Travaux dirigés Système n°3 : sockets

Éric Gaudefroy

École Hexagone
Cursus Cyberdéfense - M1

Objectif

L'objectif de ce TP est de manipuler les fonctions élémentaire de gestion des sockets de la bibliothèque standard Python *socket*, par le développement d'une application client/serveur d'échange de messages textuels.

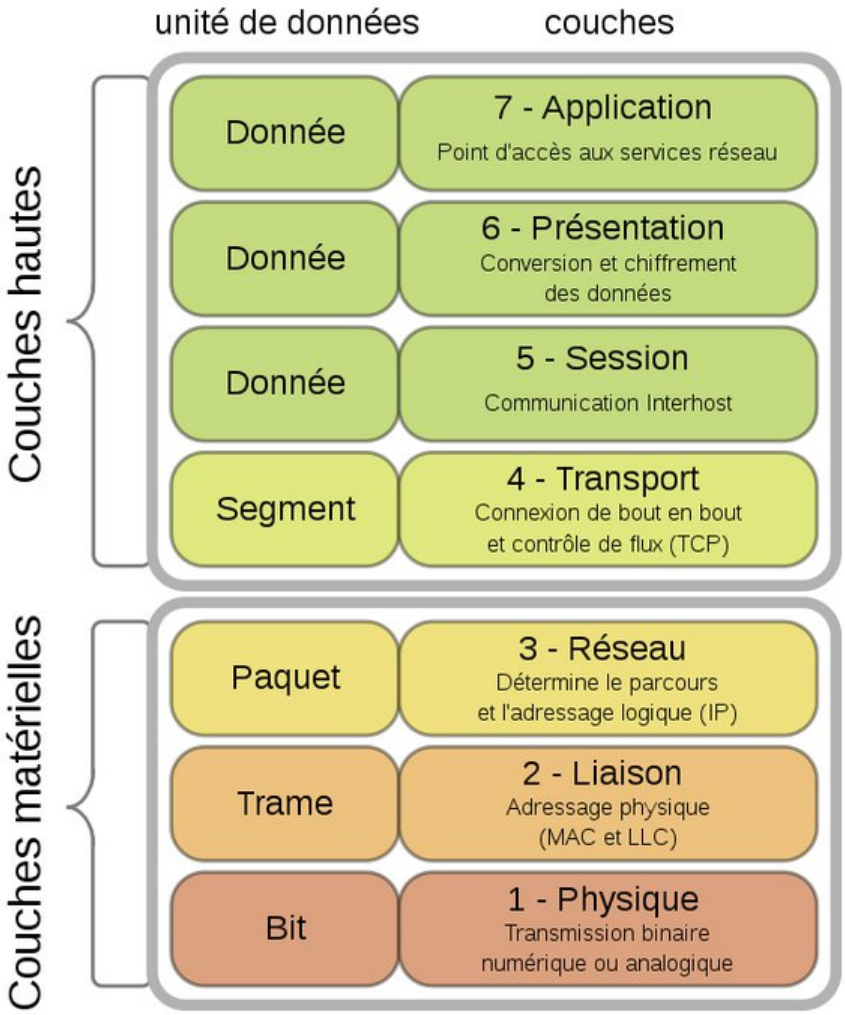


FIGURE 1 – Modèle OSI.

Le présent TD s'appuiera sur le modèle d'organisation des couches réseau *Open Systems Interconnection*, modèle de communications entre ordinateurs proposé par l'ISO (Organisation internationale de normalisation) qui décrit les fonctionnalités nécessaires à la communication et l'organisation de ces fonctions.

Exercice n°1 : création de l'application « Serveur »

1.1. création de la socket TCP/IP sur les couches OSI 1 et 2

Comme vu durant le cours « sockets », le point de départ pour la création d'une application utilisant les sockets est la création d'un objet « socket ». Sous Python, l'appel à la fonction *socket.socket()* créé un objet « socket » qui supporte un type de contexte. Il n'est pas nécessaire de clore la socket avec *s.close()* :

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    pass  # Use the socket object without calling s.close().
```

Les arguments passés à *socket()* sont des constantes précisant la famille d'adressage (IPv4 ou IPv6) et le type de socket. AF_INET est la famille d'adresse Internet correspondant à IPv4, SOCK_STREAM est le type de socket TCP. C'est ce protocole qui va être utilisé pour le transport des messages dans le réseau.

1.2. montage des couches protocolaires de niveau OSI 3 et 4

La méthode *.bind()* permet d'associer la socket avec une interface réseau physique spécifique et définir le port sur lequel le serveur va se placer en écoute :

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    # ...
```

Les paramètres passés à *.bind()* dépendent de la famille d'adresse définie pour la socket. Dans notre exemple, nous utilisons socket.AF_INET (IPv4). Donc, *bind()* attend un tuple (hôte, port).

