

Introducción a la programación en Python

Tema 12. Sockets

Autor: Antonio Villena Martín

Contenido

1.	Introducción.....	3
2.	Módulo Socket	3
3.	Recursos.....	10

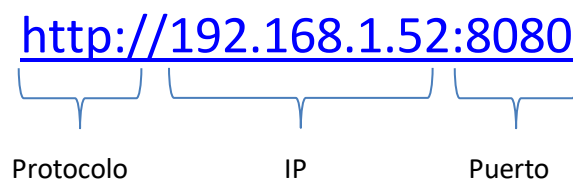
1. Introducción

El presente y el futuro está basado en las redes, Internet, la nube, por nombrar algunos. Python es un lenguaje actualizado y pensado a mejorar, por lo que nos permite realizar programas que puedan interactuar con redes, para ello nos ofrece varios módulos que solo se deben importar y empezar a trabajar con ellos.

Gracias a estas opciones de módulos y librerías disponibles, solo hay que concentrarse en la lógica de nuestro programa, lo que nos da como ventaja poder dedicarle el tiempo a lo que realmente nos interesa.

La comunicación entre diferentes entidades en Python se basa en sockets, los cuales se refieren al punto final de una conexión.

Un socket se define por la dirección IP de la máquina, el puerto de escucha y por el protocolo que utiliza. Por ejemplo:



El módulo socket de Python proporciona los tipos y funciones necesarios para trabajar con sockets, como se verá a continuación.

2. Módulo Socket

El socket es un componente básico en las comunicaciones de redes y se denomina canal de información, ya que permite hacer un intercambio entre el servidor y el cliente en un puerto en específico.

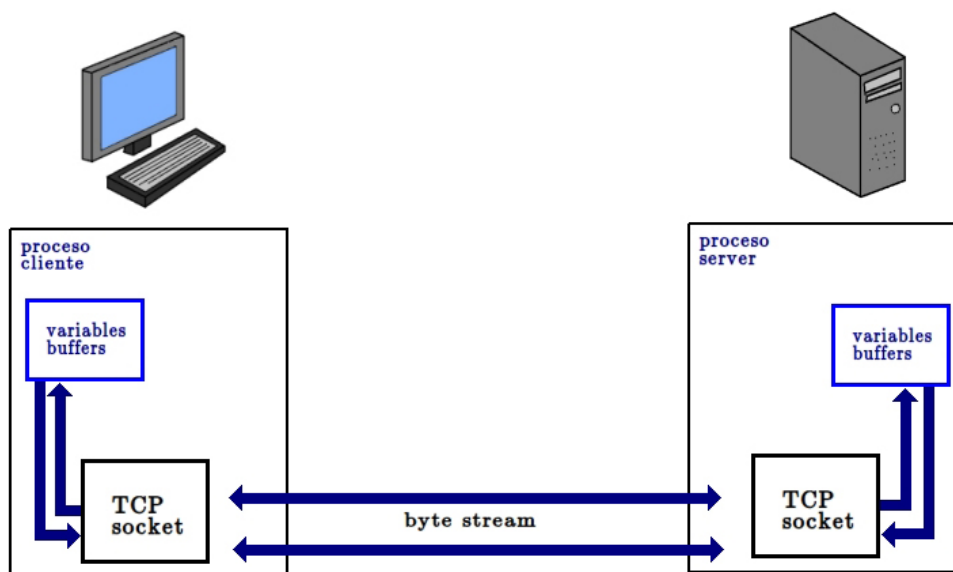
Socket: un socket es un terminal para permitir la comunicación entre dos máquinas. Un socket no es más que una tubería que entregará las cosas de un extremo a otro. El medio puede ser el área local de Newtork, la red de área amplia o Internet. Los sockets pueden usar protocolos TCP y UDP.

Servidor: un servidor es una máquina que espera las peticiones de un cliente y las sirve o las procesa.

Cliente: un cliente, por el contrario, es el que realiza las peticiones al servicio.

Hay dos tipos de sockets, los de flujo (`socket.SOCK_STREAM`) y los sockets de datagramas (`socket.SOCK_DGRAM`) en función de si están orientados a conexión (TCP¹) o a sin conexión (UDP²) respectivamente. El más usado es el de flujo y es que se tratará en este curso.

