HOW TO - Programación con sockets

Versión 3.10.13

Guido van Rossum
and the Python development team

noviembre 14, 2023

Python Software Foundation
Email: docs@python.org

Índice general

Autor Gordon McMillan

Resumen

Los sockets son usados casi en todas partes pero son una de las tecnologías más incomprendidas. Esta es una descripción general de los sockets. No es realmente un tutorial, todavía tendrás trabajo para hacer que las cosas funcionen. No cubre los pequeños detalles (y hay muchos de ellos) pero espero que pueda dar suficiente información para comenzar a usarlos decentemente.

1 Sockets

Solo voy a hablar de los sockets *INET* (como IPv4), pues solo ellos cubren el 99% del uso de los sockets. Y solo voy a hablar sobre los sockets *STREAM* (como TCP), a menos que realmente sepas lo que haces (y en ese caso esta guía no es para ti), tendrás mejor comportamiento y rendimiento con un socket *STREAM* que con cualquier otro. Voy a tratar de aclarar el misterio de que es un socket, además de algunas ideas sobre como trabajar con sockets bloqueantes y no bloqueantes. Pero voy a comenzar hablando de los sockets bloqueantes, necesitarás saber como funcionan antes de lidiar con los no bloqueantes.

Parte del problema para entenderlos es que «socket» puede significar un número de cosas ligeramente diferentes dependiendo del contexto. Entonces, primero vamos a hacer una distinción entre sockets «cliente» - un extremo de una conversación, y un socket «servidor», que es más como una central de teléfonos. La aplicación cliente (tu navegador, por ejemplo) usa sockets «cliente» exclusivamente; el servidor web con quien se está comunicando usa sockets «cliente» y «servidor».

1.1 Historia

De las varias formas de comunicación entre procesos (IPC (Inter Process Communication)) los sockets son, por mucho, la más popular. En cualquier plataforma es probable que existan otras formas de IPC más rápidas, pero en comunicación multiplataforma los sockets son los únicos competidores.

Fueron inventados en Berkeley como parte del sabor BSD de Unix. Se propagaron como la pólvora con Internet. Con razón: la combinación de sockets con INET hace que hablar con máquinas arbitrarias de todo el mundo sea increíblemente fácil (al menos en comparación con otros esquemas).

2 Creando un socket

De manera general, cuando hiciste click en el enlace que te trajo a esta página tu navegador hizo algo como lo siguiente:

```
# create an INET, STREAMing socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# now connect to the web server on port 80 - the normal http port
s.connect(("www.python.org", 80))
```

Cuando `connect` termina, el socket `s` puede ser usado en una petición para traer el texto de la página. El mismo socket leerá la respuesta y luego será destruido. Así es, destruido. Los sockets cliente son normalmente usados solo para un intercambio (o un pequeño numero se intercambios secuenciales).

Lo que sucede en el servidor web es un poco más complejo. Primero, el servidor web crea un «socket servidor»:

```
# create an INET, STREAMing socket
serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
# bind the socket to a public host, and a well-known port
serversocket.bind((socket.gethostname(), 80))
# become a server socket
serversocket.listen(5)
```

Un par de cosas que señalar: usamos `socket.gethostname()` para que el socket fuera visible al mundo exterior. Si hubiésemos usado `s.bind(('localhost', 80))` o `s.bind(('127.0.0.1', 80))` habríamos tenido un socket servidor pero solo habría sido visible en la misma máquina. `s.bind(('', 80))` especifica que el socket es accesible desde cualquier dirección que tenga la máquina.

Algo más para señalar: los números de puerto bajos son normalmente reservados para servicios «conocidos» (HTTP, SNMP, etc.). Si estás probando los sockets usa un número grande (4 dígitos).

Finalmente, el argumento que se le pasa a `listen` le indica a la librería del socket que queremos poner en cola no más de 5 solicitudes de conexión (el máximo normal) antes de rechazar conexiones externas. Si el resto del código está escrito correctamente eso debería ser suficiente.

Ahora que tenemos un socket servidor escuchando en el puerto 80 ya podemos entrar al bucle principal del servidor web:

```
while True:
    # accept connections from outside
    (clientsocket, address) = serversocket.accept()
    # now do something with the clientsocket
    # in this case, we'll pretend this is a threaded server
    ct = client_thread(clientsocket)
    ct.run()
```

Existen en realidad 3 maneras generales en las cuales este bucle puede funcionar - despachar un hilo para manejar `clientsocket`, crear un proceso nuevo para manejar `clientsocket` o reestructurar esta aplicación para usar sockets no bloqueantes y multiplexar entre nuestro «socket servidor» y cualquier `clientsocket` activo usando `select`. Más sobre esto después. Lo importante a entender ahora es: esto es *todo* lo que un «socket servidor hace». No manda ningún dato. No recibe ningún dato. Solo produce «sockets clientes». Cada `clientsocket` es creado en respuesta a algún otro «socket cliente» que hace `connect()` al host y al puerto al que estamos vinculados. Tan pronto como hemos credo ese `clientsocket` volvemos a escuchar por más conexiones. Los dos «clientes» son libres de «conversar» entre ellos - están usando algún puerto asignado dinámicamente que será reciclado cuando la conversación termine.

