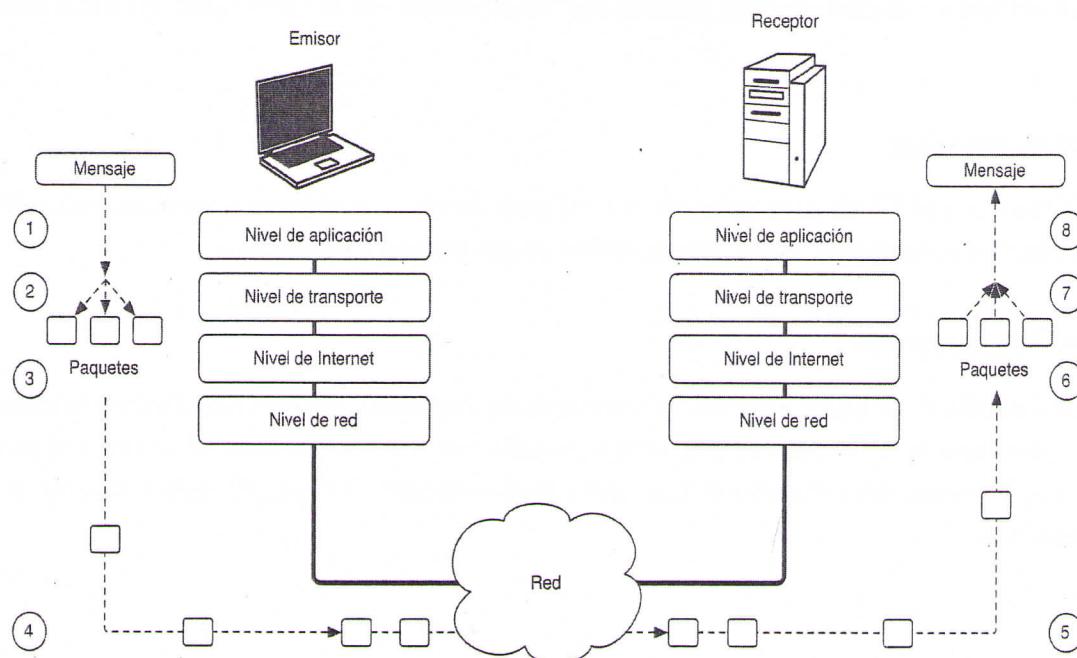


Figura 3.4. Transmisión de mensajes usando la pila de protocolos IP



¿SABÍAS QUE...?

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

ACTIVIDADES 3.2

Busca más información sobre la pila de protocolos OSI. ¿En qué se diferencia de la pila IP?

¿Sabes cómo se transmiten los mensajes a través de Internet? Busca información sobre cómo funcionan los mecanismos de encaminamiento del protocolo IP en el nivel de Internet.

PROTOCOLO TCP

El **protocolo de transporte TCP⁴** es el más comúnmente utilizado en la pila IP, hasta el punto de que muchas veces se denomina a esta como “pila TCP/IP”. Como cualquier protocolo de transporte, TCP se encarga de subdividir los mensajes que recibe del nivel del aplicación, creando paquetes que se envían al nivel de Internet, y combinar los paquetes recibidos de este mismo nivel para formar mensajes que se pasan al nivel superior (nivel de aplicación). Las características principales de TCP son:

✓ Garantiza que los datos no se pierden, siempre y cuando la comunicación sea posible. Esto quiere decir que, siempre que los niveles inferiores de la pila de protocolos funcionen, los mensajes enviados llegarán a su destino.

✓ Garantiza que los mensajes llegarán en orden, siempre y cuando la comunicación sea posible. Esto implica que, no solo todos los mensajes que se envían llegarán (como se indica en el punto anterior), además lo harán siempre en el orden en que fueron enviados.

✓ Se trata de un **protocolo orientado a conexión**. Esto repercute en la forma que se crea y se gestiona el canal de comunicación, y se explica en detalle a continuación.

3.2.2.1 Protocolos orientados a conexión

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

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



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

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

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

Cierre de la conexión: este es el paso final, y se realiza solo cuando se desea interrumpir las comunicaciones.

