TP N°1 : Communication inter-process

Objectif

Implémenter une communication inter-process en utilisant l'API socket en mode TCP et UDP

Exercice 1:

Dans cet exercice, nous allons implémenter deux scripts, un représentant le client et l'autre le serveur (mode TCP).
On procède en plusieurs étapes, décrites schématiquement ci-après :

Partie 1 : Côté Serveur

1) Créer un dossier «TP1»;
2) Ouvrir un éditeur de texte et enregistrer le fichier sous le nom «serveur.py»**;**
3) Importer le module «socket», qui contient toutes les méthodes et les classes nécessaires;

4) Création de socket :

```
Sock_serv=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- *AF_INET* : le type d'adresses : AF_INET (pour IPv4) et AF_INET6 (pour IPv6);
- *SOCK_STREAM* : le type de service : flux d'octets (pour TCP), SOCK_DGRAM (pour UDP).

5) Associer le socket à une adresse IP et un numéro de port :

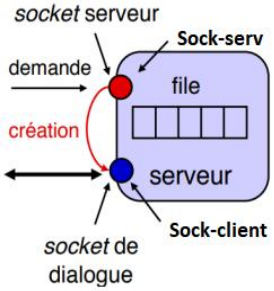
```
Sock_serv.bind(('127.0.0.1',60000))
```

Les valeurs passées à «bind()» est un «tuple[1]» à deux : (hôte, port).

6) Mettre le socket en mode écoute :

```
Sock_serv.listen()
```

- La fonction listen() ne s'utilise qu'en mode connecté (donc avec le protocole TCP).
- La fonction correspond à une «ouverture passive»: il n'y a pas de création effective d'un canal de communication mais le serveur affiche sa présence pour -sur demande - créer un tel canal.
- L'argument (qui est optionnel) qu'on lui transmet indique le nombre maximum de connexions à accepter en parallèle.



7) Accepter les demandes de connexions:

```
Sock_client, add_client=Sock_serv.accept()
```

- **`Sock_client`**:est la référence d'un nouvel objet de la classe «socket()»;
- **`add_client`** : un tuple contenant l'adresse IP et le n° de port du client.

[1] Les tuples ressemblent aux listes, mais on ne peut pas les modifier une fois qu'ils ont été créés.

8) Réception des données:

```
msgClient=Sock_client.recv(1024)
```

«1024» indique le nombre maximum d'octets à réceptionner en une fois. Si la taille du message reçu dépasse la taille du buffer, les octets surnuméraires sont mis en attente dans un tampon. Ils sont transmis lorsque la même méthode «recv()» est appelée à nouveau.

9) Émettre des données:

```
Sock_client.send(b"Welcome to this server")
```

- La méthode «send()» renvoie le nombre d'octets expédiés;
- Les données à envoyer en utilisant la fonction «send» doivent être de type «bytes» (le b devant une chaîne de caractères);
- Ou faire appel à la fonction «encode()» pour convertir une chaîne de caractère en bytes :

```
Sock_client.send("Welcome to this server".encode())
```

10) Afficher les données reçues

Vous remarquez la lettre b à côté du message reçu. Car le message est de type bytes.
Donc on utilise la méthode «decode()» pour convertir la donnée de «bytes» en une chaîne de caractère.

```
print("Serveur a reçu : ",msgClient.decode()," -- du client : ",Sock_client)
```

11) Fermeture

```
Sock_client.close()

Sock_serv.close()
```

Partie 2 : Côté client

1) Ouvrir un éditeur de texte et enregistrer le fichier sous le nom «client.py»
2) Importer le module «socket»
3) Création du socket :
   Créer un socket au niveau du client, comme cela a été fait au niveau du serveur et l'appeler «s»

4) Demande de connexion au serveur :

```
s.connect(('127.0.0.1', 60000))
```

La fonction «connect()» ne s'utilise qu'en mode connecté (donc avec le protocole TCP).
Il est possible d'utiliser «'localhost'» à la place du «'127.0.0.1'»

5) Émettre des données avec la méthode «send()»;
6) Réception des données grâce à la méthode «recv()»;
7) Fermeture du socket client avec la méthode «close()».

Exécution

1) Démarrer l'invite de commandes de votre machine;
2) À l'aide de la commande **cd** entrer dans le répertoire où se trouve le script «serveur.py»
3) Utiliser la commande **dir** (sous Windows) et **ls** (sous Linux) pour vérifier que vous êtes dans le bon répertoire;
4) Démarrer un deuxième invite de commandes;
5) À l'aide de la commande **cd** entrer dans le répertoire où se trouve le script «client.py»
6) Exécuter le script «serveur.py» à l'aide de la commande python:

