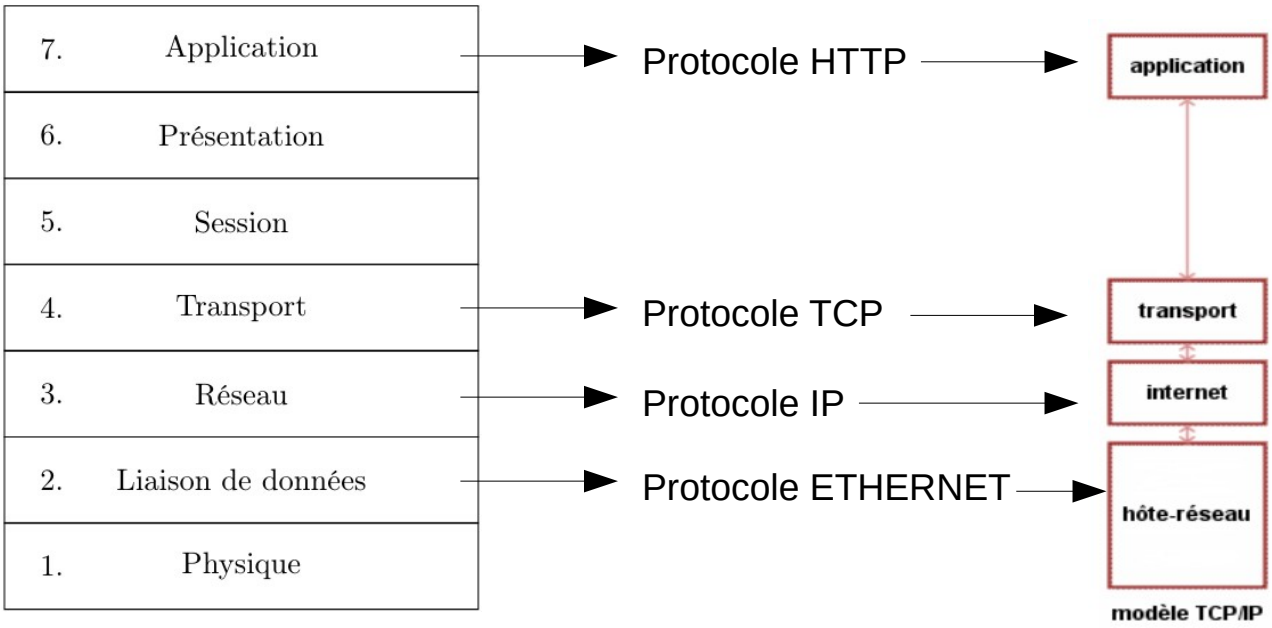
TP R107 – Python3
Programmation réseau avec Python3

1 Introduction

Notions de réseau :

Dans ce TP, vous allez devoir implémenter des serveurs et des clients utilisant un protocole réseaux : **TCP**. Ce protocole est situé au niveau de la **couche 4** du **modèle OSI** : la **couche transport**. Nous utiliserons **IP** pour la **couche 3** (couche réseau) et, la plupart du temps, **Ethernet** pour la **couche 2** (liaison de données). Ainsi que le protocole applicatif **HTTP**, **couche 7 Application**.

les 7 couches du modèle OSI :