L'hôte peut être un *hostname*, une adresse IP ou une chaine vide. Si une adresse IP est proposée, l'hôte doit être une chaine d'adresse IPv4 formattée. L'adresse IP 127.0.0.1 est l'adresse IPv4 standard de l'interface *loopback* qui n'autorise que les communications internes à la machine. Si une chaine vide est passée en argument, le serveur acceptera toutes les connexions depuis toutes les interfaces. Cela représente un sérieux risque sur la sécurité, notamment en phase de développement. Le port représente le numéro de port TCP qui accepte les connexions des clients. Il permet ainsi de définir l'application en couches hautes qui recevra les données reçues. Il s'agit d'un entier de 1 à 65535, 0 étant réservé au système. Dans leur grande majorité, les systèmes d'exploitation obligent de disposer des droits *root* pour accéder aux ports de 1 à 1024.

Il est à noter que l'utilisation d'un *hostname* est tributaire du système de résolution d'adresse, soit le fichier *hostname* local, soit une architecture DNS.

1.3. mise en écoute de la socket

La fonction *.listen()* autorise le serveur à accepter les connexions externes. Il permet ainsi de proposer une « socket en écoute » :

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    # ...
```

La méthode *.listen()* dispose d'un paramètre *backlog*. Il permet de spécifier le nombre de connexions en attente avant de refuser les nouvelles demandes. Depuis Python 3.5, c'est optionnel.
Si le serveur reçoit un grand nombre de connexions simultanées, augmenter la valeur du backlog peut aider en définissant le taille maximale de la file d'attente pour les connexions en attente. Cette valeur maximale dépend de l'OS. Sous Linux, voir /proc/sys/net/core/somaxconn.

1.4. attente de connexion d'un client

La méthode *.accept()* bloque l'exécution et attend l'arrivée de connexions entrantes. Lorsqu'un client se connecte, la méthode renvoie un nouvel objet « socket » représentant la connexion et le tuple

contenant l'adresse IP du client. Il contient l'hôte et le port pour IPv4 ou (host, port, flowinfo, scopeid) pour IPv6.

Point important : il est impératif de comprendre qu'après *.accept()*, vous disposez d'une nouvelle socket pour communiquer avec le client. Elle est distincte de la socket en écoute qui permet au serveur d'accepter de nouvelles connexions.

1.5. code Serveur

```
import socket

HOST = "127.0.0.1"  # Standard loopback interface address
PORT = 65432  # Port to listen on (non-privileged ports are > 1023)

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Après qu'*accept()* ait offert au client un objet socket nommé conn, une boucle infinie permet pour outrepasser le blocage imposé par la fonction **conn.recv()**. Elle lit n'importe quel data envoyé par le client et la renvoie à l'expéditeur grâce à la fonction **conn.sendall()**.
Si **conn.recv()** renvoie un objet *byte* vide ('b'), cela indique que le client à terminé la connexion et que la boucle infinie est terminée. La fonction *with* est utilisée pour fermer automatiquement la socket à la fin du block.

1.6. Tests de bon fonctionnement

Afin de vérifier le bon fonctionnement de notre code Serveur, nous allons l'exécuter et contrôler l'existence d'un port en écoute. Positionner vous dans le répertoire contenant le script serveur et exécuter le :

```
python3 server_1.py
```

Dans une autre fenêtre shell, contrôler l'existence de la socket :

```
netstat -an
```

Vous obtenez la liste des sockets ouvertes, et notamment :

```
Connexions Internet actives (serveurs et etablies)
Proto Recv-Q Send-Q Adresse locale          Adresse distante Etat
...
tcp        0      0 127.0.0.1:65432         0.0.0.0:*        LISTEN
...
```

Il est à noter que l'adresse locale correspond ici à *localhost*, conformément à la configuration choisie lors de l'appel à la fonction **.bind()**.

Afin de permettre à votre serveur d'écouter sur toutes les interfaces réseau, modifier le script comme suit :

```
HOST = ""
```

En relançant la commande **netstat -an**, vous obtenez :

```
Connexions Internet actives (serveurs et etablies)
Proto Recv-Q Send-Q Adresse locale           Adresse distante Etat
...
tcp         0      0 0.0.0.0:65432           0.0.0.0:*        LISTEN
...
```

L'adresse générique « 0.0.0.0 » indique que le port est atteignable depuis toutes les adresses IP connues de la machine. Il pourra donc être atteint depuis une autre machine située dans l'un des réseaux supportés par le serveur.
La commande *lsof* permet d'en savoir plus sur le processus à l'origine de la socket :

```
lsof -i -n
```

