Socket Programming HOWTO

Version 2.7.16

Guido van Rossum
and the Python development team

octobre 07, 2019

Python Software Foundation
Email : docs@python.org

Table des matières

1 Interfaces de connexion (*sockets*) **2**
1.1 Historique 2

2 Créer un *socket* **2**
2.1 Communication Entre Processus 3

3 Using a Socket **3**
3.1 Données binaires 5

4 Déconnexion **5**
4.1 When Sockets Die 5

5 Non-blocking Sockets **6**
5.1 Performance 7

Author Gordon McMillan

Résumé

Sockets are used nearly everywhere, but are one of the most severely misunderstood technologies around. This is a 10,000 foot overview of sockets. It's not really a tutorial - you'll still have work to do in getting things operational. It doesn't cover the fine points (and there are a lot of them), but I hope it will give you enough background to begin using them decently.

1 Interfaces de connexion (*sockets*)

I'm only going to talk about INET sockets, but they account for at least 99% of the sockets in use. And I'll only talk about STREAM sockets - unless you really know what you're doing (in which case this HOWTO isn't for you !), you'll get better behavior and performance from a STREAM socket than anything else. I will try to clear up the mystery of what a socket is, as well as some hints on how to work with blocking and non-blocking sockets. But I'll start by talking about blocking sockets. You'll need to know how they work before dealing with non-blocking sockets.

Part of the trouble with understanding these things is that « socket » can mean a number of subtly different things, depending on context. So first, let's make a distinction between a « client » socket - an endpoint of a conversation, and a « server » socket, which is more like a switchboard operator. The client application (your browser, for example) uses « client » sockets exclusively ; the web server it's talking to uses both « server » sockets and « client » sockets.

1.1 Historique

Of the various forms of IPC (Inter Process Communication), sockets are by far the most popular. On any given platform, there are likely to be other forms of IPC that are faster, but for cross-platform communication, sockets are about the only game in town.

They were invented in Berkeley as part of the BSD flavor of Unix. They spread like wildfire with the Internet. With good reason — the combination of sockets with INET makes talking to arbitrary machines around the world unbelievably easy (at least compared to other schemes).

2 Créer un *socket*

Roughly speaking, when you clicked on the link that brought you to this page, your browser did something like the following :

```
#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.mcmillan-inc.com", 80))
```

When the `connect` completes, the socket `s` can be used to send in a request for the text of the page. The same socket will read the reply, and then be destroyed. That's right, destroyed. Client sockets are normally only used for one exchange (or a small set of sequential exchanges).

What happens in the web server is a bit more complex. First, the web server creates a « server socket » :

```
#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)
```

Quelques remarques : nous avons utilisé `socket.gethostname()` pour que le connecteur soit visible par le monde extérieur. Si nous avions utilisé `s.bind(('localhost', 80))` ou `s.bind(('127.0.0.0.1', 80))` nous aurions encore un connecteur « serveur », mais qui ne serait visible que sur la machine même. `s.bind('', 80)]` spécifie que le socket est accessible par toute adresse que la machine possède.

Une deuxième chose à noter : les ports dont le numéro est petit sont généralement réservés aux services « bien connus » (HTTP, SNMP, etc.). Si vous expérimentez, utilisez un nombre suffisamment élevé (4 chiffres).

Enfin, l'argument `listen` indique à la bibliothèque de connecteurs que nous voulons qu'elle mette en file d'attente jusqu'à 5 requêtes de connexion (le maximum normal) avant de refuser les connexions externes. Si le reste du code est écrit correctement, cela devrait suffire.

Now that we have a « server » socket, listening on port 80, we can enter the mainloop of the web server :

```
while 1:
    #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()
```

There's actually 3 general ways in which this loop could work - dispatching a thread to handle `clientsocket`, create a new process to handle `clientsocket`, or restructure this app to use non-blocking sockets, and multiplex between our « server » socket and any active `clientsockets` using `select`. More about that later. The important thing to understand now is this : this is *all* a « server » socket does. It doesn't send any data. It doesn't receive any data. It just produces « client » sockets. Each `clientsocket` is created in response to some *other* « client » socket doing a `connect()` to the host and port we're bound to. As soon as we've created that `clientsocket`, we go back to listening for more connections. The two « clients » are free to chat it up - they are using some dynamically allocated port which will be recycled when the conversation ends.