El socket se divide en dos conceptos, el **server socket** y el **client socket**, se puede inferir que el `server socket` hace la función de servidor, es decir, es quien despacha la información, en cuanto que el `client socket`, es el socket cliente el cual se encarga de hacer peticiones al server, según lo que se haya programado. A continuación, se muestra una imagen que detalla mejor lo antes mencionado:



La aplicación cliente (un navegador web, por ejemplo) usa exclusivamente **sockets “cliente”**; el servidor web con el que habla usa tanto sockets “servidor” como sockets “cliente”.

Como se ha comentado anteriormente, un socket queda definido por la dirección IP de la máquina, el puerto en el que escucha, y el protocolo que utiliza.

¹ TCP: *Transfer Control Protocol*. Es un protocolo de comunicación altamente confiable para transferir datos útiles ya que toma el acuse de recibo de la información enviada. Y vuelve a enviar los paquetes perdidos si los hay. Es más lento que UDP al tener que establecer una conexión. Es más robusto y fiable que UDP, garantizando la entrega de los paquetes en el orden correcto.

² UDP: *User Datagram Protocol*. Protocolo sin conexión, es decir, si el paquete enviado se pierde, no solicitará su retransmisión y el ordenador de destino recibirá un dato corrupto. Por lo tanto, UDP es un protocolo poco fiable. Es más rápido y simple que TCP. Se suele usar para transmisión de audio y vídeo.

Un socket se usa para enviar una petición para el texto de una página. El mismo socket puede leer la respuesta y después se destruye. Los sockets cliente normalmente solo se usan para un intercambio.

Lo que ocurre en el **socket servidor** es un poco más compleja. Para crear un objeto socket, se usa el constructor `socket.socket()` que tiene varios parámetros opcionales, como son la familia, el tipo y el protocolo.

```
mySocket = socket.socket()
```

Mediante el método `bind`, se indica en el socket servidor en qué puerto se va a mantener a la escucha. Para sockets IP, el argumento de `bind` es una tupla que contiene el **host** y el **puerto**. El host se puede dejar vacío, indicando al método que puede utilizar cualquier nombre que esté disponible.

```
mySocket.bind(("127.0.0.1", 5000))
```

Con `listen` se le indica al socket que acepte conexiones entrantes. El argumento de `listen` indica el número máximo de conexiones que se quieren aceptar. En el siguiente ejemplo, le decimos que solamente puede aceptar una conexión. El resto de conexiones las rechazará.

```
mySocket.listen(1)
```

El método `accept` le indica al socket que se mantenga a la espera de conexiones entrantes, bloqueando la ejecución hasta que llegue un mensaje. Este método debe estar enlazado a una dirección (host y puerto) y a la escucha de conexiones. El valor de retorno es un par `(conn, addr)` donde `conn` es un nuevo objeto socket que se usa para enviar y recibir datos en la conexión, y `addr` es la dirección enlazada al socket en el otro extremo de la conexión.

```
conn, addr = mySocket.accept()
```

Hay dos conjuntos de primitivas para usar en la comunicación. Se usa `send()` para enviar un mensaje y `recv()` para recibirlo. El método `send` toma como parámetros los datos a enviar y el método `recv` toma como parámetro el número máximo de bytes a aceptar.

Un **socket cliente** es más sencillo, ya que solo se necesita crear el objeto socket y usar el método `connect` para conectar con el servidor y usar los métodos `send` y `recv`. El argumento de `connect` es una tupla con el host y el puerto al que nos queremos conectar.

```
mySocket = socket.socket()

mySocket.connect(("127.0.0.1", 5000))
```

Una vez que se ha terminado de trabajar con un socket, se cierra mediante el método **close**. Cuando hay una comunicación entre dos sockets y uno de ellos cae bruscamente (sin ejecutar el `close()`) seguramente el socket de este extremo se quede colgado. Esto pasa con los sockets bloqueantes.

El socket cliente de un navegador web y el socket cliente del servidor web son idénticos. Una conversación entre iguales. Normalmente el socket que conecta comienza la conversación, enviando una petición, o puede que una señal de inicio.

Como se ve, el concepto de cliente y servidor muchas veces se diluye, ya que un servidor puede trabajar tanto como servidor, como cliente.