Una vez cerrado el canal, si se desea reanudar las comunicaciones, hay que volver a empezar, repitiendo el primer paso.

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

PROTOCOLO UDP

El protocolo de transporte UDP⁵ es otro de los componentes fundamentales de la pila IP. Funciona de manera similar a TCP, pero tiene una serie de características fundamentales diferentes:

Al contrario que TCP, se trata de un **protocolo no orientado a conexión**. Esto lo hace más rápido que TCP, ya que no es necesario establecer conexiones, etc.

No garantiza que los mensajes lleguen siempre.

No garantiza que los mensajes lleguen en el mismo orden que fueron enviados.

Permite enviar mensajes de 64 KB **como máximo**.

En UDP, los mensajes se denominan “datagramas” (*datagrams* en inglés).

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

ACTIVIDADES 3.3

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

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

3.3 SOCKETS

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



¿SABÍAS QUE...?

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

3.3.1 FUNDAMENTOS

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

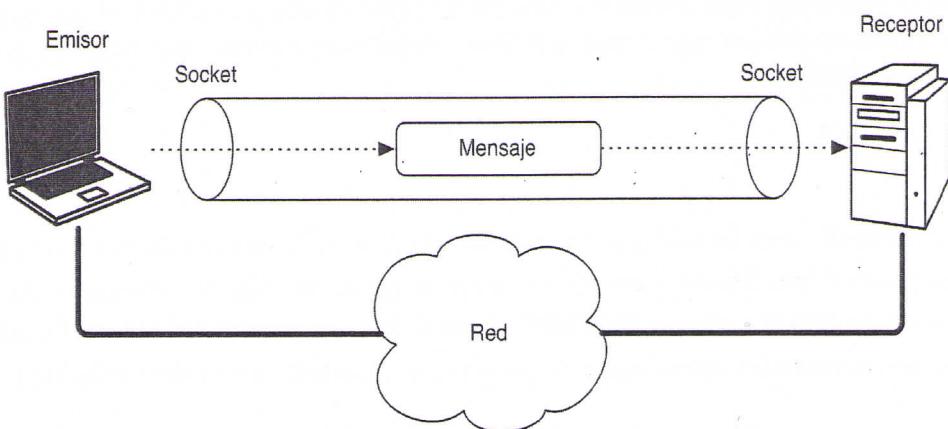


Figura 3.5. Ejemplo de comunicación usando sockets



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

3.3.1.1 Direcciones y puertos

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

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

¿SABÍAS QUE...?

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

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

¿SABÍAS QUE...?