2.1 Communication Entre Processus

If you need fast IPC between two processes on one machine, you should look into whatever form of shared memory the platform offers. A simple protocol based around shared memory and locks or semaphores is by far the fastest technique.

If you do decide to use sockets, bind the « server » socket to `'localhost'`. On most platforms, this will take a shortcut around a couple of layers of network code and be quite a bit faster.

3 Using a Socket

The first thing to note, is that the web browser's « client » socket and the web server's « client » socket are identical beasts. That is, this is a « peer to peer » conversation. Or to put it another way, *as the designer, you will have to decide what the rules of etiquette are for a conversation*. Normally, the `connect`ing socket starts the conversation, by sending in a request, or perhaps a signon. But that's a design decision - it's not a rule of sockets.

Now there are two sets of verbs to use for communication. You can use `send` and `recv`, or you can transform your client socket into a file-like beast and use `read` and `write`. The latter is the way Java presents its sockets. I'm not going to talk about it here, except to warn you that you need to use `flush` on sockets. These are buffered « files », and a common mistake is to `write` something, and then `read` for a reply. Without a `flush` in there, you may wait forever for the reply, because the request may still be in your output buffer.

Now we come to the major stumbling block of sockets - `send` and `recv` operate on the network buffers. They do not necessarily handle all the bytes you hand them (or expect from them), because their major focus is handling the network buffers. In general, they return when the associated network buffers have been filled (`send`) or emptied (`recv`). They then tell you how many bytes they handled. It is *your* responsibility to call them again until your message has been completely dealt with.

When a `recv` returns 0 bytes, it means the other side has closed (or is in the process of closing) the connection. You will not receive any more data on this connection. Ever. You may be able to send data successfully ; I'll talk more about this later.

Un protocole comme HTTP utilise un connecteur pour un seul transfert. Le client envoie une demande, puis lit une réponse. C'est tout. Le connecteur est mis au rebut. Cela signifie qu'un client peut détecter la fin de la réponse en recevant 0 octet.

Mais si vous prévoyez de réutiliser votre connecteur pour d'autres transferts, vous devez réaliser que il n'y a *pas* d"EOT (End of Transfer) sur un connecteur. Je répète : si un connecteur `send` ou `recv` retourne après avoir manipulé 0 octets, la connexion a été interrompue. Si la connexion n'a *pas* été interrompue, vous pouvez attendre sur un `recv` pour toujours, car le connecteur ne vous dira pas qu'il n'y a plus rien à lire (pour le moment). Maintenant, si vous y réfléchissez un peu, vous allez vous rendre compte d'une vérité fondamentale sur les connecteurs : *les messages doivent être de longueur fixe* (beurk), *ou être délimités* (haussement d'épaules), *ou indiquer de quelle longueur ils sont* (beaucoup mieux), *ou terminer en coupant la connexion*. Le choix est entièrement de votre côté, (mais certaines façons sont plus justes que d'autres).

En supposant que vous ne vouliez pas terminer la connexion, la solution la plus simple est un message de longueur fixe :

```
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 == '':
                raise RuntimeError("socket connection broken")
            chunks.append(chunk)
            bytes_recd = bytes_recd + len(chunk)
        return ''.join(chunks)
```

The sending code here is usable for almost any messaging scheme - in Python you send strings, and you can use `len()` to determine its length (even if it has embedded `\0` characters). It's mostly the receiving code that gets more complex. (And in C, it's not much worse, except you can't use `strlen` if the message has embedded `\0`s.)