2.1 IPC

Si necesitas conexiones IPC rápidas entre dos procesos en una misma máquina puedes revisar los *pipes* o la memoria compartida. Si decides usar sockets `AF_INET`, vincula el servidor con `"localhost"`. En la mayoría de las plataformas, esto tomará un atajo alrededor de algunas capas del código de red y será un poco más rápido.

Ver también:

El módulo `multiprocessing` integra IPC multiplataforma en un API de alto nivel.

3 Usando un socket

Lo primero a señalar es que el «socket cliente» del navegador y el «socket cliente» del servidor web son bestias idénticas. Es decir, esta es una conversación *peer to peer*. O para decirlo de otra manera, *como diseñador, tendrás que decidir cuáles son las reglas de etiqueta para una conversación*. Normalmente, el socket que se conecta inicia la conversación, enviando una solicitud o tal vez un inicio de sesión. Pero esa es una decisión de diseño: no es una regla de los sockets.

Hay dos conjuntos de verbos que se usan para la comunicación. Puedes usar `send` y `recv` o puedes transformar tu socket cliente en algo similar a un archivo y usar `read` y `write`. Esta última es la forma en la que Java presenta sus sockets. No voy a hablar acerca de eso aquí, excepto para advertirte que necesitas usar `flush` en los sockets. Estos son archivos en buffer, y un error común es usar `write` para escribir algo y luego usar `read` para leer la respuesta. Sin usar `flush` en este caso, puedes terminar esperando la respuesta por siempre porque la petición estaría aún en el buffer de salida.

Ahora llegamos al principal problema de los sockets - `send` y `recv` operan en los buffers de red. Ellos no manejan necesariamente todos los bytes que se les entrega (o espera de ellos), porque su enfoque principal es manejar los buffers de red. En general, ellos retornan cuando los buffers de red asociados se han llenado (`send`) o vaciado (`recv`). Luego ellos dicen cuántos bytes manejaron. Es *tu* responsabilidad llamarlos nuevamente hasta que su mensaje haya sido tratado por completo.

Cuando `recv` retorna 0 bytes significa que el otro lado ha cerrado (o está en el proceso de cerrar) la conexión. No recibirás más datos de esta conexión. Nunca. Es posible que puedas mandar datos exitosamente. De eso voy a hablar más tarde.

Un protocolo como HTTP usa un socket para una sola transferencia. El cliente manda una petición, luego lee la respuesta. Eso es todo. El socket es descartado. Esto significa que un cliente puede detectar el final de la respuesta al recibir 0 bytes.

Pero si planeas reusar el socket para más transferencias, tienes que darte cuenta que *no hay* EOT (End of Transfer) *en un socket.* Repito: si la llamada a `send` o `recv` de un socket retorna después de manejar 0 bytes, la conexión se ha interrumpido. Si la conexión *no* se ha interrumpido, puedes esperar un `recv` para siempre, porque el socket no te dirá cuando no hay más nada por leer (por ahora). Ahora, si piensas sobre eso un poco, te darás cuenta de una verdad fundamental de los sockets: *los mensajes deben ser de longitud fija* (ouch), *o ser delimitados* (ouch), *o indicar que tan largo son* (mucho mejor), *o terminar cerrando la conexión.* La elección es completamente tuya (pero hay algunas vías más correctas que otras).

Asumiendo que no quieres terminar la conexión, la solución más simple es un mensaje de longitud fija:

```
class MySocket:
    """demonstration class only
      - coded for clarity, not efficiency
    """

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
                            socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
        self.sock.connect((host, port))

    def mysend(self, msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError("socket connection broken")
            totalsent = totalsent + sent

    def myreceive(self):
        chunks = []
        bytes_recd = 0
        while bytes_recd < MSGLEN:
            chunk = self.sock.recv(min(MSGLEN - bytes_recd, 2048))
            if chunk == b'':
                raise RuntimeError("socket connection broken")
            chunks.append(chunk)
            bytes_recd = bytes_recd + len(chunk)
        return b''.join(chunks)
```

El código de envío aquí es usable para prácticamente cualquier esquema de mensajería - en Python envías cadenas y usas `len()` para determinar su longitud (incluso si tiene caracteres `\0` incrustados). Es principalmente el código receptor el que se vuelve más complejo. (Y en C no es mucho peor, excepto que no puedes usar `strlen` si el mensaje tiene `\0` incrustados).