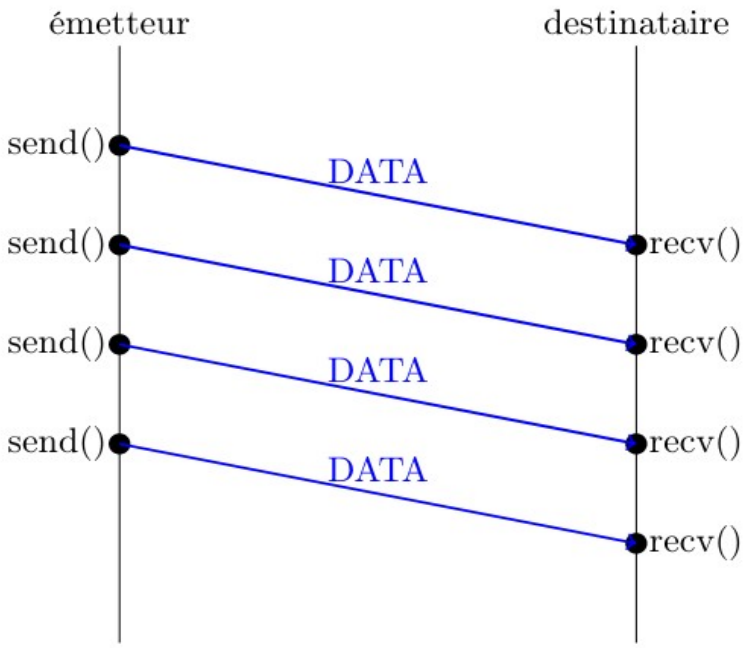


Le protocole UDP

Le protocole UDP est un protocole simple et non-fiable. Il **fonctionne en mode non-connecté** et les messages envoyés ne sont pas acquittés (ce qui signifie que le récepteur n'envoie pas automatiquement de message pour indiquer qu'il a bien reçu). Ce protocole ne garantie pas à l'émetteur d'un message que le destinataire l'a bien reçu, ni que celui-ci n'a pas été altéré.

L'absence de phase de connexion permet à l'émetteur de commencer l'envoi de ses données dès le premier paquet (en TCP, nous verrons que l'établissement de la connexion suppose l'échange de plusieurs messages, ce qui demande du temps).

Comme le montre le schéma ci-dessous, les paquets sont simplement envoyés entre les deux processus sans aucun paquet supplémentaire ajouté par le protocole. Les paquets du message sont envoyés directement dès le début de la communication. Lorsque la communication est terminée, on arrête simplement d'envoyer des paquets.



Le protocole UDP est donc léger et efficace, et se limite essentiellement à introduire la notion de port à IP : le numéro de port permettra de distinguer le processus (ou service) destinataire du paquet. UDP est en général utilisé par les applications qui privilégient la performance à la fiabilité : par exemple, des applications multimédia qui peuvent se permettre de perdre des paquets (qu'il ne vaut pas la peine de retransmettre puisque de toutes façons ils arriveraient trop tard : Visio, Vidéo, son). UDP utilise donc un système d'adressage sur la machine permettant à une machine donnée d'être impliquée dans plusieurs communications sans les mélanger : ce sont les ports réseau.

Le protocole TCP :

Le **protocole TCP** fonctionne en **mode connecté** : une connexion doit être établie entre deux processus pour que des messages puissent être envoyés entre eux.

On a alors la notion de client et de serveur : le serveur attend les connexions des clients et le client se connecte à un serveur.

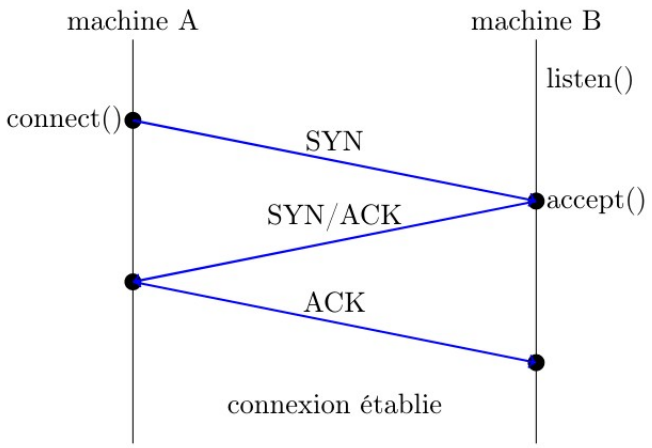
Une fois la connexion établie, les messages peuvent être envoyés du **client** vers le **serveur** ou du **serveur** vers le **client**, sans distinction (**liaison bidirectionnelle, dite full duplex**).

À la fin de la séquence de communication, il est impératif de fermer la connexion.