The easiest enhancement is to make the first character of the message an indicator of message type, and have the type determine the length. Now you have two `recv`s - the first to get (at least) that first character so you can look up the length, and the second in a loop to get the rest. If you decide to go the delimited route, you'll be receiving in some arbitrary chunk

size, (4096 or 8192 is frequently a good match for network buffer sizes), and scanning what you've received for a delimiter.

One complication to be aware of : if your conversational protocol allows multiple messages to be sent back to back (without some kind of reply), and you pass `recv` an arbitrary chunk size, you may end up reading the start of a following message. You'll need to put that aside and hold onto it, until it's needed.

Prefixing the message with its length (say, as 5 numeric characters) gets more complex, because (believe it or not), you may not get all 5 characters in one `recv`. In playing around, you'll get away with it ; but in high network loads, your code will very quickly break unless you use two `recv` loops - the first to determine the length, the second to get the data part of the message. Nasty. This is also when you'll discover that `send` does not always manage to get rid of everything in one pass. And despite having read this, you will eventually get bit by it !

In the interests of space, building your character, (and preserving my competitive position), these enhancements are left as an exercise for the reader. Lets move on to cleaning up.

3.1 Données binaires

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, a Motorola chip will represent a 16 bit integer with the value 1 as the two hex bytes 00 01. Intel and DEC, however, are byte-reversed - that same 1 is 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 32 bit machines, the ascii representation of binary data is frequently smaller than the binary representation. That's because a surprising amount of the time, all those longs have the value 0, or maybe 1. The string « 0 » would be two bytes, while binary is four. Of course, this doesn't fit well with fixed-length messages. Decisions, decisions.

4 Déconnexion

Strictly speaking, you're supposed to use `shutdown` on a socket before you `close` it. The `shutdown` is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean « I'm not going to send anymore, but I'll still listen », or « I'm not listening, good riddance ! ». Most socket libraries, however, are so used to programmers neglecting to use this piece of etiquette that normally a `close` is the same as `shutdown(); close()`. So in most situations, an explicit `shutdown` is not needed.

One way to use `shutdown` effectively is in an HTTP-like exchange. The client sends a request and then does a `shutdown(1)`. This tells the server « This client is done sending, but can still receive. » The server can detect « EOF » by a receive of 0 bytes. It can assume it has the complete request. The server sends a reply. If the `send` completes successfully then, indeed, the client was still receiving.

Python takes the automatic shutdown a step further, and says that when a socket is garbage collected, it will automatically do a `close` if it's needed. But relying on this is a very bad habit. If your socket just disappears without doing a `close`, the socket at the other end may hang indefinitely, thinking you're just being slow. *Please* `close` your sockets when you're done.

4.1 When Sockets Die

Probably the worst thing about using blocking sockets is what happens when the other side comes down hard (without doing a `close`). Your socket is likely to hang. SOCKSTREAM is a reliable protocol, and it will wait a long, long time before giving up on a connection. If you're using threads, the entire thread is essentially dead. There's not much you can do about it. As long as you aren't doing something dumb, like holding a lock while doing a blocking read, the thread isn't really consuming much in the way of resources. Do *not* try to kill the thread - part of the reason that threads are

more efficient than processes is that they avoid the overhead associated with the automatic recycling of resources. In other words, if you do manage to kill the thread, your whole process is likely to be screwed up.

5 Non-blocking Sockets

Si vous avez compris ce qui précède, vous savez déjà tout ce que vous devez savoir sur la mécanique de l'utilisation des connecteurs. Vous utiliserez toujours les mêmes appels, de la même façon. C'est juste que, si vous le faites bien, votre application sera presque dans la poche.

En Python, vous utilisez `socket.setblocking(0)` pour le rendre non-bloquant. En C, c'est plus complexe, (pour commencer, vous devez choisir entre la version BSD `O_NONBLOCK` et la version Posix presque impossible à distinguer `O_NDELAY`, qui est complètement différente de `TCP_NODELAY`), mais c'est exactement la même idée. Vous le faites après avoir créé le connecteur mais avant de l'utiliser (en fait, si vous êtes fou, vous pouvez alterner).