La mejora más fácil es hacer que el primer caracter del mensaje un indicador del tipo de mensaje y que el tipo determine la longitud. Ahora tienes dos `recv` - el primero para obtener (al menos) ese primer caracter para conocer la longitud, y el segundo en un bucle para obtener el resto. Si decides ir por el camino del delimitador, estarás recibiendo un fragmento de tamaño arbitrario (4096 o 8192 son a menudo buenas elecciones para tamaños de buffers de red) y escaneando lo que recibas en busca del delimitador.

Hay una complicación de la que estar consiente: si el protocolo conversacional permite mandar múltiples mensajes con-

secutivos (sin ningún tipo de respuesta), y pasas a `recv` un tamaño de fragmento arbitrario poder terminar leyendo el inicio de un próximo mensaje. Tendrás que dejarlo aparte y guardarlo hasta que sea necesario.

Prefijar el mensaje con su longitud (por ejemplo, 5 caracteres numéricos) se vuelve más complicado porque (créalo o no), puede que no recibas los 5 caracteres en una llamada a `recv`. Para proyectos pequeños te saldrás con la tuya; pero con altas cargas de red, tu código se romperá rápidamente a menos que uses dos `recv` en bucle - el primero para determinar la longitud, el segundo para obtener la parte del mensaje. Sucio. También será cuando descubras que `send` no siempre logra enviar todo de una sola vez. Y a pesar de haber leído esto eventualmente te va a morder!

Con interés de espacio, la construcción de tu carácter (y preservar mi posición competitiva), estas mejoras se dejan como un ejercicio para el lector. Pasemos a la limpieza.

3.1 Datos binarios

It is perfectly possible to send binary data over a socket. The major problem is that not all machines use the same formats for binary data. For example, network byte order is big-endian, with the most significant byte first, so a 16 bit integer with the value `1` would be the two hex bytes `00 01`. However, most common processors (x86/AMD64, ARM, RISC-V), are little-endian, with the least significant byte first - that same `1` would be `01 00`.

Socket libraries have calls for converting 16 and 32 bit integers - `ntohl, htonl, ntohs, htons` where «n» means *network* and «h» means *host*, «s» means *short* and «l» means *long*. Where network order is host order, these do nothing, but where the machine is byte-reversed, these swap the bytes around appropriately.

In these days of 64-bit machines, the ASCII representation of binary data is frequently smaller than the binary representation. That's because a surprising amount of the time, most integers have the value 0, or maybe 1. The string `"0"` would be two bytes, while a full 64-bit integer would be 8. Of course, this doesn't fit well with fixed-length messages. Decisions, decisions.

4 Desconectando

Estrictamente hablando, se supone que debes usar `shutdown` en un socket antes de cerrarlo con `close`. `shutdown` es un aviso para el socket en el otro lado. Dependiendo del argumento que se le pase, puede significar «No voy a mandar más datos, pero voy a escuchar» o «No estoy escuchando, adiós!». La mayoría de bibliotecas para sockets, sin embargo, están tan acostumbradas a que los programadores ignoren esta parte de la etiqueta que normalmente `close` es lo mismo que `shutdown(); close()`. Por tanto en la mayoría de las situaciones usar `shutdown` de manera explícita no es necesario.

Una forma de usar `shutdown` de manera efectiva es en un intercambio similar a *HTTP*. El cliente manda una petición y entonces hace un `shutdown(1)`. Esto le dice al servidor «El cliente terminó de enviar, pero todavía puede recibir». El servidor puede detectar «EOF» (Fin del Archivo) al recibir 0 bytes. Puede asumir que se completó la petición. El servidor envía una respuesta. Si el `send` termina satisfactoriamente entonces, en efecto, el cliente todavía estaba recibiendo.

Python lleva el apagado automático un paso más allá, y dice que cuando un socket es eliminado por el recolector de basura, automáticamente llama a `close` si es necesario. Pero confiar en esto es un mal hábito. Si tu socket simplemente desaparece sin llamar a `close`, el socket del otro lado puede colgarse indefinidamente, pensando que solo estas siendo lento. *Por favor* cierra los sockets cuando termines.

