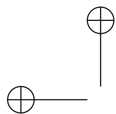
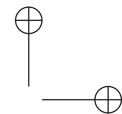



Capítulo 14

Cliente-Servidor

Al terminar este capítulo, entenderás:

- Qué características tiene el modelo cliente-servidor.
- Cómo se crean servidores concurrentes multiproceso o multihilo
- Qué es el *preforking* y el *pool* de procesos e hilos.
- Qué consecuencias tienen las operaciones bloqueantes.
- Cómo utilizar los mecanismos de programación asíncrona para conseguir concurrencia eficiente.

En el capítulo 6 ya vimos las ideas básicas de modelo cliente-servidor como forma de acercarnos a los sockets. Ahora profundizaremos en este modelo y veremos sus posibilidades y limitaciones. Así pues, vamos a continuar en el punto donde lo dejamos en ese capítulo.

El modelo (o paradigma) cliente-servidor (Figura 14.1) es, sin ninguna duda, el enfoque más simple y habitual para crear aplicaciones distribuidas. La mayoría de los **protocolos de aplicación** clásicos de TCP/IP: HTTP, DNS, IMAP, SMTP, etc. están basados en este modelo, y a día de hoy, también la mayoría de aplicaciones que se desarrollan lo siguen usando; tal es el caso de la omnipresente arquitectura REST.

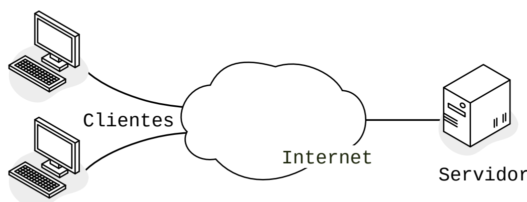
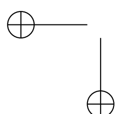
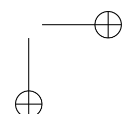


FIGURA 14.1: Modelo Cliente-Servidor

Una *aplicación distribuida* es una que está compuesta por varios componentes que se ejecutan en distintos nodos y colaboran entre sí mediante el

intercambio de mensajes para realizar alguna tarea. En esa definición cabe la gran mayoría de aplicaciones que utilicen la red para comunicarse, que a su vez son la mayoría de las que usamos hoy en día. Sí, casi todo el software que usamos a diario son aplicaciones distribuidas.

En este modelo cliente-servidor aparecen exactamente esos dos **roles** bien diferenciados que le dan el nombre:

- El **servidor** es la parte pasiva. Permanece inactiva a la espera de una petición para realizar una tarea o proporcionar un recurso a través de la red.
- El **cliente** es la parte activa. Envía una petición a un servidor para que éste realice la tarea especificada, o bien le proporcione acceso a un recurso remoto.

Que sean 'roles' es importante porque un mismo programa puede tomar uno, otro o ambos en distintos momentos o circunstancias, es decir, no definen tipos de aplicaciones. Es cierto que coloquialmente hablamos de 'servidores' y 'clientes' refiriéndonos a programas concretos: *p. ej.* el servidor web y el cliente de correo. Pero eso se debe simplemente a que uno de los roles destaca muy claramente sobre el otro, y ciertamente también es habitual encontrar programas que solo actúan como cliente.

También es común hablar de nodos como 'clientes' y, sobre todo, como 'servidores' para denotar que su actividad principal es la de alojar ese tipo de programas. Aunque la arquitectura hardware de un nodo de cómputo puede ser exactamente la misma independientemente de si ejecuta programas servidores o clientes, es fácil identificar algunas características diferenciadoras: el que llamamos 'servidor' suele disponer de una mejor conexión a la red, mayor ancho de banda, está montado en un *rack*, tiene fuente de alimentación redundante y no dispone de pantalla, teclado, ratón u otros periféricos habituales, ya que ninguna persona lo va a utilizar como PC de escritorio. Es hardware diseñado para funcionar ininterrumpidamente porque los programas servidores suelen estar pensados para ejecutarse de forma continua e indefinida.

Sin embargo, en un entorno de desarrollo las aplicaciones servidoras se ejecutan en los PC de los desarrolladores o bien en máquinas virtuales o contenedores. Incluso en un entorno doméstico es relativamente común que los usuarios ejecuten en sus máquinas servidores de diversa índole, aunque por supuesto es más habitual que ejecuten clientes.

El modelo cliente-servidor implica un patrón de comunicación característico conocido como **petición-respuesta**. En éste, el cliente realiza una

petición —que suele incluir una operación y un identificador de recurso— y el servidor devuelve una respuesta con un resultado o un valor de retorno, indicando si la operación se pudo realizar satisfactoriamente o se produjo un error. El formato de estos mensajes de petición y respuesta está especificado en un protocolo de aplicación. Por ejemplo, el protocolo HTTP encaja bastante bien en esta idea. Las operaciones son `GET`, `POST`, `PUT` y `DELETE`, etc., y los recursos se identifican con una URL. Por supuesto, existe una gran cantidad protocolos, pero la mayoría siguen esta pauta. Los recursos que ofrece un servidor pueden ir desde una impresora hasta elementos de menor granularidad, como los registros de una base de datos o los archivos de un directorio. Sin embargo, en este capitulo no vamos a hablar de protocolos de aplicación.

En su lugar, vamos a hablar de productividad y rendimiento. En la mayor parte de las situaciones, el modelo cliente-servidor implica un servidor atendiendo a múltiples clientes simultáneamente. Para lograr eso con un forma razonable, se requiere algún mecanismo de ejecución concurrente, ya sean procesos, hilos, o E/S asíncrona. Todo eso es lo que vamos a tratar en este capítulo.

14.1. Upper

Para concretar las técnicas que vamos a estudiar, utilizaremos un servicio muy básico: un conversor a mayúsculas, es decir, el servidor devuelve una versión en mayúsculas de la cadena de texto que el cliente le envía. Es equivalente al típico servicio *echo*, pero en lugar de simplemente devolver la misma cadena que se le envía, ese pequeño cambio de pasar a mayúsculas que introduce *upper* demuestra que el servidor está haciendo trabajo, por simple que sea. El funcionamiento del cliente es trivial: obtiene cadenas desde la consola, las envía al servidor, recibe las respuestas y las imprime en pantalla.

Para explorar las distintas opciones vamos a empezar por TCP, ya que al estar orientado a conexión, da más juego a la hora de evaluar su rendimiento. Después veremos qué puede hacer UDP y qué diferencias prácticas existen.

