Capítulo 15

Publicador-Suscriptor

Al terminar este capítulo, entenderás:

- Qué es el modelo de interacción publicador-suscriptor.
- Cómo implementar un chat con este modelo de interacción. tanto con UDP como con TCP.
- Qué es y qué ofrece el protocolo MQTT para implementar este tipo de comunicación.

El segundo modelo de interacción más habitual, por detrás del cliente-servidor, es publicador-suscriptor[1]. En lugar de aplicar una interacción basada en petición-respuesta, este modelo asume existen dos tipos de participantes: los *publicadores* envían mensajes a un espacio común (el *canal*) y los *suscriptores* reciben los mensajes que les interesan en función de a qué estén suscritos. El matiz clave es que esos mensajes se envían sin esperar una respuesta. Si los publicadores esperan algún tipo de información de vuelva, lo harán también con un rol de suscriptores, pero no como parte de la misma interacción, por lo que no podemos considerar que se trate de respuestas.

El modelo se caracteriza por el uso de un componente intermedio llamado **broker** que es quien se encarga de recibir los mensajes que proceden de los publicadores y hacerlos llegar a los suscriptores. Lógicamente, no tiene sentido llamar *peticiones* a estos mensajes porque no buscan un resultado. En su lugar se les llama *publicaciones* o más comúnmente *eventos*.

El hecho de que haya mensajes solo en un sentido y que todos ellos pasen por el *broker* desacopla completamente a publicadores y suscriptores. Lo único que deben conocer todos los participantes es el endpoint del broker.

[1] Abreviado a menudo como «pubsub».

La otra peculiaridad de este modelo es que puede haber múltiples suscriptores para un mismo canal. El broker es el encargado de realizar y distribuir las copias necesarias para entregar a cada suscriptor..

15.1. Chat UDP para dos

Para entender cómo funciona este modelo, vamos a desarrollar paso a paso un ejemplo práctico típico: un *chat*. Es un programa sencillo que permite a varias personas enviar mensajes a una *sala* de modo que todos los usuarios presentes en la sala pueden verlos.

Empezaremos desde una versión extremadamente simple con solo dos participantes. Después iremos añadiendo funcionalidad y resolviendo los problemas que vayan apareciendo. Para facilitar el desarrollo inicialmente le vamos a dar la estructura cliente-servidor, pero conforme evolucione, incorporaremos un broker y se convertirá en publicación-suscripción. También por simplicidad, empezaremos con una versión UDP.

15.1.1. Paso 1: Mensaje unidireccional

En este primer paso los objetivos son crear:

- Un servidor que recibe un único mensaje, lo imprime en consola y termina.
- Un cliente que envía la cadena `hello` al servidor y termina.

Servidor

Las tareas que realiza el servidor son muy simples:

1. Crea un socket UDP.
2. Vincula el socket a un puerto libre.
3. Espera un datagrama que contiene un mensaje de texto.
4. Imprime el mensaje en consola.

Estas tareas corresponden línea a línea con el Listado 15.1.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('', 12345))
message, client = sock.recvfrom(1024)
print(message.decode(), client)
sock.close()
```

LISTADO 15.1: Servidor de chat UDP básico
/chat-udp/server1.py

No hay mucho que comentar. Es un programa aún más simple de los que ya habíamos visto. Recuerda que al ser UDP, el argumento de `recvfrom()` limita el tamaño de los mensajes que podrán intercambiar los usuarios, pero para un chat sencillo como este, 1024 bytes parece más que suficiente.

Para ejecutarlo simplemente escribe el siguiente comando en la consola:

```
$ python3 chat-udp/server1.py
```

El programa (y la consola) queda inmediatamente bloqueado, en concreto en la invocación del método `recvfrom()`, a la espera de recibir el mensaje de un cliente.

Cliente

Antes de escribir el programa cliente en Python es buena idea probar que el servidor funciona como debe. Y para eso puedes utilizar `ncat` (ver 6.12).

Con el siguiente comando puedes ejecutar un cliente UDP que envía la cadena `hola` al servidor que está escuchando en el puerto 12345 (el de acabas de poner en marcha). En este instante el servidor debería imprimir el saludo y terminar.

