# Blog

# PROGRAMACIÓN DE REDES EN PYTHON: SOCKETS

**Objetivo del 1º tutorial de Curso de Redes en Python**

Definir conceptos básicos y funciones de la librería socket

Introducción a la **programación de sockets, como crearlos y crear conexiones entre ellos.**

Crear un modelo cliente-servidor básico con sockets

En el siguiente tutorial aprenderemos sobre programación de redes. Exploraremos el **modelo cliente-servidor** que se usa en la World Wide Web, e-mail y muchas otras aplicaciones.

El **modelo cliente-servidor** es un framework de comunicación distribuida de procesos de redes entre solicitantes, clientes y proveedores de servicio. Una conexión cliente-servidor es normalmente establecida a traves de una red de Internet.

Este modelo es un concepto clave en la **computación de redes** y en la construcción de funcionalidades para intercambio de e-mails y acceso Web/bases de datos. Algunas tecnologías web y protocolos creados a partir del modelo cliente-servidor son :

- Hypertext Transfer Protocol (**HTTP**)
- Domain Name System (**DNS**)
- Simple Mail Transfer Protocol (**SMTP**)
- Telnet

# Conceptos

Un **cliente** es todo programa que hace solititudes a un servidor y recive información de este, como por ejemplo: navegadores web, aplicaciones de chat y correo electrónico, entre otras.

Un **servidor** es un programa que recive y maneja las peticiones de los clientes para entregar cada pieza de información al cliente que la solicitó. Algunas aplicaciones servidor son: la misma web, bases de datos, chats y correos electrónicos, etc.

Los **sockets** son los extremos de un canal de comunicación bidireccional. Los sockets se pueden comunicar dentro de un proceso, entre procesos dentro de la misma máquina o entre procesos de máquinas de continentes diferentes.

Los sockets pueden ser implementados a traves de un diferente número de canales: sockets de dominio UNIX, TCP, UDP, etc. La librería `socket` de **python** provee clases específicas para manejar el transporte común asi como también una interfaz genérica para controlar todo lo demás.

El módulo `socket` de Python provee una interfaz para la API de los sockets Berkeley (otro nombre para los sockets de Internet). Varias de las operaciones principales para usar sockets con este módulo son:

- `socket()`
- `bind()`
- `listen()`
- `accept()`
- `connect()`
- `connect_ex()`
- `send()`
- `recv()`
- `close()`

Python provee una muy conveniente y consistente API que redirecciona estas funciones del sistema a sus contrapartes en C.

# Iniciando en la programación de redes con Python

La **programación de redes** en Python depende de los objetos **socket**. Para crear un objeto de este tipo en Python, debemos utilizar la función `socket.socket()` disponible en el módulo *socket,* con la siguiente sintaxis:

```
socket_0 = socket.socket(socket_family, socket_type, protocol=0)
```

Veamos una descripción detallada de los parámetros:

**socket_family:** es la familia de protocolos que es usada como mecanismo de transporte. Estos valores son constantes tales como **AF_INET, PF_INET, PF_UNIX, PF_X25,** entre otras.

**socket_type:** el tipo de comunicación entre los dos extremos de la conexión, usualmente se usa **SOCK_STREAM** para protocolos orientados a conexiones y **SOCK_DGRAM** para protocolos sin conexiones.

**protocol:** Normalmente es 0, este parámetro es usado para identificar la variante de un protocolo dentro de una familia y tipo de socket.

# Métodos de los objetos socket

**socket.bind() ->** este método vincula una dirección (hostname, número de puerto) a un socket.

**socket.listen() ->** configura e inicia un oyente TCP.

**socket.accept() ->** esta función acepta pasivamente una conexión de cliente TCP, esperando hasta que la conexión llegue.

Para una información más detallada en cuanto a los métodos en el módulo **socktet**, puedes visitar la documentación en este link. (https://docs.python.org/3/library/socket.html)

# Sockets TCP

Como verás en un momento, crearemos objetos **socket** usando la función `socket.socket()` y especificando el tipo de socket como `socket.SOCK_STREAM`. Cuando hacemos esto, el protocolo predeterminado que usa es el *Protocolo de Control de Transmisión* (TCP).

Pero, ¿por qué deberíamos usar TCP?:

**Es confiable:** los paquetes caídos en la red son detectados y reenviados por el remitente.

**Tiene una entrega de datos ordenada:** los datos son leídos por tu aplicación en el orden en el que los envió el remitente.

