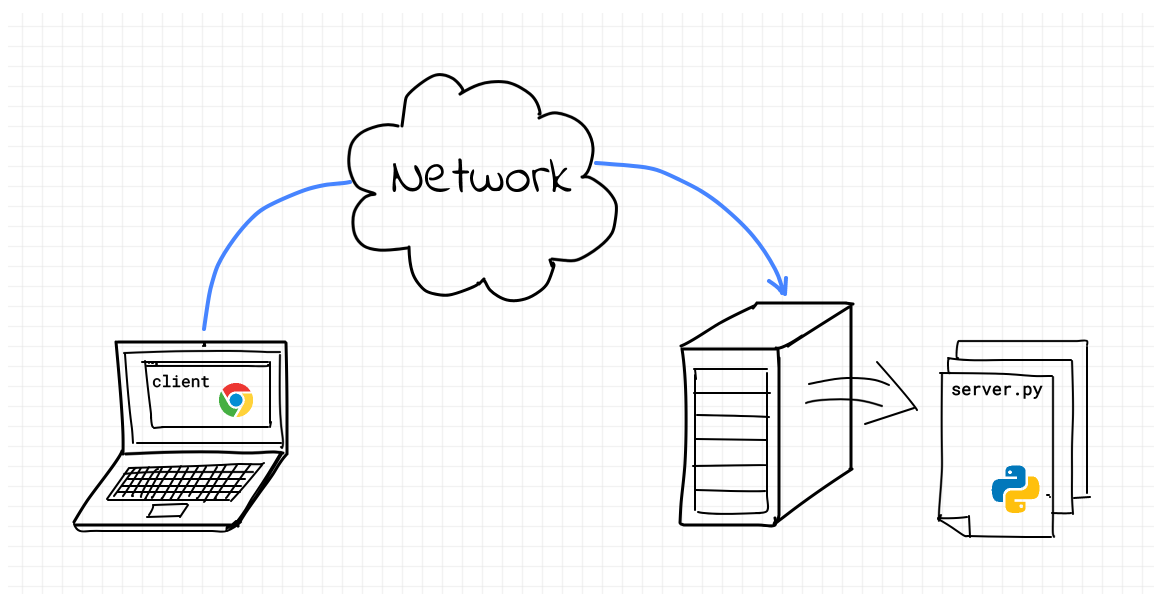


# Escribir servidor web en Python: sockets

## ¿Qué es un servidor Web?

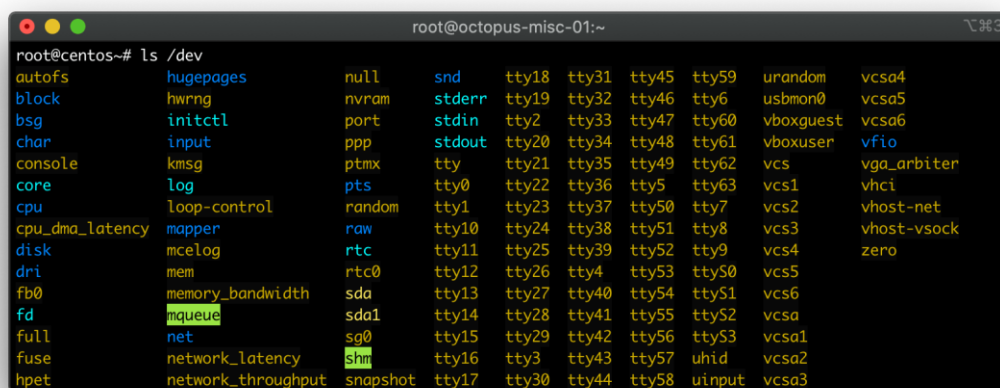
Comencemos respondiendo a la pregunta: ¿Qué es un servidor web ? En primer lugar, es un servidor (sin juego de palabras). Un servidor es un proceso [ *sic* ] que sirve a los clientes. Sorprendentemente o no, un servidor no tiene nada que ver con el hardware. Es solo una pieza de software normal ejecutada por un sistema operativo. Como la mayoría de los otros programas, un servidor obtiene algunos datos de su entrada, los transforma de acuerdo con alguna lógica empresarial y luego produce algunos datos de salida. En el caso de un *servidor web* , la entrada y salida ocurren a través de la red a través *del Protocolo de transferencia de hipertexto (HTTP)* . Para un servidor web, la entrada consiste en solicitudes HTTP de sus clientes: navegadores web, aplicaciones móviles, dispositivos IoT o incluso otros servicios web. Y la salida consta de respuestas HTTP, a menudo en forma de páginas HTML, pero también se admiten otros formatos.



¿Qué es este *protocolo de transferencia de hipertexto* ? Bueno, en este punto, sería suficiente pensar en él como un protocolo de intercambio de datos basado en texto (es decir, legible por humanos). Y la palabra *protocolo* puede explicarse como una especie de convención entre dos o más partes sobre el formato y las reglas de transferencia de datos. No se preocupe, habrá un artículo que cubrirá los detalles de HTTP más adelante, mientras que el resto de este artículo se centrará en cómo las computadoras envían datos arbitrarios a través de la red.