```
$ echo hola | ncat --udp --send-only 127.0.0.1 12345
```

Ahora que has comprobado que el servidor funciona, puedes pasar a escribir un cliente que imite esta misma funcionalidad. También es muy simple: crea el socket UDP, envía un mensaje y termina. Lo puede ver en el Listado 15.2.

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto("hola".encode(), ('127.0.0.1', 12345))
sock.close()
```

LISTADO 15.2: Cliente de chat UDP básico
`chat-udp/client1.py`

Como puedes comprobar por el segundo parámetro de la función `sendto()`, el cliente envía el mensaje al endpoint `127.0.0.1:12345`, es decir, estamos asumiendo que vas a ejecutar el servidor y el cliente en la misma máquina, algo recomendable al menos mientras escribes y pruebas el programa.

Lo puedes ejecutar como sigue, y el efecto debería ser el mismo que con `ncat`.

```
$ python3 chat-udp/client.py
```

15.1.2. Paso 2: Devuelve el saludo

El servidor del paso anterior sólo imprime el mensaje recibido. Con un pequeño cambio podrá devolver el saludo al cliente.

Utilizando la dirección del cliente, que se obtiene como valor de retorno del método `recvfrom()`, la aplicación puede a su vez utilizar el método `sendto()` y enviar un mensaje de vuelta (Listado 15.3).

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.bind(('', 12345))
message, peer = sock.recvfrom(1024)
print(message.decode(), peer)
sock.sendto("qué tal?".encode(), peer)
sock.close()
```

LISTADO 15.3: Servidor de chat UDP con respuesta
/chat-udp/server2.py

El cliente del paso anterior terminaba inmediatamente después de enviar su mensaje. Ahora debe esperar para recibir el mensaje del servidor. Como es lógico, basta con imitar lo que hace el servidor: usar el método `recvfrom()` (Listado 15.4).

```
import socket

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
sock.sendto("hola".encode(), ('127.0.0.1', 12345))
message, peer = sock.recvfrom(1024)
print("{} from {}".format(message.decode(), peer))
sock.close()
```

LISTADO 15.4: Cliente de chat UDP con respuesta
/chat-udp/client2.py

15.1.3. Paso 3: Libertad de expresión

Con este paso los usuarios que ejecutan cliente y servidor tendrán realmente la posibilidad de conversar. En lugar de enviar una cadena literal, ambos programas van a leer de consola lo que el usuario teclee y lo van a enviar hacia el interlocutor. La conversación se mantendrá hasta que cualquiera de ellos envíe la cadena `bye`. El Listado 15.5 muestra el código completo del servidor después de este cambio:

```
import socket
QUIT = b"bye"

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

```
sock.bind(('', 12345))

while 1:
    message_in, peer = sock.recvfrom(1024)
    print(message_in.decode())

    if message_in == QUIT:
        break

    message_out = input().encode()
    sock.sendto(message_out, peer)

    if message_out == QUIT:
        break

sock.close()
```

Listado 15.5: Servidor de chat UDP por turnos
/chat-udp/server3.py

La diferencia principal respecto a las versiones anteriores es el bucle `while`. Este bucle termina tanto si el usuario local introduce la cadena `bye` como si es recibida a través del socket. Para leer una cadena de texto de la consola se utiliza la función `input()`. El código del cliente (Listado 15.6) es muy similar.

```
import socket
QUIT = b"bye"

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server = ('', 12345)

while 1:
    message_out = input().encode()
    sock.sendto(message_out, server)

    if message_out == QUIT:
        break

    message_in, peer = sock.recvfrom(1024)
    print(message_in.decode())

    if message_in == QUIT:
        break
```

Listado 15.6: Cliente de chat UDP por turnos
/chat-udp/client3.py

Únicamente hay dos diferencias entre cliente y servidor: sólo el servidor ejecuta `bind()` y es el que empieza recibiendo un mensaje en el socket, mientras el cliente empieza leyendo de teclado. En UDP, el cliente tiene que ser el primero en enviar un mensaje. Como no hay conexión previa, el

servidor no puede saber quién es el cliente (su endpoint) hasta que reciba algo.

Esta versión tiene un problema grave en cuanto a la interacción entre los usuarios. Tanto el cliente como el servidor tienen dos puntos diferentes en los que el programa queda bloqueado: la función `input()` para leer de consola y el método `recvfrom()` para leer del socket. Es el mismo problema que ya vimos en los servidores del capítulo 14: la imposibilidad de atender múltiples entradas asíncronas simultaneamente[2]. Pero en este caso concreto no tiene que ver con concurrencia. Aquí implica que los usuarios han de esperar a que su interlocutor envíe un mensaje antes de poder escribir de nuevo. O dicho de otro modo, los usuarios no pueden enviar 2 o más mensajes consecutivos, que es lo que se espera de un chat.

