Introducción a la Programación con PYTHON

José Rodriguez

Unidad 12 - Sockets con Python

UNIDAD 12 – SOCKETS EN PYTHON

Índice

1.- Introducción a los sockets

Los sockets y la API de socket se utilizan para enviar mensajes a través de una red. Proporcionan una forma de comunicación entre procesos (IPC). La red puede ser una red local lógica para la computadora, o una que esté conectada físicamente a una red externa, con sus propias conexiones a otras redes.

Los socket tienen una larga historia. Su uso se originó con ARPANET en 1971 y luego se convirtió en una API en el sistema operativo Berkeley Software Distribution (BSD) lanzado en 1983 llamado Berkeley sockets.

Cuando Internet despegó en la década de 1990 con la World Wide Web, también lo hizo la programación de redes. Los navegadores y servidores web no eran las únicas aplicaciones que aprovechaban las redes recién conectadas y usaban sockets. Se generalizaron las aplicaciones cliente-servidor de todos los tipos y tamaños.

En la actualidad, aunque los protocolos subyacentes utilizados por la API de socket han evolucionado a lo largo de los años y hemos visto otros nuevos, la API de bajo nivel sigue siendo la misma.

El tipo más común de aplicaciones de socket son las aplicaciones cliente-servidor, donde un lado actúa como servidor y espera las conexiones de los clientes. Este es el tipo de aplicación que cubriré en este tutorial. Más específicamente, veremos la API de sockets para sockets de Internet, a veces llamados sockets Berkeley o BSD. También hay sockets de dominio Unix, que solo se pueden usar para comunicarse entre procesos en el mismo host.

2.- Sockets TCP

Como verá en breve, crearemos un objeto socket usando socket.socket () y especificaremos el tipo de socket como socket.SOCK_STREAM. Cuando lo

hace, el protocolo predeterminado que se utiliza es el Protocolo de control de transmisión (TCP). Este es un buen valor predeterminado y probablemente sea lo que desea.

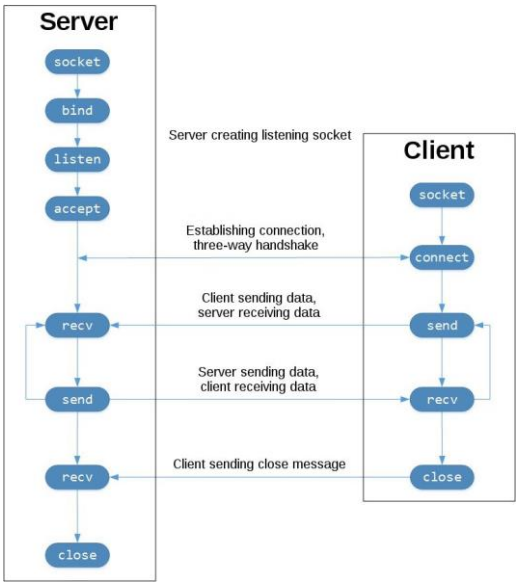
¿Por qué debería utilizar TCP

- Es confiable: los paquetes caídos en la red son detectados y retransmitidos por el remitente.
- Tiene entrega de datos en orden: su aplicación lee los datos en el orden en que fueron escritos por el remitente.

Por el contrario, los sockets del User DatagramProtocol (UDP) creados con socket.SOCK_DGRAM no son confiables, y los datos leídos por el receptor pueden estar desordenados con respecto a las escrituras del remitente.

¿Porque es esto importante? Las redes son un sistema de entrega de mejor esfuerzo. No hay garantía de que sus datos lleguen a su destino o de que recibirá lo que se le envió.

Los dispositivos de red (por ejemplo, enrutadores y conmutadores) tienen un ancho de banda finito disponible y sus propias limitaciones inherentes al sistema. Tienen CPU, memoria, buses y búfer de paquetes de interfaz, al igual que nuestros clientes y servidores. TCP le evita tener que preocuparse por la pérdida de paquetes, los datos que llegan fuera de orden y muchas otras cosas que suceden invariablemente cuando se comunica a través de una red.



La columna de la izquierda representa al servidor. En el lado derecho está el cliente.

Comenzando en la columna superior izquierda, ten en cuenta las llamadas a la API que hace el servidor para configurar un socket de "escucha":

- socket()
- bind()
- listen()
- accept()

Un enchufe de escucha hace exactamente lo que parece. Escucha las conexiones de los clientes. Cuando un cliente se conecta, el servidor llama a accept () para aceptar o completar la conexión.

El cliente llama a connect () para establecer una conexión con el servidor e iniciar el protocolo de enlace de tres vías. El paso del apretón de manos es importante ya que asegura que cada lado de la conexión sea accesible en la red, en otras palabras, que el cliente pueda llegar al servidor y viceversa. Puede ser que solo un host, cliente o servidor, pueda llegar al otro.