## ¿Qué programación de redes se parece?

En los sistemas operativos similares a Unix, es bastante común [tratar los dispositivos de E / S como archivos](#) . Al igual que los archivos normales en el disco, los ratones de computadora, las impresoras, los módems, etc. se pueden abrir, leer / escribir y luego cerrar.



```
root@centos~# ls /dev
autofs          hugepages      null           snd            tty18          tty31          tty45          tty59          urandom        vcsa4
block           hwrng         nvram          stderr         tty19          tty32          tty46          tty6           usbmon0        vcsa5
bsg             initctl        port           stdin          tty2           tty33          tty47          tty60          vboxguest      vcsa6
char            input          ppp            stdout         tty20          tty34          tty48          tty61          vboxuser       vfio
console         kmsg          ptmx           tty            tty21          tty35          tty49          tty62          vcs            vga_arbiter
core            log            pts            tty0           tty22          tty36          tty5           tty63          vcs1           vhci
cpu             loop-control   random         tty1           tty23          tty37          tty50          tty7           vcs2           vhost-net
cpu_dma_latency mapper          raw            tty10          tty24          tty38          tty51          tty8           vcs3           vhost-vsock
disk            mcelog         rtc            tty11          tty25          tty39          tty52          tty9           vcs4           zero
dri             mem            rtc0           tty12          tty26          tty4           tty53          tty50          vcs5
fb0             memory_bandwidth sda            tty13          tty27          tty40          tty54          tty51          vcs6
fd              queue          sda1           tty14          tty28          tty41          tty55          tty52          vcsa
full            net            sg0            tty15          tty29          tty42          tty56          tty53          vcsa1
fuse            network_latency shm             tty16          tty3           tty43          tty57          uhid           vcsa2
hpet            network_throughput snapshot        tty17          tty30          tty44          tty58          uinput         vcsa3
```

Para cada archivo abierto, el sistema operativo crea un denominado [descriptor de archivo](#) . Simplificando un poco, un descriptor de archivo es solo un identificador entero único de un archivo dentro de un proceso. El sistema operativo proporciona un conjunto de llamadas al sistema para manipular archivos que aceptan un descriptor de archivo como argumento. Aquí hay un ejemplo canónico con operaciones `read()` y `write()`:

```
// C-ish pseudocode

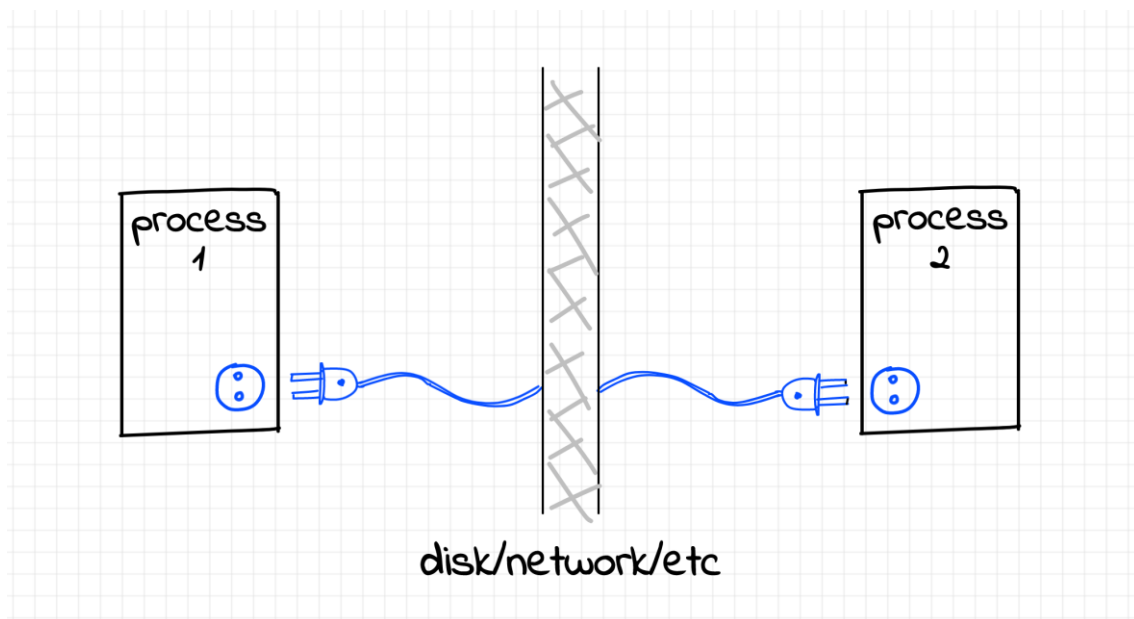
int fd = open("/path/to/my/file", ...);
```

```
char buffer[2048];
read(fd, buffer, 2048);
write(fd, "some data", 10);

close(fd);
```

Dado que la comunicación de red también es una forma de E / S, sería razonable esperar que también se redujera a trabajar con archivos. Y, de hecho, hay un tipo especial de archivo para eso llamado *sockets*.

Un socket es otra pieza de abstracción proporcionada por el sistema operativo. Como suele suceder con las abstracciones informáticas, el concepto se tomó prestado del mundo real, más específicamente de las tomas de corriente CA. Un par de sockets permiten que dos procesos se comuniquen entre sí. En particular, a través de la red. Se puede abrir un socket, los datos se pueden escribir en el socket o leer desde él. Y, por supuesto, cuando el socket ya no se necesita, debe cerrarse.



