Python_Socket

jeudi 19 janvier 2023 09:54

Formateur : Loup FORMENT

https://docs.python.org/fr/3/howto/sockets.html

https://python.doctor/page-reseaux-sockets-python-port

https://www.digitalocean.com/community/tutorials/python-socket-programming-server-client

Guide pratique : programmation avec les *sockets*

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.

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 ».

Création d'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 toujours un connecteur « serveur », mais qui ne serait visible qu'à l'intérieur de la même machine. `s.bind(('', 80))` précise que le connecteur est accessible par n'importe quelle 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 passé à `listen` indique à la bibliothèque de connecteurs que nous voulons mettre 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.

IPC (Communication Entre Processus)

Si vous avez besoin d'une communication rapide entre deux processus sur une même machine, vous devriez regarder comment utiliser les tubes (*pipe*, en anglais) 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 contourne quelques couches réseau et est un peu plus rapide.

> **Voir aussi:** Le `multiprocessing` intègre de l'IPC multiplateformes dans une API de plus haut niveau.

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.