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

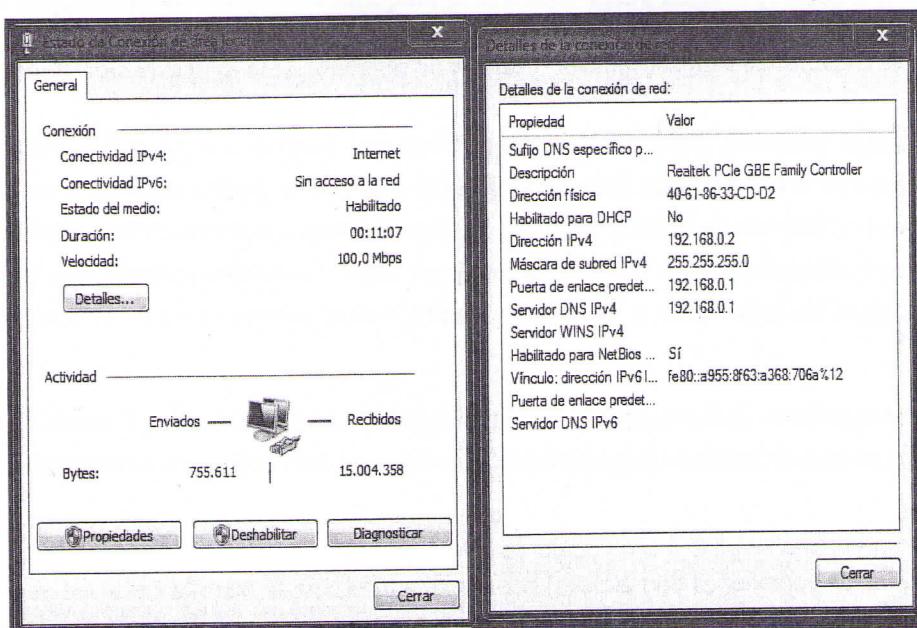


Figura 3.6. Propiedades de un adaptador de red en Windows 7

1. Nuestra máquina tiene la siguiente dirección IPv4:

- 01100100110100110101110100011001

2. Separada en octetos, quedaría de la siguiente forma:

- 01100100 11010011 01011101 00011001

3. Ahora escribimos cada octeto en decimal:

- 100.211.93.25

El resultado final es la dirección 100.211.93.25



¿SABÍAS QUE...?

