Programación paralela

práctica evaluable
sockets y arquitectura cliente-servidor

Óscar Martín
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid

marzo-abril de 2024

1. Introducción a los *sockets* en Python

Los *sockets* son un mecanismo de comunicación entre dos máquinas a través de alguna red, o entre dos programas en la misma máquina. Existen casi en cualquier lenguaje de programación. Son un estándar independiente de lenguajes y sistemas. Se pueden usar, por ejemplo, para comunicar un programa C++ ejecutándose en un Mac con un programa Python en Linux. En Python se usan con el módulo `socket`, que suele estar incluido en la instalación básica de Python.

En cada comunicación a través de *sockets*, una máquina actúa como servidor y otra como cliente. El cliente solicita una conexión al servidor; el servidor la acepta y la comunicación puede comenzar. Generalmente, la comunicación consiste en que el cliente hace peticiones que el servidor atiende.

1.1. Esquema de uso en el cliente

```
from socket import socket
sckt = socket()
srvr_ip = "localhost"
srvr_puerto = 12345
sckt.connect((srvr_ip, srvr_puerto))
# aquí tiene lugar el envío y recepción de mensajes
sckt.close()
```

- `socket()` crea un *socket*.
- `srvr_ip` es la dirección IP del servidor con el que se quiere conectar.
- `srvr_puerto` es el puerto por el que el servidor está escuchando.
- `connect(...)` solicita la conexión al servidor a través de la IP y el puerto especificados. Es una operación bloqueante.
- `close()` acaba la conexión y cierra el *socket*.

1.2. Esquema de uso en el servidor

```
from socket import socket
srvr_socket = socket()
ip_addr = "localhost"
puerto = 12345
srvr_socket.bind((ip_addr, puerto))
srvr_socket.listen()
cl_socket, _ = srvr_socket.accept()
# aquí tiene lugar el envío y recepción de mensajes
cl_socket.close()
srvr_socket.close()
```

- `ip_addr` es la dirección IP del servidor. El servidor *escucha* en ella, y el cliente necesita conocerla. La IP especial `"localhost"` sirve cuando el cliente y el servidor están en la misma máquina.
- El servidor elige un número de puerto que el cliente debe conocer. Debe ser un número entre 1.024 y 65.535.
- `bind(...)` asocia el *socket* a una IP y un puerto.
- `listen()` hace que el servidor empiece a escuchar.
- `accept()` es bloqueante y queda esperando que un cliente solicite una conexión. Como resultado de `accept` se crea un nuevo *socket*, que es el que se usa para la comunicación con el cliente (y otro resultado que no nos interesa). El primer *socket* solo sirve para escuchar y establecer la conexión.

1.3. Envío y recepción de mensajes

- `sckt.send(datos)` envía `datos`. Se ejecuta igual desde el cliente que desde el servidor. `datos` debe ser una cadena de octetos, es decir, en Python, un objeto del tipo `bytes`.
- `s.encode()` devuelve una cadena de octetos a partir de una cadena de caracteres `s`.
- `o.decode()` devuelve una cadena de caracteres a partir de una cadena de octetos `o`.
- `b"soy una cadena de octetos"`: la `b` inicial hace que esto sea una cadena de octetos.
- La instrucción `res = sckt.recv(tam)` es bloqueante, y hace que la máquina que la ejecuta quede esperando recibir datos desde la otra máquina. El número `tam` indica la cantidad máxima de octetos que queremos recibir. `res` es la cadena de octetos que se ha recibido.
- `recv` detecta el cierre de la conexión en el otro extremo. Cuando ocurre, devuelve una cadena de octetos vacía.

1.4. Acerca de las IPs y los puertos

Una máquina puede tener varias direcciones IP, correspondientes a distintas interfaces de red. Por ejemplo, nuestros ordenadores suelen tener una conexión *wifi* y una por cable. Cada una tiene una IP distinta. El servidor puede usar (hacer el `bind`) cualquiera de sus IPs, o bien puede usar la IP `"0.0.0.0"`, que hace que el servidor escuche en todos los interfaces de red de la máquina.

Para averiguar la IP de una máquina, en Linux y MacOS se puede ejecutar `ifconfig` y buscar `inet`. En Windows se puede ejecutar `ipconfig` y buscar `IPv4 Address`.

Acerca de los puertos, un servidor puede usar diferentes puertos para ofrecer diferentes servicios. Los números de puerto válidos son del 0 al 65.535, pero los inferiores a 1.024 están reservados. Cualquier número de 5 cifras puede ser elegido con confianza.

La IP especial `localhost` sirve para conectar un ordenador consigo mismo como si la conexión se estuviera estableciendo a través de una red, pero los mensajes no salen de la máquina. Se debe usar para comunicar programas ejecutándose en la misma máquina.

Conectar programas ejecutándose en distintos ordenadores requiere un poco de trabajo. Todos nuestros ordenadores suelen tener instalado un *cortafuegos* (*firewall*). Su misión es impedir que otros ordenadores, potencialmente maliciosos, puedan acceder al nuestro. Si queremos usar *sockets* entre varios ordenadores, debemos decirle al cortafuegos del ordenador que hace de servidor que permita esa comunicación. En la configuración del cortafuegos, hay que *abrir* el puerto que vayamos a usar. Para máquinas en la misma red (conectados al mismo *router*), eso es todo. Para conectar máquinas a través de internet el proceso es algo más complicado.

