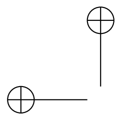
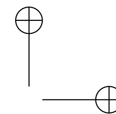



Capítulo 6

Sockets

Al terminar este capítulo, entenderás:

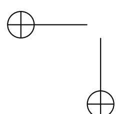
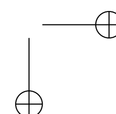
- Qué son los sockets UDP y TCP, y cuáles son sus diferencias.
- Cómo se vincula un socket a un puerto.
- Cómo enviar y recibir datos con sockets UDP y TCP.
- Qué es la E/S parcial y cómo afecta a la programación de redes.
- Cómo delimitar los mensajes en una comunicación TCP.
- Qué es `netcat` y cómo usarlo para probar servidores UDP y TCP.

Un socket es un punto de conexión a la red de comunicaciones, de forma análoga a como un enchufe —que por cierto es la traducción de *socket*— es un punto de conexión a la red eléctrica. Ahí acaba el parecido porque obviamente los sockets de los que hablamos aquí no son dispositivos físicos, sino una mera abstracción de programación.

Para ser más precisos, técnicamente 'socket' es un API, 'un conjunto de funciones', para hacer posible el intercambio de datos entre dos procesos. Los sockets se parecen un poco a las tuberías (*pipes*) de POSIX; el proceso emisor escribe en un extremo y el proceso receptor lee del otro extremo.

A menudo se les llama «BSD sockets» porque el sistema operativo BSD 4.2 fue el primero en incluirlos allá por 1983. A partir de ahí se convirtió en el estándar de facto y fue adoptado por otros sistemas operativos como GNU/Linux, Windows, macOS, etc. Y así sigue siendo en la actualidad.

Ambos, tuberías y sockets, son mecanismos de IPC (Inter-Process Communication), pero a diferencia de las tuberías, que solo permiten comunicar procesos que se ejecutan en un mismo nodo, los sockets comunican procesos que se ejecutan en nodos diferentes (incluso en lugares opuestos del planeta), y también son bidireccionales, entre otras importantes diferencias.

Aunque los sockets son una abstracción relativamente simple, son la base de todo lo demás. No es descabellado decir que prácticamente todo el software que utiliza la red para comunicarse, con cualquier SO, escrito en cualquier lenguaje, y sobre cualquier plataforma, lo hace a través de sockets o de librerías que se basan en sockets.

6.1. Programación de redes

La *programación de redes* como traducción directa del inglés *network programming* engloba las técnicas, herramientas, librerías, patrones, etc. que permiten crear aplicaciones que se comunican a través de la red, o más específicamente, a la parte del desarrollo de una aplicación que se encarga de todo eso. La programación de redes engloba dos aspectos clave: los sockets y los modelos de interacción. El más utilizado de estos modelos es cliente-servidor, pero existen muchos otros como publicador-suscriptor o comunicación de pares (*peer-to-peer*).

En este capítulo nos centraremos en los sockets y las ideas básicas del modelo cliente-servidor. Vamos a tratar *solo lo esencial* acerca de los sockets porque el objetivo es que el lector adquiera las nociones que le permitan aplicar una perspectiva pragmática a los conceptos que veremos en el resto del libro.

Dejaremos para más adelante otros asuntos más avanzados como la serialización de datos, el rendimiento, la seguridad, etc. Aún así «lo esencial» no es poca cosa. La programación de redes, y los sockets en particular, son un campo amplio, lleno de sutilezas y rincones oscuros.

6.2. La clase `socket`

La librería estándar de Python ofrece soporte para sockets en el módulo `socket`. En concreto este módulo proporciona una clase que también se llama `socket`. Incluye los métodos necesarios para manejar conexiones, enviar y recibir datos, etc. El constructor de la clase es:

```
socket(family, type, proto)
```

El parámetro `family` define un conjunto particular de sockets para una tecnología o pila de protocolos de red. Hay familias para Bluetooth (`AF_BLUETOOTH`), ATM (`AF_ATMPVC`), CAN (`AF_CAN`), comunicaciones infrarrojas (`AF_IRDA`) y muchas otras. En este capítulo nos limitaremos a la familia `AF_INET` que corresponde a la pila TCP/IP con IPv4.

El parámetro `type` determina el modelo de comunicación. Esencialmente hay dos tipos:

- `SOCK_STREAM`: Para comunicaciones de flujo (*stream*). La proporcionan protocolos orientados a conexión y habitualmente confiables.
- `SOCK_DGRAM`: Para comunicaciones de datagramas. La proporcionan protocolos no orientados a conexión y normalmente no confiables.

El parámetro `proto` indica qué protocolo quieres usar. Como en la familia `AF_INET` el único protocolo de tipo `SOCK_DGRAM` es UDP y el único protocolo de tipo `SOCK_STREAM` es TCP, es decir, como solo hay un protocolo de cada, este parámetro se puede y se suele omitir.

Resumiendo, puedes crear sockets UDP con:

```
socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Y sockets TCP con:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Pero como `AF_INET` es la familia por defecto, y `SOCK_STREAM` es el tipo por defecto, se puede simplificar la creación de un socket TCP a:

```
sock = socket.socket()
```

6.3. Puertos

Ambos, UDP y TCP, son protocolos de transporte, es decir, sirven para mover datos entre procesos, que es el objetivo principal de los sockets. Para lograrlo, cada socket necesita «direccionar» al otro extremo —el proceso remoto. Esto se logra con dos datos: la dirección IP del nodo remoto y el **puerto** vinculado al proceso remoto. Los puertos son enteros de 16 bits y aparecen en las cabeceras de ambos protocolos. Lógicamente, el puerto origen identifica al proceso emisor.

La tupla (IP, puerto) que identifica a un proceso globalmente en Internet se denomina «endpoint» o también «socket»[1]. De este modo, cualquier flujo de comunicación entre esos dos procesos está determinado por una pareja de tuplas: (IP origen, puerto origen) e (IP destino, puerto destino). Ambas direcciones IP aparecen en la cabecera IP, y ambos puertos aparecen en la cabecera de transporte (TCP o UDP).

[1] En este libro utilizaremos «endpoint» para evitar la ambigüedad de «socket».

Los routers encaminan los paquetes hacia el nodo al que corresponde la dirección destino que aparece en la cabecera IP. Cuando el paquete llega la nodo, el SO extrae el mensaje de transporte y entrega la carga útil al proceso al que corresponda el puerto destino que aparece en esa cabecera de transporte. Este mecanismo, por el cual el SO determina a qué proceso corresponden los datos en función de los puertos, se llama *multiplexación*.

Para vincular un socket a un puerto se utiliza el método `bind()`. Esta operación es imprescindible para crear un servidor porque el servidor tiene el rol pasivo de la comunicación. El servidor espera a que un cliente le contacte y por eso su endpoint debe ser conocido de antemano por el cliente. El servidor por su parte puede averiguar el endpoint del cliente en el momento que este le contacte. El Listado 6.1 muestra cómo utilizar `bind()` para un socket UDP, aunque es igual para un socket TCP.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('170.10.12.4', 2000))
```

LISTADO 6.1: Vinculando un socket a un puerto UDP

El argumento de `bind()` es una tupla (ip, puerto)[2], no solo el puerto. Esto deber ser así porque el nodo dispone de varias interfaces de red, cada una con su dirección IP, y puedes querer que el socket quede vinculado solo a alguna de esas interfaces. Cuando quieras que el servidor sea accesible a través de cualquier interfaz del nodo —algo bastante frecuente— puedes usar la dirección especial `INADDR_ANY` (`0.0.0.0`), aunque en Python se puede usar también una cadena vacía o `None`, como en `('', 2000)`.

```
sock.bind(('', 2000))
```

El puerto 2000 no tiene nada de especial, es solo un ejemplo. Técnicamente hablando puedes usar cualquier puerto libre entre 1 y 65 535, aunque hay ciertos convenios. Los puertos inferiores a 1 024 están reservados para servicios importantes y requieren privilegios especiales. Por ejemplo, el puerto 80 es el puerto asignado para el protocolo HTTP y el 443 para HTTPS. Entre el 1 024 y el 49 151 son puertos registrados, asociados también a servicios o protocolos específicos[3], aunque cualquier usuario puede ejecutar un servidor que los utilice. Por último, los puertos por encima de 49 151 son «puertos efímeros» y son asignados automáticamente por el SO cuando no se especifica, que es lo habitual en los clientes.