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

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

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

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

ACTIVIDADES 3.4

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

3.3.1.2 Tipos de *sockets*

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

3.3.1.2.1 *Sockets stream*

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

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

Proceso cliente

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

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

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

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

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

Proceso servidor

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

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

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

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

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

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

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

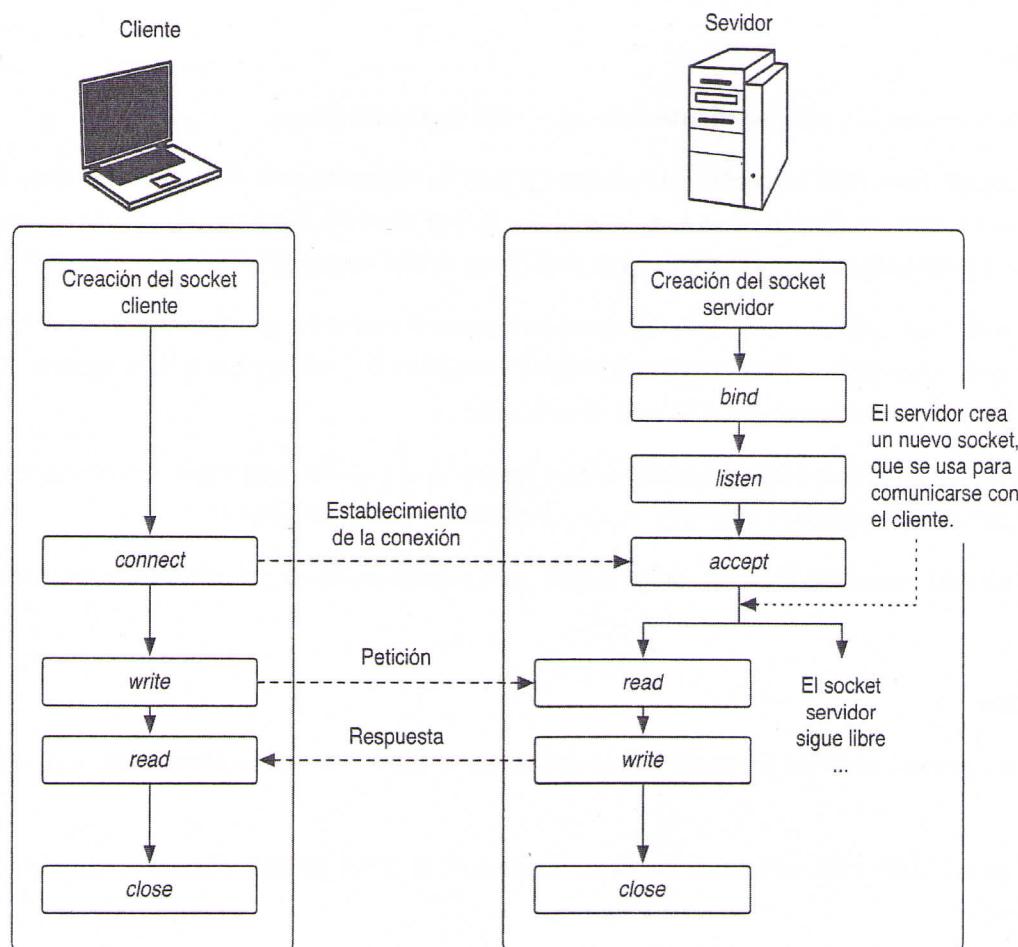


Figura 3.7. Operaciones realizadas durante las comunicaciones con sockets stream

3.3.1.2.2 Sockets datagram

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

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

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

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

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

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

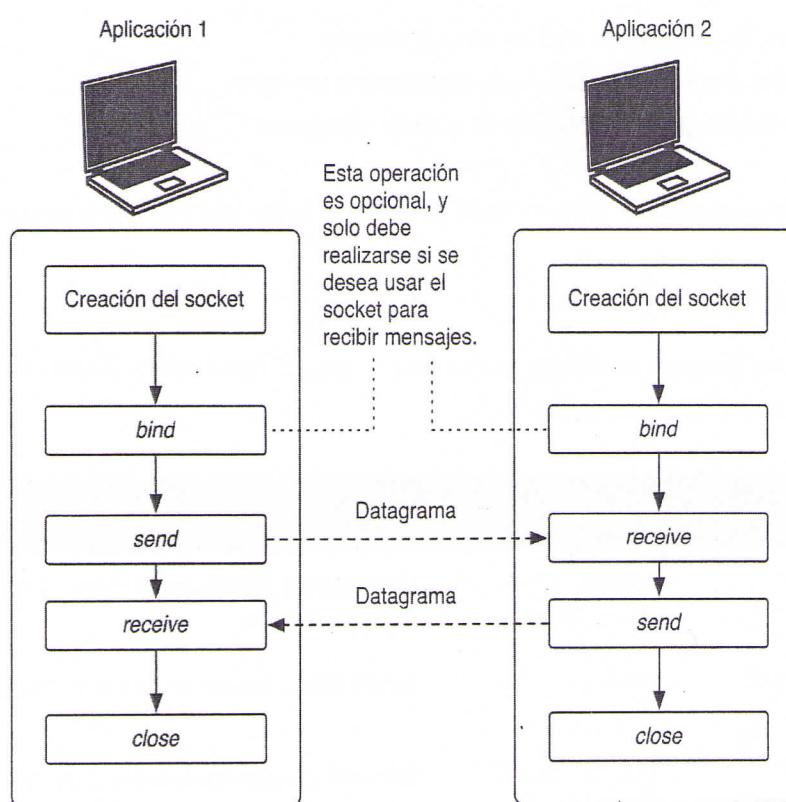


Figura 3.8. Operaciones realizadas durante las comunicaciones con sockets datagram

¿SABÍAS QUE...?

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



ACTIVIDADES 3.5

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

PROGRAMACIÓN CON SOCKETS

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

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

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

3.3.2.1 La clase *Socket*

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

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

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



EJEMPLO 3.2

Proceso cliente usando *sockets stream*:

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

public class ClienteSocketStream {

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

            Socket clientSocket = new Socket();

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

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

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

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

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

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

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

            clientSocket.close();

            System.out.println("Terminado");

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

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

3.3.2.2 La clase ServerSocket

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

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

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

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



SOCKET STREAM

Proceso servidor usando *sockets stream*:

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

public class ServeridorSocketStream {

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

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

            ServerSocket serverSocket = new ServerSocket();

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

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

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

            Socket newSocket = serverSocket.accept();

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

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

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

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

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

            newSocket.close();