En el medio está la sección de ida y vuelta, donde los datos se intercambian entre el cliente y el servidor mediante llamadas a send () y recv ().

En la parte inferior, el cliente y el servidor cierran () sus respectivos sockets.

3.- Funcionamiento de un Socket

socket.socket () crea un objeto socket que admite el tipo de administrador de contexto, por lo que puede usarlo en una declaración with. No es necesario llamar a s.close ():

```
withsocket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
pass
```

Los argumentos pasados a socket () especifican la familia de direcciones y el tipo de socket. AF_INET es la familia de direcciones de Internet para IPv4. SOCK_STREAM es el tipo de socket para TCP, el protocolo que se utilizará para transportar nuestros mensajes en la red.

bind () se utiliza para asociar el socket con una interfaz de red y un número de puerto específicos:

```
HOST='127.0.0.1'
PORT=65432

# ...

s.bind((HOST,PORT))
```

Los valores pasados a bind () dependen de la familia de direcciones del socket. En este ejemplo, usamos socket.AF_INET (IPv4). Entonces espera una tupla de 2: (host, puerto).

host puede ser un nombre de host, una dirección IP o una cadena vacía. Si se utiliza una dirección IP, el host debe ser una cadena de direcciones con formato IPv4. La dirección IP 127.0.0.1 es la dirección IPv4 estándar para la interfaz de bucle invertido, por lo que solo los procesos del host podrán conectarse al servidor. Si pasa una cadena vacía, el servidor aceptará conexiones en todas las interfaces IPv4 disponibles.

puerto debe ser un número entero de 1-65535 (0 está reservado). Es el número de puerto TCP para aceptar conexiones de clientes. Algunos sistemas pueden requerir privilegios de superusuario si el puerto es <1024.

Continuando con el ejemplo del servidor, listen () permite que un servidor acepte () conexiones. Lo convierte en un enchufe de "escucha":

```
s.listen()
conn,addr=s.accept()
```

listen () tiene un parámetro de backlog. Especifica el número de conexiones no aceptadas que permitirá el sistema antes de rechazar nuevas conexiones. A partir de Python 3.5, es opcional. Si no se especifica, se elige un valor de trabajo pendiente predeterminado.

Si su servidor recibe muchas solicitudes de conexión simultáneamente, aumentar el valor de la acumulación puede ayudar al establecer la longitud máxima de la cola para las conexiones pendientes. El valor máximo depende del sistema. Por ejemplo, en Linux, consulte / proc / sys / net / core / somaxconn.

accept () bloquea y espera una conexión entrante. Cuando un cliente se conecta, devuelve un nuevo objeto de socket que representa la conexión y una tupla que contiene la dirección del cliente. La tupla contendrá (host, puerto) para conexiones IPv4 o (host, puerto, flowinfo, scopeid) para IPv6. Consulte Familias de direcciones de socket en la sección de referencia para obtener detalles sobre los valores de tupla.

Una cosa que es imperativo entender es que ahora tenemos un nuevo objeto socket de accept (). Esto es importante, ya que es el enchufe que utilizará para comunicarse con el cliente. Es distinto del socket de escucha que usa el servidor para aceptar nuevas conexiones:

```
conn,addr=s.accept()
withconn:
print('Connected by',addr)
whileTrue:
  data=conn.recv(1024)
  ifnotdata:
    break
conn.sendall(data)
```

Después de obtener el objeto del conector del cliente conn de accept (), se usa un bucle while infinito para recorrer las llamadas de bloqueo a conn.recv (). Esto lee los datos que envía el cliente y los repite usando conn.sendall ().

Si conn.recv () devuelve un objeto de bytes vacío, b '', entonces el cliente cerró la conexión y el bucle finaliza. La instrucción with se usa con conn para cerrar automáticamente el socket al final del bloque.

En comparación con el servidor, el cliente es bastante simple. Crea un objeto de socket, se conecta al servidor y llama a s.sendall () para enviar su mensaje. Por último, llama a s.recv () para leer la respuesta del servidor y luego la imprime.