[2] Fíjate que hay otra pareja de paréntesis dentro de los de la llamada a `bind()`.

[3] La lista oficial completa de todos los puertos y sus asignaciones en `https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.txt`

Tener un número limitado de puertos implica también una cantidad limitada de procesos (en un mismo nodo) con capacidad de comunicarse con la red. Para ser exactos, un nodo puede tener un máximo de 65 535 procesos utilizando sockets TCP y otros 65 535 utilizando sockets UDP. No es un problema, es más que suficiente para cualquier uso razonable.

Desde el momento que se invoca `bind()`, se dice que el estado del puerto es «abierto». Por contra, si no hay ningún proceso vinculado a determinado puerto, decimos que está «cerrado». Es posible comprobar el estado de los puertos con el comando `ss`[4] como se ve en el Listado 6.2. Solo un puerto abierto es susceptible de recibir y aceptar mensajes. Es el primer paso para crear un servidor UDP. El servidor TCP es algo más complejo, lo veremos enseguida.

```
$ ss -lpun
State    Recv-Q Send-Q     Local Address:Port     Peer Address:Port
UNCONN   0      0               0.0.0.0:2000            0.0.0.0:*
```

Listado 6.2: Comprobando un socket UDP abierto con `ss`

El programa `ss` utiliza los servicios del SO para averiguar esta información, de modo que puedes considerarla 100 % fidedigna. Desde fuera del nodo, también es posible averiguar el estado de los puertos, aunque en ese caso los programas especializados —llamados escáners de puertos (*port scanners*)— recurren a técnicas de sondeo que no son tan fiables. Uno de los escáneres de puertos más conocidos es `nmap` y lo veremos más adelante en el libro.

Para liberar un puerto, se debe invocar el método `socket.close()`, aunque también ocurre cuando el proceso termina. Cada puerto solo puede estar vinculado a un socket, y por tanto a un proceso. Si se intenta ejecutar `bind()` con el mismo puerto en otro socket se producirá un error:

```
OSError: [Errno 98] Address already in use
```

6.4. Comunicación UDP

Como vimos ya en § 5.8, UDP es un protocolo muy simple. Su cabecera solo tiene cuatro campos: los puertos origen y destino, la longitud del mensaje y un checksum (que además es opcional). La única funcionalidad real que ofrece UDP respecto a IP es precisamente esa *multiplexación* de la que hablábamos antes.

[4] `ss` sustituye a `netstat`, aunque este último sigue estando disponible aún en muchas distribuciones de GNU/Linux.

En la sección anterior decíamos que el SO asigna un puerto al socket cliente cuando contacta con el servidor. En el caso de UDP, eso ocurre cuando envía el primer mensaje. Veámoslo con un ejemplo práctico en el Listado 6.3.

```
1  $ python
2  >>> import socket
3  >>> sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
4  >>> sock.getsockname()
5  ('0.0.0.0', 0)
6  >>> sock.sendto(b'hello', ('170.10.12.4', 2000))
7  >>> sock.getsockname()
8  ('0.0.0.0', 48845)
```

LISTADO 6.3: Cliente UDP mínimo

El método `getsockname()` devuelve el endpoint *local* actual, es decir, la IP y puerto al que está asociado el socket en ese momento. El puerto 0 que aparece en la **línea 5** significa que el socket aún no tiene puerto asociado. En la **línea 6** se utiliza el método `sendto()`, que es la manera de enviar datos con un socket UDP. El primer argumento son los datos a enviar y el segundo, el endpoint destino al que deben enviarse. Al hacer esto, el SO le asigna un puerto al socket. Puedes ver cuál ejecutando de nuevo `getsockname()` como hace el listado.

Al ser UDP un protocolo sin garantías, es posible ejecutar sin quejas este código sin que de hecho exista un servidor escuchando en el endpoint destino indicado. Por supuesto, el mensaje se perderá, pero esto es lo que ofrece UDP.

Y otra cuestión importante. Como UDP es un protocolo sin conexión, no hay un destino prefijado para los datos. Por eso, es necesario indicar el endpoint en cada llamada al método `sendto()`. Eso implica que un cliente UDP puede indicar destinos diferentes para cada mensaje que envía, y también que el servidor puede recibir mensajes de clientes distintos en cada recepción.

Un servidor UDP mínimo (Listado 6.4) tiene una estructura sencilla. Después de crear y vincular el socket a un puerto, se utiliza un bucle que recibe un mensaje, lo procesa y devuelve una respuesta al emisor.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('', 2000))

while 1:
    data, client = sock.recvfrom(1024)
    print(f"Se ha recibido el mensaje '{data}' desde '{client}'")
    sock.sendto(b"Enviaste {len(data)} bytes", client)
```

Listado 6.4: Un servidor UDP mínimo

El método `recvfrom()` es la contraparte del método `sendto()`. Su valor de retorno es una tupla con los datos recibidos y el endpoint del emisor. A continuación el código del listado imprime lo recibido en pantalla y devuelve al cliente un mensaje que indica la longitud de los datos recibidos. Tanto `sendto()` como `recvfrom()` manejan secuencias de bytes (no texto), por eso los mensajes llevan el prefijo `b`, que es el modo que tiene Python para indicar que son `bytes`[5].

El argumento de `recvfrom()` indica la cantidad máxima de memoria que el programa está dispuesto a utilizar para almacenar el mensaje recibido. Si el emisor envía un mensaje mayor, el receptor solo recibirá los primeros 1024 bytes (para el caso del ejemplo) y el resto se perderá. Esto es propio de UDP al ser un protocolo orientado a datagramas. Los mensajes son y se tratan de forma independiente, y sin garantías. El programador debe decidir cuál es el tamaño de mensaje adecuado para el propósito concreto de la aplicación que está desarrollando y asegurarse de que no se truncan.

El cliente UDP mínimo (Listado 6.5) es aún más sencillo. Solo necesita enviar un mensaje y esperar la respuesta.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto(b'hola', ('127.0.0.1', 2000))
reply, client = sock.recvfrom(1024)
print(f"El servidor ha respondido: {reply}")
sock.close()
```

Listado 6.5: Cliente UDP mínimo

Este cliente envía un mensaje con el texto `'hola'` al servidor y espera a recibir una respuesta, que imprime en pantalla. Por supuesto, podría enviar y recibir varios mensajes, y como ya se ha dicho podría usar ese mismo socket para comunicarse con otros servidores diferentes durante la misma ejecución.

La implementación de los sockets por parte del SO se está encargando de la mayor parte del trabajo que implica la construcción y reconocimiento de las cabeceras y el envío de esos datos a través de la red. Recuerda que los datagramas UDP se encapsulan en paquetes IP, que a su vez se encapsulan en alguna tecnología de enlace. Esos protocolos tienen sus propias cabeceras que contienen muchos detalles como las direcciones, el tamaño de los mensajes, checksums, etc. Gracias a los sockets, el programador no tiene

[5] «`bytes`» es el tipo Python para secuencias de bytes.

que preocuparse de nada de todo eso. Lo único que tiene que especificar en el cliente es el endpoint del servidor y los datos a enviar. Del resto se encarga el SO.