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

            serverSocket.close();
        }
    }
}
```



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

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

ACTIVIDADES 3.6

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

3.3.2.3 La clase DatagramSocket

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

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



EJEMPLO 3.4

Proceso que envía un mensaje usando *sockets datagram*:

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

public class EmisorDatagram {
    public static void main(String[] args) {
        try {
            System.out.println("Creando socket datagram");
            DatagramSocket datagramSocket = new DatagramSocket();
            System.out.println("Enviando mensaje");
            String mensaje = "mensaje desde el emisor";
            InetAddress addr = InetAddress.getByName("localhost");
            DatagramPacket datagrama =
                new DatagramPacket(mensaje.getBytes(),
                                   mensaje.getBytes().length,
                                   addr, 5555);
            datagramSocket.send(datagrama);
            System.out.println("Mensaje enviado");
            System.out.println("Cerrando el socket datagrama");
            datagramSocket.close();
            System.out.println("Terminado");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

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

ACTIVIDADES 3.7

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



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

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

public class Ej32ReceiveSend {

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

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

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

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

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

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

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

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

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

            datagramSocket.close();

            System.out.println("Terminado");

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

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

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

ACTIVIDADES 3.8

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

MODELOS DE COMUNICACIONES

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

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



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



MODELO CLIENTE/SERVIDOR

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



TIEMPO REAL La velocidad de respuesta es crucial en la programación de comunicaciones en red.

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

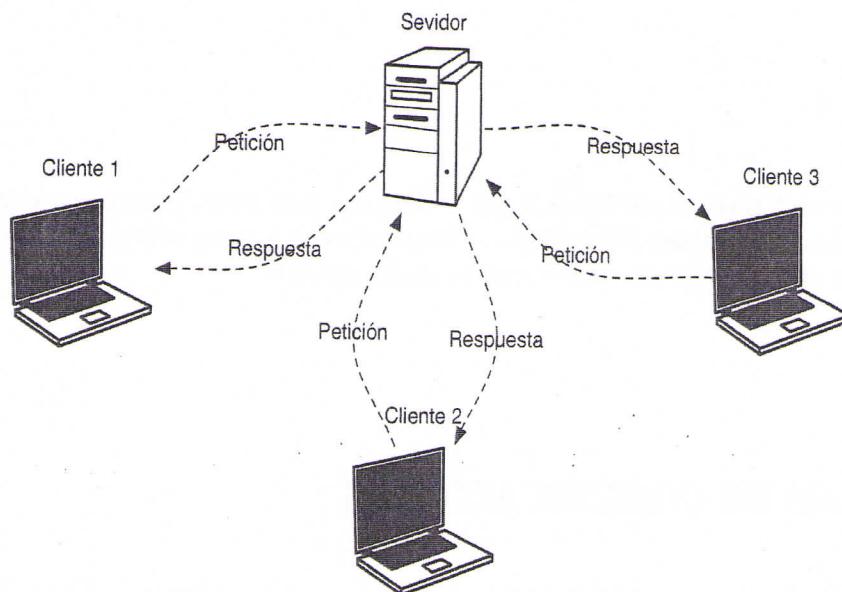


Figura 3.9. Ejemplo de comunicación cliente/servidor

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



¿SABÍAS QUE...?

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

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



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

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

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

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

3.4.1.1 Comparativa con *sockets stream*

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

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

MODELO DE COMUNICACIÓN EN GRUPO

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

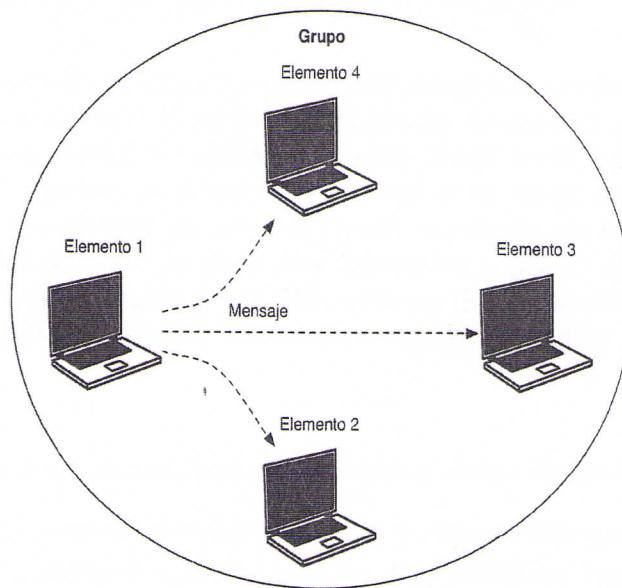


Figura 3.10. Ejemplo de comunicación en grupo



EJEMPLO 3.8

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

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



¿SABÍAS QUE...?

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