Y con esto además puedes ver que la situación no encaja con un protocolo petición-respuesta. No queremos que haya una correspondencia uno a uno entre los mensajes que envía el cliente y los que envía el servidor. Los mensajes que envía el servidor no son respuestas a solicitudes del cliente, cada usuario debe poder enviar tantos mensajes como quiera en cualquier momento.

15.1.4. Paso 4: Habla cuando quieras

Para resolver el problema de la E/S asíncrona vamos a utilizar hilos, que es la más sencilla al menos a nivel de código. Atenderemos la entrada procedente de la consola en un hilo adicional (el envío), mientras que el hilo principal se encargará de atender el socket (la recepción). El código del servidor aparece en el Listado 15.7.

```
4  import socket
5  import _thread
6  server = ('', 12345)
7  QUIT = b"bye"
8
9  class Chat:
10     def __init__(self, sock, peer):
11         self.sock = sock
12         self.peer = peer
13
14     def run(self):
15         _thread.start_new_thread(self.sending, ())
16         self.receiving()
17         self.sock.close()
18
19     def sending(self):
20         while 1:
21             message = input().encode()
```

[2] a menos que se utilicen hilos, procesos o E/S asíncrona.

```
22              self.sock.sendto(message, self.peer)
23
24              if message == QUIT:
25                  break
26
27      def receiving(self):
28          while 1:
29              message, peer = self.sock.recvfrom(1024)
30              print(message.decode())
31
32              if message == QUIT:
33                  self.sock.sendto(QUIT, self.peer)
34                  break
35
36  if __name__ == '__main__':
37      sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
38      sock.bind(server)
39      message, client = sock.recvfrom(0, socket.MSG_PEEK)
40      Chat(sock, client).run()
```

Listado 15.7: Servidor de chat UDP simultaneo
/chat-udp/server4.py

El programa está compuesto por una clase `Chat` con tres métodos, aparte del constructor. El método `sending()` se ocupa de leer líneas de texto de la consola y enviarlas a través del socket. El método `receiving()` lee líneas de texto del socket y las imprime en la consola. En ambos casos, si el mensaje leído o recibido es `bye`, la función termina. En el caso de la función `receiving()` además devuelve el mensaje para que el hilo de recepción del otro extremo también termine. El método `run()` crea y arranca el hilo para la tarea de envío y ejecuta la tarea de recepción.

En cuanto a la función principal, la llamada a `recvfrom()` (**línea 37**) se utiliza únicamente para obtener el endpoint del cliente, pero sin consumir nada del buffer del socket gracias al flag MSG_PEEK. La **línea 38** crea la instancia de la clase `Chat` pasando el socket y el endpoint del cliente como parámetros.

El cliente solo difiere en la creación del socket de modo que se puede reutilizar la clase `Chat` del servidor tal cual. Simplemente crea el socket y la instancia de `Chat`, a la que le pasa el socket y el endpoint del servidor. Puedes verlo en el Listado 15.8.

```
import socket
from server4 import Chat, server

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
Chat(sock, server).run()
```

Listado 15.8: Cliente de chat UDP simultáneo
/chat-udp/client4.py

Sigue habiendo un pequeño problema estético. Como el punto de entrada del usuario y el lugar donde se escriben los mensajes que se reciben es el mismo (la salida estándar) es fácil que se mezclen, dificultando la lectura de la salida. Para solucionarlo habría que hacer que las tareas de recepción y envío escribieran en partes diferentes de la pantalla, o quizá construir un pequeño GUI. Sin embargo, ésta es una cuestión al margen de la finalidad de este ejemplo.

