université de BORDEAUX

TP5b - Socket Serveur

1 Serveur Echo en Python

Nous allons maintenant étudier la programmation d'un serveur TCP en Python, en prenant l'exemple du service *echo*.

Le service `echo` est un service des plus simples : il répète ce qu'on lui dit. Il existe à la fois en version UDP et en version TCP (et même en AppleTalk). Quel est son numéro de port ?

On ne pourra pas lancer notre serveur sur ce port-là car les ports de numéro inférieurs à 1024 sont réservés à l'utilisateur `root`. On utilisera donc le port 7777.

1.1 Petit rappel

Il sera utile de se référer aux documentations suivantes :

— socket : https://docs.python.org/3/library/socket.html

— string : https://docs.python.org/3/library/stdtypes.html#str

Par ailleurs, il faut rappeler que les fonctions `send() & recv()` de la bibliothèque *socket* manipulent uniquement des tableaux d'octets. Par conséquent, on aura souvent besoin de faire des conversions entre des tableaux d'octets (`b"h\xc3\xa9ho"`) et des chaînes de caractères (`"hého"`). La différence entre les deux est simplement la notion d'encodage des caractères, on va prendre comme convention que les tableaux d'octets encodent les chaînes de caractères en UTF-8 :

Pour convertir une chaîne de caractères en un tableau d'octets avant de pouvoir envoyer vers le réseau :

```
s = "hého"
c = s.encode("utf-8")
```

Et inversement, quand on a lu un tableau d'octets depuis le réseau et que l'on veut l'afficher :

```
c = b"h\xc3\xa9ho"
s = c.decode("utf-8")
```

1.2 Version TCP

Un serveur TCP fonctionne presque de la même manière qu'un client TCP, la différence essentielle est que l'on doit gérer à la fois une socket d'écoute des connexions et les sockets pour les clients. On se contente ici de gérer un client à la fois.

— Créer une *socket* à l'aide de la fonction `socket.socket()`, de famille `socket.AF_INET6` (qui gère à la fois en v4 et en v6), de type `socket.SOCK_STREAM`, et laisser le protocole 0 pour que le système choisisse automatiquement TCP.
— Utiliser la méthode `bind` de la socket pour greffer notre *socket* au port 7777. Pour l'adresse, il suffit de spécifier le paramètre `("", 7777)` pour être à l'écoute sur toutes les cartes réseaux de la machine. Attention, il faut donc mettre 2 parenthèses, pour que `("", 7777)` soit un seul argument.
— Appeler la méthode `listen` pour indiquer au système qu'on va accepter des connexions. Un backlog de 1 suffira.
— Dans une boucle infinie,
  • Appeler la méthode `accept` pour accepter une connexion entrante. Notez bien que cette fonction vous retourne un tuple contenant en premier une *nouvelle* socket, représentant la connexion acceptée, dont on va appeler les méthodes `recv` et `send`, et en deuxième son adresse qui ne nous sera pas utile. Il faut bien sûr conserver la socket initiale pour le prochain `accept`, pour l'instant on s'occupe seulement de cette connexion.
  • Dans une boucle infinie,
    — Utiliser la méthode `recv` pour réceptionner des données, en lui passant comme taille 1500. Si la longueur des données reçues est 0, c'est que le client s'est déconnecté et l'on peut utiliser `break` pour sortir de la boucle.
    — utiliser la méthode `send` pour ré-expédier les données à l'envoyeur. Il suffit de passer le message obtenu !
  • Fermer la *socket* de la connexion à l'aide de la méthode `close`.

Cette version passe ainsi son temps à effectuer `accept`, une boucle de `send/recv`, et `close`. Pour tester, utilisez `nc localhost 7777` dans un terminal à côté pendant que votre serveur tourne. Utilisez `netstat -tuap` (ou `ss -tuap`) pour remarquer la présence de votre serveur. Relancez `nc` plusieurs fois, pour constater que le numéro de port côté client change effectivement à chaque fois.

Il se peut que `bind` échoue avec l'erreur `Address already in use`. Si vous regardez dans netstat ou ss, vous verrez une ligne du genre

```
tcp6       0      0 ::1:7777                ::1:49346               FIN_WAIT2   -
```

Cela signifie donc que le système préfère éviter que vous relanciez un serveur sur ce port alors qu'il reste encore des connexions qui ne se sont pas terminées, et il faut alors attendre quelques minutes. Pour éviter que le système soit si précautionneux, avant l'appel à `bind` utilisez appeler la méthode `setsockopt` pour mettre l'option `socket.SO_REUSEADDR` (dans le *level* `socket.SOL_SOCKET`) à 1. Notez que si vous avez eu ce problème, cette option ne va pas fonctionner immédiatement, car il aurait fallu que cette option soit positionnée par l'instance du serveur que vous aviez lancée. Patientez quelques minutes, et les instances suivantes, lancées avec l'option, permettront d'en lancer d'autres.

