université de BORDEAUX

TP4

1 Gestion de multiples clients avec un serveur TCP

Reprenons le code Python 3 de notre serveur *echo* (version TCP) écrit lors du TP3.

```
#!/usr/bin/python3
import socket

s = socket.socket(socket.AF_INET6, socket.SOCK_STREAM, 0)
s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
s.bind(('', 7777))
s.listen(1)

while True:
    sc, a = s.accept()
    print("new client:", a)
    while True:
        msg = sc.recv(1500)
        if len(msg) == 0:
            print("client disconnected")
            sc.close()
            break
        sc.sendall(msg)

s.close()
```

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*.

1.1 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 `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, déplacez cette boucle (ainsi que la fermeture de la socket) dans sa propre fonction `handle(sc)`, qui prend juste en paramètre la socket "cliente" *sc*. Il suffit alors de créer un thread *t* et de le démarrer 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 !

1.2 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.

Repartez du programme de départ, du début du TP.

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 Serveur de *chat*

L'objectif de ce TP est de réaliser un mini-serveur de discussion en ligne (ou *chat*, en anglais).

2.1 Une première version simple

Le serveur de *chat* que l'on se propose d'écrire est dans un premier temps simpliste : chaque message envoyé par un client est renvoyé tel quel à tous les autres clients.

Pour ce faire, nous allons repartir du code de notre serveur *echo* en TCP dans sa version `select`. Il est alors très simple de transformer ce programme en un serveur de *chat* : il suffit d'écrire le message reçu sur toutes les sockets "clientes", hormis la socket qui a envoyé le message.

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

2.2 A propos d'une version thread

Que pourrait-il se passer si l'on utilisait la version *thread* ? Que pourrait être une solution ? On ne demande pas d'implémenter cette version, qui pose quelques difficultés techniques...

2.3 Extensions du serveur de *chat*

Il s'agit maintenant d'étendre le serveur de *chat* pour qu'il soit plus intéressant. Pour cela, plutôt que d'envoyer du simple texte, on va échanger des *commandes* entre les clients et le serveur. Toutes les commandes envoyées au serveur respecteront le format suivant : `<cmd> <arg>`, avec `<arg>` un argument optionnel séparé de la commande `<cmd>` par un espace. Dans ce contexte, le serveur doit analyser les lignes reçues, pour effectuer la commande souhaitée.

Pour rappel, commencez par appeler la méthode `decode` pour convertir le tableau d'octets reçu (classe `bytes`) en chaine de caractères (classe `str`).

Pour analyser la ligne de commande, vous aurez besoin des fonctions de la classe `str`, et en particulier de la fonction *split*. Voire la documentation directement avec la commande `pydoc3 str`, ou en ligne : https://docs.python.org/3/library/stdtypes.html#str.

Pour simplifier l'exercice, on ne spécifie pas le format des réponses que le serveur transmet aux clients, et qui pourront être dans un format brut, la plus simple possible. Dans

vos tests, vous utiliserez *netcat* comme client, qui se contentera d'afficher des réponses brutes.