15.1.5. Paso 5: Todo en uno

El servidor y el cliente del paso anterior utilizan la misma clase `Chat` para resolver la mayor parte del problema. Lo único diferente entre servidor y cliente es el código de la función principal (lo que está fuera de clase `Chat`). Eso significa que es posible crear un único programa que se comporte como servidor o cliente en función de un parámetro de línea de comandos. Puedes verlo en el Listado 15.9.

```
import sys
import socket
from threading import Thread

SERVER = ('', 12345)
QUIT = b'bye'

class Chat:
    def __init__(self, sock, peer):
        self.sock = sock
        self.peer = peer

    def run(self):
        sender_thread = Thread(target=self.sending, daemon=True)
        sender_thread.start()
        self.receiving()
        sender_thread.join()
        self.sock.close()

    def sending(self):
        while True:
            message = input().encode()
            self.sock.sendto(message, self.peer)
            if message == QUIT:
                break

    def receiving(self):
        while 1:
            message, _ = self.sock.recvfrom(1024)
            print("other> {}".format(message.decode()))
            if message == QUIT:
                self.sock.sendto(QUIT, self.peer)
                break

if __name__ == '__main__':
```

```
    if len(sys.argv) != 2:
        print("usage: %s [--server|--client]".format(sys.argv[0]))
        sys.exit()

    mode = sys.argv[1]
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    if mode == '--server':
        sock.bind(SERVER)
        message, client = sock.recvfrom(0, socket.MSG_PEEK)
        Chat(sock, client).run()
    else:
        Chat(sock, SERVER).run()
```

LISTADO 15.9: Chat UDP multihilo (servidor y cliente)
/chat-udp/chat-thread.py

15.2. Chat asíncrono con `select()`

Como alternativa a los hilos, veamos como implementar el chat anterior con `select()`. Como hemos visto, las dos entradas asíncronas a vigilar son la consola y el socket, así que le podemos pasar esos descriptores a `select()`. Es un uso más simple que el del servidor TCP de $14.13, pero que sigue la misma idea: proporcionar un único punto de bloqueo que permite atender cualquiera de las dos situaciones. La llamada sería algo como:

```
while 1:
    ready = select([sys.stdin, sock], [], [])[0]
    for fd in ready:
        if fd == sock:
            # leer del socket
        else:
            # leer de consola
```

El bucle `for` es necesario porque ambos descriptores podrían estar listos para leer al mismo tiempo, o muy próximos en el tiempo. Para leer se va a seguir usando las funciones `recvfrom()` e `input()`. Recuerda que, después de que `select()` haya confirmado que hay datos disponibles, tienes garantía de que esas funciones no van a bloquear el programa. Puedes ver el programa completo (que se sigue pudiendo usar como cliente o servidor) en el Listado 15.10.

```
import sys
import socket
from select import select
SERVER = ('', 12345)
QUIT = b'bye'

class Chat:
    def __init__(self, sock, peer):
        self.sock = sock
```

```
        self.peer = peer

    def run(self):
        fds = [sys.stdin, self.sock]
        while 1:
            ready = select(fds, [], [])[0]
            for fd in ready:
                if self.sock in ready:
                    msg = self.receiving()
                else:
                    msg = self.sending()

                if msg == QUIT:
                    return

    def sending(self):
        message = input().encode()
        self.sock.sendto(message, self.peer)
        return message

    def receiving(self):
        message, peer = self.sock.recvfrom(1024)
        print("other> {}".format(message.decode()))
        return message

if __name__ == '__main__':
    if len(sys.argv) != 2:
        print("usage: %s [--server|--client]".format(sys.argv[0]))
        sys.exit()

    mode = sys.argv[1]
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    if mode == '--server':
        sock.bind(SERVER)
        message, client = sock.recvfrom(0, socket.MSG_PEEK)
        Chat(sock, client).run()

    else:
        Chat(sock, SERVER).run()
```

LISTADO 15.10: Chat UDP (servidor y cliente) con `select()`
`/chat-udp/chat-select.py`

Aquí los métodos `sending()` y `receiving()` actúan como *manejadores* de los eventos de lectura de consola y del socket respectivamente. A diferencia de la versión con hilos, no hay bucles. Son funciones que se ejecutan cuando se detecta cada una de las condiciones, hacen una única lectura, procesan el mensaje y terminan.

Es sencillo adaptarlo para usar el módulo `selectors`, que indudablemente gana en limpieza, cambiando el método `run()` a algo como:

```
def run(self):
    selector = selectors.DefaultSelector()
```

```
    selector.register(sys.stdin, selectors.EVENT_READ, self.sending)
    selector.register(self.sock, selectors.EVENT_READ, self.receiving)

    while 1:
        for key, mask in selector.select():
            if key.data() == QUIT:
                return
```

Puedes ver el código completo del servidor en el archivo /chat-udp/chat-selectors.py.