14.2. Servidor TCP

En esta primera versión (Listado 14.1) tenemos un bucle al final del listado que acepta conexiones y las delega a una función `handle()` que se encarga de la comunicación con el cliente hasta su desconexión. Este esquema se

llama patrón *acceptor* y es el mismo que vimos en el Listado 6.6 y que se repite en todos los ejemplos del capítulo.

```
import sys
import time
import socket

def upper(msg):
    time.sleep(1)  # simulates a complex job
    return msg.upper()

def handle(sock, client):
    print(f"Client connected: {client}")
    while 1:
        data = sock.recv(32)
        if not data:
            break

        sock.sendall(upper(data))

    sock.close()
    print(f"Client disconnected: {client}")

def main(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('', port))
    sock.listen(5)

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

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")
```

LISTADO 14.1: Servidor TCP
/upper/tcp_server.py

En un servidor monoproceso y monohilo (como este) existen dos puntos de bloqueo: esperar un nuevo cliente —en `accept()`—, o esperar un nuevo mensaje del cliente conectado —en `recv()`. y el proceso solo puede estar en uno de ellos. Por supuesto, eso también implica que solo puede haber un cliente conectado; hasta que no acabe el bucle de la función `handle()`, no podrá ejecutar la siguiente iteración del bucle de la función `main()`.

Se le llama servidor *iterativo*, porque *itera* atendiendo cliente tras cliente, propiciando una *serie* de conexiones. Dicho de otro modo: un cliente no

será atendido hasta que el servidor termine con el anterior. Aquí 'iterativo' es por definición lo contrario a 'concurrente'.

Un detalle relevante que hay que comentar es la forma en la que se hace la conversión a mayúsculas mediante la función `upper()`. Aparte de llamar al método `bytes.upper()`, que es la que realmente hace el trabajo, invoca `time.sleep(1)` para provocar una pausa artificial de 1 segundo. El motivo es simular que pasar a mayúsculas es una tarea compleja, para así poder apreciar más fácilmente el efecto de las técnicas de concurrencia que veremos después y la mejora que suponen.

Otro detalle que te puede llamar la atención es la llamada a `setsockopt()`. Como su nombre indica sirva para modificar el valor de una *opción* del socket. Estas opciones sirven para modificar ciertos comportamientos del socket. En concreto esta (`SO_REUSEADDR`) permite arrancar un servidor en un puerto que había sido utilizado recientemente. Esto es solo una conveniencia para los ejemplos y no es recomendable en producción. Veremos este asunto con detalle más adelante en § 12.3.1.

14.3. Cliente TCP

El cliente, después de conectar con el servidor, ejecuta un bucle que lee una línea de texto desde consola —llamada `sys.stdin.readline()`—, la envía al servidor y espera la respuesta.

El tratamiento de la E/S parcial en este caso es simple. Los mensajes que se mueven entre cliente y servidor no necesitan límites definidos, no importa si se trocean o dónde, al final todo el texto llegará al servidor y toda la secuencia de respuestas llegará de vuelta al cliente. En todo caso, para que el funcionamiento resulte más intuitivo en cualquier circunstancia, el cliente se asegura de que recibe una respuesta del mismo tamaño que el último mensaje enviado; en el bucle `while len(msg) <sent` al final de la función `main()`.

```
import sys
from socket import socket

def main(host, port):
    with socket() as sock:
        sock.connect((host, port))

        while 1:
            data = sys.stdin.readline().strip().encode()
            if not data:
                break

            sock.sendall(data)
```

```
            msg = bytes()
            while len(msg) < len(data):
                msg += sock.recv(32)

            print("Reply is '{0}'".format(msg.decode()))

if len(sys.argv) != 3:
    print("Usage: {0} <host> <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(sys.argv[1], int(sys.argv[2]))
except KeyboardInterrupt:
    print("shut down")
```

LISTADO 14.2: Cliente TCP

/upper/tcp_client.py

Para probar el programa simplemente ejecuta servidor y cliente en consolas distintas tal como aparece a continuación. Para terminar la conexión simplemente cierra la entrada estándar del cliente, lo que puedes conseguir pulsando `Ctrl+D`.

Cliente

```
$ ./tcp_client.py
hola
Reply is 'HOLA'
```

Servidor

```
$ ./tcp_server.py 2000
Client connected: ('127.0.0.1', 43172)
Client disconnected: ('127.0.0.1', 43172)
```

FIGURA 14.2: Servidor y cliente upper TCP

Un pequeño detalle al que debes prestar atención es la codificación de los mensajes. La función `readline()` devuelve una cadena de caracteres (`str`) y del mismo modo la función `print()` acepta una cadena. Sin embargo, los métodos `send()` y `recv()` del socket solo aceptan y devuelven secuencias de tipo `bytes`. Por eso es necesario convertir entre estos tipos con los métodos `encode()` y `decode()` respectivamente. Curiosamente el servidor no ha tenido que hacer estas conversiones porque también existe un método `bytes.upper()`, que es el que está usando la función `upper()` del Listado 14.1. Veremos más sobre esta cuestión en el capítulo 17.

Mientras tienes un cliente conectado, puedes ejecutar un nuevo cliente en otra consola, para así comprobar que efectivamente el servidor no acepta esta segunda conexión hasta que la primera termine.

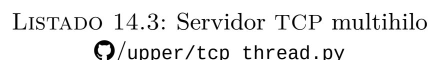
Por cierto, este cliente es tan simple que de hecho lo puedes sustituir directamente con `ncat`, y el funcionamiento es idéntico.

14.4. Servidor TCP multihilo

Un enfoque muy habitual para permitir que el servidor atienda múltiples clientes es crear un hilo (una instancia de la clase `threading.Thread`) en el que ejecutar la función `handle()` para cada nueva conexión. El código completo de este servidor multihilo (*threading server*) es muy similar al servidor iterativo. Lo puedes ver en el Listado 14.3.

```
import sys
import time
import socket
from threading import Thread

def upper(msg):
    time.sleep(1)
    return msg.upper()

def handle(sock, client, n):
    print(f"Client {n:>3} connected: {client}")
    while True:
        data = sock.recv(32)
        if not data:
            break
        sock.sendall(upper(data))

    sock.close()
    print(f"Client {n:>3} disconnected: {client}")

def main(port):
    sock = socket.socket()
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('', port))
    sock.listen(30)

    n = 0
    while True:
        conn, client = sock.accept()
        t = Thread(
            target=handle, args=(conn, client, n := n+1), daemon=True)
        t.start()

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")
```

Listado 14.3: Servidor TCP multihilo


/upper/tcp_thread.py

El programa no limita el número de hilos, aunque por supuesto el SO sí que lo hará. Si quieres detectar un exceso de hilos debes capturar la excepción

`RuntimeError`. Este límite tiene que ver principalmente con la cantidad de memoria que el SO asigna a cada hilo. Los hilos son muy ligeros porque todos ellos comparten el mismo espacio de direcciones (la misma memoria) que el proceso original y solo necesitan memoria adicional para la 'traza de pila' (*call stack*). La traza de pila almacena las llamadas a las funciones con sus argumentos, es decir, supone poca memoria, con lo que es posible crear varios miles de hilos sin problema.

Crear y destruir hilos es rápido, de modo que si la tarea que realiza el servidor es sencilla y corta, el servidor multihilo es bastante eficiente.