Los sockets son bastante diversos y hay muchas formas de usarlos para [la comunicación entre procesos](#). Por ejemplo, [los sockets de red](#) se pueden utilizar cuando dos procesos residen en máquinas diferentes. Para procesos locales, [los sockets de dominio Unix](#) pueden ser una mejor opción. Pero incluso estos dos tipos de sockets pueden ser de diferentes tipos: datagrama (o UDP), stream (o TCP), sockets sin

formato, etc. Esta variedad puede parecer complicada al principio, pero afortunadamente hay un enfoque más o menos genérico sobre cómo utilizar sockets de cualquier tipo en el código. Aprender a programar uno de estos tipos de conectores le permitirá extrapolar el conocimiento a otros.

```
// C-ish pseudocode again

int fd = socket(SOCK_TYPE_TCP);

sockaddr serv_addr = { /* ... */ };
connect(fd, serv_addr);

char buffer[2048];
read(fd, buffer, 2048);
write(fd, "some data", 10);

close(fd);
```

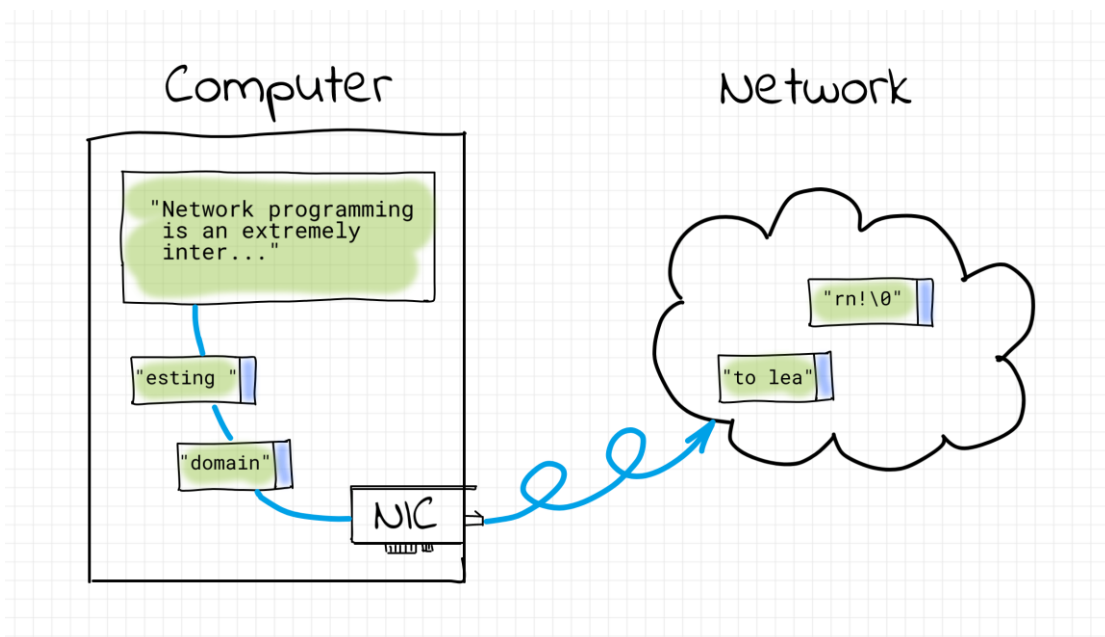
Más adelante en este artículo, nos centraremos en una forma de comunicación cliente-servidor a través de sockets de red utilizando una pila de protocolos TCP / IP. Aparentemente, este es el formulario más utilizado en la actualidad, en particular, porque los navegadores lo utilizan para acceder a sitios web.

## Cómo se comunican los programas a través de la red

Imagine que hay una aplicación que quiere enviar un texto relativamente largo a través de la red. Supongamos que el socket ya se ha abierto y el programa está a punto de `write()`, en lenguaje de redes, `send()` estos datos en el socket. ¿Cómo se transmitirán estos datos?

Las computadoras viven en un mundo discreto. [Las tarjetas de interfaz de red \(NIC\)](#) transmiten datos en pequeñas porciones: unos pocos cientos de bytes a la vez. Al mismo tiempo, en general, el tamaño de los datos que un programa puede querer enviar no está limitado de ninguna manera y puede exceder los cientos de gigabytes. Para transmitir un dato grande arbitrario a través de la red, es necesario fragmentarlo y cada

fragmento debe enviarse por separado. Lógicamente, el tamaño máximo del fragmento no debe exceder la limitación del adaptador de red.



