Université Grenoble Alpes

Programmation d'applications client-serveur avec *sockets* TCP/IP (Observation). Illustration avec Python 3 et Linux

1 Préambule

Le but de ce sujet est d'étudier la programmation réseau avec le modèle client-serveur et les *sockets* TCP/IP, *avec des programmes écrits en Python 3 pour le système d'exploitation Linux*.
Plusieurs exemples de programmes (complets et prêts à l'emploi) sont fournis. L'objectif de ce TP est de comprendre leur fonctionnement en combinant la lecture du code source et leur utilisation pratique.
Des pistes d'approfondissement (exercices de programmation) sont proposées à la fin du document.

Le code à étudier et à tester est fourni dans les fichiers suivants :

- Pour l'application "écho" :
  - `echo_client_tcp.py`
  - `echo_server_tcp.py`
  - `echo_client_tcp_multiprocess.py`
- Pour l'application "service de fichiers"
  - `simple_file_client.py`
  - `simple_file_server.py`
- Pour toutes les applications :
  - `my_utils.py` qui fournit diverses fonctions utilitaires

2 Application "écho"

On considère ici une application client-serveur très simple nommée "écho", basée sur les caractéristiques suivantes :

- Protocole simple requête-réponse : le client envoie une requête puis attend de recevoir la réponse du serveur avant d'envoyer éventuellement une nouvelle requête.
- Plusieurs échanges requête-réponse sont véhiculés au sein de la même connexion TCP.
- Une requête correspond simplement à l'envoi d'une ligne de texte (terminée par une séquence de fin de ligne) que le client a récupérée à partir du clavier. Le serveur répond en renvoyant exactement la même ligne de texte en majuscules. Le client affiche ensuite la réponse qu'il a reçue.

2.1 Serveur itératif

Étudier le contenu des fichiers fournis (`echo_client_tcp.py`, `echo_server_tcp.py`, et les fonctions utilisées dans `my_utils.py`). Puis observer l'exécution de l'application : lancer

le serveur et le client dans deux terminaux différents (voire sur deux machines différentes).

Pour lancer le serveur, choisir un port TCP disponible (8765 dans l'exemple ci-dessous) [1] :

```
$ ./echo_server_tcp.py 8765
```

Pour lancer le client, il faut indiquer le nom (ou l'adresse IP) de la machine serveur (dans cet exemple, on prendra le nom `localhost` ou l'adresse IPv4 `127.0.0.1`, qui correspondent à la machine locale), ainsi que le numéro de port de l'application serveur :

```
$ ./echo_client_tcp.py 127.0.0.1 8765
```

Remarques :

- Pour arrêter le client, il faut fermer son flux d'entrée clavier en pressant `Ctrl-d`.
- Le code fourni ne gère par l'arrêt propre du serveur. On pourra ignorer cet aspect dans le cadre de ce TP et se contenter d'arrêter brutalement le serveur en pressant `Ctrl-c` sous Linux (cela déclenche l'envoi d'un signal Unix `SIGINT` au processus, et la réaction par défaut consiste à détruire immédiatement le processus).
- Pour voir la liste des paramètres/options disponibles, on peut lancer les programmes ci-dessus sans aucun arguments. Exemple :
  ```
  $ ./echo_client_tcp.py
  usage: echo_client_tcp.py [-h] serverhost serverport
  echo_client_tcp.py: error: the following arguments are required:
  serverhost, serverport
  ```
- Le code fourni est (en principe) compatible avec les réseaux utilisant le protocole IP version 4 (IPv4) ou version 6 (IPv6).
  - Par défaut, le code fourni pour l'application serveur associe la socket serveur à toutes les adresses disponibles sur la machine serveur (qu'il s'agisse d'adresses IPv4 ou IPv6). On peut modifier ce comportement en utilisant l'option `-ipv` pour ne considérer qu'un seul type d'adresse (par exemple `-ipv 4` pour sélectionner uniquement les adresses IPv4).
  - En ce qui concerne le client, le comportement adopté pour contacter le serveur varie en fonction de la façon dont l'identifiant du serveur est passé dans les paramètres. Si l'identifiant du serveur est passé sous forme d'adresse IP (IPv4 ou IPv6) alors le client contactera directement le serveur via cette adresse, et utilisera comme adresse source sa propre adresse associée au même numéro de version IP (si le client ne dispose que d'une adresse IPv4 et le serveur ne dispose que d'une adresse IPv6 ou inversement, alors ils ne pourront pas communiquer ensemble). Si l'identifiant du serveur est passé sous forme de nom symbolique, alors le client utilisera le DNS pour obtenir la ou les adresses IP (IPv4 ou IPv6) associées à ce nom puis effectuera une tentative de connexion en utilisant successivement chacune de ces adresses, jusqu'à obtenir un succès ou épuiser la liste.

1. Ici, le symbole `$` représente l'invite de commandes du shell

Étudier également le cas dans lequel plusieurs clients tentent de se connecter simultanément au serveur. Si nécessaire, ajouter des traces dans le code pour mieux suivre les étapes.

Remarquer également les numéros de ports TCP utilisés par le client et par le serveur. Faire éventuellement le lien avec les informations affichées par l'utilitaire `netstat` (sous Linux).

2.2 Serveur concurrent

On souhaite maintenant faire évoluer le fonctionnement du serveur de façon à pouvoir traiter plusieurs clients de manière concurrente en utilisant des processus.