Python tiene una limitación técnica importante debida al GIL, que impide que varios hilos puedan ejecutar código Python al mismo tiempo, es decir, impide el paralelismo. Esto es un grave problema para programas intensivos en CPU, pero no lo es en absoluto para programas intensivos en E/S, que es precisamente lo que son los servidores.

14.5. Forzando los límites

Para dejar patente la mejora de rendimiento que supone el servidor multihilo, se proporciona un programa llamado `/upper/tcp_stress_client.py` que crea la cantidad de clientes que se le solicita y trata de conectarlos al servidor.

En paralelo, cada uno de esos clientes envía 8 palabras. En concreto cada uno envía *twenty tiny tigers take two taxis to town* en 8 mensajes distintos y al terminar, desconecta. Con el siguiente comando puedes probar el cliente de *stress* con el servidor iterativo. Verás que aparecen las 8 respuestas a los mensajes del 'cliente 0' (están identificados con un índice entre corchetes) antes de empezar a ver los mensajes del 'cliente 1'.

Cliente

```
$ ./tcp_stress_client.py 127.0.0.1 2000 10
- [  0] Reply: TWENTY
- [  0] Reply: TINY
...
- [  0] Reply: TO
- [  0] Reply: TOWN
- [  1] Reply: TWENTY
- [  1] Reply: TINY
- [  1] Reply: TIGERS
...
```

Servidor

```
$ ./tcp_server.py 2000
Client connected: ('127.0.0.1', 47900)
Client disconnected: ('127.0.0.1', 47900)
Client connected: ('127.0.0.1', 42700)
Client disconnected: ('127.0.0.1', 42700)
Client connected: ('127.0.0.1', 42714)
Client disconnected: ('127.0.0.1', 42714)
Client connected: ('127.0.0.1', 42724)
Client disconnected: ('127.0.0.1', 42724)
Client connected: ('127.0.0.1', 42738)
```

FIGURA 14.3: Servidor y cliente upper TCP

Para que los clientes no esperen indefinidamente, el programa fija un *timeout* para la conexión, que de expirar provoca que el cliente desista. Lo verás si lo ejecutas con demasiados clientes.

Si los 10 clientes consiguen conectarse, la ejecución completa necesitará unos 80 segundos (10 clientes × 8 mensajes × 1 segundo por mensaje).

Para ver la diferencia con el nuevo servidor, que es la intención, ejecuta ahora el mismo comando con el servidor multihilo de esta sección. En este caso la ejecución se completará en algo más de 8 segundos. Mientras el servidor sea capaz de crear tantos hilos como clientes se conecten a la vez, el tiempo total será aproximadamente esos mismos 8 segundos.

Te recomendamos que pruebes con distintas cantidades de clientes y observes el comportamiento del servidor. Llévalo al límite, prueba con cantidades de clientes cada vez mayores y trata de entender lo que ocurre en cada caso. Y por supuesto, no olvides probarlo también con los servidores que veremos en las siguientes secciones.

14.6. Servidor TCP multiproceso

El servidor multiproceso (*forking server*) es la alternativa mas frecuente al multihilo. El código es más complejo por la necesaria gestión de procesos. El programa en sí también es más costoso que el multihilo. Los procesos son mucho más pesados que los hilos. Implican una copia completa de la memoria del proceso padre. Por otro lado, los procesos son independientes unos de otros, de modo que los errores de uno no afectan a los demás y no hay ningún GIL que afecte a las tareas intensivas en CPU.

Respecto al código, la diferencia clave es que la función `handle()` se ejecuta ahora dentro de un nuevo proceso hijo. Aparte de eso, las funciones `main()`, `handle()` y `upper()` son prácticamente las mismas.

```
import sys
import os
import time
import socket

class ProcessThrottler(object):
    def __init__(self, max_procs=40):
        self.max_procs = max_procs
        self.procs = []

    def collect_children(self):
        # from socketserver module
        while self.procs:
            opts = os.WNOHANG if len(self.procs) < self.max_procs else 0
            pid, status = os.waitpid(0, opts)
```

```
            if not pid:
                break

            self.procs.remove(pid)

    def start_new_process(self, func, args):
        self.collect_children()
        pid = os.fork()
        if pid:
            self.procs.append(pid)
        else:
            func(*args)
            sys.exit()

def upper(msg):
    time.sleep(1)  # simulates a complex job
    return msg.upper()

def handle(sock, client, n):
    print(f"Client {n:>3} connected: {client}")
    while 1:
        data = sock.recv(32)
        if not data:
            break
        sock.sendall(upper(data))

    sock.close()
    print(f"Client {n:>3} disconnected: {client}")

def main(port):
    sock = socket.socket()
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('', port))
    sock.listen(5)

    throttler = ProcessThrottler()
    n = 0

    while 1:
        conn, client = sock.accept()
        throttler.start_new_process(
            handle, (conn, client, n := n+1))

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")
```

LISTADO 14.4: Servidor TCP multiproceso
/upper/tcp_fork.py

La creación de los procesos recae exclusivamente en la clase `ProcessThrottler`. La clase guarda una lista de los PID de los procesos que va crean-

do (`procs`). El método `start_new_process()` es el que ejecuta la llamada `os.fork()` e invoca la función indicada como argumento. El método `collect_children()`[1] se encarga de limitar la cantidad de procesos que se crearán, invocando `os.waitpid()` de forma bloqueante si se alcanza el máximo, evitando así que se cree un nuevo proceso hasta que no haya terminado uno de los existentes. Ese máximo se indica en el constructor con el parámetro `max_procs` (por defecto: 40).

Python dispone de algunas abstracciones de más alto nivel para la creación y gestión de procesos. Por ejemplo, la clase `multiprocessing.Process` ofrece un API similar a la de `threading.Thread` para manejar un único proceso, mientras que `multiprocessing.Pool` y `ProcessPoolExecutor` del módulo `concurrent.futures` permiten crear un *poll*[2] de procesos en los que se puede ejecutar la función proporcionada.

El Listado 14.5 muestra una versión del servidor multiproceso que utiliza la clase `multiprocessing.Pool`. Sin embargo, este programa funciona de un modo diferente. A diferencia del servidor anterior (Listado 14.4) que crea y destruye un proceso por cada conexión, en este los procesos existen desde que se instancia la clase `Pool` y se reutilizan una y otra vez para las nuevas conexiones. Por eso a esta técnica se le llama *preforking* y la ventaja es evidente: la aplicación se ahorra la sobrecarga de crear y destruir procesos continuamente, y puede ser una diferencia significativa si el servidor recibe muchas peticiones que requieren poco procesamiento. También tiene una desventaja: el consumo de memoria es el mismo aunque el servidor esté completamente ocioso.

```
import sys
import time
import socket
from multiprocessing import Pool

MAX_PROCS = 10

def upper(msg):
    time.sleep(1)  # simulates a complex job
    return msg.upper()

def handle(sock, client):
    print(f"Client connected: {client}")
    while 1:
        data = sock.recv(32)
        if not data:
            break
        sock.sendall(upper(data))
```

[1] Este código pertenece al módulo `socketserver` de la librería estándar.

