Guide pratique : programmation avec les sockets

Version 3.6.15

Guido van Rossum
and the Python development team

septembre 05, 2021

Python Software Foundation
Email : docs@python.org

Table des matières

Auteur Gordon McMillan

Résumé

Les connecteurs (*sockets*, en anglais) sont utilisés presque partout, mais ils sont l'une des technologies les plus méconnues. En voici un aperçu très général. Ce n'est pas vraiment un tutoriel — vous aurez encore du travail à faire pour avoir un résultat opérationnel. Il ne couvre pas les détails (et il y en a beaucoup), mais j'espère qu'il vous donnera suffisamment d'informations pour commencer à les utiliser correctement.

1 Interfaces de connexion (*sockets*)

Je ne vais aborder que les connecteurs INET (i.e. IPv4), mais ils représentent au moins 99% des connecteurs (*socket* en anglais) utilisés. Et je n'aborderai que les connecteurs STREAM (i.e. TCP) — à moins que vous ne sachiez vraiment ce que vous faites (auquel cas ce HOWTO n'est pas pour vous !), vous obtiendrez un meilleur comportement et de meilleures performances avec un connecteur STREAM que tout autre. Je vais essayer d'éclaircir le mystère de ce qu'est un connecteur, ainsi que quelques conseils sur la façon de travailler avec des connecteurs bloquants et non bloquants. Mais je vais commencer par aborder les connecteurs bloquants. Nous avons besoin de savoir comment ils fonctionnent avant de traiter les connecteurs non bloquants.

Une partie de la difficulté à comprendre ces choses est que « connecteur » peut désigner plusieurs choses très légèrement différentes, selon le contexte. Faisons donc d'abord une distinction entre un connecteur « client » — point final d'une conversation — et un connecteur « serveur », qui ressemble davantage à un standardiste. L'application cliente (votre navigateur par exemple) utilise exclusivement des connecteurs « client » ; le serveur web avec lequel elle parle utilise à la fois des connecteurs « serveur » et des connecteurs « client ».

1.1 Historique

Parmi les différentes formes d'IPC (Inter Process Communication), les connecteurs sont de loin les plus populaires. Sur une plate-forme donnée, il est probable que d'autres formes d'*IPC* soient plus rapides, mais pour la communication entre plates-formes, les connecteurs sont à peu près la seule solution valable.

Ils ont été inventés à Berkeley dans le cadre de la déclinaison *BSD* d'Unix. Ils se sont répandus comme une traînée de poudre avec Internet. Et pour cause : la combinaison des connecteurs avec *INET* rend le dialogue avec n'importe quelle machine dans le monde entier incroyablement facile (du moins par rapport à d'autres systèmes).

2 Créer un connecteur

Grosso modo, lorsque vous avez cliqué sur le lien qui vous a amené à cette page, votre navigateur a fait quelque chose comme ceci :

```
# 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))
```

Lorsque l'appel à `connect` est terminé, le connecteur `s` peut être utilisé pour envoyer une requête demandant le texte de la page. Le même connecteur lira la réponse, puis sera mis au rebut. C'est exact, mis au rebut. Les connecteurs clients ne sont normalement utilisés que pour un seul échange (ou un petit ensemble d'échanges séquentiels).

Ce qui se passe dans le serveur web est un peu plus complexe. Tout d'abord, le serveur web crée un « connecteur serveur » :

```
# 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(('127.0.0.1', 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.

Maintenant que nous avons un connecteur « serveur », en écoute sur le port 80, nous pouvons entrer dans la boucle principale du serveur 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()
```