Nous y voyons notamment le nom de la commande, son PID, le propriétaire du processus et, evidemment, le port en écoute.

```
COMMAND  PID USER   FD   TYPE DEVICE SIZE/OFF NODE NAME
python3 5302 user    3u  IPv4  72567      0t0  TCP *:65432 (LISTEN)
```

1.6. code Client

La partie Client répond à la même logique. En comparaison au serveur, le client est bien plus simple. Il créé un objet socket, utilise *.connect()* pour se connecter au serveur en écoute et appelle *.sendall()* pour envoyer ses datas. Enfin, il execute *.recv()* pour obtenir les éléments lui ayant été envoyés et les afficher.

```
import socket

HOST = "127.0.0.1"  # The server's hostname or IP address
PORT = 65432  # The port used by the server

with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((HOST, PORT))
    s.sendall(b"Hello, world")
    data = s.recv(1024)

print(f"Received {data!r}")
```

En exécutant le serveur, vous constatez qu'il apparait bloqué. Il est en effet suspendu par *.accept()* et attend la connexion d'un client.

1.6. communication locale

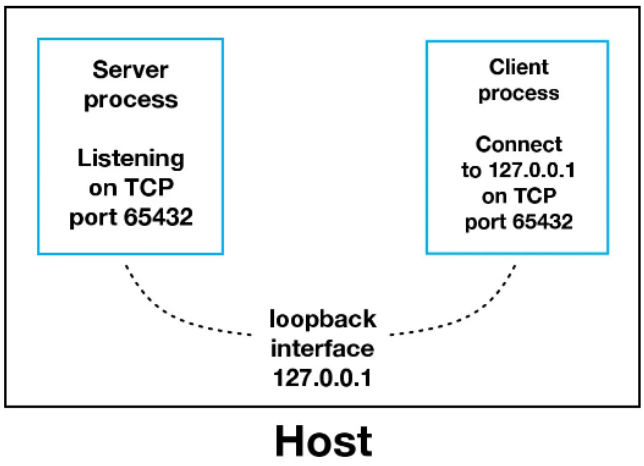


FIGURE 2 – Schéma global.

En utilisant l'interface *loopback* (adresse IPv4 127.0.0.1 ou IPv6 : :1), les données ne quittent amais l'hôte vers un réseau externe. Dans la figure 2, l'interface *loopback* est contenue dans l'hôte. Cela montre le caractère interne de cette interface. Cette interface représente donc une solution sécurisée pour créer des communications internes à un ordinateur.

Dans un premier temps, nous allons tenter de nous connecter à la socket serveur en écoute grâce au script client. Pour cela, dans une première fenêtre Shell, lancer le serveur :

```
python3 server_1.py
```

Cette application est à présent en attente de connexion client. Dans une seconde fenêtre Shell, lancer le client et observer le fonctionnement :

```
python3 client_1.py
```

Comme choisi dans le script client, ce dernier va établir une connexion sur le port 65432 de la machine locale (*localhost*) grâce à une socket en mode TCP sur IPv4. Lorsque cette connexion est établie, le script client envoie une chaine de caractère "Hello, world" vers le serveur puis ouvre un buffer de 1024 bits prêt à recevoir les données transmises par le serveur. Lorsque ces données sont reçues, elles sont affichées, puis le script se termine.

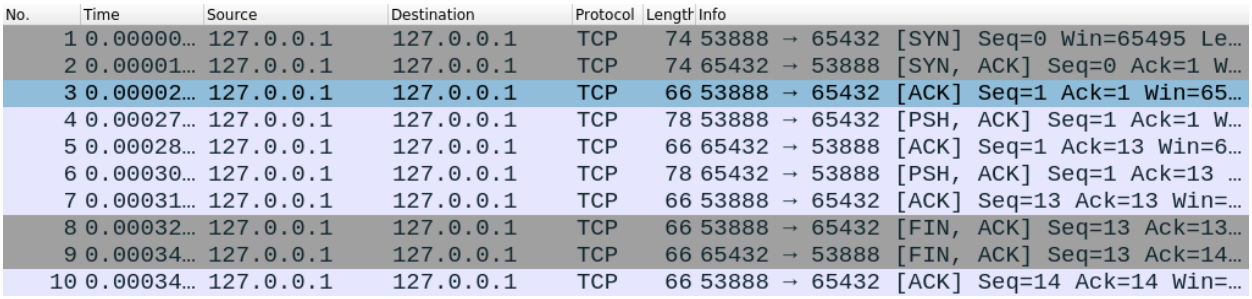
Côté Serveur, l'arrivée d'une connexion cliente permet de lever le blocage de la fonction **.accept()**. Le serveur va alors afficher l'IP du client puis lancer une boucle de réception des données transmises par le client. Le script utilise un buffer de 1024 bits pour stocker ces éléments. Lorsque toutes les données ont été reçues, le script renvoie vers le client le contenu des données reçues grâce à la fonction **.sendall()** : le message du client est collationné puis le serveur se termine.

