TP5 - Mini-serveur *chat*

L'objectif de ce TP est de réaliser un mini-serveur de discussion en ligne. Pour cela on commencera d'abord par un simple serveur `echo`, puis on passera à un serveur multi-client à l'aide de *threads* et de `select()`, et enfin au serveur de *chat* proprement dit.

Pour vous simplifier la vie, la programmation se fera en python.

Bien sûr, il sera utile de se référer à la documentation des sockets python sur
`https://docs.python.org/3/library/socket.html`
`https://docs.python.org/3/library/select.html`
des threads python sur
`https://docs.python.org/3/library/threading.html`
et des chaînes de caractères sur
`https://docs.python.org/3/library/stdtypes.html#str`

On aura donc besoin dans le TP d'utiliser les imports suivants :

```
import socket
import select
import threading
```

Enfin, on aura besoin de convertir entre tableaux d'octets (`b"h
xc3
xa9ho"`) et 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 depuis une chaîne de caractères vers 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'imprimer :

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

1 Serveur 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 Version UDP

La version UDP est relativement simple à réaliser.

— 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_DGRAM`, et laisser le protocole 0 pour que le système choisisse automatiquement UDP.

— 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.

— Dans une boucle infinie,

— Appeler la méthode `recvfrom` de la socket, en lui passant comme taille 1500. On récupère à la fois le message et l'adresse qui l'a envoyé.
— Appeler la méthode `sendto` de la socket pour ré-expédier le message à l'envoyeur. Il suffit de redonner les mêmes paramètres !

Pour tester, vous pouvez utiliser `nc -u localhost 7777` . Utilisez `strace` pour bien observer les appels effectués par votre serveur. Utilisez `netstat -Ainet -Ainet6 -ap` 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.

1.2 Version TCP

La version TCP est plus compliquée puisqu'il faut réceptionner les connexions, mais le début est tout à fait le même.

— créer la *socket* à l'aide de `socket`, cette fois en type `socket.SOCK_STREAM`.
— utiliser `bind` de la même façon.
— appeler `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 une *nouvelle* socket, représentant la connexion acceptée (ainsi que 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, utiliser `send` pour les ré-expédier. 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.
  — 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`, et de nouveau utilisez `strace` et `netstat`.

Il se peut que `bind` échoue avec l'erreur `Address already in use`. Si vous regardez dans netstat, 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.

1.3 Clients TCP multiples

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.

Le plus simple est de déporter la boucle `read/write` dans un nouveau *thread*, ce qui permet donc au *thread* principal de retourner immédiatement effectuer l'`accept` suivant. Pour faire tourner la boucle dans un thread, déplacez cette boucle et la fermeture de la socket dans sa propre fonction `f`, qui prend juste en paramètre la socket. Il suffit alors d'importer le module threading, et de créer un thread avec

```
threading.Thread(None, f, None, (s2,)).start()
```

qui passe la socket `s2` à la fonction `f`.

1.3.1 Version `select`

La version *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 utilise une liste de socket `l`, que l'on initialise à la liste vide.

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 plus la socket d'écoute, et deux listes vides. Elle retourne trois liste, c'est la première qui nous intéresse, elle contient les sockets disponibles en lecture.
- On utilise une boucle `for` pour parcourir les sockets.
  - Si c'est la socket d'écoute, c'est qu'un client vient de se connecter. On peut appeler la méthode `accept` de notre socket d'écoute, qui nous retourne une nouvelle socket pour ce client, que l'on peut ajouter à notre liste.
  - Sinon, c'est qu'un client nous a envoyé des données. Il suffit d'appeler `recv` et `send` comme avant. Dans le cas où la longueur reçue est zéro, il faut non seulement fermer la socket, mais aussi l'enlever de la liste avec la méthode `remove` de la liste.

Testez !

2 Serveur de *chat*

Le serveur de *chat* que l'on se propose d'écrire est dans un premier temps simpliste : chaque donnée envoyée par un client est renvoyée telle quelle à tous les autres clients.

Maintenant que l'on a un serveur `echo` en TCP passé en version `select`, il est très simple d'en faire un tel serveur de *chat* : il suffit d'écrire non seulement sur la socket qui a envoyé le message, mais aussi à toutes les autres sockets.

Que pourrait-il se passer si l'on utilisait la version *thread* ?

Que pourrait être une solution ?

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 qu'envoyer du simple texte, on va échanger des *commandes* entre les clients et le serveur. Pour commencer, les lignes de texte simple (à envoyer à tous les clients) doivent commencer par `MSG`. Il faut alors *analyser* les lignes reçues : vous pouvez utiliser pour cela les fonctions de la classe `str`, cf la commande `pydoc str`

- Ajoutez une notification d'entrée/sortie : lorsqu'un client se connecte ou se déconnecte, une commande `JOIN` et `PART` indiquant son adresse est envoyée à tous les autres clients.
- Ajoutez un « nick » : lorsqu'un client se connecte, il doit utiliser une commande `NICK` pour annoncer son nom au serveur. Le serveur peut ainsi utiliser ce nom-là plutôt que son adresse. Il faudra bien sûr stocker ces noms à côté des sockets correspondantes.
- Ajoutez une commande `LIST` qui permet d'obtenir la liste de tous les clients connectés.
- Ajoutez une commande `KILL` qui permet d'éjecter un client : le serveur ferme la socket correspondante après lui avoir envoyé un message d'adieu.
- Gérez des canaux de discussion distincts : chaque client peut utiliser des commandes `JOIN` et `PART` pour rejoindre ou quitter des canaux (le serveur doit donc se souvenir de la liste des canaux que chaque client a rejoints). La commande `MSG` doit donc désormais préciser le canal où l'on parle
- Ajoutez une commande `KICK` qui permet d'éjecter un client d'un canal : il reste connecté, mais n'est plus abonné à ce canal.