Il y a en fait trois façons générales de faire fonctionner cette boucle : mobiliser un fil d'exécution pour gérer les `clientsocket`s, créer un nouveau processus pour gérer les `clientsocket`s, ou restructurer cette application pour utiliser des connecteurs non bloquants, et multiplexer entre notre connecteur « serveur » et n'importe quel `clientsocket` actif en utilisant `select`. Plus d'informations à ce sujet plus tard. La chose importante à comprendre maintenant est la suivante : c'est *tout* ce que fait un connecteur « serveur ». Il n'envoie aucune donnée. Il ne reçoit aucune donnée. Il ne fait que produire des connecteurs « clients ». Chaque `clientsocket` est créé en réponse à un *autre* connecteur « client » qui se connecte à l'hôte et au port auxquels nous sommes liés. Dès que nous avons créé ce `clientsocket`, nous retournons à l'écoute pour d'autres connexions. Les deux « clients » sont libres de discuter — ils utilisent un port alloué dynamiquement qui sera recyclé à la fin de la conversation.

2.1 Communication Entre Processus

Si vous avez besoin d'une communication rapide entre deux processus sur une même machine, vous devriez regarder comment utiliser les *pipes* ou la mémoire partagée. Si vous décidez d'utiliser les connecteurs AF_INET, liez le connecteur « serveur » à `'localhost'`. Sur la plupart des plates-formes, cela court-circuite quelques couches réseau et est un peu plus rapide.

Voir aussi :

Le `multiprocessing` intègre de l'IPC multiplateforme dans une API de plus haut niveau.

3 Utilisation d'un connecteur

La première chose à noter, c'est que la prise « client » du navigateur web et la prise « client » du serveur web sont des bêtes identiques. C'est-à-dire qu'il s'agit d'une conversation « pair à pair ». Ou pour le dire autrement, *en tant que concepteur, vous devrez décider quelles sont les règles d'étiquette pour une conversation*. Normalement, la connexion via `connect` lance la conversation en envoyant une demande, ou peut-être un signe. Mais c'est une décision de conception — ce n'est pas une règle des connecteurs.

Il y a maintenant deux ensembles de verbes à utiliser pour la communication. Vous pouvez utiliser `send` et `recv`, ou vous pouvez transformer votre connecteur client en une bête imitant un fichier et utiliser `read` et `write`. C'est la façon dont Java présente ses connecteurs. Je ne vais pas en parler ici, sauf pour vous avertir que vous devez utiliser `flush` sur les connecteurs. Ce sont des « fichiers », mis en mémoire tampon, et une erreur courante est d'"« écrire » via `write` quelque chose, puis de « lire » via `read` pour obtenir une réponse. Sans un `flush`, vous pouvez attendre la réponse pour toujours, parce que la requête peut encore être dans votre mémoire tampon de sortie.

Nous arrivons maintenant au principal écueil des connecteurs — `send` et `recv` fonctionnent sur les mémoires tampons du réseau. Ils ne traitent pas nécessairement tous les octets que vous leur passez (ou que vous attendez d'eux), car leur principal objectif est de gérer les tampons réseau. En général, leur exécution se termine lorsque les tampons réseau associés ont été remplis (`send`) ou vidés (`recv`). Ils vous indiquent alors combien d'octets ils ont traité. Il est de *votre* responsabilité de les rappeler jusqu'à ce que votre message ait été complètement traité.

Lorsqu'un `recv` renvoie 0 octet, cela signifie que l'autre partie a fermé (ou est en train de fermer) la connexion. Vous ne recevrez plus de données sur cette connexion. Jamais. Vous pouvez peut-être envoyer des données avec succès. J'en parlerai plus tard.

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

Le code d'envoi ici est utilisable pour presque tous les systèmes de messagerie — en Python, vous envoyez des chaînes

de caractères, et vous pouvez utiliser `len()` pour en déterminer la longueur (même si elle contient des caractères `\0`). C'est surtout le code de réception qui devient plus complexe. (Et en C, ce n'est pas bien pire, sauf que vous ne pouvez pas utiliser `strlen` si le message contient des `\0`s).