Reprendre les étapes précédentes en utilisant cette fois-ci la seconde version du serveur : `echo_server_tcp_multiprocess.py`

Dans cette version, il y a un processus principal, nommé ici le "veilleur", qui attend les connexions des clients (en appelant la fonction `accept`). Lorsqu'une nouvelle connexion est disponible, le veilleur crée un nouveau processus fils "assistant" (*handler*), qui sera chargé du dialogue avec le nouveau client, puis le veilleur se remet en attente de nouvelles connexions. Lorsqu'un processus assistant a fini de dialoguer avec un client, il se termine immédiatement.

3 Service de fichiers

Étudier maintenant la seconde application : un service de fichiers implémenté par les programmes `simple_file_client.py` et `simple_file_server.py`.

Le principe de cette application est le suivant :

— Un client peut contacter le serveur pour demander à télécharger un fichier présent sur la machine serveur (seuls les fichiers présent dans le répertoire courant de l'application serveur sont disponibles).

— Contrairement à l'application précédente (écho), le protocole applicatif choisi ici ne gère qu'un seul échange requête-réponse au sein d'une même connexion TCP. Si un utilisateur souhaite télécharger plusieurs fichiers, il devra lancer plusieurs fois l'application cliente (un seul téléchargement de fichier par connexion TCP établie).

— L'application cliente commence par demander à l'utilisateur de saisir le nom du fichier demandé. Cette chaîne de caractères est ensuite envoyée au serveur.

— Le serveur analyse ensuite la requête du client et envoie au client un en-tête de réponse en fonction de la situation. Cet en-tête contient deux lignes de texte :

  - Si la demande est incorrecte (fichier inexistant ou inaccessible) : la ligne `KO` suivie d'une description du problème rencontré.
  - Si la demande est recevable : la ligne `OK` suivie de la taille du fichier (en nombre d'octets).

- Dans le cas d'une demande recevable :
  - Le serveur envoie ensuite le contenu du fichier. Puisque le fichier peut contenir des données de type arbitraire (pas obligatoirement du texte), il est nécessaire de transmettre son contenu sous forme d'octets bruts plutôt que de chaînes de caractères. Par ailleurs, étant donné que la taille du fichier peut être arbitrairement grande, il peut être nécessaire de transmettre le fichier par morceaux (dont la taille est bornée par celle du tampon en mémoire dans lequel on lit les données à partir du disque).
  - Le client crée un fichier local (le répertoire par défaut est `/tmp`) et y écrit les données lues à partir du la socket.

4 Pour aller plus loin ...

Des pistes d'exercices d'approfondissement sont proposés ci-dessous :

Variante d'un protocole requête-réponse : Modifier légèrement la spécification de l'un des protocoles applicatifs (par exemple le protocole du service "écho") et introduire les modifications nécessaires dans l'implémentation du programme client et du programme serveur. Par exemple, une fois la connexion établie, le serveur commence par envoyer un message de bienvenue au client et le client attend d'avoir reçu et affiché ce message avant d'envoyer sa première requête d'écho.

Serveur de fichiers concurrent : Développer une version multiprocessus du serveur de fichiers, afin de gérer plusieurs clients en même temps.

Serveur concurrent multi-threads : Développer une variante du serveur écho concurrent en utilisant des threads plutôt que des processus. Les opérations de création/destruction de threads sont moins coûteuses que les opérations équivalentes pour les processus. Noter cependant qu'avec Python, contrairement à d'autres langages, l'implémentation des threads ne permet généralement pas de tirer efficacement parti du parallélisme matériel (processeurs multi-cœurs) [2].

Serveur concurrent avec pool de threads : La création systématique de nouveaux threads peut être coûteuse. Une autre approche consiste à pré-allouer un groupe (*pool*) de threads et à faire en sorte que chacun d'entre eux puisse traiter successivement plusieurs clients (lorsqu'un client est parti, le thread peut s'occuper d'un nouveau client). Plusieurs

2. Pour plus de détails sur le sujet, voir par exemple le lien suivant : `https://realpython.com/python-gil/`

architectures sont envisageables dans cette optique :

— **Conserver une architecture avec un veilleur et des threads assistants.** Dans ce cas, il faut trouver un moyen pour que le veilleur puisse transmettre le canal de communication au thread assistant choisi (contrairement à la version précédente, on ne peut pas fournir ces informations lors de la création du thread assistant). Pour cela, on peut utiliser un tampon producteurs-consommateurs (cf. cours sur la synchronisation). Ici, il n'y a qu'un seul producteur et plusieurs consommateurs : le veilleur dépose les informations concernant l'arrivée d'un nouveau client et ces informations sont extraites du tampon par le premier thread assistant disponible. Les accès concurrents au tampon effectués par les différents threads devront être synchronisés de manière adéquate afin d'éviter les problèmes d'incohérences.

— **Utiliser une architecture symétrique dans laquelle tous les threads jouent le même rôle.** Chaque thread récupère une nouvelle connexion d'un client (en appelant directement `accept`) puis gère le dialogue complet avec ce client avant de recommencer avec un autre client. Cette architecture est un peu plus simple à mettre en œuvre. Noter que des appels concurrents à la primitive `accept` sont automatiquement synchronisés par le systèmes d'exploitation.