Réseaux - (ré)introduction aux sockets

Un *socket* est un objet permettant d'ouvrir une connexion et de communiquer avec une machine locale ou distante. Il a été introduit dans les distributions de Berkeley (un système UNIX).

Ils permettent la réception et l'émission de messages à partir de plusieurs protocoles de communications, comme PF_INET, AF_INET, PF_UNIX, et bien d'autres. Dans le cas d'une communication client-serveur, il y aura deux programmes distincts.

La documentation de Python propose deux programmes qu'on retrouvera ci-dessous.

```python
# Echo server program
import socket

HOST = ''                 # Symbolic name meaning all available interfaces
PORT = 50007              # Arbitrary non-privileged port
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen(1)
    conn, addr = s.accept()
    with conn:
        print('Connected by', addr)
        while True:
            data = conn.recv(1024) #we receive data by 1024 bits chunks
            if not data: break
            print("received:", data)
            conn.sendall(data)
```

```python
# Echo client program
import socket

HOST = 'localhost'    # The remote host
PORT = 50007          # The same port as used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b'Hello, world')
    print("message sent to server")
    data = s.recv(1024) #we receive data by 1024 bytes chunks
    print('Received', repr(data))
```

Observations :

— On prendra soin d'utiliser un port non-réservé (> 1024) pour être sûr de ne pas bloquer les autres applications.

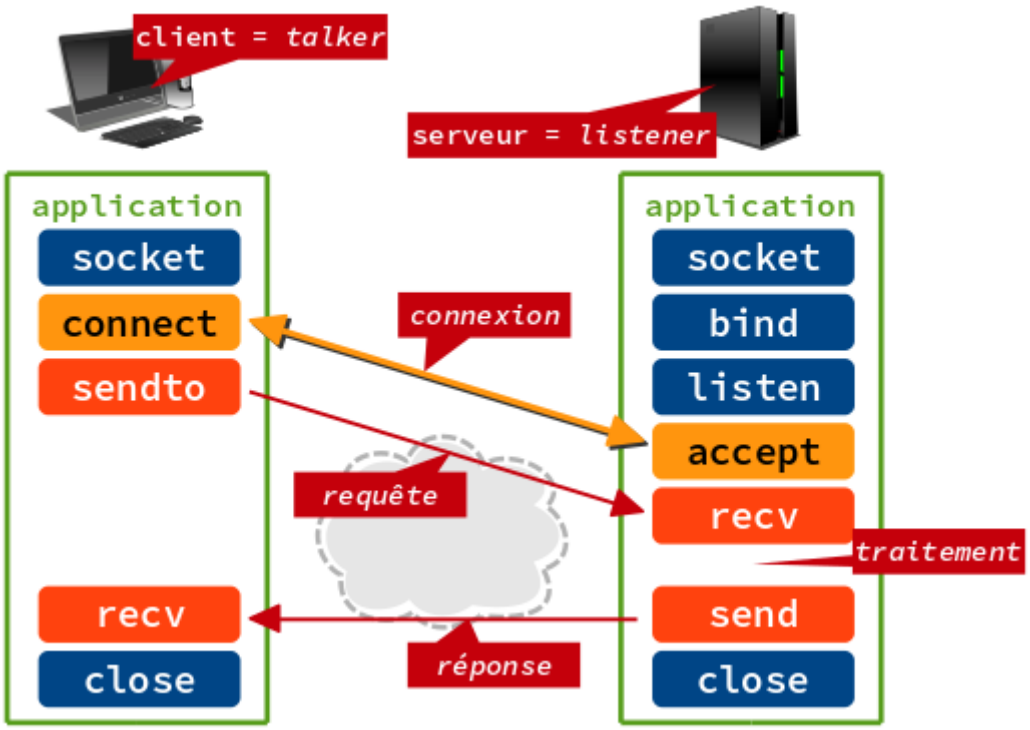

FIGURE 1 – Schéma d'une connexion TCP entre un client et un serveur. Les noms de fonctions de la librairie `socket` correspondant aux étapes sont indiqués.

— La méthode `bind` du socket est **bloquante**[1] : elle se libère lorsqu'un client se connecte.

— Les données envoyées et reçues le sont sous forme d'octets. Il faut donc convertir la chaîne de caractères en liste de *bytes* (en la précédant de la lettre b) pour l'envoyer.

— Le socket est muni d'un *buffer* dans lequel on lit les données avec la méthode `recv`, en indiquant le nombre d'octets qui seront lus.

— Dans cet exemple, le serveur accepte la connexion d'un client, attend son message, l'affiche dans la console, puis lui renvoie. Les deux sockets se terminent alors.

Pour que le socket ne se déconnecte pas après l'opération, il faut utiliser une boucle après avoir établi la connexion.

Exercice 1 — Serveur textuel simple

En reprenant les codes présentés ci-dessus, rédiger deux programmes client-serveur où le client envoie des requêtes sous forme de chaîne de caractères au serveur, puis attend la réponse. Le serveur traite la requête puis envoie la réponse. Les programmes tournent tant que l'un des deux sockets n'est pas fermé.

Exemple de commande :

— `UPPER <str>` et `LOWER <str>` : retourne la chaîne où tous les caractères seront transformés en majuscules (resp. minuscules) ;

— `DATENOW` : retourne le timestamp actuel ;

1. Une instruction bloquante met en pause le fil d'exécution jusqu'à ce qu'un certain événement se réalise : ici une connexion.

Exercice 2 — Nombre mystère en ligne

On rappelle les règles du jeu du nombre mystère : on choisit un nombre entre 1 et n. Un joueur dispose de k essais pour le trouver. A chaque fois que le joueur fait une proposition, le serveur doit donner une indication : plus grand ou plus petit. Le jeu s'arrête si le joueur a fait k propositions sans trouver, ou qu'il trouve le nombre avant.

Implémenter les programmes client/serveur.

Note : Lorsque la connexion est établie, le serveur envoie au client les paramètres du jeu avant d'écouter sa proposition.

Ces deux applications peuvent se contenter du terminal pour afficher les données car il n'y a pas de communication simultanée. Nous étudierons plus tard comment gérer une communication asynchrone.