El diagrama de la Figura 6.1 muestra la estructura y esquema de iteración típico de un cliente y un servidor UDP básicos.

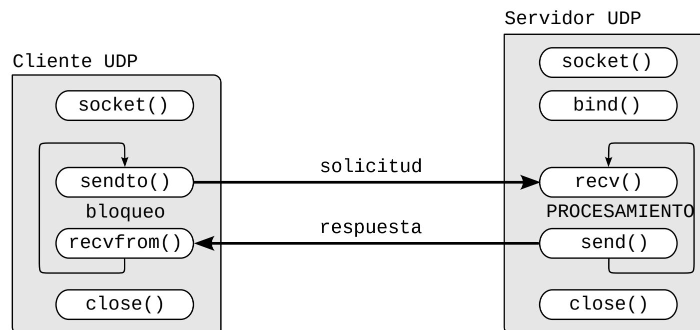


FIGURA 6.1: Diagrama de interacción de un cliente y un servidor UDP

La única diferencia significativa respecto a los ejemplos mínimos anteriores es que representa bucles para el par de invocaciones `sendto()`/`recvfrom()` y su contraparte en el servidor, que encaja con el comportamiento muy habitual en aplicaciones reales.

También es interesante mencionar que en el servidor, después de la recepción de un mensaje lo habitual es realizar algún tipo de procesamiento o acceso/modificación de un recurso antes de devolver un resultado o un código que indique el resultado de la operación. Mientras eso ocurre, lo habitual es que es cliente quede bloqueado en espera de la respuesta.

6.5. Servidor TCP

TCP es un protocolo mucho más complejo que UDP. El intercambio de datos no es posible si no se realiza primero un proceso de conexión exitoso. El protocolo garantiza que todos los datos llegan al otro extremo, sin errores, en el orden correcto, sin partes ausentes ni duplicadas.

Aunque se utiliza la misma clase `socket` y se usan métodos similares, o incluso los mismos que con UDP, el modo en que funcionan y la forma en que logran su objetivo es significativamente diferente. Por eso es importante conocer esas diferencias para evitar confusiones y errores.

Empecemos con un servidor TCP mínimo (Listado 6.6). Suele tener dos partes bien diferenciadas: la primera parte se llama *acceptor* y se encarga de atender nuevas conexiones —el bucle `while` con la llamada `accept()`— y la segunda parte (el manejador) se encarga de gestionar la conversación con un cliente concreto con la función `handle()`. Esta estructura se suele respetar incluso en servidores mucho más complejos.

```
1  import socket
2
3  def handle(conn):
4      data = conn.recv(1024)
5      print(f"Se ha recibido el mensaje '{data}'")
6      conn.send(b"Enviaste {len(data)} bytes")
7      conn.close()
8
9  sock = socket.socket()
10 sock.bind(('', 2000))
11 sock.listen(5)
12
13 while 1:
14     conn, client = sock.accept()
15     handle(conn)
```

Listado 6.6: Servidor TCP mínimo

Además de la llamada a `bind()`, que funciona igual que con UDP, con TCP es necesario llamar también a `listen()`. Este método le pide al SO que ponga el socket a la escucha. Su argumento fija el tamaño del *backlog*, que es una cola en la que se insertan las conexiones en espera de ser aceptadas. Una cola de tamaño 5, como la del ejemplo, significa que podrá haber un máximo de 5 clientes a la espera. Si un sexto cliente intenta conectar, será rechazado.

Al socket `sock`, sobre el que se invoca `bind()` y `listen()`, se le llama *listening socket* o *master socket*, y solo sirve para eso: aceptar nuevas conexiones. No se puede utilizar para enviar o recibir datos.

Para aceptar conexiones (**línea 14**) se invoca el método `accept()`. El proceso queda bloqueado en ese punto hasta que un cliente inicie una conexión. Cuando eso ocurre, el método retorna una tupla con dos elementos: un nuevo socket para intercambiar mensajes con ese cliente (`conn`) y el endpoint de ese cliente (`client`).

Una vez establecida la conexión, el bucle llama a `handle()` pasando como argumento el socket `conn`. Esta función se encarga de manejar la conexión con ese cliente hasta su desconexión.

En este servidor tan simple, la función `handle()` recibe datos del cliente —con el método `recv()`—, los imprime en pantalla e invoca `send()` para

enviar una respuesta indicando la longitud[6]. Solo hay un mensaje de petición y uno de respuesta, pero como en el esquema de UDP, habitualmente la comunicación suele implicar varios mensajes. Para acabar cierra la conexión.

Una vez termina la función `handle()`, el bucle `while` llama de nuevo a `accept()` y espera al siguiente cliente. Como puedes ver, se trata de un bucle infinito, que es lo habitual en un servidor. En una versión en producción, el servidor tendría algún mecanismo para terminar de forma ordenada, habitualmente mediante un manejador de señal.

Es un programa sencillo, y todo parece correcto, salvo que... **NO LO ES**. Este código está pasando por alto la forma en que funciona realmente la comunicación TCP. Volveremos sobre el asunto enseguida, después de echar un vistazo al cliente.

6.6. Cliente TCP

La estructura del cliente TCP para enviar un único mensaje al servidor y recibir una respuesta es muy similar al cliente UDP. Se muestra en el Listado 6.7.

```
import socket

sock = socket.socket()
sock.connect(('127.0.0.1', 2000))

sock.send(b'hola')
reply = sock.recv(1024)
print(f"El servidor ha respondido: '{reply}'")
sock.close()
```

LISTADO 6.7: Cliente TCP mínimo

La diferencia más evidente es que el cliente TCP invoca `connect()` para establecer la conexión, que es la contraparte de `accept()` en el servidor. También es una llamada bloqueante, el proceso queda bloqueado hasta que el servidor acepta la petición y la conexión queda establecida. A partir de ese momento, puede comenzar el intercambio de mensajes, y como puedes comprobar a diferencian del servidor, todo se hace con el mismo socket.

Este esquema de interacción entre cliente y servidor TCP está bien representado en la figura 6.2. Como en el caso de UDP, representa una situación más general en la que cliente y el servidor intercambian varios mensajes. Por lo demás, corresponde con la idea básica de los programas cliente y servidor que acabamos de ver.

[6] como en el ejemplo de UDP

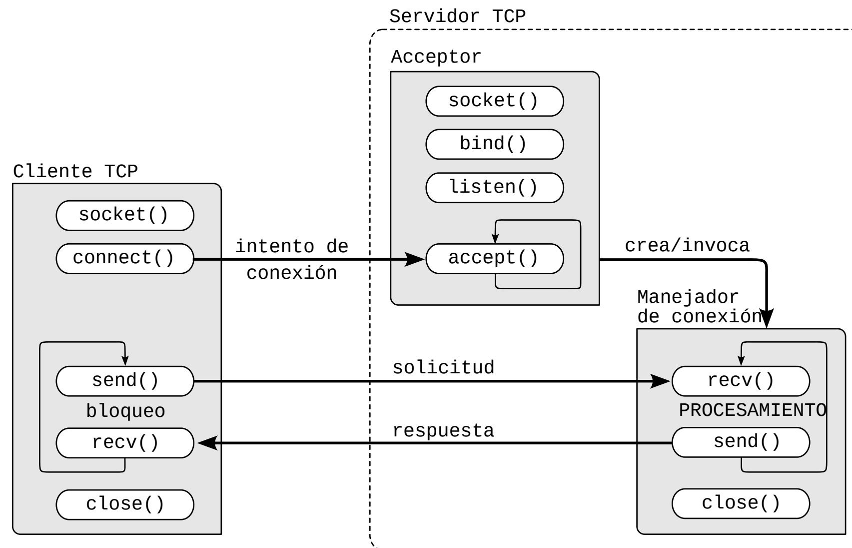


Figura 6.2: Diagrama de interacción de un cliente y un servidor TCP

Las dos cajas grises en el lado del servidor representan los dos elementos que hemos visto: el acceptor y el manejador de la conexión. En el Listado 6.6 el acceptor invoca directamente la función `handle`, pero como veremos más adelante en el capítulo 14, es muy habitual que esa función se ejecute en un nuevo hilo o proceso, como forma de conseguir un mayor rendimiento.