Le plus simple est de faire du premier caractère du message un indicateur du type de message, et de faire en sorte que le type détermine la longueur. Vous avez maintenant deux `recv`s — le premier pour obtenir (au moins) ce premier caractère afin de pouvoir déterminer la longueur, et le second dans une boucle pour obtenir le reste. Si vous décidez de suivre la route délimitée, vous recevrez un morceau de taille arbitraire (4096 ou 8192 est fréquemment une bonne valeur pour correspondre à la taille de la mémoire tampon du réseau), et vous analyserez ce que vous avez reçu pour trouver un délimiteur.

Une subtilité dont il faut être conscient : si votre protocole de conversation permet de renvoyer plusieurs messages les uns à la suite des autres (sans aucune sorte de réponse), et que vous passez à `recv` une taille de morceau arbitraire, vous pouvez en arriver à lire le début du message suivant. Vous devrez alors le mettre de côté et le conserver, jusqu'à ce que vous en ayez besoin.

Préfixer le message avec sa longueur (disons, sous la forme de 5 caractères numériques) devient plus complexe, parce que (croyez-le ou non), vous pouvez ne pas recevoir les 5 caractères en un seul `recv`. Pour une utilisation triviale, vous vous en tirerez à bon compte ; mais en cas de forte charge réseau, votre code se cassera très rapidement, à moins que vous n'utilisiez deux boucles `recv` — la première pour déterminer la longueur, la deuxième pour obtenir la partie « données » du message. Vilain. C'est aussi à ce moment que vous découvrirez que « l'envoi » via `send` ne parvient pas toujours à tout évacuer en un seul passage. Et bien que vous ayez lu cet avertissement, vous finirez par vous faire avoir par cette subtilité !

Pour garder une longueur raisonnable à cette page, pour forger votre caractère (et afin de garder l'avantage concurrentiel que j'ai sur vous), ces améliorations ne seront pas abordées et sont laissées en exercice au lecteur. Passons maintenant au nettoyage.

3.1 Données binaires

Il est parfaitement possible d'envoyer des données binaires sur un connecteur. Le gros problème est que toutes les machines n'utilisent pas les mêmes formats pour les données binaires. Par exemple, une puce Motorola code l'entier 1, sous 16 bits, comme les deux octets hexadécimaux 00 01. Intel et DEC, cependant, utilisent l'ordre d'octets inverse — ce même 1 est codé 01 00. Les bibliothèques de connecteurs ont des appels pour convertir des entiers de 16 et 32 bits — `ntohl, htonl, ntohs, htons` où `n` signifie *réseau* (*network*, en anglais) et `h` signifie *hôte*, `s` signifie *court* (*short*, en anglais) et `l` signifie *long*. Lorsque l'ordre du réseau est l'ordre de l'hôte, ceux-ci ne font rien, mais lorsque la machine utilise l'ordre d'octets inverse, ceux-ci échangent les octets de manière appropriée.

De nos jours, avec les machines 32 bits, la représentation *ASCII* des données binaires est souvent plus compacte que la représentation binaire. C'est parce qu'un nombre surprenant de fois, tous ces *longs* ont la valeur 0, ou peut-être 1. La chaîne « 0 » serait codée sur deux octets, alors qu'elle le serait sur quatre en binaire. Bien sûr, cela ne fonctionne pas très bien avec les messages de longueur fixe. Ah, les décisions, les décisions…

4 Déconnexion

À proprement parler, vous êtes censé utiliser `shutdown` sur un connecteur pour l'arrêter avant de le fermer via `close`. Le `shutdown` est un avertissement au connecteur de l'autre côté. Selon l'argument que vous lui passez, cela peut signifier « Je ne vais plus envoyer, mais je vais quand même écouter », ou « Je n'écoute pas, bon débarras ! ». La plupart des bibliothèques de connecteurs, cependant, sont tellement habituées à ce que les programmeurs négligent d'utiliser ce morceau d'étiquette que normalement un `close` est équivalent à `shutdown() ; close()`. Ainsi, dans la plupart des situations, un `shutdown` explicite n'est pas nécessaire.