4.- Diseñando un socket

Un modelo sencillo de servidor-cliente

```
import socket

HOST = '127.0.0.1'  # Standard loopback interface address (localhost)
PORT = 65432        # Port to listen on (non-privileged ports are > 
1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()

    with conn:
        print('Conectado IP:', addr)
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Analizamos el código

La función bind() se utiliza para asociar el socket a un interfaz de red determinado y un puerto en concreto. Los valores pasados a la función bind() dependen de la familia de direcciones del socket. Nosotros estamos usando socket.AF_INET (IPv4), por tanto, se espera la tupla (host, puerto)

Address Family	Protocol	Address Tuple	Description
socket.AF_INET	IPv4	(host, port)	host is a string with a hostname like 'www.example.com' or an IPv4 address like '10.1.2.3'. port is an integer.

La pareja (host,puerto) de un determinado servidor web la podemos obtener a partir de

```
import socket
print(socket.getaddrinfo("www.google.com", 80,
proto=socket.IPPROTO_TCP))
```

dando como resultado

[(<AddressFamily.AF_INET: 2>, 0, 6, '', ('216.58.211.228', 80))]

La función accept() obliga al servidor a aceptar conexiones devolviendo la IP y el puerto del cliente. En este caso guardamos estos valores en dos variables para poder usarlos después.

```
s.listen()
conn, addr = s.accept()
```

Las funciones recv() recoge los datos del cliente y sendall() envía un mensaje a todos los clientes que esperan.

El cliente abre un puerto para la conexión cuyo valor dependerá de la máquina en la que se encuentre.

El cliente sería

```
import socket

HOST = '127.0.0.1'  # The server's hostname or IP address
PORT = 65432        # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    data = s.recv(1024)

print('Received', repr(data))
```

5.- Sockets con websockets

Websockets es un modulo que incluye elementos para construer un socket que funciones a través de la web.

En primer lugar debemos instalarlo. Desde una consola ejecuta la siguiente línea.

```
pip install websockets
```

el servidor tendría la siguiente estructura

```
import asyncio
import websockets

async def hello(websocket, path):
    nombre = await websocket.recv()
    print(f"< {name}")

    greeting = f"Hola {name}!"

    await websocket.send(greeting)
    print(f"> {greeting}")

start_server = websockets.serve(hello, "localhost", 8765)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Mientras que el cliente sería

```
import asyncio
import websockets

async def hello():
    uri = "ws://localhost:8765"
    async with websockets.connect(uri) as websocket:
        name = input("Cual es tu nombre? ")

        await websocket.send(name)
        print(f"> {name}")

        greeting = await websocket.recv()
        print(f"< {greeting}")

asyncio.get_event_loop().run_until_complete(hello())
```

La arquitectura de la aplicación será un servidor Python que espera conexiones a través de la web y un cliente que será una página web que accede al servidor.

En primer lugar creamos un servidor web que envía de forma periódica información a los clientes

`Servidor.py`

```
import asyncio
import datetime
import random
import websockets

async def time(websocket, path):
    while True:
        now = datetime.datetime.utcnow().isoformat() + "Z"
        await websocket.send(now)
        await asyncio.sleep(random.random() * 3)

start_server = websockets.serve(time, "127.0.0.1", 5678)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()

import asyncio
import random
import websockets

async def lista(websocket, path):
    while True:
        numero='7'
        print (numero)
        await websocket.send(numero)
        await asyncio.sleep(random.random())

start_server = websockets.serve(lista, "127.0.0.1", 5678)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

La página web que se conecta al servidor es la siguiente

```
<!DOCTYPE html>
<html>
    <head>
        <title>WebSocket demo</title>
    </head>
    <body>
        <script>
            var ws = new WebSocket("ws://127.0.0.1:5678/"),
                messages = document.createElement('ul');
            ws.onmessage = function (event) {
                var messages = document.getElementsByTagName('ul')[0],
                    message = document.createElement('li'),
                    content = document.createTextNode(event.data);
                message.appendChild(content);
                messages.appendChild(message);
            };
            document.body.appendChild(messages);
        </script>
    </body>
</html>
```

Otro ejemplo del servidor sería

Servidor2.py

```
import asyncio
import datetime
import random
import websockets

async def time(websocket, path):
    while True:
        now = datetime.datetime.utcnow().isoformat() + "Z"
        await websocket.send(now)
        await asyncio.sleep(random.random() * 3)

start_server = websockets.serve(time, "127.0.0.1", 5678)

asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

El resultado en el navegador es

- 2020-08-19T17:18:12.116856Z
- 2020-08-19T17:18:13.195038Z
- 2020-08-19T17:18:15.523107Z
- 2020-08-19T17:18:15.835608Z
- 2020-08-19T17:18:15.882483Z

El siguiente servidor envía números enteros (entre 0 y 9) aleatorios.

```
import asyncio
import random
import websockets

async def lista(websocket, path):
    while True:
        cadena=str(int(random.random()*10))
        print (cadena)
        await websocket.send(cadena)
        await asyncio.sleep(random.random())

start_server = websockets.serve(lista, "127.0.0.1", 5678)

asyncio.get event loop().run until complete(start server)
asyncio.get_event_loop().run_forever()
```