6.7. Flujos de datos y E/S parcial

En el servidor y en el cliente TCP hemos pasado por alto —a propósito— un detalle muy importante. Las primitivas `send()` y `recv()` no funcionan como en principio cabría esperar. Desde luego, distan bastante de cómo lo hacen `sendto()` y `recvfrom()` de UDP.

Por extraño que pueda sonar, invocar `send(datos)` de TCP no garantiza ni mucho menos que se vaya a enviar un segmento cuya carga útil sean exactamente esos `datos`. Aunque en condiciones de baja/media carga es probable que sí ocurra, habrá situaciones en las que varias llamadas a `send()` impliquen el envío de un único segmento (o ninguno), o en las que un único `send()` conlleve el envío de varios segmentos; a fin de cuentas, no hay correspondencia alguna entre los límites de los datos que se pasan a `send()` y la carga útil de los segmentos que se envían a la red. Aunque la probabilidad de que se den estas situaciones sea relativamente baja, el programa debe estar preparado para funcionar correctamente cuando ocurran o fallará, y

eso pasa por entender exactamente cómo funcionan estas primitivas. Y sí, eso también implica que el código que acabamos de ver en los listados 6.6 y 6.7 está **MAL**.

6.7.1. Envío TCP

La comunicación TCP trata los datos como un flujo (*stream*). Los datos que el emisor pasa como argumento a `send()` no se envían inmediatamente a la red como ocurre en UDP. En su lugar van a parar a un buffer de envío (*sending buffer*) bajo el control del SO, y cuando él lo considere oportuno, construirá un segmento e incluirá en él la parte de ese buffer que estime adecuada. Hay varias variables y condicionantes no triviales que influyen en estas decisiones y que veremos en próximos capítulos.

Además de la incertidumbre que conlleva para el programador el mecanismo de construcción y envío de segmentos, ocurre que el método `send()` funciona de un modo peculiar (o no tanto). En realidad es muy similar a la llamada al sistema `write()`[sc] de POSIX y sufre del mismo efecto, llamado «E/S parcial» (*partial I/O*). Implica que en el momento de la invocación, el SO no asegura que pueda aceptar el bloque de datos completo que se le pasa, sino solo una parte. Puede que en ese momento el driver de dispositivo no esté listo o no tenga suficiente espacio en su buffer interno, entre otros motivos. Por esa razón, el valor de retorno de `send()`, la función `os.write()` de Python o la llamada al sistema `write()`[sc] devuelven un entero que indica cuántos bytes pudieron escribir realmente. En realidad `os.write()` es un envoltorio Python para `write()`[sc] así que es lógico que tenga un comportamiento análogo.

La consecuencia directa de la E/S parcial es que potencialmente son necesarias varias llamadas para asegurar que se ha enviado todo el bloque de datos. Por suerte, la solución a este problema es sencilla, al menos para envío/escritura. En un bucle se puede comprobar si quedan datos por enviar, y de ser así, invocar de nuevo `send()` con los datos que «no entraron». El bucle termina cuando el SO acepta por fin todos los datos que se pretendían enviar. Este patrón de envío es tan común que Python lo ofrece como otro método de la clase `socket` llamado `sendall()`. El Listado 6.8 podría ser una implementación simplificada a efectos explicativos.

```python
def sendall(sock, data):
    sent = 0
    while sent < len(data):
        sent += sock.send(data[sent:])
```

LISTADO 6.8: implementación simplificada de `sendall()`

Una solución similar la aplican muchos lenguajes de programación cuando se escribe en archivos convencionales almacenados en disco[7]. Así, el método *file*.`write()`[8] de las clases para manipulación de archivos de muchos lenguajes internamente efectúan varias invocaciones a `os.write()` hasta conseguir escribir todos los datos.

6.7.2. Recepción TCP

La recepción TCP funciona de forma análoga. Cuando se invoca `recv()`, los límites de los datos que se obtienen no tienen por qué coincidir con los segmentos recibidos y menos aún con lo que el emisor colocó en cada llamada a `send()`. También pueden ser necesarias varias llamadas a `recv()` para obtener todos los datos esperados, del mismo modo que pueden ser necesarias varias invocaciones a `os.read()` para leer la cantidad deseada de bytes desde un archivo. Por eso, el método `recv()`, la función `os.read()` y `read()`[sc] devuelven la cantidad de bytes que realmente se han recibido/leído. Como antes, no pierdas de vista que los métodos tipo *file*.`read()` de las clases de manejo de archivos internamente pueden realizar varias llamadas a `os.read()` para conseguir leer la cantidad de datos que se le solicita.

Lamentablemente no hay una solución genérica para implementar un hipotético método `recvall()`, como contraparte de `sendall()`, para resolver esta situación.

El argumento de `recv()` no es de mucha ayuda en esto. Poner un número muy grande no garantiza en absoluto que vaya a obtener el mensaje completo, porque simplemente los datos esperados pueden no haber llegado aún. Como en el caso de UDP, ese argumento simplemente le dice al SO cuanto espacio como máximo estás dispuesto a utilizar para almacenar esos datos. A diferencia de UDP, si los datos recibidos no caben en ese espacio, no se pierden, sino que se quedan en el buffer de recepción (*receiving buffer*) a la espera de que invoques `recv()` de nuevo.

Lo importante aquí es que es responsabilidad del programador delimitar los mensajes y decidir hasta cuando debe seguir llamando a `recv()` y eso no siempre es sencillo. Depende completamente del objetivo y finalidad de la aplicación, o siendo más precisos, del protocolo de aplicación. Si la aplicación envía mensajes de una longitud fija, es sencillo, solo hay que recibir hasta conseguir esa cantidad de bytes. Pero si los mensajes son de longitud variable o impredecible, la cosa se complica.

[7] Utilizamos la expresión ‘en disco’ como una forma común para referirnos al almacenamiento secundario, a pesar de que la tecnología actual no involucra discos.

[8] Usamos *file* para referirnos en genérico a clases para manipulación de archivos. Python llama a esto *file object*, aunque no existe ningún tipo `file`.

En nuestro pequeño servicio contador de caracteres (Listado 6.6), el objetivo del servidor es informar de la longitud del mensaje que recibe, y obviamente es desconocido para él. Eso significa que el servidor no puede saber cuántas veces debe llamar a `recv()` para obtener el mensaje completo. Al cliente le pasa algo parecido cuando recibe: no puede saber qué tamaño tendrá el mensaje de respuesta.

El problema de fondo es que el programador diseña su aplicación para intercambiar mensajes que tienen sentido para el objetivo del programa. Algo como:

```
cliente: "hola"
servidor: "Enviaste 4 bytes"
```

Pero TCP no entiende nada de eso. Para TCP todo es una única secuencia de bytes que abarca toda la conexión y que tiene que llevar de un lado al otro del modo más eficiente posible. Para TCP no hay mensajes individuales, solo un flujo continuo de datos en cada sentido.