15.3. Chatroom UDP

Ahora que están claros los mecanismos para el envío y recepción de mensajes, podemos plantear una solución general para un chat con múltiples usuarios. Llamaremos a esta modalidad 'chatroom' para distinguirlo de la versión para dos usuarios. Esta «sala de chat» es el *canal* al que los publicadores envían sus mensajes.

El nuevo servidor va a tomar el papel de *broker* que recibirá los mensajes de una cantidad arbitraria de participantes. Estos participantes serán a la vez publicadores, por su tarea de envío; y suscriptores, por su tarea de recepción.

La Figura 15.1 representa esta idea, pero ten en cuenta que los óvalos que representan a los participantes aparecen por duplicado, en sus dos roles de publicadores y suscriptores, para enfatizar el flujo de los mensajes.

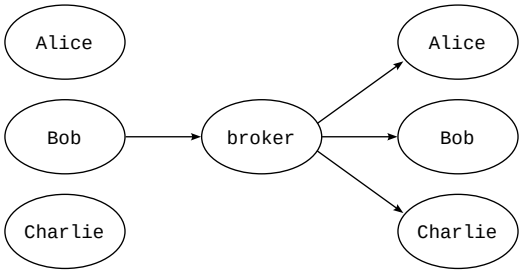

FIGURA 15.1: Arquitectura del chatroom

Puedes ver el código del broker en el Listado 15.11. Mantiene un diccionario llamado `members` que asocia el endpoint de cada participante con su *nick*. Al recibir el primer mensaje de un participante, lo añade al diccionario. Los siguientes mensajes ya se envían a todos los demás participantes precediendo el nombre del emisor. Por último, si el mensaje que envía el participante es `bye`, el broker lo elimina del diccionario y de la sala de chat.

Este programa es un buen ejemplo de cómo un socket UDP, puede operar múltiples flujos (es concurrente) a pesar de trabajar en un único hilo de ejecución.

Es interesante que el broker no necesita ningún mecanismo de concurrencia o gestión asíncrona porque solo usa un socket, el participante sí lo necesita (`selectors` en este caso) porque tiene dos entradas que atender: consola y socket.

```
import socket
QUIT = 'bye'

def main():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.bind(('', 12345))
    members = {}

    while 1:
        data, endpoint = sock.recvfrom(1024)
        message = data.decode()
        if endpoint not in members.keys():
            members[endpoint] = message
            print("User '{}' has joined the chat.".format(message))
            continue

        sender = members[endpoint]
        for member_endpoint, nick in members.items():
            if member_endpoint == endpoint:
                continue

            print(nick, '<-', message)
            encoded = "{}: {}\n".format(sender, message).encode()
            sock.sendto(encoded, member_endpoint)

        if message == QUIT:
            del members[endpoint]
            print("User '{}' has left the chat.".format(sender))

try:
    main()
except KeyboardInterrupt:
    print("shut down.")
```

LISTADO 15.11: Broker de chatroom UDP
/chat-udp/chatroom-broker.py

El cliente (en el rol de participante) es en realidad muy parecido al del paso 4 del chat de dos usuarios (lo tienes en el archivo /chat-udp/chatroom-member.py). Te animamos a probar el broker y unos cuantos participantes en terminales diferentes para que puedas comprobar cómo funciona.

Broker

```
$ ./chatroom-broker.py
New connection from ('127.0.0.1', 41656)
User 'alice' has joined the chat.
New connection from ('127.0.0.1', 53202)
User 'bob' has joined the chat.
New connection from ('127.0.0.1', 47864)
User 'charlie' has joined the chat.
alice <- hi everyone
bob <- hi everyone
```

Alice

```
$ ./chatroom-member.py 127.0.0.1 2000
alice
charlie: hi everyone
```

Bob

```
$ ./chatroom-member.py 127.0.0.1 2000
bob
charlie: hi everyone
```

Charlie

```
$ ./chatroom-member.py 127.0.0.1 2000
charlie
hi everyone
```

FIGURA 15.2: Chatroom con tres participantes

15.4. Chatroom TCP

Por supuesto también tiene mucho sentido hacer una versión TCP, no ya por rendimiento, si no por fiabilidad. Recuerda que UDP puede perder cualquier mensaje en cualquier momento, lo que no es lo que más conviene para una aplicación de mensajería.