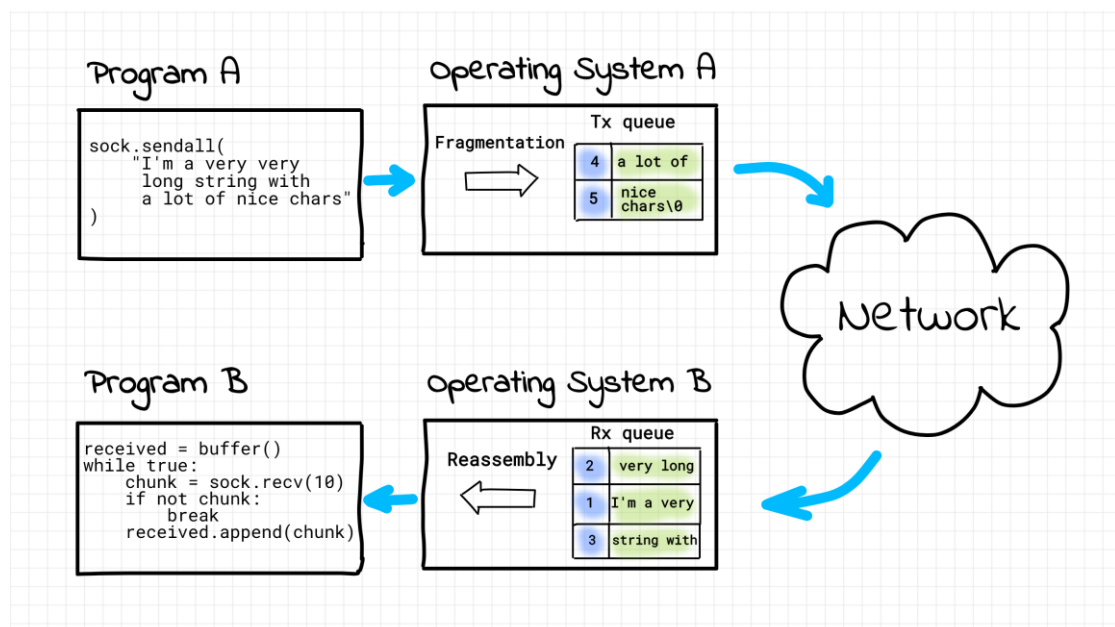
Cada fragmento consta de dos partes: información de control y carga útil. La información de control incluye las direcciones de origen y destino, el tamaño del fragmento, una suma de comprobación, etc. Aunque la carga útil es, bueno, los datos reales que el programa desea enviar.

La mayoría de las veces, para dirigirse a las computadoras en la red, se les asignan las llamadas direcciones IP . El acrónimo *IP* significa *Protocolo de Internet* , un protocolo famoso que hizo posible la **interconexión** de **redes** dando origen a **Internet** . El Protocolo de Internet es el principal responsable de 3 cosas:

- direccionamiento de interfaces de host;
- encapsular los datos de la carga útil en paquetes (es decir, *fragmentación* antes mencionada );
- enrutar paquetes desde un origen a un destino a través de una o más redes IP.

En su camino desde el origen hasta el destino, un paquete IP generalmente pasa por un puñado de hosts intermedios. Esta serie de

hosts constituye una *ruta* . Podría haber (y suele haber) más de una ruta para un par arbitrario (origen, destino). Y dado que múltiples rutas son posibles al mismo tiempo, está perfectamente bien que los paquetes IP con el mismo par (origen, destino) tomen rutas diferentes. Volviendo al problema de enviar un texto largo a través de la red, puede suceder que los *fragmentos*, es decir, los paquetes IP, en los que se dividió el texto, tomarán diferentes rutas en su camino hacia el host de destino. Sin embargo, diferentes rutas pueden tener diferentes retrasos. Además de eso, siempre existe la probabilidad de pérdida de paquetes porque ni los hosts intermedios ni los enlaces son completamente confiables. Por tanto, los paquetes IP pueden llegar al destino en un orden alterado.



En términos generales, no todos los casos de uso requieren un pedido estricto de paquetes. Por ejemplo, el tráfico de voz y video está diseñado para tolerar cierta cantidad de pérdida de paquetes porque la retransmisión de paquetes daría lugar a un aumento inaceptable de la latencia. Sin embargo, cuando un navegador carga una página web usando HTTP, esperamos que las letras y palabras en ella se ordenen exactamente de la misma manera en que lo hizo el creador de la página. Así es como surge la necesidad de un mecanismo de entrega de paquetes confiable, ordenado y con verificación de errores.

Como probablemente ya haya notado, los problemas en el dominio de las redes tienden a resolverse mediante la introducción de más y más protocolos. Y de hecho, existe otro protocolo de Internet famoso llamado *Protocolo de control de transmisión* o simplemente *TCP*. TCP se basa en su protocolo subyacente, IP. El objetivo principal de TCP es proporcionar la entrega confiable y ordenada de un flujo de bytes entre aplicaciones. Por lo tanto, si alimentamos nuestro texto (codificado) a un socket TCP en una máquina, se puede leer sin alterar desde el socket en la máquina de destino. Para no preocuparse por los problemas de entrega de paquetes, HTTP se basa en las capacidades de TCP.



Para lograr la entrega en orden y confiable, TCP aumenta la información de control de cada fragmento con el número de secuencia de incremento automático y la suma de verificación. En el lado de la recepción, el reensamblaje de los datos no se basa en el orden de llegada de los paquetes, sino en el número de secuencia de TCP. Además, la suma de comprobación se utiliza para validar el contenido de los fragmentos que llegan. Los fragmentos con formato incorrecto simplemente se rechazan y no se reconocen. Se espera que el lado de envío retransmita los fragmentos que no han sido reconocidos. Obviamente, se requiere algún tipo de almacenamiento en búfer en ambos lados para implementar esto.