Une façon d'utiliser efficacement le `shutdown` est d'utiliser un échange de type HTTP. Le client envoie une requête et effectue ensuite un `shutdown(1)`. Cela indique au serveur que « ce client a fini d'envoyer, mais peut encore recevoir ».

Le serveur peut détecter *EOF* par une réception de 0 octet. Il peut supposer qu'il a la requête complète. Le serveur envoie une réponse. Si le `send` se termine avec succès, alors, en effet, le client était encore en train de recevoir.

Python pousse l'arrêt automatique un peu plus loin, et dit que lorsqu'un connecteur est collecté par le ramasse-miette, il effectue automatiquement une fermeture via `close` si elle est nécessaire. Mais c'est une très mauvaise habitude de s'appuyer sur ce système. Si votre connecteur disparaît sans avoir fait un `close`, le connecteur à l'autre bout peut rester suspendu indéfiniment, pensant que vous êtes juste lent. Fermez vos connecteurs quand vous avez terminé *s'il vous plait*.

4.1 Quand les connecteurs meurent

Le pire dans l'utilisation de connecteurs bloquants est probablement ce qui se passe lorsque l'autre côté s'interrompt brutalement (sans faire de fermeture via `close`). Votre connecteur risque d'attendre infiniment. TCP est un protocole fiable, et il attendra très, très longtemps avant d'abandonner une connexion. Si vous utilisez des fils d'exécution, le fil entier est pratiquement mort. Il n'y a pas grand-chose que vous puissiez faire à ce sujet. Du moment que vous ne faites rien de stupide, comme tenir un verrou verrouillé pendant une lecture bloquante, le fil ne consomme pas vraiment beaucoup de ressources. N'essayez *pas* de tuer le fil — si les fils sont plus efficients que les processus, c'est en partie parce qu'ils évitent les coûts significatifs liés au recyclage automatique des ressources. En d'autres termes, si vous parvenez à tuer le fil, tout votre processus risque d'être foutu.

5 Connecteurs non bloquants

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).

La différence majeure de fonctionnement est que *send*', `recv`, `connect` et `accept` peuvent rendre la main sans avoir rien fait. Vous avez (bien sûr) un certain nombre de choix. Vous pouvez vérifier le code de retour et les codes d'erreur et, en général, devenir fou. Si vous ne me croyez pas, essayez un jour. Votre application va grossir, boguer et vampiriser le processeur. Alors, évitons les solutions vouées à l'échec dès le départ et faisons les choses correctement.

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)
```

Vous passez à `select` trois listes : la première contient tous les connecteurs dont vous souhaiter lire le contenu ; la deuxième tous les connecteurs sur lesquels vous voudriez écrire, et la dernière (normalement laissée vide) ceux sur lesquels vous voudriez vérifier s'il y a des erreurs. Prenez note qu'un connecteur peut figurer dans plus d'une liste. L'appel à `select` est bloquant, mais vous pouvez lui donner un délai d'attente. C'est généralement une bonne chose à faire — donnez-lui un bon gros délai d'attente (disons une minute), à moins que vous n'ayez une bonne raison de ne pas le faire.

En retour, vous recevrez trois listes. Elles contiennent les connecteurs qui sont réellement lisibles, inscriptibles et en erreur. Chacune de ces listes est un sous-ensemble (éventuellement vide) de la liste correspondante que vous avez transmise.

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.

En fait, `select` peut être pratique même avec des connecteurs bloquants. C'est une façon de déterminer si vous allez bloquer — le connecteur 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.

Alerte de portabilité : Sous Unix, `select` fonctionne aussi bien avec les connecteurs qu'avec les fichiers. N'essayez pas cela sous Windows. Sous Windows, `select` ne fonctionne qu'avec les connecteurs. Notez également qu'en C, la plupart des options de connecteurs les plus avancées se font différemment sous Windows. En fait, sous Windows, j'utilise habituellement des fils d'exécution (qui fonctionnent très, très bien) avec mes connecteurs.