TP 09

INTRODUCTION AUX SOCKETS

OBJECTIF

Vous allez dans ce TP faire communiquer en mode connecté deux processus applicatifs (un client et un serveur au moyen de **sockets**.

1. LES SOCKETS

Les sockets constituent une interface de programmation applicative (API) permettant la communication inter-processus (IPC), apparue initialement en 1983 dans les systèmes UNIX BSD afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'au travers d'un réseau, et répandue aujourd'hui dans la plupart des systèmes d'exploitation. Dans ce TP, vous utiliserez le module socket de Python (qu'il vous faudra donc importer).

Les étapes dans l'utilisation des sockets en mode connecté sont données ci-dessous :

Serveur	Client
1. crée une socket	
2. attend une connexion de la part du client	1. crée une socket
3. prend connaissance de la connexion TCP une fois établie avec le client	2. se connecte au serveur
4. échange des informations avec le client	3. échange des informations avec le serveur
5. ferme la connexion	4. ferme la connexion

Les méthodes Python que vous utiliserez sont les suivantes :

1.1. Côté Serveur

- Créer un socket : socket.socket (socket.AF_INET, socket.SOCK_STREAM)
  Le constructeur de l'objet socket attend 2 paramètres :
  - la famille d'adresses : socket.AF_INET = adresses IPv4
  - le type du socket : socket.SOCK_STREAM = protocole TCP pour une utilisation en mode connecté

  socket () renvoie le descripteur du socket créé, à utiliser dans les appels prochains.
- Lier le socket à un port d'écoute : méthode .bind ()
  Le serveur se met en attente des connexions entrantes, mais il faut indiquer sur quel port il va placer son écoute et le configurer en conséquence, à l'aide de la méthode .bind() qui attend un tuple (nom_d_hote, port) :
  - mettre ici une chaîne vide comme nom d'hôte (ce qui est équivalent à l'adresse IP 0.0.0.0 : i.e. le socket est lié à toutes les interfaces locales de la machine)
  - mettre n'importe quelle valeur pour le port d'écoute, comprise entre 1024 et 65535 : ici, on prendra 65432
- Mettre le socket à l'état d'écoute : méthode .listen ()
  Le serveur est prêt à écouter, mais il n'écoute pas encore. Il faut encore lui préciser le nombre maximum de connexions qu'il peut recevoir sans les accepter, à l'aide de la méthode .listen()

qui a un seul paramètre *n* (si *n* clients se connectent et que le serveur n'accepte aucune de ces connexions, aucun autre client ne pourra se connecter ; mais généralement, très peu de temps après que le client a demandé la connexion, le serveur l'accepte : il peut donc avoir bien plus de *n* clients connectés)

- Prendre connaissance des connexions TCP établie à l'initiative d'un client : méthode .accept()
  Lorsque TCP établit une connexion avec un client, le serveur prend connaissance de la connexion, à l'aide de la méthode .accept()
  - l'exécution du programme est bloquée tant qu'aucun client ne s'est connecté
  - la méthode .accept() renvoie 2 informations :
    - le socket de communication qui vient de se créer (et qui va permettre de communiquer avec le client tout juste connecté)
    - un tuple (nom_d_hote, port) représentant l'adresse IP et le port du client
- Fermer une connexion : méthode .close()

1.2. Côté Client

- Créer un socket : socket.socket (socket.AF_INET, socket.SOCK_STREAM)
  Il faut créer le socket qui va donner accès au serveur.
- Connecter le client : méthode .connect()
  Il suffit de demander la connexion au serveur (dont on connaît le nom d'hôte et le port d'écoute), à l'aide de la méthode .connect(). Ici, étant donné que l'application serveur et l'application client tournent sur la même machine, le nom d'hôte est *localhost*.

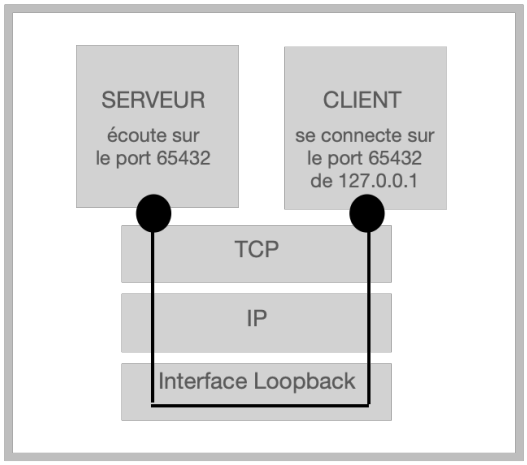
1.3. Echanges entre le client et le serveur

Une fois la connexion entre client et serveur établie, chacun peut envoyer et recevoir des données. Les données échangées sont de type bytes (chaînes d'octets). En Python, les chaînes d'octets sont représentées comme des chaînes de caractères (limités au codage ASCII) préfixées par un b (ex : b'Bonjour !').

- Envoyer des données : la méthode .send() attend un argument de type bytes et renvoie le nombre d'octets effectivement envoyés
- Recevoir des données : la méthode .recv() attend comme argument un entier donnant la taille maximale de la chaîne d'octets à recevoir et renvoie la chaîne d'octets (type bytes) effectivement reçue
  Attention : .recv() est une méthode bloquante : l'exécution de programme s'arrête jusqu'à ce que des données soient reçues : si les données envoyées sont de taille inférieure à la taille maximale spécifiée (généralement 1024), les données reçues feront exactement la taille des données envoyées ; si les données envoyées sont de taille supérieure à la taille maximale spécifiée, les données reçues auront la taille maximale spécifiée et il faudra refaire un .recv() pour "récupérer" le reste des données envoyées ;

2. ENVIRONNEMENT

Dans un souci de simplification, vous vous limiterez ici à un serveur et à un seul client, tous deux locaux à votre machine (le client est supposé connaître le numéro de port du serveur, à savoir 65432).



Le scénario à considérer sera le suivant :

Le serveur doit :

- se mettre en attente d'une connexion de la part du client sur le port 65432
- dès que le client s'est connecté, lui envoyer le message "Bonjour Client, je suis le serveur 65432, la connexion est etablie"
- recevoir un message du client et l'imprimer
- recevoir un second message du client et l'imprimer
- fermer la connexion

Le client doit :

- se connecter au port 65432 du serveur
- recevoir un message de la part du serveur et l'imprimer
- envoyer le message "Bonjour Serveur 65432, nous sommes donc dans la phase de transfert" au serveur
- envoyer le message "Serveur 65432, je vous dis au revoir"
- fermer la connexion