# Código para iniciar un servidor

El siguiente código iniciará un servidor web usando la librería **sockets.** El script espera a que una conexión sea hecha y si esta es recibida, mostrará los bytes recibidos.

```
1.  import socket
2.
3.  host = socket.gethostname() # Esta función nos da el nombre de la máquina
4.  port = 12345
5.  BUFFER_SIZE = 1024 # Usamos un número pequeño para tener una respuesta rápida
6.
7.  '''Los objetos socket soportan el context manager type
8.  así que podemos usarlo con una sentencia with, no hay necesidad
9.  de llamar a socket_close()
10. '''
11. # Creamos un objeto socket tipo TCP
12. with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as socket_tcp:
13.
14.     socket_tcp.bind((host, port))
15.     socket_tcp.listen(5) # Esperamos la conexión del cliente
16.     conn, addr = socket_tcp.accept() # Establecemos la conexión con el cliente
17.     with conn:
18.         print('[*] Conexión establecida')
19.         while True:
20.             # Recibimos bytes, convertimos en str
21.             data = conn.recv(BUFFER_SIZE)
22.             # Verificamos que hemos recibido datos
23.             if not data:
24.                 break
```

```
25.             else:
26.                 print('[*] Datos recibidos: {}'.format(data.decode('utf-8')))
27.             conn.send(data) # Hacemos echo convirtiendo de nuevo a bytes
```

En la **programación de redes en Python**, para escribir servidores de internet creamos un objeto **socket** en nuestro código y luego usamos este para llamar a otras funciones del módulo.

Veamos que hace detalladamente este script.

1. **Definimos** el **host** (huesped), el **puerto** y el **tamaño del buffer** de datos que recibirá la conexión
2. **Vinculamos** estas variables a nuestro objeto **socket** con el método `socket.bind()`
3. **Establecemos** la conexión, aceptamos los datos y mostramos los mismos.

Este script no nos muestra ningún resultado si lo ejecutamos ya que hace falta una pieza importante en nuestro modelo **cliente-servidor:** el cliente. El programa solo se ejecuta hasta que llamamos a la función `socket_0.accept()` ya que necesita un programa cliente que se conecte a él.

# Código para iniciar un cliente

Vamos a escribir un programa que defina un cliente que abra la conexión en un puerto y host dado. Esto es muy simple de hacer con la función `socket.connect(hostname, port)` que abre una conexión TCP al *hostname* en el puerto *port*. Una vez hayamos abierto un objecto **socket** podemos leer y escribir en este como cualquier otro objeto de entrada y salida(**IO**), siempre recordando cerrarlo tal como cerramos archivos después de trabajar con estos.

```
1.  import socket
2.
3.  # El cliente debe tener las mismas especificaciones del servidor
4.  host = socket.gethostname()
5.  port = 12345
```

```
6.  BUFFER_SIZE = 1024
7.  MESSAGE = 'Hola, mundo!' # Datos que queremos enviar
8.
9.  with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as socket_tcp:
10.     socket_tcp.connect((host, port))
11.     # Convertimos str a bytes
12.     socket_tcp.send(MESSAGE.encode('utf-8'))
13.     data = socket_tcp.recv(BUFFER_SIZE)
```

Este script es parecido al anterior, solo que esta vez, definimos una variable `MESSAGE` que simulan los paquetes de datos, realizamos la conexión igual que antes y llamamos al método `socket.send(data)` después de convertir nuestra **str a bytes,** para asegurar la integridad de nuestros datos.

Para ejecutar este par de scripts de ejemplo, primero tenemos que ejercutar el servidor:

```
1.  (aprendePython) ➜  python server.py &
```

Anexamos el ampersand (&) para que se ejecute esa línea y quede el proceso abierto esperando otro comando (al presionar Enter, se ejecutará el servidor hasta que ejecutemos el cliente) y luego iniciamos el cliente:

```
1.  (aprendePython) ➜  python client.py
```

El resultado que tenemos es el siguiente:

```
1.  (aprendePython) ➜ python server.py &
2.  [1] 15024
3.  (aprendePython) ➜ python client.py
4.  [*] Conexión establecida
5.  [*] Datos recibidos: Hola, mundo!
```

# Limitaciones en el código

Si ejecutamos estos scripts e intentamos conectarnos a ese mismo servidor desde otra terminal, este simplemente rechazará la conexión. También debemos tener en cuenta que cuando el cliente realiza la llamada a `socket_tcp.recv(1024)`, es posible que la función retorne solo un byte **b'H'** de todo el mensage **b'Hola mundo!'.**

La variable `BUFFER_SIZE` de valor 1024 es la cantidad máxima de datos que pueden ser recibidos de una sola vez. Pero esto no significa que la función retornará 1024 bytes. La función `send()` también tiene este comportamiento. `send()` retorna el número de bytes enviados, los cuales pueden ser menos que el tamaño de los datos que se envían. Debemos controlar ambas deficiencias en nuestro código.

Normalmente en la programación de redes para hacer que un servidor maneje múltiples conexiones al mismo tiempo, se implementa la **concurrencia o paralelismo**.