L'établissement de la connexion se fait en utilisant le protocole dit « **triple handshake** »:

- Le client envoie une requête de connexion : paquet SYN
- Le serveur accepte : paquet SYN/ACK
- Le client acquitte la réception de l'acceptation : paquet ACK

Schémat où on l'on voit les paquets échangés lorsque la machine A se connecte en TCP à la machine B



2 Comment faire une connexion avec Python3

Les sockets :

Une socket est un point terminal sur le réseau. C'est la **structure de donnée** qui permet de communiquer sur le réseau.

Lorsque l'on écrit un **programme communiquant** sur le réseau, on **manipule des sockets** pour **établir ou fermer les connexions** dans les cas des protocoles connectés, **envoyer et recevoir des données sur le réseau**.

Programmation réseau sur les sockets :

En Python, les fonctions associées aux **sockets** sont fournies dans le **module socket**. Vous devez donc, en premier lieu, **importer ce module**.

Votre script Python devra commencer par :

```
#!/usr/bin/env python3

import socket
```

3 Programmation client-serveur :

Le serveur UPD en Python3 :

Nous avons vu que le **protocole UDP** fonctionne, en **mode non connecté** :

La socket à ouvrir pour une communication **UDP** est de type « **SOCK_DGRAM** ».

```
serveur = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Voici un exemple de programme serveur « ***simpleUDPServeur*.py** » qui ouvre un **socket UDP** et l'associe au port de votre choix (ici **3000**) :

```
#!/usr/bin/env python3

import socket

serveur = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# permet de libérer le PORT après un CTRL + C
serveur.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

serveur.bind(('localhost', 3000))     # Utilise le port réseau 3000
print("Serveur UDP en écoute sur 3000\n")

while True :
    # utilisation de recvfrom pour récupérer le tuple address
    request, address = serveur.recvfrom(1024)
    print("Message client : ", request.decode("utf-8"))
    print("IP du client connecté : ", address)
    serveur.sendto(b"i am the server", address)

serveur.close()
```

Nous avons maintenant un mini serveur qui accepte les connexions **UDP** sur le PORT **3000** et afficher dans la console l'IP du client qui s'y connecte.

Tester : Enregistrez ce fichier dans votre VM Linux et exécutez le via la commande :

~$ python3 ./*simpleUDPServeur*.py

dans un terminal.

Le serveur affiche : « **Serveur UDP en écoute sur 3000** » et attend des messages d'un client (boucle infinie : **while True**)

Pour tester, vous pouvez utiliser la commande linux « **netcat -u localhost 3000** », l'option

Ensuite tapez un mot et faites « Entrée » le serveur affichera votre chaîne de caractères et l'IP de votre poste client et répondra sur la console du client « **i am the server** »

client UPD en Python3

Pour envoyer des messages au serveur UDP, nous devons définir un « **tuple** » que l'on appelle **addrPort** et qui contient l'IP du serveur et le Port réseau sur lequel il écoute.

```
addrPort = ("127.0.0.1", 3000)
```

Puis, nous créons un **« socket »** client de type UDP, comme pour le serveur, avec le type **SOCK_DGRAM**

```
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

Enfin, il suffit d'envoyer des messages au serveur, via notre variable « **addrPort** » et la fonction « **sendto(…)** » sans avoir à réaliser une connexion, puisque **UDP est en mode déconnecté** :

```
client.sendto(b"Hello from client", addrPort)
```

On peut également attendre une réponse du serveur via la fonction « **recv(1024)** » que l'on décode en « **utf-8** », car les données transitent au format « **bytes** » sur le socket, puis on affiche le msg du serveur dans la console :

```
msgServer = client.recv(1024).decode('utf-8')

print("Message du serveur : ", msgServer)
```

Pour terminer, on ferme le « socket » client que l'on a ouvert au début :

```
client.close()
```

Voici le code complet de notre client UDP « **simpleUDPclient.py** » :

```
#!/usr/bin/env python3

import socket

addrPort = ("127.0.0.1", 3000)

client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

client.sendto(b"Hello from client", addrPort)

msgServer = client.recv(1024).decode('utf-8')

print("Message du serveur : ", msgServer)

client.close()
```

client et serveur TCP

Nous avons vu que le **protocole TCP** fonctionne, en **mode connecté** :

il est nécessaire de commencer par établir une connexion entre le client et le serveur, et de terminer l'échange en fermant la connexion.