[2] No conozco ninguna traducción razonable para esto y me niego a llamarlo *piscina de procesos*, lo siento.

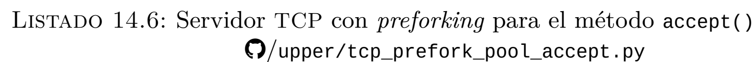
```
    sock.close()
    print(f"Client disconnected: {client}")

def main(port):
    sock = socket.socket()
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('', port))
    sock.listen(5)

    with Pool(MAX_PROCS) as pool:
        while 1:
            conn, client = sock.accept()
            pool.apply_async(handle, (conn, client))

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")
```

LISTADO 14.5: Servidor TCP con *preforking*
/upper/tcp_prefork_pool.py

Hay otro modo más agresivo de implementar *preforking* con TCP que consiste en compartir el *master socket* entre los procesos hijos y todos ellos invocan `accept()`. Eso es posible cuando el SO ofrece soporte específico (en el caso de Linux). Lo puedes ver en el Listado 14.6.

```
import sys
import os
import time
import socket
from multiprocessing import Pool

MAX_PROCS = 10

def upper(msg):
    time.sleep(1)  # simulates a complex job
    return msg.upper()

def handle(sock, client):
    print(f"Client connected: {client}, PID: {os.getpid()}")
    while 1:
        data = sock.recv(32)
        if not data:
            break
        sock.sendall(upper(data))

    sock.close()
    print(f"Client disconnected: {client}")

def worker(sock):
```

```
    try:
        while 1:
            conn, client = sock.accept()
            handle(conn, client)
    except KeyboardInterrupt:
        sys.exit(0)

def main(port):
    sock = socket.socket()
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('', port))
    sock.listen(5)

    with Pool(MAX_PROCS) as pool:
        pool.map(worker, [sock] * MAX_PROCS)

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")
```

Listado 14.6: Servidor TCP con *preforking* para el método `accept()`

/upper/tcp_prefork_pool_accept.py

14.7. Servidor UDP

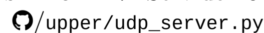
Es momento de ver las variantes UDP del servicio *upper*. El Listado 14.7 es una implementación funcional y correcta del servidor salvo porque, como de costumbre en el libro, obviamos el tratamiento de errores en favor de la legibilidad[3]. Hay unas cuántas diferencias evidentes y alguna no tan evidente que vale la pena destacar respecto al servidor TCP.

En la función `main()` hay un bucle que recibe mensajes, e imitando la estructura del servidor TCP, invoca la función `handle()` con cada mensaje[4] y el endpoint del cliente que lo ha enviado. La función `handle()` hace el trabajo y envía la respuesta. No hay un bucle es esa función, lo que deja patente que este es un servidor orientado a mensajes, no a conexiones.

Recuerda que con UDP la E/S parcial no es un problema. Si no hay pérdida de datos[5], cada llamada a `recvfrom()` devuelve un mensaje completo y cada llamada a `sendto()` envía el mensaje completo.

[3]No olvides tenerlo en cuenta si escribes código para producción.
[4]No con cada conexión, que no hay.
[5]Y recuerda que eso es factible con UDP.

```
import sys
import time
import socket

def upper(msg):
    time.sleep(1)  # simulates a complex job
    return msg.upper()

def handle(sock, msg, client, n):
    print(f"New request: {n} {client}")
    sock.sendto(upper(msg), client)

def main(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('', port))

    n = 0
    while 1:
        msg, client = sock.recvfrom(1024)
        handle(sock, msg, client, n := n+1)

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")
```

Listado 14.7: Servidor UDP
/upper/udp_server.py

El cliente UDP lo puedes ver en el Listado 14.8. Como en TCP, puedes sustituirlo por `ncat` si quieres.

```
import sys
from socket import socket, SOCK_DGRAM

def main(host, port):
    with socket(type=SOCK_DGRAM) as sock:
        server_endpoint = (host, port)

        while 1:
            data = sys.stdin.readline().strip().encode()
            if not data:
                break

            sock.sendto(data, server_endpoint)
            msg, _ = sock.recvfrom(1024)
            print("Reply is '{}'".format(msg.decode()))

if len(sys.argv) != 3:
    print("Usage: {0} <host> <port>".format(sys.argv[0]))
    sys.exit(1)

try:
```

```
    main(sys.argv[1], int(sys.argv[2]))
except KeyboardInterrupt:
    print("shut down")
```

Listado 14.8: Cliente UDP
/upper/udp_client.py

También tenemos /upper/udp_stress_client.py, el cliente de *stress* para UDP. Pruébalo con este servidor y con las variantes que vamos a ver a continuación para ver las diferencias. Aquí puedes ver cómo se comporta con el servidor UDP básico.

Cliente

```
$ ./udp_stress_client.py 127.0.0.1 2000 20
- [  0] Reply: TWENTY
- [  1] Reply: TWENTY
- [  2] Reply: TWENTY
- [  3] Reply: TWENTY
- [  4] Reply: TWENTY
- [  5] Reply: TWENTY
- [  6] Reply: TWENTY
```

Servidor

```
$ ./udp_server.py 2000
New request: 1 ('127.0.0.1', 40923)
New request: 2 ('127.0.0.1', 48631)
New request: 3 ('127.0.0.1', 40484)
New request: 4 ('127.0.0.1', 55612)
New request: 5 ('127.0.0.1', 51594)
New request: 6 ('127.0.0.1', 55917)
New request: 7 ('127.0.0.1', 55545)
```

Figura 14.4: Servidor y cliente upper TCP

A la vista de esta salida se puede apreciar esa diferencia no tan evidente de la que hablábamos antes. Resulta que este no es un servidor iterativo, al menos no en el sentido en el que lo es el TCP. Cada llamada a `recvfrom()` puede recoger un mensaje de un cliente diferente, lo que eventualmente permite progresar a todos los clientes que estén enviando solicitudes. Por eso, este es un servidor **concurrente** (ver Figura 14.5). Es importante recordar aquí la diferencia entre *concurrente* y *paralelo*. Este servidor no es paralelo, no puede ejecutar la función `handle()` exactamente al mismo tiempo para varios clientes, pero eso no es necesario para considerarlo concurrente. La tabla 14.1 resume este aspecto para los servidores que hemos visto. Es una consecuencia directa del diferente modo de trabajar de TCP y UDP.