Para reconciliar ambos enfoques aparentemente enfrentados, el programador debe incluir algo en el mensaje que le permita, en el otro extremo, saber cuándo ha recibido un mensaje completo (desde el punto de vista de la aplicación). Y es crítico, porque si no recibe suficiente, el mensaje estará incompleto; y si intenta recibir demasiado, el proceso quedará bloqueado porque no hay más datos en ese momento.

framing

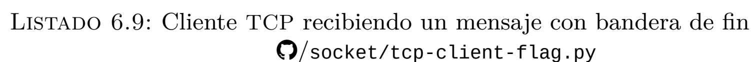
El proceso que determina el comienzo y fin de cada mensaje dentro de un flujo continuo de datos en función de la lógica propia de cada aplicación específica se denomina *framing* en inglés, y se puede traducir como *delimitación de mensajes*.

6.7.3. Bandera de fin de mensaje

Para el propósito del contador de caracteres, una posible solución sería incluir un *token* especial al final del mensaje a modo de bandera. Por ejemplo, tanto cliente como servidor podrían enviar un carácter de nueva linea (`\n`) al final de cada mensaje y el receptor simplemente tiene que seguir recibiendo hasta que aparezca. El listado 6.9 muestra una versión corregida del cliente, que aplica esta solución, y utiliza `sendall()` para el envío.

```
import socket

def read_message(conn):
    data = bytes()
    while (i := data.find(b'\n')) == -1:
        chunk = conn.recv(1024)
        if not chunk:
            return data
```

```
        data += chunk
    line = data[:i]
    return line

sock = socket.socket()
sock.connect(('127.0.0.1', 2000))

sock.sendall(b'hola\n')
reply = read_message(sock)

print(f"El servidor ha respondido: {reply.decode()}")
sock.close()
```

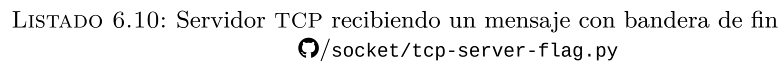


Listado 6.9: Cliente TCP recibiendo un mensaje con bandera de fin
/socket/tcp-client-flag.py

Del otro lado, el listado 6.10 muestra una versión corregida del servidor. Este tiene en cuenta lo que hemos visto sobre E/S parcial, incluye la bandera de fin de mensaje y el manejador de señal para la finalización adecuada. Con esto, si pulsas `Ctrl+C` en la consola en la que se está ejecutando el servidor, la shell enviará al proceso la señal SIGINT y el servidor ejecutará la función `int_handler()` que se encarga de cerrar el socket y terminar ordenadamente el proceso. Aunque añadir este tipo de manejador es ciertamente una práctica muy conveniente en código de producción, por legibilidad no los incluiremos en los ejemplos del libro.

```
import socket
import signal

def int_handler(sig, frame):
    print("\nSIGINT recibido. Cerrando socket y saliendo...")
    sock.close()
    exit(0)

def read_message(conn):
    data = bytes()
    while (i := data.find(b'\n')) == -1:
        chunk = conn.recv(1024)
        if not chunk:
            return data
        data += chunk
    line = data[:i]
    return line

def handle(conn):
    line = read_message(conn)
    print(f"Se ha recibido el mensaje: {line}")
    conn.sendall(f"Enviaste {len(line)} bytes\n".encode())
    conn.close()

sock = socket.socket()
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
sock.bind(('', 2000))
sock.listen(5)
```

```
signal.signal(signal.SIGINT, int_handler)

while True:
    try:
        conn, client = sock.accept()
        handle(conn)
    except OSError:
        break
```



LISTADO 6.10: Servidor TCP recibiendo un mensaje con bandera de fin
/socket/tcp-server-flag.py

La función read_message() que busca la bandera de fin de mensaje es frágil porque podría leer más de lo debido y provocar pérdida de datos, pero de momento sirve para ilustrar este ejemplo trivial. Más adelante en este mismo capítulo veremos una solución más robusta usando el método *file*.readline().

6.7.4. Tamaño del mensaje en la cabecera

Otra solución más flexible y robusta es incluir la longitud del mensaje en su cabecera. De ese modo no se necesita elegir un carácter especial como bandera. Esta solución es de las más habituales en los protocolos de aplicación (*p. ej.* HTTP).

Hasta este momento ni siquiera había una cabecera en nuestro ejemplo del contador de caracteres, solo se enviaba la carga útil y, con la solución anterior, la bandera de fin. Como este es un protocolo[9] muy simple, puede tener una cabecera también muy simple. El mensaje tendrá el formato <longitud>:<datos> siendo <longitud> un número en decimal que indica la cantidad de bytes que mide el resto del mensaje, *p. ej.* 17:you all everybody. La cabecera está formada solo por <longitud>: y el resto es cuerpo[10].

El listado 6.11 muestra un fragmento de código en el que se realiza la recepción conforme a ese formato de mensaje. Fíjate que en este caso no hay problema en que el carácter ':' aparezca en el cuerpo del mensaje, solo el primero funciona como separador entre cabecera y cuerpo.

```
size = bytes()
while b':' not in reply:
    size += sock.recv(1)

size = int(size[:-1])

body = bytes()
while len(body) < size:
```

[9] Sí, técnicamente estamos diseñando un protocolo de aplicación.
[10] Ya dije que sería una cabecera simple.

```
        body += sock.recv(size - len(body))
```

Listado 6.11: Recepción TCP con una mensaje con formato `<longitud>:<datos>`

Tanto esta solución con longitud en la cabecera como la bandera de fin en realidad son soluciones sencillas. Pero date cuenta que el verdadero problema es que aunque estas puedan ser soluciones razonables para este caso, pueden no ser adecuadas en otras situaciones. ¿Qué ocurre si el carácter que hemos elegido como bandera forma parte del mensaje? ¿Y si no es aceptable añadir nada al mensaje? ¿Y si la cabecera debe ser de tamaño fijo o es crítico mantener el tamaño del mensaje al mínimo? Este tipo de problemas son comunes en la programación de redes y es lo que justifica la necesidad de protocolos de aplicación y la razón por la que hay tantos y tan variados.

6.8. Sockets como archivos

La mayoría de los métodos de la clase `socket` que hemos estado usando son «envoltorios» para las llamadas al sistema `socket()`[sc], `connect()`[sc], `send()`[sc] cuyos prototipos aparecen en `<sys/socket.h>` de la librería estándar del lenguaje C.

No es casualidad que hayamos estado haciendo analogías entre sockets y archivos. En los sistemas POSIX se aplica un principio que reza «todo es un archivo» ya sea un dispositivo de caracteres o de bloques (un disco), tubería, terminal, zona de memoria, entrada y salida estándar, etc. Por supuesto, los sockets deben considerar muchas situaciones que no ocurren con archivos convencionales y por eso `recv`/`send` disponen de un argumento opcional `flags` que permite modificar su comportamiento. Pero, en la mayoría de situaciones las primitivas `read`/`write` son equivalentes a `recv`/`send` y, de hecho, cuando se programa en C, lo son.

En Python no existen los métodos `socket.read()`/`socket.write()`, pero si hay dos opciones que permiten tratar a los sockets como archivos en caso de que convenga.