El problema con la concurrencia es que es complicado hacer que funcione. Hay muchos matices que considerar y situaciones de las cuales protegerse. Claro, no estamos tratando de que el lector no aprenda programación concurrente, si un programa necesita escalabilidad, es casi una obligación aplicar la concurrencia para el uso de más de un procesador o núcleo.

En cambio en este tutorial usaremos algo que es más simple que el paralelismo y mucho más fácil de usar: la librería `selectors`.

# Modelo cliente-servidor de conexiones múltiples

Primero veamos como implementar un servidor que controle varias conexiones:

# Servidor

```
1. import selectors
```

```
import types
import socket

selector = selectors.DefaultSelector()

def accept_conn(sock):
    conn, addr = sock.accept()
    print('Conexión aceptada en {}'.format(addr))
    # Ponemos el socket en modo de no-bloqueo
    conn.setblocking(False)
    data = types.SimpleNamespace(addr=addr, inb=b'', outb=b'')
    events = selectors.EVENT_READ | selectors.EVENT_WRITE
    selector.register(conn, events, data=data)

def service_conn(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(BUFFER_SIZE)
        if recv_data:
            data.outb += recv_data
        else:
            print('Cerrando conexion en {}'.format(data.addr))
            selector.unregister(sock)
            sock.close()
    if mask & selectors.EVENT_WRITE:
        if data.outb:
            print('Echo desde {} a {}'.format(repr(data.outb), data.addr))
            sent = sock.send(data.outb)
            data.outb = data.outb[sent:]

if __name__ == '__main__':
    host = socket.gethostname() # Esta función nos da el nombre de la máquina
    port = 12345
    BUFFER_SIZE = 1024 # Usamos un número pequeño para tener una respuesta rápida

    # Creamos un socket TCP
    socket_tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # Configuramos el socket en modo de no-bloqueo
    socket_tcp.setblocking(False)
    socket_tcp.bind((host, port))
    socket_tcp.listen()
    print('Socket abierto en {} {}'.format(host, port))
    socket_tcp.setblocking(False)
```

```
    # Registramos el socket para que sea monitoreado por las funciones selector,.select()
    selector.register(socket_tcp, selectors.EVENT_READ, data=None)

    while socket_tcp:
        events = selector.select(timeout=None)
        for key, mask in events:
            if key.data is None:
                accept_conn(key.fileobj)
            else:
                service_conn(key, mask)
        socket_tcp.close()

    print('Conexión terminada.')
```

Detallemos un poco más nuestra implementación:

Al igual que antes **definimos** las variables necesarias a vincular con el socket, estas son: `host, port, BUFFER_SIZE, MESSAGE`

Configuramos el socket para en **modo no-bloqueo** con: `socket_tcp.setblocking(False)`. Las funciones del módulo `socket` no retornan un valor inmediatamente, estas tienen que esperar que se complete una llamada del sistema para retornar un valor. Cuando configuramos el socket en no-bloqueo, hacemos que nuestra aplicación no se detenga esperando una respuesta del sistema.

Comenzamos un ciclo **while** en el cual, la primera línea es: `events = sel.select(timeout=None)`. Esta función bloquea hasta que haya sockets listos para ser escritos/leídos. Luego retorna una lista de pares (clave, evento), uno por cada socket. La clave es un `SelectorKey` que contiene un atributo `fileobj`. `key.fileobj` es el objeto **socket** y `mask` es una máscara de evento para las operaciones que están listas.

Si `key.data` es **None,** entonces sabemos que viene del socket que está abierto y necesitamos aceptar la conexión. Llamamos a la función `accept_conn()` que hemos definido para manejar esta situación.

Si `key.data` no es **None,** entonces es un socket cliente que está listo para ser aceptado y necesitamos atenderlo. Así que llamamos a la función `service_conn()` con `key` y `mask` como argumentos, que contienen todo lo que necesitamos para operar el socket.