Serveur TCP

La socket à ouvrir pour une communication TCP est de type **SOCK_STREAM**.

Voici un exemple de programme serveur « simpleServeurSocket.py » qui ouvre une **socket TCP** et l'associe au port de votre choix (ici 3000) :

```
#!/usr/bin/env python3

import socket

# initialisation du serveur

serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serveur.bind(('', 3000))   # Ecoute sur le port 3000
serveur.listen()
```

Nb : Notre serveur n'est pas encore opérationnel...

Nous devons maintenant ajouter un appel à une fonction particulière qui permet d'attendre les connexions d'éventuels clients sur notre serveur :

```
(...)
while True :
  client, infosclient = serveur.accept()
  request = client.recv(1024)
  print("IP client connecté: ",socket.gethostbyname(socket.gethostname()))
  client.close()

serveur.close()
```

Nous avons maintenant un mini serveur qui accepte les connexions **TCP** sur le PORT **3000** et afficher dans la console l'IP du client qui s'y connecte.

Tester : Enregistrez ce fichier dans votre VM Linux et exécutez le via la commande :

~$ python3 ./simpleServeurSocket.py

dans un terminal.

Le prompt ne réaparaît pas, c'est parce que le serveur s'exécute (boucle infinie : **while True**)

Le code complet de notre « simpleServeurSocket.py »

```
#!/usr/bin/env python3

import socket

# initialisation du serveur

serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serveur.bind(('', 3000))   # Ecoute sur le port 3000
serveur.listen()

while True :
  client, infosclient = serveur.accept()
  request = client.recv(1024)
  print(request.decode("utf-8")) #affiche les données du client
  print("IP client connecté: ",socket.gethostbyname(socket.gethostname()))
  client.close()

serveur.close()
```

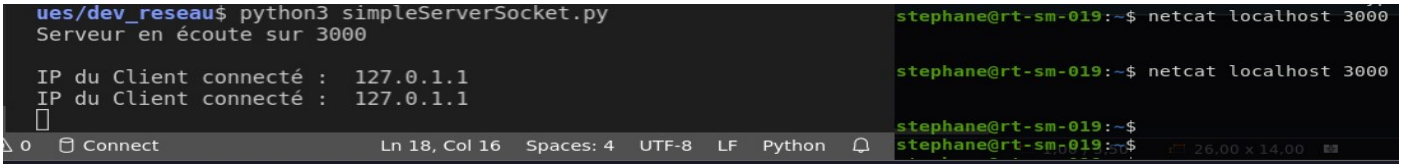
Pour pouvoir tester notre serveur, il nous faut **une commande linux** qui permet de faire un **connexion TCP** sur le **PORT 3000**.

Linux nous met à disposition un commande qui peut réaliser ce test : « **netcat** ».

Regardez la documentation de **netcat** et tester votre serveur **depuis une autre fenêtre** de Terminal.

Que voyez vous apparaître dans le terminal du serveur.

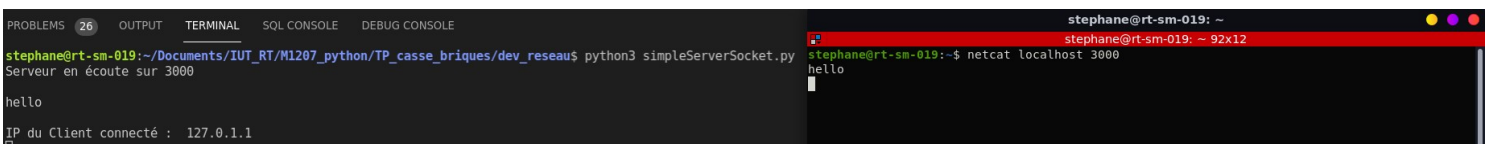
Résultat du test de connexion :



A droite le terminal qui lance le test

A gauche la console de vscode qui exécute notre programme « simpleServeur.py »

Lorsque l'on fait entrée après avoir validé la commande « **netcat** » puis tapez sur la touche « **Entrée** » pour que le serveur le serveur affiche dans la console du serveur l'adresse IP (ici le localhost). Si vous tapez une chaine de caractère dans **netcat**, le serveur l'affiche dans la console serveur.