Veamos las diferencias más importantes sobre el Listado 15.12.

```
import socket
import selectors

QUIT = 'bye'
PORT = 12345

class ChatroomBroker:
    def main(self):
        with socket.socket() as self.master:
            self.master.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
            self.master.bind(('', PORT))
            self.master.listen(5)

            self.members = {}
            self.selector = selectors.DefaultSelector()
            self.selector.register(self.master, selectors.EVENT_READ, self.acceptor)

            while True:
                for key, mask in self.selector.select():
                    key.data(key.fileobj)

    def acceptor(self, sock):
        conn, addr = self.master.accept()
        print("New connection from", addr)
        self.members[conn] = None
        self.selector.register(conn, selectors.EVENT_READ, self.receiver)
```

```
    def receiver(self, conn):
        message = conn.recv(1024).decode().strip()
        if message == QUIT or not message:
            print("User '{}' has left the chat.".format(self.members[conn]))
            self.selector.unregister(conn)
            conn.close()
            del self.members[conn]
            return

        if not self.members[conn]:
            self.members[conn] = message
            print("User '{}' has joined the chat.".format(message))
            return

        sender = self.members[conn]
        for member_conn, nick in self.members.items():
            if member_conn == conn:
                continue

            print(nick, '<-', message)
            encoded = "{}: {}\n".format(sender, message).encode()
            member_conn.sendall(encoded)

try:
    ChatroomBroker().main()
except KeyboardInterrupt:
    print("shut down.")
```

LISTADO 15.12: Broker de chatroom TCP
/chat-tcp/chatroom-broker.py

El broker mantiene el diccionario `members` cuya clave es el socket conectado y cuyo valor es el *nick* del participante. Lo hemos implementado con dos manejadores para un `selector`: `receiver()` y `reader()`. El primero, como su nombre indica, acepta cada nueva conexión, y la registra en el selector para recibir los mensajes de ese participante. El segundo es el que se encarga de esto. Tiene que manejar varias situaciones:

- Eliminación del participante cuando envía el mensaje `bye` o este cierra la conexión.
- Fijar el *nick* del participante cuando se recibe el primer mensaje.
- Reenviar a todos los demás participantes cuando se recibe un mensaje. En este caso, concatena el nick del emisor al comienzo del mensaje.

El participante también lo hemos implementado con un `selector` y dos manejadores: `sending()` y `receiving()`. El primero lee de la consola y envía el mensaje al broker, mientras que el segundo lee del socket y lo imprime en la consola. El código del participante se muestra en el Listado 15.13.

```
import sys
import socket
import selectors
```

```
QUIT = b'bye'

class ChatroomMember:
    def __init__(self, broker, nick):
        self.broker = broker
        self.nick = nick

    def sending(self):
        message = input().encode()
        self.sock.sendall(message)
        return message

    def receiving(self):
        message = self.sock.makefile().readline().strip()
        print(message)
        return message

    def run(self):
        self.sock = socket.socket()
        self.sock.connect(self.broker)
        self.sock.sendall(self.nick.encode())

        selector = selectors.DefaultSelector()
        selector.register(sys.stdin, selectors.EVENT_READ, self.sending)
        selector.register(self.sock, selectors.EVENT_READ, self.receiving)

        while 1:
            for key, mask in selector.select():
                if key.data() in [QUIT, '']:
                    return+33

if len(sys.argv) != 3:
    exit("Usage: ./chatroom-member.py <broker_address> <nick>")

try:
    broker = (sys.argv[1], 12345)
    ChatroomMember(broker, nick=sys.argv[2]).run()
except (KeyboardInterrupt, EOFError):
    print("shut down.")
```

LISTADO 15.13: Participante de chatroom
/chat-tcp/chatroom-member.py

Recuerda que con `selectors` o `select()` conseguimos un comportamiento asíncrono y concurrente (no paralelo) equivalente en ese sentido al broker UDP.

15.5. Chatroom con MQTT

En la sección anterior hemos visto como programar un broker TCP sencillo que se encarga de la distribución de mensajes desde los productores a los consumidores. MQTT es precisamente un protocolo diseñado para crear este tipo de aplicaciones. Las implementaciones de este protocolo suelen propor-

cionar un broker genérico. El más utilizado por mucho (y que también vamos a usar aquí) se llama `mosquitto`.