# Cliente

Ahora veamos una implementación de un cliente. Es bastante parecida a la implementación del servidor pero en lugar de esperar conexiones, el cliente empieza a iniciar conexiones con la función `start_connections()`.

```
import socket
import selectors
import types

selector = selectors.DefaultSelector()
messages = [b'Mensaje 1 del cliente', b'Mensaje 2 del cliente']
BUFFER_SIZE = 1024

def start_connections(host, port, num_conns):
    server_address = (host, port)
    for i in range(0, num_conns):
        connid = i + 1
        print('Iniciando conexión {} hacia {}'.format(connid, server_address))
        socket_tcp = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        # Conectamos usando connect_ex() en lugar de connect()
        # connect() retorna una excepcion
        # connect_ex() retorna un aviso de error
        socket_tcp.connect_ex(server_address)
        events = selectors.EVENT_READ | selectors.EVENT_WRITE
        data = types.SimpleNamespace(connid=connid,
                                     msg_total=sum(len(m) for m in messages),
                                     recv_total=0,
                                     messages=list(messages),
                                     outb=b'')
        selector.register(socket_tcp, events, data=data)
    events = selector.select()
    for key, mask in events:
        service_connection(key, mask)


def service_connection(key, mask):
    sock = key.fileobj
    data = key.data
    if mask & selectors.EVENT_READ:
        recv_data = sock.recv(BUFFER_SIZE) # Debe estar listo para lectura
        if recv_data:
            print('Recibido {} de conexión {}'.format(repr(recv_data), data.connid))
            data.recv_total += len(recv_data)
        if not recv_data or data.recv_total == data.msg_total:
```

```
            print('Cerrando conexión', data.connid)
            selector.unregister(sock)
            sock.close()
        if mask & selectors.EVENT_WRITE:
            if not data.outb and data.messages:
                data.outb = data.messages.pop(0)
            if data.outb:
                print('Enviando {} a conexión {}'.format(repr(data.outb), data.connid))
                sent = sock.send(data.outb) # Debe estar listo para escritura
                data.outb = data.outb[sent:]

if __name__ == '__main__':
        host = socket.gethostname() # Esta función nos da el nombre de la máquina
        port = 12345
        BUFFER_SIZE = 1024 # Usamos un número pequeño para tener una respuesta rápida

        start_connections(host, port, 2)
```

Ahora ejecutamos nuestra nueva implementación de cliente-servidor para múlples conexiones:

```
(aprendePython) ➜ python multiconnection-server.py &
[1] 9276
(aprendePython) ➜ Socket abierto en XXV 12345
python multiconnection-server.py &
(aprendePython) ➜ python multiconnection-client.py
Iniciando conexión 1 hacia ('XXV', 12345)
Iniciando conexión 2 hacia ('XXV', 12345)
Conexión aceptada en ('192.168.0.103', 44858)
Conexión aceptada en ('192.168.0.103', 44860)
Enviando b'Mensaje 1 del cliente' a conexión 1
Enviando b'Mensaje 1 del cliente' a conexión 2
Echoing b'Mensaje 1 del cliente' a ('192.168.0.103', 44858)
Echoing b'Mensaje 1 del cliente' a ('192.168.0.103', 44860)
Cerrando conexion en ('192.168.0.103', 44860)
Cerrando conexion en ('192.168.0.103', 44858)
```

Como vemos, nuestros clientes se comunican con nuestro servidor y este hace “eco” para verificar que los mensajes fueron recibidos.

➞ ¡Felicitaciones por llegar hasta el final de este artículo! Te invitamos a continuar aprendiendo sobre programación en redes con Python en nuestro **Curso de redes en Python:**

**Curso de redes en Python**

**(https://unipython.com/curso-de-network-o-redes/)**

sockets (https://unipython.com/tag/sockets/)

Share:

# 3 COMENTARIOS

## Juan Villarreal

junio 29, 2019 a 6:30 pm (https://unipython.com/programacion-de-redes-en-python-sockets/#comment-854) Responder

excelente contenido aqui seguire aprendiendo muchas gracias por este aporte desde Monterrey – Mexico and Mcallen – Texas

## rafa

marzo 24, 2020 a 9:02 pm (https://unipython.com/programacion-de-redes-en-python-sockets/#comment-1232) Responder

estaba buscando algo de soquets en python y me tope con tu pagin esta exelente las cosas me quedaron muy claras explicas muy bien muchas gracias □

## RAFA

enero 28, 2021 a 6:06 am (https://unipython.com/programacion-de-redes-en-python-sockets/#comment-2564) Responder

Estaba buscando algo de sockets y me topé con tu página, muchas gracias

# Deja una respuesta

Nombre *

E-mail *

Mensaje *

**PUBLICAR EL COMENTARIO**

## Sobre

Unipython es una plataforma de aprendizaje online dirigida a personas que quieran mejorar su carrera profesional. El objetivo de Unipython es proporcionar cursos online de calidad en los campos de la Programación, Internet de las cosas, Analisis de Datos, Inteligencia Artificial, Desarrollo Web/Apps, Testeo, Videojuegos y Tecnología Creativa.

## +Info

- Que dicen de nosotros (https://unipython.com/unipython-en-los-medios/)
- Contacto (https://unipython.com/contacto-5/)
- FAQ (https://unipython.com/faq/)
- Política de Privacidad (https://unipython.com/politica-de-privacidad/)
- Términos y condiciones (https://unipython.com/terminos-y-condiciones/)

## Para empresas

- Contrata a nuestros graduados (https://unipython.com/contrata-a-nuestros-

## Síguenos

- Linkedin (https://www.linkedin.com/company/unipython/)

graduados/)

- Servicios para empresas (https://unipython.com/contrata-nuestros-servicios/)
- Youtube (https://www.youtube.com/channel/UCe0GlySXvnv1OrLXLOoq6yA)