El método `socket.fileno()` devuelve el descriptor de archivo (un entero) asociado al socket. Con este descriptor puedes usar la primitivas `os.read()` y `os.write()` para recibir y enviar datos. Esto no cambia la forma de trabajar con el socket porque estas funciones también aplican la E/S parcial. Sin embargo, disponer del descriptor del archivo te da la posibilidad pasar el descriptor a un subproceso o de utilizar código de librería que no está pensando para sockets, pero acepta descriptores de archivo.

```
>>> sock.fileno()
12
>>> os.write(sock.fileno(), b'hello')
```

La otra opción es `socket.makefile()`. Este método crea un objeto de estilo archivo (un *file object*) para manipular el socket:

```
>>> sock.makefile('r')
<_io.TextIOWrapper name=5 mode='r' encoding='UTF-8'>
```

El método `sock.makefile()` tiene parámetros similares a los de la función `open()`.

- `mode`: `'r'`, `'w'`, `'rb'`, `'wb'`, etc.
- `buffering`: tamaño del buffer, o `0` sin buffer.
- `encoding`: Para hacer (de)codificación automática (si se ha indicado modo texto).
- `newline`: Para indicar qué carácter se usa como salto de línea (consulta `open()` en la documentación de Python).

Y como archivo de alto nivel, sus métodos tienen el comportamiento habitual, es decir, ocupándose de la E/S parcial haciendo varias llamadas a `os.read()`/`os.write()` si es necesario.

- *file*.`read(size)` lee `size` bytes.
- *file*.`readline()` lee hasta que reciba el carácter `newline` indicado en `makefile()`.
- *file*.`readlines()` lee todas las líneas hasta el final de la conexión.
- *file*.`write()` envía `data` completo, que es equivalente a `socket.sendall()`.
- *file*.`flush()` fuerza la escritura (envío) de los datos pendientes.

Pero recuerda que para que todo funcione, el protocolo de la aplicación que estas creando debe asegurar que es factible que lo que esperas recibir (sea por cantidad o formato) puede llegar eventualmente sin producir un bloqueo.

Al tener disponible el método `readline()`, puedes implementar una versión más robusta de la recepción del mensaje con bandera de fin que hemos escrito en § 6.7.3.

```
import socket

sock = socket.socket()
sock.connect(('127.0.0.1', 2000))

sock.sendall(b'hola\n')
reply = sock.makefile().readline()

print(f"El servidor ha respondido: {reply}")
```

```
sock.close()
```

Listado 6.12: Cliente TCP recibiendo un mensaje con bandera de fin usando *file*.readline()

Aparte de ahorrarnos el escribir la función `read_message()`, se encarga de convertir a texto (`reply` es una cadena ya decodificada) y si quedan datos después del salto de línea lo guarda para la siguiente invocación, algo que no hacía nuestra función.

6.9. Gestión explícita de buffers

Como corresponde a un lenguaje dinámico como Python, la memoria necesaria para los buffers de las llamadas a `send()`, `recv()` y otras se reserva y libera automáticamente. Veamos como ejemplo un emisor que envía a un receptor un archivo en bloques de 1024 bytes sobre un flujo TCP. Date cuenta que en este ejemplo no importa quién es el servidor y quién el cliente, la transferencia se puede hacer en ambos sentidos exactamente de la misma forma. En los siguientes fragmentos no se incluye la creación de los sockets, conexión o desconexión, solo la transferencia.

Emisor

```
with open('outgoing.data', 'rb') as f:
    while 1:
        chunk = f.read(1024)
        if not chunk:
            break
        sock.sendall(chunk)
```

Receptor

```
with open('incoming.data', 'wb') as f:
    while 1:
        chunk = sock.recv(1024)
        if not chunk:
            break
        f.write(chunk)
```

Figura 6.3: Transferencia de un archivo creando buffers nuevos en cada iteración

En cada iteración de estos bucles, `read()` y `recv()` crean un buffer interno de 1024 bytes para realizar la correspondiente llamada al sistema. Después crean el buffer `chunk` con los datos que han conseguido leer/recibir. Repito: ¡en cada iteración! Esto puede suponer una ineficiencia nada despreciable, especialmente en plataformas con recursos limitados.

Tanto para `read()` como para `recv()`, el 1024 indica el tamaño del buffer que crean[11] y determina la cantidad máxima de bytes que pueden retornar. Sin embargo, como hemos visto en § 6.7, no trabajan exactamente igual. El método *file*.`read()` siempre devolverá 1024 bytes a menos que no queden tantos en el archivo; mientras que `recv()` puede devolver cualquier cantidad entre 1 y 1024 aunque la conexión siga activa y queden muchos

[11] La destrucción de todos esos buffers es tarea del recolector de basura.

miles de bytes por llegar. Si en lugar de usar *file*.`read()`/*file*.`write()` usases `os.read()`/`os.write()`, los dos bloques de código tendrían una semántica análoga.

Sin embargo, es posible evitar tanta creación y destrucción de buffers. Python ofrece una alternativa que permite crear un único buffer y reutilizarlo en cada iteración del bucle mientras dure la transferencia. Quedaría algo así:

Emisor

```
buffer = bytearray(1024)
view = memoryview(buffer)

with open('outgoing.data', 'rb') as f:
    while 1:
        nbytes = f.readinto(buffer)
        if nbytes == 0:
            break
        sock.sendall(view[:nbytes])
```

Receptor

```
buffer = bytearray(1024)
view = memoryview(buffer)

with open('incoming.data', 'wb') as f:
    while True:
        nbytes = sock.recv_into(buffer)
        if nbytes == 0:
            break
        f.write(view[:nbytes])
```

FIGURA 6.4: Transferencia de un archivo reutilizando un único buffer

El tipo `bytearray` es similar a `bytes`, pero mutable. Por eso se puede reutilizar en cada iteración para almacenar los datos procedentes del archivo en el envío o desde el socket en la recepción. Como puedes ver, los métodos `readinto()` y `recv_into()`[12] son variantes diseñadas para este propósito.

El tipo `memoryview` permite acceder a una parte del buffer sin copiar nada, por lo que es mucho más eficiente que la primera alternativa en la que *file*.`read()` y `recv()` creaban buffers. La variable `nbytes` indica en ambos casos cuántos bytes se han podido obtener realmente. Como el propósito del código es enviar y recibir un archivo, realmente no supone ningún problema si `socker.recv_into()` no recupera la misma cantidad de datos en todas las iteraciones.

6.10. Fin de la comunicación TCP

En los listados anteriores ya has visto que tanto el cliente como el servidor invocan `close()` para cerrar el socket, y con ello la conexión si es TCP. El receptor TCP necesita conocer esta situación. Cuando invoca `recv()`, si el otro extremo ya ha cerrado la conexión, el método devolverá una secuencia vacía. De ese modo el receptor puede terminar también de forma ordenada. Y si estaba en un bucle recibiendo mensajes sucesivos, puede interrumpirlo como corresponda.

[12] También existe un `recvfrom_into()` para sockets de datagramas.

El siguiente listado ilustra este comportamiento. Es un receptor que va concatenando los datos que va recibiendo en el buffer `reply` y sale del bucle al recibir una secuencia vacía (`if not data`).

```
reply = bytes()
while True:
    data = sock.recv(1024)
    if not data:
        break
    reply += data
```

El socket TCP dispone de una forma más fina que le método `close()` para controlar el cierre de la conexión: el método `shutdown()`. Este acepta 3 valores:

- `socket.SHUT_RD`: El proceso ya no va a invocar `recv()`. Eso le dice al SO que a partir de ese momento puede descartar cualquier mensaje entrante sin llegar a almacenarlo en el buffer. Podrá seguir enviando mensajes.
- `socket.SHUT_WR`: El proceso ya no va a invocar `send()`. Eso implica el envío de un mensaje con el flag FIN al otro extremo. Podrá seguir recibiendo mensajes.
- `socket.SHUT_RDWR`: Es la combinación de las dos anteriores.