The major mechanical difference is that `send`, `recv`, `connect` and `accept` can return without having done anything. You have (of course) a number of choices. You can check return code and error codes and generally drive yourself crazy. If you don't believe me, try it sometime. Your app will grow large, buggy and suck CPU. So let's skip the brain-dead solutions and do it right.

Utiliser `select`.

En C, implémenter `select` est assez complexe. En Python, c'est du gâteau, mais c'est assez proche de la version C ; aussi, si vous comprenez `select` en Python, vous aurez peu de problèmes avec lui en C :

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

You pass `select` three lists : the first contains all sockets that you might want to try reading ; the second all the sockets you might want to try writing to, and the last (normally left empty) those that you want to check for errors. You should note that a socket can go into more than one list. The `select` call is blocking, but you can give it a timeout. This is generally a sensible thing to do - give it a nice long timeout (say a minute) unless you have good reason to do otherwise.

In return, you will get three lists. They contain the sockets that are actually readable, writable and in error. Each of these lists is a subset (possibly empty) of the corresponding list you passed in.

Si un connecteur se trouve dans la liste des sorties que vous pouvez lire, vous pouvez être pratiquement certain qu'un `recv` sur ce connecteur retournera *quelque chose*. Même chose pour la liste des sorties sur lesquelles vous pouvez écrire. Vous pourrez envoyer *quelque chose*. Peut-être pas tout ce que vous voudrez, mais *quelque chose* est mieux que rien. (En fait, n'importe quel connecteur raisonnablement sain retournera en écriture — cela signifie simplement que l'espace tampon réseau sortant est disponible).

Si vous avez un connecteur « serveur », mettez-le dans la liste des lecteurs potentiels. Si il apparaît dans la liste des sorties que vous pouvez lire, votre `accept` fonctionnera (presque certainement). Si vous avez créé un nouveau connecteur pour `connect` à quelqu'un d'autre, mettez-le dans la liste des éditeurs potentiels. Si il apparaît dans la liste des sorties sur lesquelles vous pouvez écrire, vous avez une bonne chance qu'il se soit connecté.

One very nasty problem with `select` : if somewhere in those input lists of sockets is one which has died a nasty death, the `select` will fail. You then need to loop through every single damn socket in all those lists and do a `select([sock], [],[],0)` until you find the bad one. That timeout of 0 means it won't take long, but it's ugly.

En fait, `select` peut être pratique même avec des connecteurs bloquants. C'est une façon de déterminer si vous allez bloquer — le socket redevient lisible lorsqu'il y a quelque chose dans les tampons. Cependant, cela n'aide pas encore à déterminer si l'autre extrémité a terminé, ou si elle est simplement occupée par autre chose.

Portability alert : On Unix, `select` works both with the sockets and files. Don't try this on Windows. On Windows, `select` works with sockets only. Also note that in C, many of the more advanced socket options are done differently on Windows. In fact, on Windows I usually use threads (which work very, very well) with my sockets. Face it, if you want any kind of performance, your code will look very different on Windows than on Unix.

5.1 Performance

There's no question that the fastest sockets code uses non-blocking sockets and select to multiplex them. You can put together something that will saturate a LAN connection without putting any strain on the CPU. The trouble is that an app written this way can't do much of anything else - it needs to be ready to shuffle bytes around at all times.

Assuming that your app is actually supposed to do something more than that, threading is the optimal solution, (and using non-blocking sockets will be faster than using blocking sockets). Unfortunately, threading support in Unixes varies both in API and quality. So the normal Unix solution is to fork a subprocess to deal with each connection. The overhead for this is significant (and don't do this on Windows - the overhead of process creation is enormous there). It also means that unless each subprocess is completely independent, you'll need to use another form of IPC, say a pipe, or shared memory and semaphores, to communicate between the parent and child processes.

Finally, remember that even though blocking sockets are somewhat slower than non-blocking, in many cases they are the « right » solution. After all, if your app is driven by the data it receives over a socket, there's not much sense in complicating the logic just so your app can wait on `select` instead of `recv`.