4.1 Cuando los sockets mueren

Probablemente lo peor de usar sockets bloqueantes es lo que pasa cuando el otro lado se apaga inesperadamente (sin llamar a `close`). Tu socket es probable que se cuelgue. TCP es un protocolo confiable, y va a esperar un largo, largo tiempo antes de rendirse con una conexión. Si estás usando hilos, todo el hilo está esencialmente muerto. No hay mucho que puedas hacer respecto a eso. A menos que no estés haciendo algo tonto, como mantener un bloqueo mientras se realiza una lectura bloqueante, el hilo realmente no estará consumiendo muchos recursos. *No* trates de matar el hilo - parte de la razón por la que los hilos son más eficientes que los procesos es que evitan la complicación asociada con el reciclaje automático de recursos. En otras palabras, si te las arreglas para matar el hilo, es muy probable que todo el proceso termine arruinado.

5 *Sockets* no bloqueantes

Si has entendido todo lo anterior, ya conoces la mayor parte de lo que necesitas saber sobre las mecánicas del uso de los sockets. Usarás las mismas llamadas, de la misma manera. Es solo eso, si lo haces correctamente, tu aplicación estará casi correcta.

En Python, usa `socket.setblocking(False)` para que no sea bloqueante. En C, es más complejo (por un lado, deberá elegir entre el sabor BSD `O_NONBLOCK` y el sabor POSIX casi indistinguible `O_NDELAY`, que es completamente diferente de `TCP_NODELAY`) , pero es exactamente la misma idea. Haz esto después de crear el socket, pero antes de usarlo. (En realidad, si estás loco, puedes cambiar de un lado a otro).

La principal diferencia mecánica es que `send`, `recv`, `connect` y `accept` pueden retornar sin haber hecho nada. Tu tienes (por supuesto) un número de elecciones. Puedes verificar el código de retorno y los códigos de error y en general volverte loco. Si no me crees pruébalo alguna vez. Tu aplicación crecerá grande, con errores y consumirá todo el CPU. Así que vamos a saltarnos las soluciones descerebradas y hacerlo correctamente.

Usando `select`.

En C, usar `select` es algo complejo. En Python es pan comido, pero está lo suficientemente cercano a la versión de C que si entiendes el `select` en Python tendrás pocos problemas con él el C:

```
ready_to_read, ready_to_write, in_error = \
               select.select(
                  potential_readers,
                  potential_writers,
                  potential_errs,
                  timeout)
```

A `select` se le pasan tres listas: la primera contiene todos los sockets que puedes intentar leer; la segunda con todos los sockets que puedes intentar escribir, y la tercera (normalmente se deja vacía) todos los que quieras verificar los errores. Debes tener en cuenta que un socket puede ir en más de una lista. La llamada a `select` es bloqueante, pero puedes darle un tiempo de espera. Esto generalmente es una cosa sensata de hacer - dale un tiempo de espera largo (un minuto por ejemplo) a menos que tengas una buena razón para no hacerlo.

En el retorno tendrás tres listas. Estas contienen los sockets que son realmente legibles, escribibles y con error. Cada una de estas lista es un subconjunto (posiblemente vacío) de la lista correspondiente que pasaste.

Si un socket está en la lista retornada legible, puedes estar tan-seguro-como-podrías-estarlo-en-este-negocio que una llamada a `recv` en este socket va a retornar *algo*. La misma idea se aplica a la lista de escribibles. Serás capaz de mandar *algo*. Tal vez no todo lo que quieras, pero *algo* es mejor que nada. (Realmente, cualquier socket socket razonablemente saludable va a retornar como escribible - eso solo significa que el espacio de salida del buffer de red está disponible)

Si tienes un socket *servidor*, ponlo en la lista de *potenciales legibles*. Se retorna en la lista de legibles, una llamada a `accept` va a funcionar (casi seguro). Se has creado un nuevo socket para llamar a `connect` para conectarte con otro,

ponlo en la lista de *potenciales escribibles*. Si retorna en la lista de escribibles, tienes una buena oportunidad de que esté conectado.

Realmente, `select` puede ser útil incluso con sockets bloqueantes. Es una manera de determinar si vas a bloquear - el socket retorna como leíble cuando hay algo en el buffer. Sin embargo, esto aun no sirve de ayuda con el problema de determinar si el otro extremo terminó, o solo está ocupado con otra cosa.

Alerta de portabilidad: En Unix, `select` funciona tanto con sockets como con archivos. No intentes esto en Windows. En Windows `select` funciona solo con sockets. También ten en cuenta que en C, muchas de las opciones más avanzadas de los sockets se hacen diferentes en Windows. De hecho, en Windows normalmente uso hilos (que funciona muy, muy bien) con los sockets.