Cuando el emisor invoca `shutdown(SHUT_WR)`, el receptor experimenta el mismo comportamiento descrito anteriormente: el método `recv()` devuelve una cadena vacía. Es decir, el receptor no puede distinguir si el emisor cerró con `SHUT_WR` o `close()`. En todo caso el proceso debería invocar `close()` para liberar el socket aunque haya invocado `shutdown()` previamente.

6.11. Manejo de errores

En los ejemplos anteriores —y en la mayoría de los que encontrarás en el libro— no se han incluido manejadores de errores y excepciones. La razón para esto es proporcionar listados más compactos y fáciles de explicar y entender. Sin embargo, en un código destinado a producción es fundamental tratar los errores de forma adecuada, porque de lo contrario el programa puede comportarse de forma errática o simplemente terminar abruptamente. Lógicamente, este tratamiento de errores es especialmente importante para los servidores, donde no puedes permitir que la operación incorrecta de un cliente provoque la caída del servidor.

La tabla 6.1 muestra los errores más habituales cuando se opera con sockets y conexiones.

Excepción	Situación
`socket.gaierror`	Error en la resolución de nombres.
`TimeoutError`	`connect()` llevó demasiado tiempo (incluso para sockets bloqueantes).
`BrokenPipeError`	Se invocó `send()` cuando el otro extremo ya había desconectado.
`ConnectionAbortedError`	La conexión se abortó durante la ejecución de `connect()` o `accept()` y no se llegó a completar.
`ConnectionResetError`	Se invocó `send()`/`recv()` cuando el otro extremo ya había cerrado de forma abrupta (RST en lugar de FIN).
`ConnectionRefusedError`	Se invocó `connect()` a un puerto cerrado.

CUADRO 6.1: Errores comunes en programación de sockets

Estos errores aparecen como excepciones que heredan de la clase `OSError` (Figura 6.13). Conocer la jerarquía es útil porque permite al programador decidir lo genérico o específico que quiere hacer el manejador.

```
Exception
 └── OSError (== socket.error)
      ├── socket.gaierror
      ├── TimeoutError
      └── ConnectionError
           ├── ConnectionAbortedError
           ├── ConnectionRefusedError
           ├── ConnectionResetError
           └── BrokenPipeError
```

LISTADO 6.13: Jerarquía de excepciones de `socket`

Por ejemplo, si capturas `ConnectionError`, eso incluye todas las causas de error de la conexión.

```
while 1:
    try:
        conn, client = sock.accept()
    except ConnectionError as e:
        print(f"Error de conexión {e} con el cliente {client}")
        continue
```

Estas excepciones son «nombres propios» para números de error (`errno`) específicos de la excepción `OSError`, pero hay muchos otros valores de `errno` que no tienen un nombre específico. En esos casos, el programador puede capturar la excepción y comprobar el valor para determinar el problema concreto.

```
try:
    [...]
except OSError as e:
    if e.errno == errno.ECONNREFUSED:
```

```
        print("Conexión rechazada")
    elif e.errno == errno.EHOSTUNREACH:
        print("Host inalcanzable")
    elif e.errno == errno.ENETUNREACH:
        print("Red inalcanzable")
    else:
        print(f"Error desconocido: {e}")
```

A continuación puedes ver algunos de ellos. Están disponibles como constantes en el módulo `errno`.

nombre	errno	descripción
EDESTADDRREQ	89	Se requiere dirección de destino.
EPROTONOSUPPORT	93	Protocolo no soportado.
EAFNOSUPPORT	97	Familia de direcciones no soportada.
EADDRNOTAVAIL	99	Dirección no disponible.
ENETDOWN	100	La red está caída.
ENETRESET	102	La conexión se ha perdido por un fallo de red.
EISCONN	106	El socket ya está conectado.
ENOTCONN	107	El socket no está conectado.
EHOSTUNREACH	113	No se puede determinar una ruta hacia el host.
ENETUNREACH	101	La red no es alcanzable.
ETIMEDOUT	110	El tiempo de conexión se ha agotado.
ECONNREFUSED	111	El destino rechaza el intento de conexión.
ECONNRESET	104	El destino ha cerrado la conexión abruptamente.

CUADRO 6.2: Códigos `errno` relacionados con errores de red

6.11.1. Context manager

Los sockets de Python se pueden usar como *context managers*, es decir, con la clausula `with`. Esto ayuda a gestionar automáticamente los recursos asociados, ahorrando al programador la invocación de `close()` y capturando excepciones comunes. El Listado 6.14 muestra un servidor para el servicio Echo que aprovecha esa característica. Tal como indica su nombre, el servidor Echo devuelve al cliente los mensajes que este le envía.

```
import socket

def handle(conn):
    while 1:
        data = conn.recv(1024)
        if not data:
            break

        conn.sendall(data)

with socket.socket() as sock:
    sock.bind(('', 7))
    sock.listen(5)
```

```
    while 1:
        conn, client = sock.accept()
        with conn:
            handle(conn)
```

LISTADO 6.14: Servidor de Echo TCP con *context manager*

Una vez el servidor acepta una conexión, recibe mensajes del cliente y los devuelve con `sendall()`. Si `recv()` retorna una cadena vacía, significa que el cliente ha desconectado. Entonces el bucle `while` de `handle()` termina con `break` y el servidor espera la siguiente conexión. Es interesante destacar que para el caso concreto de Echo utilizar `recv()` sin más no es un problema. El servidor no necesita identificar límites en los datos que recibe. Simplemente devuelve lo que recibe. Da igual si eso ocurre en bloques pequeños o grandes, iguales o diferentes, y al cliente tampoco le va a afectar, porque todo es un único flujo.

6.12. Netcat

El programa `netcat` es, a pesar de su sencillez, una de las herramientas más potentes y versátiles que existen en el campo de la programación, análisis y manipulación de redes y servicios TCP/IP. Es un recurso imprescindible para profesionales de la programación, administración o gestión de redes, expertos en seguridad y hackers.

Existen muchas implementaciones de `netcat` desarrolladas por distintos fabricantes, con sus propias características, pero todas ellas comparten una misma funcionalidad básica y simple que se resume en:

- `netcat` crea un socket TCP o UDP, que puede ser cliente o servidor en función de los argumentos que se le indiquen.
- Una vez establecida la comunicación, envía por el socket todo lo que lea de su entrada estándar y a la vez, envía a su salida estándar todo lo que reciba por ese mismo socket.

Esto lo convierte en un cliente o servidor absolutamente genérico, que combinado con la potencia de la *redirección* que ofrece la `shell` (§ 2.6) permite proporcionar conectividad de red a cualquier programa que consuma y genere datos de su entrada y salida estándar, respectivamente. Por eso se podría decir que `netcat` es un «socket» para usar desde línea de comandos.

Para que todos los usos de `netcat` que se hacen a lo largo del libro sean homogéneos, vamos a usar siempre la implementación de `insecure.org`, que se llama `ncat` y está disponible en la mayoría de distribuciones GNU/Linux en el paquete del mismo nombre.

6.12.1. Sintaxis básica

Aunque `ncat` tiene muchas opciones y funcionalidad, en la gran mayoría de las situaciones se utilizan dos comandos básicos:

- Crear un servidor TCP: `ncat -l <puerto>`
- Crear un cliente TCP: `ncat <nodo> <puerto>`

El parámetro `-l` (*listen*) indica que se trata de un servidor. Para crear un servidor o un cliente UDP simplemente añade el argumento `-u` en ambos casos.