Para una sala de chat tan simple como la de la sección anterior, solo necesitamos escribir el código del participante (Listado 15.14) que hace uso de una librería cliente de MQTT llamada `paho-mqtt`, también con mucho la más usada en Python.

```
import json
import sys
import paho.mqtt.client as mqtt

TOPIC = 'chatroom'
QUIT  = 'bye'

class ChatroomMember:
    def __init__(self, broker, nick):
        self.broker = broker
        self.nick = nick

    def on_connect(self, client, userdata, flags, reason_code, properties):
        self.client.subscribe(TOPIC)

    def on_message(self, client, userdata, msg):
        data = json.loads(msg.payload.decode())
        if data['nick'] == self.nick:
            return

        print("{}: {}".format(data['nick'], data['message']))

    def run(self):
        self.client = mqtt.Client(mqtt.CallbackAPIVersion.VERSION2)
        self.client.on_connect = self.on_connect
        self.client.on_message = self.on_message
        self.client.connect(*self.broker)
        self.client.loop_start()

        while 1:
            line = input()
            if line == QUIT:
                break

            payload = json.dumps({'nick': self.nick, 'message': line})
            self.client.publish(TOPIC, payload)

        self.client.loop_stop()
        self.client.disconnect()

if len(sys.argv) != 3:
    exit("Usage: ./chatroom-member.py <broker_address> <nick>")

try:
    broker = (sys.argv[1], 1883)
    ChatroomMember(broker, nick=sys.argv[2]).run()
except (KeyboardInterrupt, EOFError):
    print("shut down.")
```

LISTADO 15.14: Participante de chatroom con MQTT
/chat-mqtt/chatroom-member.py

Puedes apreciar lo mucho que se parece la estructura de este programa al del Listado 15.13. Hay un método `run()` en el que se configura un *callback* `on_connect()` que realiza la suscripción al canal `chatroom` y otro `on_message()` que se encarga de procesar e imprimir los mensajes recibidos.

Como puedes ver, con MQTT tanto los productores como los suscriptores utilizan la clase `Client`. En este caso, al igual que la versión TCP básica, el participante es a la vez publicador —invocando el método `publish()`— y suscriptor —con el callback de `on_message()`.

Este *callback* para `on_message()` funciona de un modo muy similar a los manejadores de `selectors`. Cuando hay datos, el *runtime* invoca el método y los pasa como argumento (`userdata`). Este mensaje puede ser de cualquier tipo que se pueda serializar como cadena o secuencia de bytes, pero lo más habitual con MQTT es usar JSON. A pesar de lo simple que es la carga útil de estos mensajes (nick, mensaje de usuario) hemos preferido usarlo también aquí.

El bucle de eventos de MQTT arranca con `loop_start()` y se ejecuta en un hilo independiente, por eso el método `run()` queda libre para la lectura de mensajes desde la entrada estándar.

El broker `mosquitto` se puede utilizar como un servicio del sistema, sin más que instalar el paquete oficial del mismo nombre. También es posible arrancarlo como un contenedor docker si lo prefieres con algo como:

```
$ docker run --rm -p 1883:1883 eclipse-mosquitto mosquitto -c /mosquitto-no-auth.conf
```

Para esta sencilla prueba, vamos a suponer que el broker está disponible como servicio del sistema en tu máquina. Puedes ejecutar varios participantes en varios terminales indicando la IP del broker y el nick de cada participante.

```
$ ./chatroom-member.py 127.0.0.1 alice
$ ./chatroom-member.py 127.0.0.1 bob
```

También hay un pequeño script `tmux` /chat-mqtt/run.sh que lanza 3 participantes e ilustra el funcionamiento del chatroom.

Y ¿qué más?

El modelo publicador-suscriptor es un muy común en aplicaciones distribuidas por su naturaleza asíncrona y desacoplada. Es especialmente útil

para aplicaciones de monitorización, notitificación o actulización de estado como por ejemplo IoT. También es relevante su buen nivel de escalibilidad y adaptabilidad ya que permite que publicadores y suscriptores se añadan y eliminen sin afectar a los demás participantes. Por otro lado, la crítica más frecuente es que su broker representa un cuello de botella y un punto único de fallo. Solucionar esa limitación lleva arquitecturas más complejas basadas en broker federados o replicados, y en úlitma instancia a arquitecturas sin broker, lo que da lugar a otro importante modelo de interacción: *peer-to-peer* (P2P), llamadas comunmente en español como *redes de pares*.