1.7. analyse réseau

Afin d'identifier précisément les échanges réseaux réalisés par nos applications, nous pouvons lancer un enregistrement réseau avec Wireshark juste avant de relancer un test client-serveur.
Pour cela, il est nécessaire de disposer du logiciel Wireshark installé. Lancer le en mode privilégié dans une nouvelle fenêtre Shell afin d'accéder à la capture de toutes les interfaces disponibles.

```
sudo wireshark
```

Notre serveur étant configuré en écoute sur *localhost*, choisir de capturer l'interface *loopback*. Lancer le script serveur, puis le client. Enfin, stopper la capture réseau. Nous obtenons l'échange suivant :



No.	Time	Source	Destination	Protocol	Length	Info
1	0.00000…	127.0.0.1	127.0.0.1	TCP	74	53888 → 65432 [SYN] Seq=0 Win=65495 Le…
2	0.00001…	127.0.0.1	127.0.0.1	TCP	74	65432 → 53888 [SYN, ACK] Seq=0 Ack=1 W…
3	0.00002…	127.0.0.1	127.0.0.1	TCP	66	53888 → 65432 [ACK] Seq=1 Ack=1 Win=65…
4	0.00027…	127.0.0.1	127.0.0.1	TCP	78	53888 → 65432 [PSH, ACK] Seq=1 Ack=1 W…
5	0.00028…	127.0.0.1	127.0.0.1	TCP	66	65432 → 53888 [ACK] Seq=1 Ack=13 Win=6…
6	0.00030…	127.0.0.1	127.0.0.1	TCP	78	65432 → 53888 [PSH, ACK] Seq=1 Ack=13 …
7	0.00031…	127.0.0.1	127.0.0.1	TCP	66	53888 → 65432 [ACK] Seq=13 Ack=13 Win=…
8	0.00032…	127.0.0.1	127.0.0.1	TCP	66	53888 → 65432 [FIN, ACK] Seq=13 Ack=13…
9	0.00034…	127.0.0.1	127.0.0.1	TCP	66	65432 → 53888 [FIN, ACK] Seq=13 Ack=14…
10	0.00034…	127.0.0.1	127.0.0.1	TCP	66	53888 → 65432 [ACK] Seq=14 Ack=14 Win=…

FIGURE 3 – Capture Wireshark d'une connexion via *loopback*.

Nous identifions les éléments suivants :

- les adresses IPv4 source et destination fixées à 127.0.0.1, ce qui est cohérent avec une communication interne à l'hôte via l'interface *loopback*, ou *localhost* ;
- le port client, fixé aléatoirement lors de la création de la socket client, ici à 53888 ;
- le port serveur, fixé à 65432 dans nos scripts Python ;
- la séquence TCP, définie ci-après.

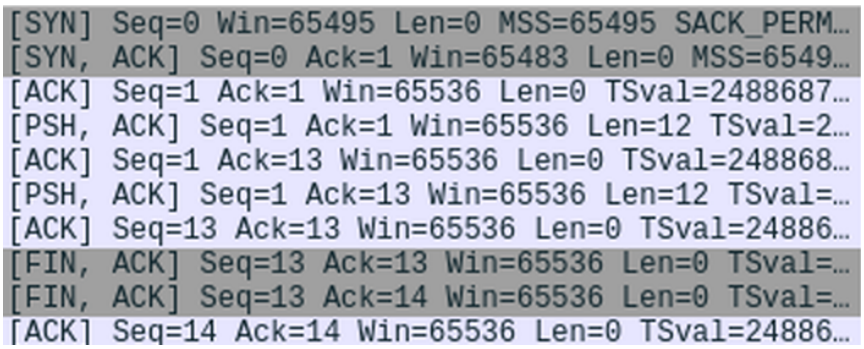


FIGURE 4 – Echange TCP.

L'échange réseau réalisé par nos applications client/serveur est caractéristique des échanges TCP :

- un *three-way handshake* SYN/SYN-ACK/ACK permettant d'établir la connexion entre le client et le serveur ;
- une séquence PUSH du client vers le serveur permettant l'envoi des données, suivie de sa séquence d'acquittement ACK ;
- une séquence PUSH du serveur vers le client correspondant à l'envoi des données de réponse du serveur, suivie de sa séquence d'acquittement ACK ;
- séquence de clôture de la communication FIN/ACK.

Hexagone 2023