- Pour commencer, les messages que souhaite envoyer un client doivent commencer par `MSG` et respecteront donc la syntaxe suivante : `MSG <message>`. Le serveur se contente alors de renvoyer ce message à tous les clients (hormis l'expéditeur).
- Toute commande inconnue doit être ignorée et pourra éventuellement renvoyer un message d'erreur du type *"invalid command"* à son expéditeur.
- Ajoutez une notification à tous les clients lorsqu'un nouveau client se connecte ou qu'un client déjà existant se déconnecte : le message précisera l'adresse du client au format `"<address>:<port>"`. Pour obtenir cette adresse, on peut utiliser la méthode `getpeername()` de la classe socket.
- Ajoutez un pseudo (ou *nick*) aux utilisateurs. Lorsqu'un client se connecte, il possède par défaut son adresse (au format `"<address>:<port>"`) comme *nick*. Pour personnaliser son nom, le client envoie la commande `NICK <nick>`. Il faudra bien sûr stocker ces noms à côté des sockets correspondantes, soit à l'aide d'une seconde liste, soit à l'aide d'un dictionnaire qui utilisera la socket "cliente" comme clé. (cf. documentation).
- Mettre à jour les notifications de connection & déconnection pour qu'elles utilisent maintenant ce *nick*. Faire en sorte que les messages relayés aux clients soit préfixés par le *nick* de son expéditeur.
- Ajoutez une commande `WHO` qui permet d'obtenir la liste de tous les clients connectés.
- Ajoutez une commande `QUIT <message>` qui permet à un client de se déconnecter *proprement* du serveur en envoyant un message d'adieu à tout le monde. Attention, le serveur devra fermer la socket correspondante et nettoyer les structures de données.
- Ajoutez une commande `KILL <nick> <message>` qui permet d'éjecter un client désagréable, après lui avoir envoyé un message d'adieu. En principe, cette commande est réservé à un client particulier, qui joue le rôle d'administrateur sur le serveur.

Voici un exemple de session entre le serveur et trois clients :

```
# server
$ python3 chat.py
Welcome to Chat Server
client connected "::ffff:127.0.0.1:51116"
client connected "::ffff:127.0.0.1:51120"
client connected "::ffff:127.0.0.1:51124"
client "::ffff:127.0.0.1:51116" => "toto"
client "::ffff:127.0.0.1:51120" => "tutu"
client "::ffff:127.0.0.1:51124" => "tata"
client disconnected "toto"
client disconnected "tutu"
client disconnected "tata"
```

```
# client 1
$ netcat localhost 7777
NICK toto
[tata] hello world!
QUIT bye bye

# client 2
$ netcat localhost 7777
NICK tutu
[tata] hello world!
[toto] bye bye
[tata] I kill you!
```

```
# client 3
$ netcat localhost 7777
NICK tata
WHO
[server] toto tutu tata
MSG hello world!
[toto] bye bye
KILL tutu I kill you!
QUIT ciao
```

Vous remarquerez que les adresses IP sont préfixées par `::ffff:`. En effet, nous avions créé une socket IPv6, mais c'est un client IPv4 qui s'est connecté ! Le système traduit alors l'adresse IPv4 en une addresse IPv6-mappant-IPv4[1] notée ainsi. Si vous utilisez `ip6-localhost` pour vous connecter, ce sera l'adresse IPv6 `::1` qui apparaîtra : le client se sera connecté vraiment en IPv6.

1. https://fr.wikipedia.org/wiki/Adresse_IPv6_mappant_IPv4

2.4 Bonus : les channels

Dans ce bonus, il s'agit de gérer des canaux de discussion distincts (ou *channels* en anglais). Chaque client peut utiliser des commandes `JOIN <channel>` et `PART` pour rejoindre ou quitter un canal. Si le canal n'existe pas, alors celui-ci est automatiquement créé par le serveur lors du premier `JOIN`. Pour lister les canaux disponibles, il faut utiliser la commande `LIST`.

Dans cette version, il n'est donc plus possible de parler sans avoir préalablement rejoint un canal. La commande `MSG` doit donc désormais préciser le canal où l'on parle et respectera donc la syntaxe suivante : `MSG <channel> <message>`.

Afin de simplifier un peu, on considère qu'un client ne pourra se connecter qu'à un seul canal à la fois. Le serveur doit donc se souvenir du canal que chaque client a rejoint (ou `""` s'il n'a rejoint encore aucun canal). Dans contexte, il devient probablement plus approprié d'utiliser une structure de type dictionnaire de liste ou dictionnaire de dictionnaire (cf. documentation).

Ajoutez pour finir une commande `KICK <nick>` qui permet d'éjecter un client d'un canal : il reste connecté, mais n'est plus abonné à ce canal. En principe, seul le créateur du canal (ou modérateur) peut utiliser cette commande.

Super Bonus : Implémentez complètement le protocole IRC (Internet Relay Chat), décrit par la RFC 1459 ;-)