1.3 Client Echo

Une fois que votre serveur TCP Echo est fonctionnel, on va coder le client Echo correspondant. Repartez du code du client *Daytime*, vu précedemment. Ce client devra lire, en boucle, les caractères saisis sur l'entrée standard pour les envoyer au serveur, et afficher les réponses en écho de ce dernier. Le client fermera la connexion lorsque

l'utilisateur saisit une chaîne de caractères vide avec un retour à la ligne (`"\n"`).

Pour lire sur l'entrée standard, on pourra utiliser le code suivant :

```
import sys
line = sys.stdin.readline()
```

1.4 Pour aller plus loin : version UDP

La version UDP est relativement simple à réaliser.

- Créer une *socket* de famille `socket.AF_INET6` et de type `socket.SOCK_DGRAM`.
- Utiliser ensuite la méthode `bind`.
- Dans une boucle infinie,
  - Appeler la méthode `recvfrom` de la socket.
  - Appeler la méthode `sendto` de la socket pour ré-expédier le message à l'envoyeur. Il suffit de passer le message et l'adresse obtenus !

Pour tester, vous pouvez utiliser `nc -u localhost 7777` . Utilisez `strace` pour bien observer les appels effectués par votre serveur. Utilisez `netstat -tuap` (ou `ss -tuap`) pour remarquer la présence de votre serveur. Relancez `nc` plusieurs fois, pour constater que le numéro de port côté client change effectivement à chaque fois ! !

2 Gestion de multiples clients avec un serveur TCP

Reprenons le code Python de notre serveur *echo* (version TCP).

Dans ce code, il faut bien distinguer :

- la socket "serveur" *s*, créée en premier, qui est à *l'écoute* et sert principalement à attendre & accepter les demandes de connexion des clients ;
- la socket "cliente" *sc*, qui est renvoyée par la fonction `accept()` et sert à dialoguer avec un client particulier.

Que se passe-t-il si vous essayez de lancer plusieurs clients TCP à la fois ? Eh oui, notre programme ne s'occupe pour l'instant que d'un client à la fois. Pour surmonter ce problème, nous allons étudier deux solutions possibles à base de *thread* ou de *select*.

2.1 Version *select*

La version avec des *threads* a l'avantage d'être très simple, elle pose cependant des problèmes de synchronisation. Elle a aussi le défaut de nécessiter un thread par client connecté, ce qui peut devenir coûteux avec de nombreux clients. Une autre version possible est `select`, où l'attente de commandes venant des clients est gérée de manière centralisée dans le thread principal.

On aura besoin pour cette nouvelle version d'utiliser le module *select* (https://docs.python.org/3/library/select.html) :

```
import select
```

Dans cette version, nous allons utiliser une liste de sockets "clientes", que l'on initialise à la liste vide :

```
l = []
```

L'initialisation de la socket "serveur" reste sinon la même, il faut par contre changer la boucle principale :

— On commence la boucle par appeller `select.select` en lui passant comme premier paramètre notre liste de clients à laquelle on ajoute à la volée la socket "serveur", et deux listes vides. Elle retourne trois listes, c'est la première qui nous intéresse, elle contient les sockets disponibles en lecture.

— On utilise une boucle `for` pour parcourir cette liste de sockets, pour chacune d'entre elles :

- Si c'est la socket "serveur", c'est qu'un client vient de se connecter. On peut appeler la méthode `accept` de notre socket "serveur", qui nous retourne une nouvelle socket "cliente", que l'on peut alors ajouter à notre liste.
- Sinon, c'est que le client de cette socket nous a envoyé des données. Il suffit d'appeler `recv` et `sendall` comme avant. Dans le cas où la longueur reçue est zéro, il faut non seulement fermer la socket "cliente", mais aussi l'enlever de la liste l avec la méthode `remove`.

Implémentez cette version et testez-la sur la machine locale avec de multiples clients *netcat* connectés simultanément !

2.2 Pour aller plus loin : version *thread*

On aura besoin pour cette version d'utiliser le module *threading* (https://docs.python.org/3/library/threading.html) :

```
import threading
```

Le plus simple est de déporter la boucle interne `recv/send` dans un nouveau *thread*, ce qui permet donc au *thread* principal de retourner immédiatement pour effectuer l'`accept()` suivant.

Pour faire tourner la boucle dans un thread, créez une nouvelle fonction `handle(sc)` (qui prend juste en paramètre la socket "cliente" *sc*), et déplacez dedans cette boucle (ainsi que la fermeture de la socket du client).

Il suffit alors de créer un thread t et de le démarrer dans la boucle principale de la manière suivante, pour qu'il s'occupe d'appeler la fonction `handle` pour vous :

```
t = threading.Thread(None, handle, None, (sc,))
t.start()
```

Attention à bien taper les parenthèses et la virgule dans `(sc,)`. Il s'agit en fait du tuple d'arguments passés à la fonction `handle()` au démarrage du thread. Au final, chaque nouveau client qui se connecte est géré côté serveur par un nouveau thread, qui se termine quand le client se déconnecte et que fonction `handle()` prend fin...

Implémentez cette version et testez-la sur la machine locale avec de multiples clients *netcat* connectés simultanément !