Servidor	Concurrente	Paralelo
TCP iterativo	No	No
TCP multihilo	Sí	Sí
TCP multiproceso	Sí	Sí
UDP básico	Sí	No

Cuadro 14.1: Concurrencia y paralelismo de los servidores TCP y UDP

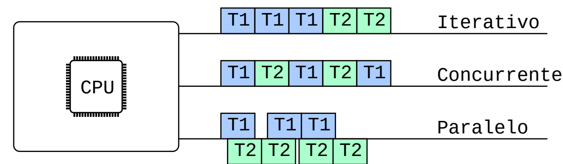


FIGURA 14.5: Comparación entre procesamiento iterativo, concurrente y paralelo

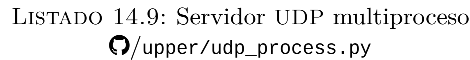
14.8. Servidor UDP multiproceso

Por supuesto es posible crear un servidor UDP que cree un proceso por cada mensaje recibido consiguiendo así un procesamiento paralelo. El Listado 14.9 muestra una posible implementación, en este caso utilizando la clase `multiprocessing.Process` en lugar de `fork()` directamente.

```
import sys
import time
import socket
import multiprocessing as mp

MAX_PROCS = 10

def start_new_process(func, args):
    for p in mp.active_children()[::-1][MAX_PROCS:]:
        p.join()

    ps = mp.Process(target=func, args=args)
    ps.start()

def upper(msg):
    time.sleep(1)  # simulates a complex job
    return msg.upper()

def handle(sock, msg, client, n):
    print(f"New request: {n} {client}")
    sock.sendto(upper(msg), client)
    sys.exit(0)

def main(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('', port))

    n = 0
    while 1:
        msg, client = sock.recvfrom(1024)
        start_new_process(handle, (sock, msg, client, n := n + 1))

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
```

```
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")
```

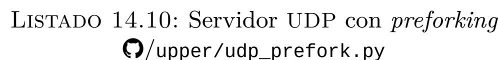
Listado 14.9: Servidor UDP multiproceso



Sin embargo, si la tarea que realiza en cada petición es simple, la sobrecarga relativa por la creación y destrucción de procesos es mucho más grave que en el caso de TCP. Aquí se está creando un proceso por ¡cada mensaje!, no por cada conexión. Lo que sí puede ser razonable es utilizar la técnica de *preforking* que evita completamente esa sobrecarga. Lo puedes ver en el Listado 14.10.

```
import sys
import time
import socket
from multiprocessing import Pool

MAX_PROCS = 20

def upper(msg):
    time.sleep(1)  # simulates a complex job
    return msg.upper()

def handle(sock, msg, client, n):
    print(f"New request: {n} {client}")
    sock.sendto(upper(msg), client)

def main(port):
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('', port))

    with Pool(MAX_PROCS) as pool:
        n = 0
        while 1:
            msg, client = sock.recvfrom(1024)
            pool.apply_async(handle, (sock, msg, client, n := n+1))

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    main(int(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")
```

Listado 14.10: Servidor UDP con *preforking*



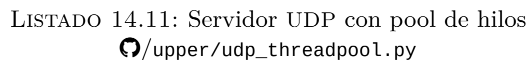
14.9. Servidor UDP multihilo

Implementar un servidor UDP creando un hilo cada vez que llega un mensaje para ejecutar en él la función `handle()` no es una buena idea. La sobrecarga de crear hilos no es despreciable, aunque no tan grave ni de lejos como con procesos. Pero el mayor problema no es ese. El problema es que en UDP todos los hilos tienen acceso al mismo socket, a diferencia de servidor TCP multihilo, en el que cada hilo opera su propio socket. Compartir el socket provoca condiciones de carrera, bloqueos, etc. Por eso, un servidor UDP multihilo de este tipo no es nada recomendable.

Sin embargo, es posible aplicar un enfoque distinto. Utilizar el paralelismo que proporcionan los hilos solo a la tarea del servidor, sin involucrar a los sockets. La idea es tener un hilo para recibir los peticiones (el principal), otro para enviar las respuestas (*responder*) y un pool para el procesamiento de los mensajes. Los mensajes recibidos se pasan como argumento a la función `handle()`, que se ejecuta en el pool. Las respuestas se envían al hilo *responder* por medio de una cola (`queue.Queue`). De ese modo, no hay ningún socket compartido, tenemos las ventajas, pero no los inconvenientes. Lo puedes ver en el Listado 14.11.

```
import sys
import time
import queue
from socket import socket, SOCK_DGRAM
from threading import Thread, current_thread
from concurrent.futures import ThreadPoolExecutor

MAX_THREADS = 20
output_queue = queue.Queue()

def upper(msg):
    time.sleep(1)  # Simulate heavy processing
    return msg.upper()

def handle(msg, client, n):
    print(f"Processing request {n} from {client}, thread {current_thread().name}")
    response = upper(msg)
    output_queue.put((response, client))

def responder(sock):
    while 1:
        response, client = output_queue.get()
        sock.sendto(response, client)

def main(port):
    sock = socket(type=SOCK_DGRAM)
    sock.bind(('', port))

    Thread(target=responder, args=(sock,), daemon=True).start()
```

```
    with ThreadPoolExecutor(max_workers=MAX_THREADS) as executor:
        n = 0
        while 1:
            msg, client = sock.recvfrom(1024)
            executor.submit(handle, msg, client, n := n+1)

if __name__ == '__main__':
    if len(sys.argv) != 2:
        print("Usage: {0} <port>".format(sys.argv[0]))
        sys.exit(1)

    try:
        main(int(sys.argv[1]))
    except KeyboardInterrupt:
        print("shut down")
```

Listado 14.11: Servidor UDP con pool de hilos

/upper/udp_threadpool.py

14.10. socketserver

El módulo `socketserver` de la librería estándar de Python ofrece clases y *mixins* para implementar servidores TCP y UDP de un modo muy sencillo. El Listado 14.12 es la versión equivalente al servidor TCP iterativo.

```
import sys
import os
import time
from socketserver import StreamRequestHandler, TCPServer

def upper(msg):
    time.sleep(1)  # simulates a complex job
    return msg.upper()

class Handler(StreamRequestHandler):
    def handle(self):
        print(f"Client connected: {self.client_address}")
        while 1:
            data = os.read(self.rfile.fileno(), 32)
            if not data:
                break

            self.wfile.write(upper(data))
        print(f"Client disconnected: {self.client_address}")

class CustomTCPServer(TCPServer):
    allow_reuse_address = True

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

server = CustomTCPServer(('', int(sys.argv[1])), Handler)
server.serve_forever()
```

Listado 14.12: Servidor TCP iterativo con `socketserver`
/upper/tcp_ss.py

La clase `TCPServer` permite crear un servidor TCP iterativo. Solo hay que pasarle como argumento el endpoint donde debe escuchar y una clase que herede de `StreamHandlerRequest` que proporcione a su vez un método `handle()` que es esencialmente equivalente a la función del mismo nombre de los ejemplos anteriores. Una diferencia muy evidente es que esa función no invoca `send()` y `recv()` sobre el socket conectado. En su lugar, invoca `read()` y `write()` sobre los *file objects* llamados `rfile` y `wfile`, pero que obviamente se refieren al socket.