Se va a construir un servidor sencillo y un cliente donde el servidor abrirá un socket y esperará a que los clientes se conecten. Una vez que el cliente está conectado, puede enviar un mensaje y el servidor procesará el mensaje y responderá el mismo mensaje, pero lo convertirá a mayúsculas para demostrar el procesamiento y la transmisión del lado del servidor. Conceptualmente, esto es bastante simple. Se van a crear dos ficheros:

```
servidor.py
cliente.py
```

A continuación se muestra el código ejemplo de la implementación en **servidor.py**:

```
import socket

def main():
    # Definición del socket del servidor
    host = "127.0.0.1"
    port = 5000

    # Instancia del objeto socket
    mySocket = socket.socket()

    # Enlace al socket definido para el servidor
    mySocket.bind((host, port))

    # Establece el número máximo de conexiones entrantes
    mySocket.listen(1)
    # Inicia la escucha
```

```
conn, addr = mySocket.accept()

# Imprime la dirección del cliente entrante
print("Conexión desde: " + str(addr))

# Bucle infinito para procesar la información entrante del cliente
while True:
    # Decodifica la información entrante y la almacena en la variable
    'data'
    data = conn.recv(1024).decode()
    if not data:
        break
    print("desde el usuario conectado: " + str(data))

    # La única operación que realiza es convertir a mayúsculas el
    texto que envia el cliente
    data = str(data).upper()
    print("enviando: " + str(data))
    # Envía de vuelta al cliente el texto modificado
    conn.send(data.encode())

# Cierra la conexión
conn.close()

if __name__ == '__main__':
    main()
```

1. Primero se importa la biblioteca `socket` de python.
2. A continuación, se define una función `main`
3. Se definen dos variables, un host y un puerto, aquí la máquina local es el anfitrión. De este modo la dirección IP es 127.0.0.1 y el puerto 5000 y se aconseja utilizar para el puerto cualquier número por encima de 1024 como servicios básicos de uso de los puertos. Los puertos con número bajo normalmente están reservados para servicios “bien conocidos” (HTTP, SNMP, FTP, etc).
4. A continuación, se define la variable `mySocket` que es una instancia de un socket de Python.
5. En el servidor es importante enlazar (`bind`) el socket al host y a un puerto.
6. Entonces se llama al método de escucha `listen` y se le pasa como parámetro ‘1’ para que escuche de perpetuamente hasta que se cierre la conexión.
7. Entonces tenemos dos variables `conn` y `addr` que llevarán a cabo la conexión del cliente y la dirección del cliente mediante el método `accept`. Este método, como se ha visto anteriormente, hace que el servidor se mantenga a la espera de conexiones entrantes,

bloqueando la ejecución hasta que llega un mensaje.

8. A continuación, se imprimen las direcciones de los clientes y se crea otra variable de datos que está recibiendo datos de la conexión y tenemos que decodificarla, este paso es necesario a partir de la versión 3 de Python, ya que la cadena normal con `str` no pasará por socket y `str` ya no implementa la interfaz de buffer.
9. Se ejecuta todo esto en un bucle infinito, así que a menos que la conexión se cierre se ejecuta esto y el servidor se mantiene a la escucha hasta que se reciben los datos que el servidor transforma en mayúsculas llamando al método `upper` y envía la cadena al cliente y se codifica también como una cadena normal, ya que en caso contrario no transmitirá correctamente.

A continuación se muestra el código de la parte del cliente. Se crea el fichero `cliente.py` y se copia el siguiente contenido:

```
import socket

def main():
    # Establece los datos del socket del servidor al que nos queremos
    # conectar
    host = '127.0.0.1'
    port = 5000

    # Crea el objeto socket
    mySocket = socket.socket()

    # Establece conexión a los datos del socket del servidor
    mySocket.connect((host, port))
    # Solicita la entrada de texto al usuario
    message = input("> ")

    while message != 'q':
        # Codifica el mensaje y lo envía al servidor
        mySocket.send(message.encode())
        # Recibe la respuesta del servidor
        data = mySocket.recv(1024).decode()
        print('Recibido del servidor: ' + data)
        message = input("> ")

    # Cierra la conexión
    mySocket.close()

if __name__ == '__main__':
    main()
```