```
>python serveur.py
```

7) Exécuter le script «client.py» à l'aide de la commande python:

```
>python client.py
```

Questions:

1) Pourquoi l'adresse IP utilisée dans ce TP est 127.0.0.1?
2) *Quelle est la condition imposée sur le choix de la valeur du numéro de port?*

Exercice 2:

1) Ajouter une instruction pour permettre l'affichage du message reçu par le client
2) Ré-exécuter le script serveur et avant de lancer le client, utiliser la commande **netstat** pour afficher les numéros ports tcp ouverts. Pour plus de détails sur cette commande, utiliser le Help.
3) Ré-exécuter le script client. Une fois les deux scripts terminés, lancer de nouveau la commande **netstat** pour voir si les sockets sont toujours ouverts.
4) Changer les deux scripts pour que le message envoyé et le numéro de port du serveur deviennent dynamiques (C'est à l'utilisateur de saisir le message à envoyer et le numéro de port).
5) Utiliser la méthode «getsockname()» de la classe «socket» pour que le script «client» affiche l'adresse IP et numéro de port attribués au socket du client.
6) Utilisez la méthode «bind()» pour attribuer une adresse IP et N° port statique au client (N°port = 62000).
   a) *Lancer le script client pour qu'il envoie un message au serveur*
   b) *Lancer de nouveau le script client.*
   c) Qu'est ce que vous remarquez?
7) Modifier le script du serveur pour que ce dernier une fois il a terminé la communication avec le client , il revient pour accepter de nouvelle demande de connexion. Ce travail se répète à l'infini.

Excercie 3 : Passage d'arguments de la ligne de commande

On souhaite récupérer la valeur du message à envoyer et N° port du serveur depuis la ligne de commande.

Exemple:

```
>python client.py 60000 Bonjour
```

- Pour cela on utilise le module **`sys`** : il fournit un accès à certaines variables utilisées et maintenues par l'interpréteur. Pour pouvoir l'utiliser, il faut importer le module **`sys`**
- **sys.argv :** représente la liste des arguments (type string) de la ligne de commande passés à un script Python. argv[0] est le nom du script (client.py), argv[1] vaut la chaîne '**60000**' et argv[2]=='Bonjour'.

```
port = int(sys.argv[1])
msg =sys.argv[2]
s.connect(('127.0.0.1', port))
s.send(msg.encode())
```

Ou

```
s.connect(('127.0.0.1', int(sys.argv[1])))
s.send(sys.argv[2].encode())
```

Excercie 4 : Socket en mode non connecté (UDP)

Implémenter une communication inter-process avec les sockets en mode non connecté («serveur_udp.py» et «client_udp.py»). Sachant que la procédure est similaire à celle du mode connecté. Enfin à certaine exceptions:

a) Pour la création de socket, on modifie l'argument qui défini le type de service par SOCK_DGRAM;
b) Dans le mode non connecté, l'établissement de connexion n'existe pas. Ainsi, les méthodes : «listen()», «accept()» et «connect()» n'existent pas;
c) C'est au moment de l'envoie qu'il faut préciser l'adresse de destination. Pour faire, on utilise la méthode «sendto()» de la classe socket :

```
sendto(bytesToSend, serverAddressPort)
```

   - bytesToSend : le message à envoyer, de type byte
   - serverAddressPort : tuple contenant l'adresse IP et le numéro de port

d) Pour la réception des données, on utilise la méthode «recvfrom(bufferSize)» qui possède la même syntaxe que «recv()» en mode connecté.

Exercice (Advanced)

1) Changer le type d'adresse utilisé de IPv4 en IPv6
2) Exécuter le script serveur et après la commande **netstat**
3) *Qu'est ce que vous remarquez?*
4) Exécuter le script client sans changer le type d'adresse
5) *Qu'est ce que vous remarquez?*
6) Changer le type d'adresse du client et exécuter le script client
7) *Qu'est ce que vous remarquez?*