Otra peculiaridad es que no se usa directamente la clase `TCPServer`. El programa define una clase `CustomTCPServer` que hereda de ella para redefinir el atributo `allow_reuse_address` como `True`. Eso es equivalente a la activación de la opción `SO_REUSEADDR`.

El módulo `socketserver` proporciona las clases `ForkingTCPServer` y `ThreadingTCPServer` alternativas a `TCPServer` para implementar directamente servidores multiproceso y multihilo, respectivamente. Y efectivamente, basta con cambiar la clase como puedes comprobar en /upper/tcp_ss_thread.py, que sería equivalente al Listado 14.3 y /upper/tcp_ss_fork.py que sería equivalente al Listado 14.4.

También proporciona la clase `UDPServer`, que puedes ver en uso en el Listado 14.13 y es el equivalente al servidor del Listado 14.7. Aquí tiene como manejador una clase que hereda de `DatagramRequestHandler` y especializa el método `handle()` que recibe una petición y envía la respuesta.

```
import sys
import time
from socketserver import DatagramRequestHandler, UDPServer

def upper(msg):
    time.sleep(1)  # simulates a complex job
    return msg.upper()

class Handler(DatagramRequestHandler):
    def handle(self):
        print(f"New request: {self.client_address}")
        msg = self.rfile.read()
        self.wfile.write(upper(msg))

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

server = UDPServer(('', int(sys.argv[1])), Handler)
server.serve_forever()
```

Listado 14.13: Servidor UDP con `socketserver`
/upper/udp_ss.py

También existen las clases `ForkingUDPServer` y `ThreadingUDPServer` para implementar servidores multiproceso y multihilo, respectivamente, pero que tienen los inconvenientes que ya hemos expuesto.

El módulo no tiene soporte para *preforking* en ninguna de las dos modalidades y tampoco son sencillas de implementar porque pasar las instancias de las clases involucradas a los subprocesos es complejo.

14.11. Operaciones bloqueantes

En § 6.8 vimos algunas de las consecuencias de que los sockets al igual que los archivos requieran de operaciones de E/S. Pero hay otra consecuencia muy importante que no habíamos abordado aún: Estas operaciones son **bloqueantes**. Igual que leer de la consola esperando que el usuario pulse una tecla puede conllevar que el proceso quede bloqueado indefinidamente, lo mismo ocurre con el método `recv()`. En estos casos, el SO bloquea el proceso que invoca estas funciones hasta que lleguen los datos esperados. Pero no es algo que afecte solo a `recv()`; también afecta a `accept()` y por supuesto a `recvfrom()`, pues también en esos casos el proceso espera a la llegada de mensajes procedentes de la red para continuar su ejecución.

Es bastante intuitivo pensar que estas operaciones de «lectura» sean bloqueantes, pero resulta que también lo son las operaciones de «escritura» como `send()`, `sendto()` o `connect()`. Aunque es menos habitual que el envío de un mensaje conlleve el bloqueo de forma indefinida[6], también son operaciones de E/S y por eso el SO puede bloquear el proceso a la espera de que la operación se complete; en este caso, que los datos sean enviados, o almacenados en el buffer correspondiente, antes de continuar la ejecución del programa.

Que estas operaciones sean bloqueantes puede sonar como un problema que hay que resolver, pero es lógico que sea así e incluso resulta conveniente. Por norma general es **más sencillo** escribir —y más eficiente ejecutar— un programa de comunicaciones, o en general intensivo en E/S, si se comporta de este modo.

14.12. Alternativas a la E/S bloqueante

Pese a lo dicho, hay algunas situaciones en las que el programa podría necesitar aprovechar los tiempos «muertos» mientras el SO espera a que

[6]Volveremos sobre esto cuando hablemos de control de flujo en el capítulo 11

estas operaciones terminen, o quizá quiera ejecutar otras operaciones de E/S. Recuerda que cuando un proceso está bloqueado, la CPU no está ociosa; el planificador del SO pone otro proceso en ejecución, pero ciertamente, tu aplicación está parada.

Normalmente hay tres opciones para aprovechar los bloqueos: operaciones no bloqueantes, timeouts y programación asíncrona. Aunque esto es potencialmente aplicable a cualquier operación de E/S, lógicamente aquí vamos a hablar de sockets. Veámoslas brevemente:

Operaciones no bloqueantes

Los sockets están por defecto en modo bloqueante y por eso se comportan como hemos visto. Se puede conseguir que un socket concreto tenga un comportamiento no bloqueante invocando su método `setblocking(False)`. A partir de ese momento, si por ejemplo se invoca `recv()` y no hay datos disponibles, eleva inmediatamente una excepción `BlockingIOError` y es responsabilidad del programador capturar la excepción e intentar la recepción de nuevo más tarde si así lo estima oportuno. No es difícil imaginar cómo esto nos puede complicar rápidamente las cosas... Además es necesario señalar que el comportamiento de los sockets no bloqueantes puede diferir entre plataformas, algo que dificulta la escritura de código portable, otra buena razón para evitarlos si no está realmente justificado.

Timeouts

Hay otras situaciones en las que las operaciones bloqueantes son adecuadas, pero no es aceptable que el proceso quede bloqueado indefinidamente. En esta situación se puede usar un timeout, es decir, definir un tiempo máximo que de alcanzarse provoca la interrupción de la operación y eleva una excepción `TimeoutError` que avisa de la situación devolviendo así el control al programa. El método `socket.settimeout()` permite fijar ese límite (expresado en segundos).

Este método acepta dos valores especiales. Si se llama con `None`, elimina el timeout y el socket queda en modo bloqueante, pero si se llama con `0`, el socket cambia a modo no bloqueante, es decir, es como invocar `setblocking(False)`.

Entrada/salida asíncrona

El SO ofrece algunas llamadas al sistema para gestionar operaciones de E/S de forma asíncrona. La más básica y rudimentaria, que veremos a continuación, es `select()`, que permite esperar a que uno o más descriptores

de archivo—en el sentido amplio del concepto— estén listos para leer o escribir.

Aparte de eso, Python proporciona un modelo de programación asíncrona basado en corutinas que funcionan en un único hilo de ejecución. La programación asíncrona es potente y compleja, y requiere del programador un enfoque muy diferente a la programación secuencial habitual.

Veremos una pequeña introducción a estas dos soluciones en las siguientes secciones.

14.13. Servidor TCP asíncrono con `select()`

Con `select()` se puede resolver de otro modo el problema del servidor iterativo: los dos puntos de bloqueo en `accept()` y `recv()`.

La función `select()`, que en Python se encuentra en el módulo del mismo nombre, permite manejar una serie de descriptores de archivo cuyo estado puede cambiar en cualquier momento. El prototipo de la función es este:

```
read_ready, write_ready, error_ready = select.select(rlist, wlist, xlist, timeout=None)
```

Los parámetros `rlist`, `wlist` y `xlist` son listas de descriptores de archivo que quieres que `select()` monitorice. En la lista `rlist` pones descriptores de los que quieres leer, en `wlist` de los que quieres escribir y en `xlist` de los que quieres comprobar si han tenido algún error. En los sistemas POSIX los descriptores de archivo incluyen muchas otras cosas como tuberías, dispositivos, terminales, y por supuesto, también sockets.