1. De manera similar, se importa el módulo de socket y se instancia una clase `socket` almacenándolo en la variable `mySocket` y la conexión se establece a través de la dirección del servidor que se pasa como 127.0.0.1 y el puerto 5000.
2. Sin embargo, una diferencia notable es que no se tiene que agregar el enlace ya que el cliente no necesita enlazar, este es un concepto muy básico pero importante de la programación de la red.
3. Luego se usa el método `input` para solicitar información.
4. Mientras el carácter introducido no sea “q”, se seguirá ejecutando el bucle y se envía el mensaje codificándolo y cuando se reciben los datos procesados, se decodifica y se imprime por consola el resultado.
5. Una cosa común a ambos ficheros es que el método `main` de Python que se ejecuta en ambos archivos, que está envuelto en el método `__main__` para que se ejecute.

A continuación, se muestra una secuencia de envío de mensajes a través de estos dos scripts.

En primer lugar, se inicia el servidor:

```
Python servidor.py
```

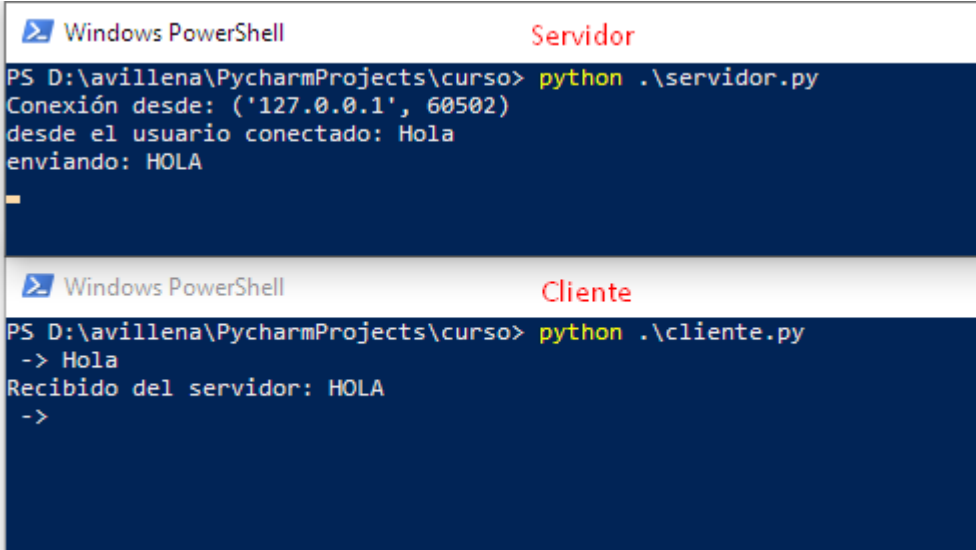
Y después el cliente en otra consola:

```
Python cliente.py
```

Inmediatamente, aparece en la consola del servidor que se ha establecido una conexión.

```
PS D:\avillena\PycharmProjects\curso> python .\servidor.py  
Conexión desde: ('127.0.0.1', 60502)
```

Desde la consola del cliente, se envía el mensaje ‘Hola’ (1). El servidor indica que ha recibido el mensaje ‘Hola’ (2) del usuario y envía un mensaje de vuelta, pero previamente convierte el texto a mayúsculas. Finalmente, el cliente recibe el mensaje de respuesta del servidor y lo imprime.



The image shows two overlapping Windows PowerShell windows. The top window, titled 'Windows PowerShell' and 'Servidor', shows the execution of a Python script. The bottom window, also titled 'Windows PowerShell' and 'Cliente', shows the execution of another Python script. The interaction between the two scripts is visible through the output text.

```
Windows PowerShell Servidor
PS D:\avillena\PycharmProjects\curso> python .\servidor.py
Conexión desde: ('127.0.0.1', 60502)
2 desde el usuario conectado: Hola
3 enviando: HOLA
1
4

Windows PowerShell Cliente
PS D:\avillena\PycharmProjects\curso> python .\cliente.py
1 -> Hola
4 Recibido del servidor: HOLA
->
```

3. Recursos

<https://docs.python.org/es/3/library/socket.html#module-socket>