El siguiente es el uso más básico, lo que acabamos de ver, que funciona como una especie de chat tremendamente básico, y que permite a cualquier de los dos extremos enviar y recibir en cualquier momento. Lógicamente, debes ejecutar primero el servidor y luego el cliente. Después puedes escribir en la entrada de cualquiera de los dos y el texto aparecerá en el otro extremo. En este caso, ambos se estarán ejecutando en tu PC, pero por supuesto puede hacerse en nodos diferentes.

Cliente

```
$ ncat 127.0.0.1 2000
hola[ENTER]
```

Servidor

```
$ ncat -l 2000
hola
```

Figura 6.5: Uso básico de `ncat`: servidor y cliente TCP

Veremos unos cuantos ejemplos para ilustrar las muchas posibilidades de `ncat`, pero ten en cuenta que para casi todos existen programas específicos que pueden lograr el mismo objetivo con más posibilidades y, sobre todo, más seguridad.

> Prueba esos ejemplos únicamente si sabes lo que estás haciendo. Algunos pueden suponer un problema para la integridad de tu sistema y tus datos.

Transferencia de archivos

Aplicando redirección puedes enviar un archivo (tanto texto como binario) del cliente al servidor y viceversa. Suponiendo que tienes el archivo `song.mp3` en el nodo cliente, puedes ejecutar lo siguiente:

Cliente

```
$ ncat sidney 2000 < song.mp3
```

Servidor

```
$ ncat -l 2000 > copia-song.mp3
```

FIGURA 6.6: Transfiriendo un archivo con `ncat`

Servidor Echo

`ncat` puede ejecutar cualquier programa conectando directamente el socket que crea con la entrada/salida de ese programa. Si el programa que ejecutas es `cat` conseguirás un comportamiento equivalente al servicio Echo, es decir, devolver al cliente todo lo que reciba de él:

Cliente

```
$ ncat sidney 2000
hola[ENTER]
hola
```

Servidor

```
$ ncat -l 2000 -e /bin/cat
```

FIGURA 6.7: Servidor Echo con `ncat`

Servidor *daytime*

También muy básico. Cuando el cliente conecta y sin enviar nada, el servidor que emula el servicio *daytime* le envía la fecha y hora actual y cierra la conexión. Para lograrlo, basta con ejecutar `date` en lugar de `cat`:

Cliente

```
$ ncat sidney 2000
jue 24 abr 2025 23:50:01 CEST
```

Servidor

```
$ ncat -l 2000 -e /bin/date
```

FIGURA 6.8: Servidor daytime con `ncat`

Lógicamente, puedes hacer lo mismo con cualquier programa que produzca una salida.

Shell remota

De forma similar a los ejemplos anteriores, si el programa que ejecutas con `ncat` es una shell, esta ejecutará los comandos que se le envíen y devolverá el resultado de la ejecución de esos comandos (Figura 6.9). Y para tener una *shell inversa* simplemente ejecuta la shell en el cliente (Figura 6.10).

Desde luego esta no es la forma más conveniente de ejecutar una shell remota en un entorno de producción, para eso existe `ssh`. Sim embargo es habitual encontrar cosas parecidas en software malicioso. Se suele utilizar

Cliente

```
$ ncat sidney 2000
hostname[ENTER]
sidney
```

Servidor

```
$ ncat -l 2000 -e /bin/bash
```

FIGURA 6.9: Shell remota con ncat

Cliente

```
$ ncat sidney 2000 -e /bin/bash
```

Servidor

```
$ ncat -l 2000
hostname[ENTER]
sidney
```

FIGURA 6.10: Shell remota inversa con ncat

como *backdoor* y normalmente explota alguna vulnerabilidad que permite al atacante puede introducir comandos en la máquina vulnerable.

Streaming multimedia

Del mismo modo que se envía el contenido de un archivo para guardarlo en disco, se puede consumir directamente por el receptor. Si dispones de un programa que reproduce audio o vídeo desde su entrada estándar (como `mplayer`) puedes enviar un archivo compatible que se irá transfiriendo a medida que se consume.

Cliente

```
$ ncat sidney 2000 | mplayer -
```

Servidor

```
$ ncat -l 2000 < song.mp3
```

FIGURA 6.11: Streaming multimedia con ncat

Web

Puedes servir una página web con `ncat`, si bien un cliente estándar espera que envíes también la cabecera HTTP. Puedes usar este archivo `index.html` poco ortodoxo porque incluye la cabecera HTTP de respuesta:

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 96

<!DOCTYPE html>
<html>
  <head><title>Ejemplo</title></head>
  <body>
  <h1>Hola desde ncat</h1>
  </body>
</html>
```

Lo puedes servir con el comando:

```
$ ncat -l 2000 < index.html
```

Y cargarlo con `firefox` o cualquier otro navegador web.

```
$ firefox http://127.0.0.1:8080
```

Fíjate que la consulta HTTP GET aparece en la salida del servidor.

```
GET / HTTP/1.1
Host: 127.0.0.1:8080
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: es-ES,es;q=0.8,en-US;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate, br, zstd
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Proxy

Puedes utilizar `ncat` para redirigir una conexión entrante a otro servidor, es decir, otro puerto y/u otro nodo:

```
$ ncat -l 2000 -e "ncat example.net 3000"
```

El trafico recibido en el puerto 2000 de este nodo se redirige al nodo `example.net`:3000. Permite incluso que el flujo entrante sea UDP pero la redirección sea a un servidor TCP o viceversa!

Medir el ancho de banda

En este caso, puedes usar un cliente que envía datos a la mayor velocidad posible —leyendo del dispositivo `/dev/zero`— y el servidor que los recibe los redirige hacía el programa `pv` que calcula la velocidad de llegada. Con esto puedes medir la velocidad máxima (el ancho de banda) entre ambos nodos.

Cliente

```
$ ncat sidney 2000 < /dev/zero
```

Servidor

```
$ ncat -l 2000 | pv > /dev/null
2,84GiB 0:00:04 [1,35GiB/s] [<=>     ]
```

FIGURA 6.12: Midiendo el ancho de banda con `ncat`

Imprimir un archivo PostScript

Este ejemplo funciona con impresoras que soportan el estándar AppSocket/JetDirect, que son la mayoría de las que se conectan por Ethernet o WIFI y, obviamente, debe estar habilitado en su configuración. El archivo debe estar en formato PostScript, pero lo puedes convertir fácilmente desde PDF con `pdf2ps`[13].

```
$ pdf2ps documento.pdf
$ cat documento.ps | ncat ip.de.la.impresora 9100
```

Y ¿qué más?

Hemos visto cómo los sockets proporcionan la base de la programación de redes en prácticamente cualquier sistema operativo y lenguaje de programación. Hemos visto cómo crear servidores y clientes UDP, un protocolo que proporciona comunicaciones sin conexión, rápidas, ligeras pero nada fiables. También servidores y clientes TCP, que proporcionan un flujo de datos orientado a conexión, con entrega garantizada y corrección de errores, incidiendo también en la importancia de la multiplexación y los puertos, o las implicaciones de la entrada/salida parcial y la gestión eficiente de los buffers.

Los sockets son en realidad solo el punto de partida. Hay muchos temas pendientes, que cubriremos en los siguientes capítulos, y que también son fundamentales para entender y construir aplicaciones de red: los modelos cliente-servidor y publicación-suscripción, la serialización de datos, el diseño de protocolos de aplicación, la gestión eficiente de múltiples conexiones concurrentes, etc. Además, conocer los sockets te va a ayudar a entender los detalles del funcionamiento del control de flujo, de errores, el control de congestión, etc., que veremos en los siguientes capítulos.

También la seguridad es un tema fundamental, y muchos de los ataques se aprovechan de errores relacionados con la programación de sockets o de su uso incorrecto por parte del programador.

[13] Paquete `ghostscript`