En una sola máquina en un momento dado, puede haber muchos procesos que se comunican a través de sockets TCP. Por lo tanto, debe haber tantos números de secuencia y búferes independientes como sesiones de comunicación. Para abordar esto, TCP introduce el concepto de *conexión*. Simplificando un poco, una *conexión TCP* es una especie de acuerdo entre los lados transmisor y receptor sobre los números de secuencia inicial y el estado actual de transmisión. Es necesario

establecer una conexión (intercambiando algunos paquetes de control al principio, el llamado *protocolo de enlace*), mantenerla (algunos paquetes deben enviarse en ambos sentidos, de lo contrario, la conexión puede expirar, lo que se denomina *keepalive*), y cuando la conexión ya no sea necesaria, debe cerrarse (intercambiando algunos otros paquetes de control).

Por último, pero no menos importante ... Una dirección IP define un host de red como un todo. Sin embargo, entre dos hosts cualesquiera, puede haber muchas conexiones TCP simultáneas. Si la única información de direccionamiento en nuestros fragmentos fueran las direcciones IP, sería prácticamente imposible determinar la afiliación de los fragmentos con las conexiones. Por lo tanto, se requiere cierta información de direccionamiento adicional. Para eso, TCP introduce el concepto de *puertos*. Cada conexión obtiene un par de números de puerto (uno para el remitente, otro para el receptor) que identifica de forma única la conexión TCP entre un par de IP. Por lo tanto, cualquier conexión TCP puede ser plenamente identificado por la siguiente tupla: (source IP, source port, destination IP, destination port).

## Implementación de un servidor TCP simple

---

¡Es hora de practicar! Intentemos crear nuestro propio servidor TCP diminuto en Python. Para eso, necesitaremos el `socket` módulo de la biblioteca estándar.

Para un novato, la principal complicación con los sockets es la existencia de un ritual aparentemente mágico de preparar los sockets para que funcionen. Sin embargo, la combinación de los antecedentes teóricos del comienzo de este artículo con la parte práctica de esta sección debería convertir la magia en una secuencia de acciones significativas.

En el caso de TCP, los flujos de trabajo de socket del lado del servidor y del lado del cliente son diferentes. Un servidor espera pasivamente a que los clientes se conecten. A priori, la dirección IP y el puerto TCP del servidor son conocidos por todos sus potenciales clientes. Por el contrario, el servidor no conoce las direcciones de sus clientes hasta el momento en que se conectan. Es decir, los clientes desempeñan el papel



de iniciadores de la comunicación al conectarse activamente a los servidores.

Sin embargo, hay más que eso. En el lado del servidor, en realidad hay dos tipos de sockets involucrados: el socket del servidor antes mencionado esperando conexiones y, sorpresa, sorpresa, ¡los sockets del cliente! Por cada conexión establecida, hay un socket más creado en el lado del servidor, simétrico a su contraparte del lado del cliente. Por lo tanto, para  $N$  clientes conectados, siempre habrá  $N + 1$  sockets en el lado del servidor.

## Crear socket TCP del servidor

Entonces, creemos un socket de servidor:

```
# python3

import socket

serv_sock = socket.socket(
    socket.AF_INET,      # set protocol family to 'Internet' (INET)
    socket.SOCK_STREAM,  # set socket type to 'stream' (i.e. TCP)
    proto=0              # set the default protocol (for TCP it's IP)
)

print(type(serv_sock))  # <class 'socket.socket'>
```

Espera un minuto ... ¿Dónde está lo prometido `int fd = open("/path/to/my/socket")`? La verdad es que la llamada al sistema `open()` es demasiado limitada para el caso de uso de socket porque no permite pasar todos los parámetros necesarios, como la familia de protocolos, el tipo de socket, etc. Por lo tanto, para los sockets, `socket()` se ha introducido una llamada al sistema dedicada. De manera similar `open()`, después de crear un punto final para la comunicación, devuelve un descriptor de archivo que se refiere a ese punto final. A partir de la `fd = ...` parte perdida, Python es un lenguaje orientado a objetos. En lugar de funciones, tiende a utilizar clases y métodos. El `socket` módulo de la biblioteca estándar de Python es en realidad un [delgado envoltorio OO](#) alrededor del [conjunto de llamadas relacionadas con sockets](#). Simplificando demasiado, se puede pensar en algo como eso:

```
class socket: # Yep, the name of the class starts from a lowercase letter...
    def __init__(self, sock_family, sock_type, proto):
        self._fd = system_socket(sock_family, sock_type, proto)

    def write(self, data):
        system_write(self._fd, data)

    def fileno(self):
        return self._fd
```

Es decir, si alguien realmente lo necesita, la descripción del archivo entero se puede obtener de la siguiente manera:

```
print(serv_sock.fileno()) # 3 or some other small integer
```

## Vincular el socket del servidor a la interfaz de red

Dado que, en general, una sola máquina servidor puede tener más de un adaptador de red, debería haber una forma de *vincular* el socket del servidor a una interfaz en particular asignando una dirección local de esta interfaz al socket:

```
# Use '127.0.0.1' to bind to localhost
# Use '0.0.0.0' or '' to bind to ALL network interfaces simultaneously
# Use an actual IP of an interface to bind to a specific address.

serv_sock.bind(('127.0.0.1', 6543))
```

Además de eso, `bind()` requiere que se especifique un puerto. El servidor estará esperando o, en lenguaje de redes, *escuchando* las conexiones del cliente en ese puerto.