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

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

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

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

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



¿SABÍAS QUE...?

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

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

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



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

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

3.4.3.1 Limitaciones de los modelos fundamentales

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

Reparto de roles fijo.

Mecanismo de comunicación único.

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

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

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





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

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

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



¿SABÍAS QUE...?

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

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

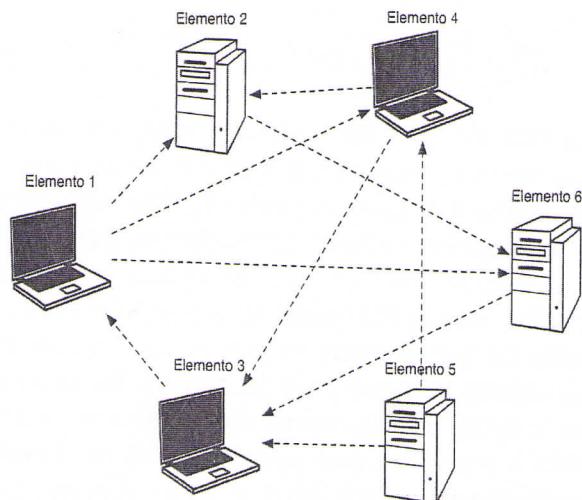


Figura 3.11. Ejemplo de una red peer-to-peer



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

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



¿SABÍAS QUE...?

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

ACTIVIDADES 3.9

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



RESUMEN DEL CAPÍTULO

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

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

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

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



EJERCICIOS PROPUESTOS

1. Escribe una pareja de programas (A y B) que transfieran un fichero entre ellos. El programa A deberá leer un fichero de texto del disco y enviarlo a B. Be recibirá el contenido del fichero y lo imprimirá por su salida estándar. Utiliza para ello *sockets stream*.
2. Escribe un programa que conteste a preguntas. El programa creará un *socket stream* y aguardará conexiones. Cuando llegue una conexión, leerá los mensajes recibidos, byte a byte, hasta que encuentre el carácter ASCII "?" (signo de final de interrogación). Cuando esto ocurra, construirá una frase con todos los bytes recibidos y contestará con un mensaje. El contenido del mensaje dependerá de la frase recibida:
- Si la frase es “¿Cómo te llamas?”, responderá con la cadena “Me llamo Ejercicio 2”.
 - Si la frase es “¿Cuántas líneas de código tienes?”, responderá con el número de líneas de código que tenga.
 - Si la frase es cualquier otra cosa, responderá “No he entendido la pregunta”.
- Escribe, además, dos programas de prueba que verifiquen que el programa responde bien a todas las preguntas.
3. Escribe un programa que responda a saludos usando *sockets datagram*. El programa escuchará por el *socket* mensajes que contengan la cada de texto “Hola”. Cuando reciba uno, responderá a su emisor con otro mensaje que contenga la cadena “¿Qué tal?”. Escribe además un programa adicional para probar el funcionamiento de este.
4. Escribe una pareja de programas (A y B) que usen *sockets datagram* para intercambiar un mensaje llamado *token*. Al arrancarse, el programa A enviará un mensaje al B con la palabra “token”. Cuando el B lo reciba, enviará de vuelta a A un mensaje con la palabra “recibido”, y terminará. Cuando A reciba el mensaje de B, terminará también.
5. Escribe un programa que cuente el número de conexiones que vaya recibiendo. Este programa dispondrá de un *socket stream* servidor. Cada vez que un *socket cliente* se conecte, este le enviará un mensaje con el número de clientes conectados hasta ahora. Así pues, el primer cliente que se conecte recibirá un 1, el segundo un 2, el tercero un 3, etc.