El valor de retorno es una tupla con tres listas, que contienen los descriptores que efectivamente están listos para ser leídos, escritos o han tenido un error, es decir, subconjuntos de los argumentos de la función.

La function `select()` bloquea el proceso hasta que ocurre algo en algún descriptor de cualquiera de las listas. Eso proporciona un único punto de bloqueo en el programa y por tanto, permite manejar múltiples entradas asíncronas simultáneas sin necesidad de hilos u otros mecanismos para conseguir paralelismo.

Puedes ver el código del servidor completo en el Listado 14.14. Como hay dos tipos de sockets: el máster y los conectados a los clientes, el programa debe hacer un tratamiento diferente para cada uno, que corresponde con los métodos `master_handler()` y `child_handler()` respectivamente. La lista `socks` contiene inicialmente el socket maestro, y conforme el servidor acepta conexiones, los nuevos sockets se añaden a esa misma lista. La lista `socks`

se pasa como primer argumento a `select()` porque en este programa tan sencillo solo nos interesa evitar el bloqueo relacionado con las lecturas.

```
import sys
import socket
import time
import select
from utils import show_select_status

def upper(msg):
    time.sleep(1)  # simulates a complex job
    return msg.upper()

class Server:
    def __init__(self, port):
        self.master = socket.socket()
        self.master.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
        self.master.bind(('', port))
        self.master.listen(5)
        self.socks = [self.master]

    def master_handler(self):
        conn, client = self.master.accept()
        self.socks.append(conn)
        print(f"- Client connected: {client}, Total {len(self.socks)} sockets")

    def child_handler(self, conn):
        data = conn.recv(32)
        if not data:
            self.socks.remove(conn)
            conn.close()
            return

        conn.sendall(upper(data))

    def run(self):
        while 1:
            read_ready = select.select(self.socks, [], [])[0]
            show_select_status(self.socks, read_ready)
            for s in read_ready:
                if s == self.master:
                    self.master_handler()
                else:
                    self.child_handler(s)

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print("Usage: {0} <port>".format(sys.argv[0]))
        exit(1)

    try:
        server = Server(int(sys.argv[1]))
        server.run()
    except KeyboardInterrupt:
        print("\nServer shutting down.")
        exit(0)
```

Listado 14.14: Servidor TCP con select()
/upper/tcp_select.py

Si pruebas este servidor con el cliente de stress podrás ver que su comportamiento es, salvando las diferencias, similar al del servidor UDP básico. Ofrece concurrencia: todos los clientes progresan, pero no paralelismo: la función `upper()` no se ejecuta a la vez para varios clientes.

```
$ ./tcp_stress_client.py 127.0.0.1 2000 4
- [  0] Reply: TWENTY
- [  1] Reply: TWENTY
- [  2] Reply: TWENTY
- [  3] Reply: TWENTY
- [  0] Reply: TINY
```

La librería estándar de Python incluye un módulo llamado `selectors` con un nivel de abstracción algo más alto para conseguir lo mismo. Puedes ver y probar un servidor para el servicio upper con `selectors` en el archivo /upper/tcp_selectors.py. Se basa en la creación de un objeto de clase `DefaultSelector` en el que se registran manejadores para un descriptor de archivo y un motivo específico.

```
selector = selectors.DefaultSelector()
selector.register(f, selectors.EVENT_READ, handler)
```

Esto le indica al `selector` que si hay datos disponibles para leer en el descriptor de fichero `f`, debe invocar la función `handler`.

14.14. Servidor TCP asíncrono con asyncio

Python, como muchos otros lenguajes modernos, ofrece una solución más general: concurrencia basada en *corutinas* o tareas cooperativas. Con este modelo, el código se organiza de un modo similar a un programa secuencial convencional, pero cuando se invoca una primitiva de E/S bloqueante, el proceso no se bloquea, sino que suspende la ejecución de esa función y cede el control a otra. De ese modo, el proceso puede aprovechar los tiempos «muertos» sin bloqueos y sin necesidad de crear procesos u hilos adicionales.

El programador debe indicar explícitamente qué funciones son corutinas con la palabra clave `async`, y debe llamarlas con `await`. Las corutinas pueden a su vez invocar otras corutinas, mientras que la función más externa la debe gestionar un bucle de eventos. Este bucle es el que se encarga de ejecutar las corutinas, suspenderlas y reanudarlas cuando corresponda. El bucle de eventos y otras muchas utilerías las ofrece el módulo `asyncio`, que literalmente es la abreviatura de *asynchronous I/O*.

Existen dos modalidades para implementar un servidor TCP con `asyncio`:

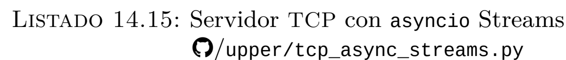
- *Streams*
- *Transports and Protocols*

Streams es la de más alto nivel y cómo su nombre indica se basa en el manejo de dos flujos: uno de lectura y otro de escritura. El Listado 14.15 aplica ese enfoque. Se proporciona una función —que aquí hemos llamado `handle()`— que recibe los flujos (`reader` y `writer`). Para crear el servidor se utiliza `asyncio.start_server()`. Fíjate que todas las funciones bloqueantes (o las que las llaman) son asíncronas (declaradas `async`) y son invocadas con `await`. Aunque se parece bastante al servidor TCP iterativo, este puede intercalar la recepción de mensajes, que se hace en `reader.read`, con la aceptación de nuevas conexiones.

Con `async` la función que simula un `upper()` costoso no puede ser una llamada a `sleep()` porque el bucle de eventos utilizaría esa pausa para ejecutar otras corutinas, de modo que no conseguiría su propósito de simular una tarea costosa. En su lugar se ha añadido un bucle vacío que dura 1 segundo —función `fake_heavy_upper()`.

```
import sys
import asyncio
import time

async def upper(msg):
    def fake_heavy_upper(msg):
        start = time.time()
        while time.time() - start < 1:
            pass
        return msg.upper()

    return await asyncio.to_thread(fake_heavy_upper, msg)

async def handle(reader, writer):
    peername = writer.get_extra_info('peername')
    print(f"Client connected: {peername}")

    try:
        while 1:
            data = await reader.read(32)
            if not data:
                break
            writer.write(await upper(data))
            await writer.drain()

    except asyncio.CancelledError:
        pass
    finally:
        print(f"Client disconnected: {peername}")
        writer.close()
```

```
        await writer.wait_closed()

async def main(port):
    server = await asyncio.start_server(handle, '', port)

    async with server:
        await server.serve_forever()

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    asyncio.run(main(sys.argv[1]))
except KeyboardInterrupt:
    print("shut down")
```


LISTADO 14.15: Servidor TCP con asyncio Streams
/upper/tcp_async_streams.py

La segunda modalidad para implementar un servidor es *Transports and Protocols*, que es de más bajo nivel que *Streams*. El *transporte* representa el medio para transmitir los mensajes (el socket), mientras que el *protocolo* dice qué información hay que transmitir y cómo se debe interpretar.