## Espere las conexiones del cliente

A continuación, el socket debe convertirse explícitamente en un estado de *escucha*:

```
serv_sock.listen(10) # 10 - is a size of the connections queue, so-called backlog size
```

Después de esta llamada, el sistema operativo prepara el socket del servidor para *aceptar* conexiones entrantes. Sin embargo, nuestro código aún no está listo para eso. Pero primero, vamos a tocar brevemente la parte del trabajo pendiente.

Como ya sabemos, la comunicación en red ocurre mediante el envío de paquetes. Al mismo tiempo, TCP requiere el *establecimiento* de conexiones. Por lo tanto, para establecer una conexión TCP, un cliente y un servidor deben intercambiar algunos paquetes de control (es decir, sin datos comerciales) (el llamado *protocolo de enlace*). Y debido a los retrasos de la red, no es un procedimiento instantáneo.

Por lo general, TCP se implementa en el nivel del sistema operativo y, en nuestros programas, no nos ocupamos de los detalles de nivel inferior, como el protocolo de enlace. Los parámetros del backlog definen el tamaño de la cola de las conexiones establecidas. Hasta que el número de clientes conectados sea menor que el tamaño de la acumulación, el sistema operativo establecerá nuevas conexiones y las pondrá en cola. Sin embargo, cuando el número de la conexión establecida alcanza el tamaño de la acumulación, cualquier conexión nueva será rechazada explícitamente o ignorada implícitamente (depende de las configuraciones del sistema operativo).

## Aceptar conexión de cliente

Para obtener una conexión establecida de la cola de trabajos pendientes, debemos hacer lo siguiente:

```
client_sock, client_addr = serv_sock.accept()
```

Sin embargo, la cola de conexiones establecidas puede estar vacía. En tal caso, la `accept()` llamada bloqueará la ejecución del programa hasta que el próximo cliente se conecte (o el programa sea interrumpido por una señal, pero no es el tema de este artículo).

Después de *aceptar* la primera conexión de cliente, habrá dos sockets en el lado del servidor: el ya familiar `serv_socket` en el `LISTEN` estado y el nuevo `client_sock` en el `ESTABLISHED` estado. Curiosamente, el `client_sock` del lado del servidor y el socket correspondiente del lado

del cliente son los denominados puntos finales del mismo nivel. Es decir, son del mismo tipo, los datos se pueden escribir o leer desde cualquiera de ellos, y ambos se pueden cerrar usando una `close()` llamada que termina la conexión de manera eficiente. Ninguna de estas acciones afectará la escucha de `serv_socket` todos modos.

## Obtener el puerto y la IP del socket del cliente

Echemos un vistazo a las direcciones de los extremos del servidor y del cliente. Cada socket TCP se puede identificar por dos pares de números: (local IP, local port) y (remote IP, remote port). Para conocer la IP remota y el puerto del cliente recién conectado, el servidor puede inspeccionar la `client_addr` variable devuelta por la `accept()` llamada exitosa :

```
print(client_addr) # E.g. ('127.0.0.1', 54614)
```

Alternativamente, se puede utilizar el `socket.getpeername()` método del punto final del mismo nivel del lado del servidor `client_sock` para conocer la dirección *remota* del cliente conectado. Y para conocer la dirección *local* que el sistema operativo del servidor asignó para el punto final del mismo nivel del lado del servidor, se puede usar el `socket.getsockname()` método.

En el caso de nuestro servidor, puede verse así:

```
serv_sock:
    laddr (ip=<server_ip>, port=6543)
    raddr (ip=0.0.0.0, port=*)

client_sock: # peer
    laddr (ip=<client_ip>, port=51573) # 51573 is a random port
    assigned by the OS
    raddr (ip=<server_ip>, port=6543) # it's a server's listening port
```

## Enviar y recibir datos a través de socket

Aquí hay un ejemplo simple de recibir algunos datos del cliente y luego enviarlos de vuelta (el llamado servidor de eco):

```
# echo-server  
  
data = client_sock.recv(2048)  
client_sock.send(data)
```

Bueno, ¿dónde están las prometidas `read()` y las `write()` llamadas? Si bien es posible usar estas dos llamadas con descriptores de archivos de socket, como con el `socket()` llamada al sistema, no permiten especificar todas las posibles opciones necesarias. Por lo tanto, se han introducido para sockets `send()` y `recv()` llamadas al sistema. Desde el man 2 `send`:

La única diferencia entre `send ()` y `write ()` es la presencia de banderas. Con un argumento de banderas cero, `send ()` es equivalente a `write ()`.

... Y man 2 `recv`:

La única diferencia entre `recv ()` y `read ()` es la presencia de banderas. Con un argumento de banderas cero, `recv ()` es generalmente equivalente a `read ()`.

Detrás de la aparente simplicidad del fragmento anterior hay un problema grave. Tanto `recv()` y `send()` son llamadas que en realidad su trabajo es a través de los denominados buffers de red. La llamada a `recv()` regresa tan pronto como aparecen *algunos* datos en el búfer en el lado receptor. Y, por supuesto, *alguna rara vez* significan *todos*. Por lo tanto, si el cliente desea transmitir, digamos 1800 bytes de datos, `recv()` puede regresar tan pronto como se reciban los primeros 1500 bytes (los números son arbitrarios en este ejemplo) porque la transmisión se dividió en dos porciones.