1.5. Ejemplo mínimo

Este es el código de un sencillo servidor. El servicio que ofrece es transformar cadenas de caracteres a mayúsculas. Los clientes le hacen llegar sus cadenas de caracteres (aunque codificadas como cadenas de octetos) y el servidor lo transforma y lo reenvía en mayúsculas (como cadena de octetos).

```
from socket import socket

srvr_socket = socket()
ip_addr = "localhost"
puerto = 54321
srvr_socket.bind((ip_addr, puerto))
srvr_socket.listen()

cl_socket, _ = srvr_socket.accept()

msj_rec = cl_socket.recv(1024)
peticion = msj_rec.decode()
respuesta = peticion.upper()
msj_env = respuesta.encode()
```

```
cl_socket.send(msj_env)

cl_socket.close()

srvr_socket.close()
```

Y este es un sencillo cliente que hace uso de ese servicio:

```
from socket import socket

sckt = socket()
srvr_ip = "localhost"
srvr_puerto = 54321

sckt.connect((srvr_ip, srvr_puerto))

peticion = "Hola"
msj_env = peticion.encode()
sckt.send(msj_env)
msj_rec = sckt.recv(1024)
respuesta = msj_rec.decode()
print(f"Según el servidor, '{peticion}' en mayúsculas es '{respuesta}'.")

sckt.close()
```

Como resultado visible de ejecutar el servidor y el cliente, se obtiene el mensaje

```
Según el servidor, 'Hola' en mayúsculas es 'HOLA'.
```

2. Ejercicios

2.1. Primer paso: varias peticiones en la misma conexión

Con una conexión establecida, el cliente y el servidor pueden intercambiar cualquier cantidad de mensajes que consideren necesario, no solo un mensaje como en el ejemplo de arriba. Como primer paso de esta práctica, modifica el cliente de forma que tome peticiones de una (persona) usuaria introducidas a través del terminal. Cada petición ha de enviarla al servidor, esperar la respuesta e imprimir el resultado. Es necesario también modificar el servidor para que sea capaz de recibir y contestar peticiones de un cliente en bucle.

Importante: si el servidor está esperando datos de un cliente (ejecutando `recv`) y el cliente cierra su *socket* (porque no necesita nada más del servidor), el servidor recibe un mensaje vacío. Esta es la manera que puede usar el servidor para saber cuándo salir de su bucle y cerrar sus *sockets*. Por ejemplo:

```
msj_rec = cl_socket.recv(1024)
while msj_rec:
    ...
    msj_rec = cl_socket.recv(1024)
```

2.2. Segundo paso: varios tipos de peticiones

Sobre el código resultante del paso anterior, modifica lo necesario para que se puedan solicitar al servidor (al menos) tres servicios distintos: transformar a mayúsculas, transformar a minúsculas e invertir la cadena de caracteres dada. Los mensajes del cliente deben dejar claro cuál es su petición en cada caso. Como las peticiones provienen en última instancia de una usuaria, habrá de ser ella quien deje clara cuál es la operación que está solicitando y sobre qué cadena de caracteres ha de ejecutarse.

2.3. Tercer paso: varios clientes gestionados por hebras

Un servidor puede aceptar conexiones por *sockets* de más de un cliente, y atender a todos a la vez. El servidor ha de crear una hebra para atender a cada cliente con el que se conecte. La parte principal del servidor puede tener esta forma:

```
while True:
    ⟨aceptar una conexión de un cliente⟩
    ⟨crear y poner en marcha una hebra que se encargue de ese cliente⟩
```

Un bucle sin fin no es aceptable en otros contextos, pero los servidores con frecuencia están diseñados para estar dispuestos a dar servicio a sus clientes permanentemente, y no tienen un fin de ejecución programado.

2.4. Cuarto paso: acceso a datos del servidor

El servidor puede tener almacenados datos que todos los clientes pueden querer modificar y consultar. Algo que podríamos llamar una base de datos; por ejemplo, de datos de personas. Un diccionario de Python puede ser una buena forma de almacenar una base de datos de personas. Las operaciones típicas en una base de datos son alta, baja, modificación y consulta; implementa las cuatro. El sistema debe comportarse bien en caso de errores, por ejemplo, cuando se intente dar de baja un registro que no existe en la base de datos. Si conoces excepciones, puedes usarlas, pero no es necesario. El código para una baja podría seguir una estructura parecida a esta:

```
función para dar de baja el registro con clave clave:
    si clave existe en la base de datos
        eliminar ese registro de la base de datos
        devolver un mensaje confirmatorio
    si no
        devolver un mensaje de error
```

El mensaje devuelto por esa función, eventualmente, será enviado al cliente para informarle del resultado de su petición.

Ten en cuenta que, con varias hebras accediendo a los mismos datos simultáneamente, se ha de tener cuidado con los problemas habituales de la concurrencia.

Te sugiero que crees una clase, llamada, por ejemplo, `BaseDeDatos`, en un archivo `.py` independiente, que contenga los cuatro métodos para alta, baja, modificación y consulta, y que el servidor importe y use esa clase.

2.5. Paso extra (opcional): bloqueo selectivo

En una gran base de datos empresarial, de un banco, por ejemplo, puede haber miles de puestos de trabajo (clientes) haciendo operaciones a la vez. En esas condiciones, no es aceptable que la base de datos entera se trate como un solo recurso compartido, y que toda ella se bloquee para cada operación. En lugar de eso, se hacen bloqueos por registro. Si se trata, por ejemplo, de datos de personas, cada operación bloquea los datos de una sola persona.