En el caso más sencillo consiste en implementar una clase que hereda de `asyncio.Protocol` y encapsula el comportamiento del protocolo de aplicación. Se consigue especializando los métodos de la clase `Protocol`: `connection_made()`, `data_received()`, etc., que son invocados cuando ocurren esos eventos. A esto se le llama programación basada en *hooks*. El Listado 14.16 muestra el servidor TCP con esta modalidad.

```
import sys
import asyncio
import time

async def upper(msg):
    def fake_heavy_upper(msg):
        start = time.time()
        while time.time() - start < 1:
            pass
        return msg.upper()

    return await asyncio.to_thread(fake_heavy_upper, msg)

class UpperProtocol(asyncio.Protocol):
    def connection_made(self, transport):
        self.peername = transport.get_extra_info("peername")
        print(f"Client connected: {self.peername}")
        self.transport = transport

    def data_received(self, data):
        message = data.decode()
        loop = asyncio.get_running_loop()
```

```
        loop.create_task(self.handle_message(message))

    async def handle_message(self, message):
        reply = await upper(message)
        self.transport.write(reply.encode())

    def connection_lost(self, exc):
        print(f"Client disconnected: {self.peername}")
        self.transport.close()

async def main(port):
    loop = asyncio.get_running_loop()
    server = await loop.create_server(UpperProtocol, '', port)

    async with server:
        await server.serve_forever()

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    asyncio.run(main(int(sys.argv[1])))
except KeyboardInterrupt:
    print('shut down')
```

Listado 14.16: Servidor TCP con `asyncio` Transport and Protocols
`/upper/tcp_async_protocol.py`

14.15. Cliente TCP asíncrono con asyncio

Una posible versión asíncrona del cliente TCP que hemos visto en 14.3 podría ser el del Listado 14.17. Es destacable el uso del método `run_in_executor()` que permite ejecutar la función `stdin.readline()`, que es bloqueante, en un hilo independiente para evitar que bloquee el bucle de eventos. En realidad aunque el envío y la recepción son concurrentes, no aporta ninguna ventaja en este caso y no tiene demasiado interés práctico. Evitar el bloqueo de la lectura desde la consola y desde el socket no va a mejorar el rendimiento del cliente, seguirá pudiendo enviar un mensaje cada segundo como máximo.

```
import sys
import asyncio

def readline():
    return sys.stdin.readline().strip().encode()

async def main(host, port):
    reader, writer = await asyncio.open_connection(host, port)

    try:
        while True:
```

```
            data = await asyncio.get_event_loop().run_in_executor(None, readline)
            if not data:
                break

            writer.write(data)
            await writer.drain()

            msg = b''
            while len(msg) < len(data):
                chunk = await reader.read(32)
                if not chunk:
                    break
                msg += chunk

            print(f"Reply is '{msg.decode()}'")

    except asyncio.CancelledError:
        pass
    finally:
        writer.close()
        await writer.wait_closed()

if len(sys.argv) != 3:
    print("Usage: {0} <host> <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    asyncio.run(main(sys.argv[1], int(sys.argv[2])))
except KeyboardInterrupt:
    print("shut down")
```

LISTADO 14.17: Cliente TCP con `asyncio`
`/upper/tcp_client_async.py`

Eso no significa que no sea interesante en otros casos. Si el cliente tiene otras cosas que hacer, puede suponer una importante ventaja. Un ejemplo práctico de esto es el cliente de *stress* que hemos estado usando a lo largo del capítulo: `/upper/tcp_stress_client.py`. Este cliente envía mensajes a varios servidores, de modo que mientras espera la respuesta de uno, puede estar enviando mensajes a otros. Te recomendamos estudiar ese programa y compararlo con el cliente 14.17 que acabamos de ver.

14.16. Servidor UDP asíncrono con `asyncio`

También puedes implementar un servidor UDP con la modalidad *Transports and Protocols* (ver Listado 14.18), heredando de `asyncio.DatagramProtocol`. Sin embargo, no es posible implementar un servidor UDP con *Streams* porque como vimos antes, está diseñado solo para transferencia de datos orientada a flujo.

```
import sys
import asyncio
import time

async def upper(msg):
    def fake_heavy_upper(msg):
        start = time.time()
        while time.time() - start < 1:
            pass
        return msg.upper()

    return await asyncio.to_thread(fake_heavy_upper, msg)

class UpperUDPProtocol(asyncio.DatagramProtocol):
    def connection_made(self, transport):
        self.transport = transport

    def datagram_received(self, data, addr):
        print(f"New request: {addr}")
        asyncio.create_task(self.handle_request(data, addr))

    async def handle_request(self, data, addr):
        response = await upper(data.decode())
        self.transport.sendto(response.encode(), addr)

    def error_received(self, exc):
        print(f"Comm error: {exc}")

async def main(port):
    loop = asyncio.get_running_loop()

    transport, _ = await loop.create_datagram_endpoint(
        lambda: UpperUDPProtocol(),
        local_addr=('', port)
    )

    try:
        await asyncio.sleep(3600)
    except asyncio.CancelledError:
        pass
    finally:
        transport.close()

if len(sys.argv) != 2:
    print("Usage: {0} <port>".format(sys.argv[0]))
    sys.exit(1)

try:
    asyncio.run(main(int(sys.argv[1])))
except KeyboardInterrupt:
    print("shut down")
```

LISTADO 14.18: Servidor UDP con `asyncio` *Transport and Protocols*
/upper/udp_async_protocol.py

Y ¿qué más?

Hemos visto las posibilidades *nativas* de concurrencia que ofrece un servidor UDP debido su naturaleza orientada a mensajes individuales. Al mismo tiempo eso implica limitaciones debido a que es necesario realizar todas sus operaciones sobre un único socket, por la contención que supone protoger el socket adecuadamente. A pesar de ello, hemos visto técnicas como el *pool* de hilos o el *preforking* que permiten un alto nivel de paralelismo evitando el acceso compartido. Por contra, TCP ofrece más opciones para hacer servidores de alto rendimiento al estar basado en un modelo de conexiones que permite que cada una de esas conexiones disponga de su propio socket, con el asilamiento que ello conlleva. Sea como sea, no se deben despreciar las posibilidades de concurrencia que ofrece la gestión de E/S asíncrona, ya sea con `select` y sus derivados, o el soporte integrado en el lenguaje gracias a `asyncio`.

Aunque solo hemos visto una pincelada de cada una de estas técnicas, es sencillo imaginar las diversas opciones disponibles, tanto con Python como con otros lenguajes. Te invitamos a experimentar por ti mismo y aplicar estas y otras técnicas más avanzadas a tus propios proyectos.