Lo mismo ocurre con el `send()` método. Devuelve el número real de bytes que se han escrito en el búfer. Sin embargo, si el búfer tiene menos espacio disponible que el dato que se intentó, solo se escribirá una parte. Por lo tanto, depende del remitente asegurarse de que el resto de los datos se transmitirán eventualmente. Afortunadamente, Python proporciona un `socket.sendall()` ayudante útil que realiza el bucle de envío bajo el capó.

En realidad, esto conduce a [consideraciones interesantes cuando se trata de diseñar el intercambio de datos a través de TCP](#) :

los mensajes deben tener una longitud fija (asco), o estar delimitados (encogerse de hombros), o indicar cuánto duran (mucho mejor), o terminar cerrando la conexión.

## Detectar que el cliente ha terminado con el envío (apagado)

Tenga en cuenta que las tres primeras opciones aún pueden conducir a una situación en la que el socket del lado del servidor estará esperando que la `recv()` llamada regrese por tiempo indefinido. Puede suceder si el servidor quisiera recibir  $K$  mensajes del cliente mientras que el cliente solo quisiera enviar  $M$  mensajes, donde  $M < K$ . Por lo tanto, depende de los diseñadores de protocolo de nivel superior decidir las reglas de comunicación.

Sin embargo, [existe una forma sencilla de indicar que el cliente ha terminado con el envío](#). El socket de cliente puede `shutdown(how)` la conexión especificando `SHUT_WR` para `how`. Esto dará lugar a una `recv()` llamada en el lado del servidor que devuelve 0 bytes. Por lo tanto, podemos reescribir el código de recepción de la siguiente manera:

```
chunks = []
while True:
    data = client_sock.recv(2048)
    if not data:
        break
    chunks.append(data)
```

## Cerrar sockets

Cuando hayamos terminado con un socket, debería estar cerrado:

```
socket.close()
```

Cerrar un socket de forma explícita conducirá a vaciar sus búferes y cerrar la conexión correctamente.

## Ejemplo de servidor TCP simple

Finalmente, aquí está el código completo del servidor de eco TCP:

```
# python3

import socket

# Create server socket.
serv_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM, proto=0)

# Bind server socket to loopback network interface.
serv_sock.bind(('127.0.0.1', 6543))

# Turn server socket into listening mode.
serv_sock.listen(10)

while True:
    # Accept new connections in an infinite loop.
    client_sock, client_addr = serv_sock.accept()
    print('New connection from', client_addr)

    chunks = []
    while True:
        # Keep reading while the client is writing.
        data = client_sock.recv(2048)
        if not data:
            # Client is done with sending.
            break
        chunks.append(data)

    client_sock.sendall(b''.join(chunks))
    client_sock.close()
```

Guárdelo `server.py` ejecútelo `python3 server.py`.

## Implementación de un cliente TCP simple

Las cosas son mucho más sencillas del lado del cliente. No existe un conector de escucha en el lado del cliente. Solo necesitamos crear un único punto final de socket y `connect()` enviarlo al servidor antes de enviar algunos datos:

```

# python3

import socket

# Create client socket.
client_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect to server (replace 127.0.0.1 with the real server IP).
client_sock.connect(('127.0.0.1', 6543))

# Send some data to server.
client_sock.sendall(b'Hello, world')
client_sock.shutdown(socket.SHUT_WR)

# Receive some data back.
chunks = []
while True:
    data = client_sock.recv(2048)
    if not data:
        break
    chunks.append(data)
print('Received', repr(b''.join(chunks)))

# Disconnect from server.
client_sock.close()

```

Guárdelo `client.py` ejecútelo `python3 client.py`.

## Servidor de socket vs servidor HTTP

El servidor que implementamos antes es claramente un servidor TCP simple. Sin embargo, no es un servidor web (todavía). Si bien (¿casi?) Todos los servidores web son servidores TCP, no todos los servidores TCP son, por supuesto, un servidor web. Para convertir este servidor en un servidor web, necesitaríamos enseñarle cómo manejar HTTP. Es decir, los datos transmitidos a través de sockets deben formatearse de acuerdo con el conjunto de reglas definidas por el Protocolo de transferencia de hipertexto y nuestro código debe saber cómo analizarlo.



# Terminando

---

Memorizar cosas sin entenderlas es una mala estrategia para un desarrollador. La programación de sockets es un ejemplo perfecto cuando mirar el código sin los antecedentes teóricos puede ser simplemente abrumador. Sin embargo, una vez que se adquiere la comprensión de las partes móviles y las limitaciones, todas estas manipulaciones mágicas con la API de socket se convierten en un conjunto significativo de acciones. Y no tenga miedo de dedicar tiempo a lo básico. La programación de redes es un conocimiento fundamental que es vital para el desarrollo exitoso y la resolución de problemas de servicios web avanzados.