6. Crea una versión generalizada de los programas del Ejercicio 4, para hacer posible que el token se pase entre un grupo de 2 o más programas, en forma de anillo. Cada uno de los programas se debe arrancar indicando como argumentos de entrada su posición en el anillo y el tamaño del anillo. El programa que se encuentre en la posición numero 1 (el primero), generará el mensaje “token” y se lo enviará al programa 2. Cuando este lo reciba se lo pasará al 3, y así sucesivamente. Cuando lo reciba el último programa, lo enviará de vuelta al número 1. Cuando el número 1 lo reciba, la secuencia se interrumpirá (el *token* habrá dado una vuelta completa al anillo). Se deben cumplir además las siguientes restricciones:
- Todos los programas deben tener el mismo código fuente. Se trata, por tanto, del mismo programa, pero ejecutado con distintos parámetros.
 - El programa debe permitir un número variable de elementos en el anillo. El tamaño del anillo se especificará de antemano.
- Se puede programar usando *sockets stream* o *sockets datagram*.

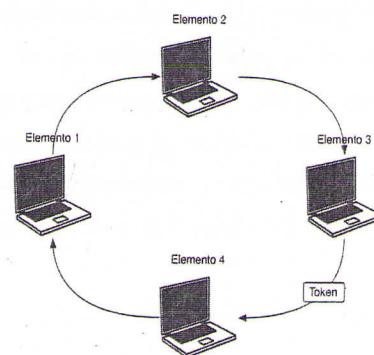


Figura 3.12. Ejemplo de un anillo de procesos (Ejercicio 6)



TEST DE CONOCIMIENTOS

1 ¿Cuál de las siguientes NO es una característica fundamental de TODOS los sistemas distribuidos?

- a) Los elementos que forman el sistema no están sincronizados.
- b) Está formado por más de un elemento computacional.
- c) Los elementos que lo forman siguen el modelo de comunicación cliente/servidor.
- d) Los elementos que lo forman están conectados a una red de comunicaciones.

2 De los siguientes, ¿cuál es el elemento fundamental de la comunicación entre aplicaciones?

- a) Emisor.
- b) Paquete.
- c) Canal.
- d) Todos los anteriores.

3 En la pila de protocolos IP, ¿cuál es la función principal del nivel de Internet?

- a) Crear el canal de comunicación, descomponer el mensaje en paquetes y gestionar la transmisión entre emisor y receptor.
- b) Garantizar la seguridad de las comunicaciones mediante algoritmos de cifrado asimétrico.
- c) Proporcionar los elementos hardware de comunicaciones y sus controladores básicos, sobre los que se efectúan las comunicaciones.
- d) Dirigir los paquetes por la red y hacer que estos lleguen a su destino.

4 De las siguientes afirmaciones, ¿cuáles son ciertas acerca del protocolo TCP?

- a) Es un protocolo orientado a conexión.
- b) Se sitúa en el nivel de transporte de la pila de protocolos IP.

5 a) Garantiza que los datos no se pierden y que lleguen en el mismo orden en que han sido enviados, siempre y cuando la comunicación sea posible.

- b) Permite enviar mensajes de 64 KB como máximo.

6 En la pila de protocolos IP, ¿cuál es la función principal del nivel de transporte?

- a) Crear el canal de comunicación, descomponer el mensaje en paquetes y gestionar la transmisión entre emisor y receptor.
- b) Garantizar la seguridad de las comunicaciones mediante algoritmos de cifrado asimétrico.
- c) Proporcionar los elementos hardware de comunicaciones y sus controladores básicos, sobre los que se efectúan las comunicaciones.
- d) Dirigir los paquetes por la red y hacer que estos lleguen a su destino.

7 De las siguientes afirmaciones sobre los mecanismos de funcionamiento de los sockets, ¿cuáles son ciertas?

- a) En los *sockets datagram*, no siempre es necesario realizar la operación *bind* antes de poder usarlos.
- b) Existen dos clases de *sockets stream*: los *sockets servidor* y los *sockets cliente*.
- c) La operación *accept* crea un nuevo *socket*, conectado al *socket cliente* que realizó el *connect*.
- d) Los *sockets datagram* solo sirven para enviar mensajes.

8 En Java, ¿qué clase debemos usar para crear *sockets stream cliente*?

- a) *ServerSocket*.
- b) *DatagramSocket*.
- c) *Socket*
- d) *MegaSocket*.

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

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

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

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

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

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