## Ejecutando los Scripts

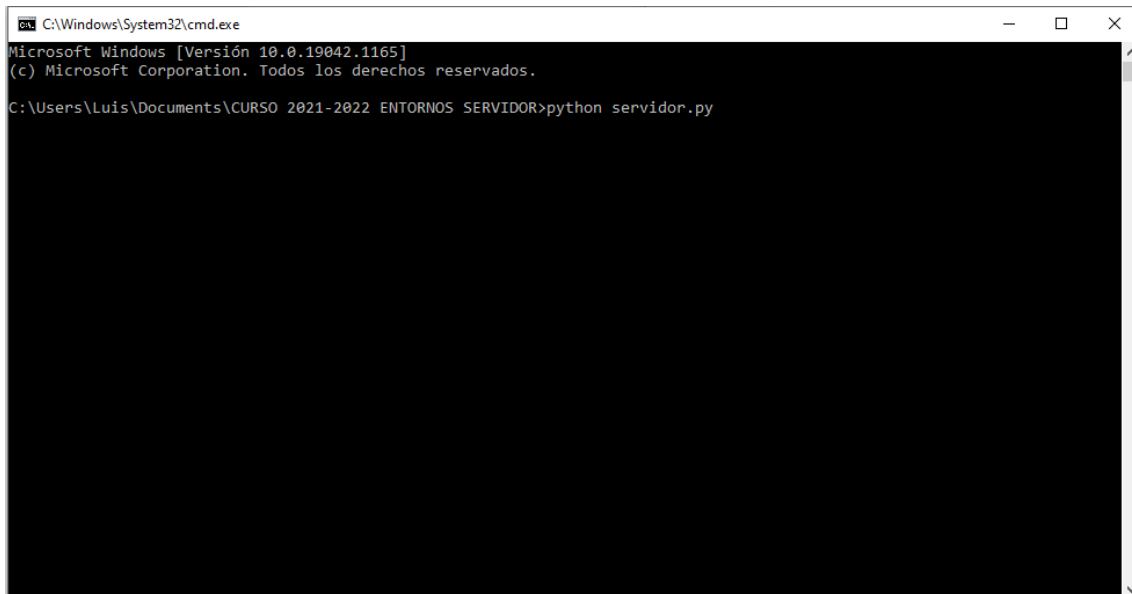
Ahora que ya tenemos los 2 scripts, vamos a ejecutarlos, se deben ejecutar de manera separada cada uno en una terminal CMD de Windows.

En una terminal vamos a ejecutar el servidor y en otra terminal ejecutamos el cliente.

### **Terminal con el servidor**

Hay que escribir en la consola cmd de Windows > python server.py

Se quedara el servidor esperando peticiones del cliente.

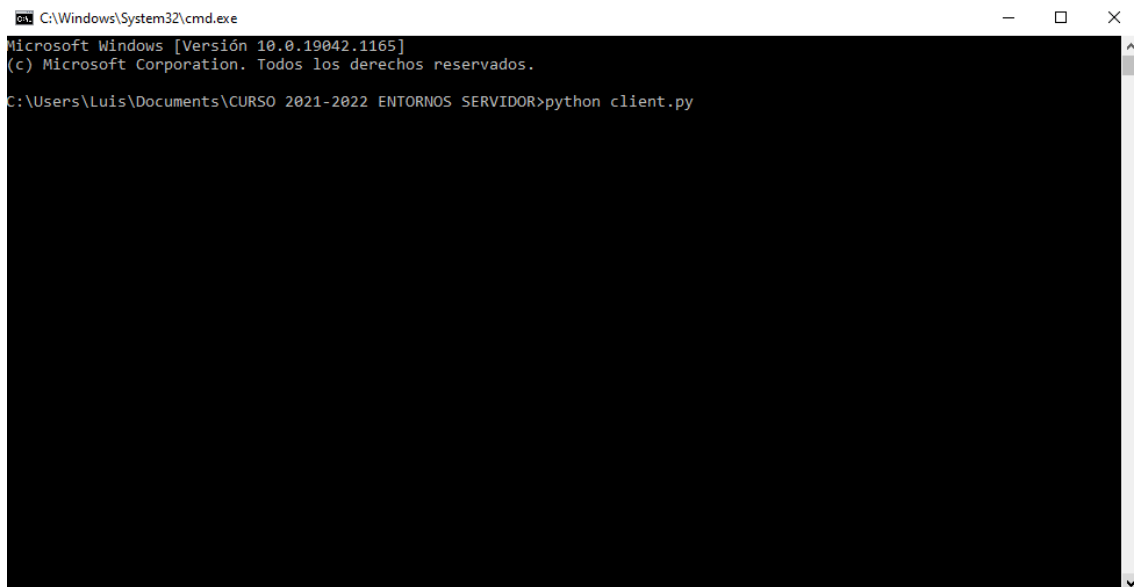


```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.19042.1165]
(c) Microsoft Corporation. Todos los derechos reservados.
C:\Users\Luis\Documents\CURSO 2021-2022 ENTORNOS SERVIDOR>python servidor.py
```

## Terminal con el cliente

En el cliente escribimos > python client.py

Veremos la respuesta que nos devuelve el servidor en la terminal del cliente.



```
C:\Windows\System32\cmd.exe
Microsoft Windows [Versión 10.0.19042.1165]
(c) Microsoft Corporation. Todos los derechos reservados.
C:\Users\Luis\Documents\CURSO 2021-2022 ENTORNOS SERVIDOR>python client.py
```