Le client TCP :

Maintenant que nous avons testé notre serveur TCP via la commande « **netcat** » on peut développer un client TCP en Python3.

Pour se faire il nous faut également un variable « **socket** » TCP comme sur le serveur :

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

Puis on utilise la fonction « **connect(...)** »

```
client.connect(("localhost", 3000))
```

Enfin on utilise la fonction « **sendall(...)** » au format « **bytes** » pour envoyer des données sur le serveur

```
client.sendall(b"hello serveur\n")
```

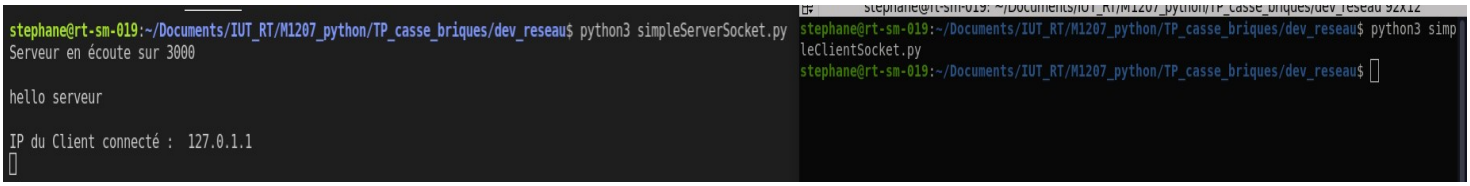
Testez le client après avoir démarrer le serveur.

Voici le code complet de « simpleclientSocket.py »

```
#!/usr/bin/env python3
import socket

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(("localhost", 3000))
client.sendall(b"hello serveur\n")
client.close()
```

Voici le résultat du test de notre programme « **simpleclientSocket.py** »



Conclusion :

Nous avons vu comment mettre en œuvre une intéraction entre deux programmes, donc deux processus système via une connexion TCP avec le langage Python3.

Notre programme est volontairement simple pour se concentrer sur les API de Python qui permettent de faire écouter un programme sur un PORT logique via une boucle sans fin. Vous avez donc créez votre premier « **daemon** » Linux.

Pour améliorer notre programme nous allons voir comment le rendre **multi-utilisateurs**, c'est à dire, que plusieurs clients puissent se connecter sur le serveur et échanger des informations entre eux.

4 Le Multi-Tread, pour un serveur multi connexions

Mais également un client multi thread capable de recevoir les messages du serveur tout en attendant la saisie de l’utilisateur.

*** En cours de rédaction, l’approche du « **multi thread** » ci-dessous les codes sources fonctionnels **

Code source du programme « **multiThreadServerSocketTCP.py** » :

```
#!/usr/bin/env python3
# Serveur TCP Multi Thread
import socket
import os
from _thread import *

ServerSocket = None
host = '127.0.0.1'
port = 9090
clients = []
nbclients = 0
numclient = None

def main():
    global nbclients
    ServerSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ServerSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    try:
        ServerSocket.bind((host, port))
    except socket.error as e:
        print(str(e))
    finally:
        print('Waiting for a Connection..')
        ServerSocket.listen(5)
    while True:
        client, address = ServerSocket.accept()
        print('Connected to: ' + address[0] + ':' + str(address[1]))
        client.send(str.encode(str(nbclients)))
        clients.append(client)
        print("Liste clients : ", clients)
        start_new_thread(threaded_client, (client, ))
        nbclients+=1
        print('Thread Number: ' + str(nbclients))

def threaded_client(connection):
    global nbclients
    print("connection", connection)
    while True:
        data = connection.recv(2048)
        reply = '\n>>' + data.decode('utf-8') + '\n'
        for client in clients:
            client.sendall(str.encode(reply))

        if data == "quit":    # Bogue sur le quit !
            numclient = int(connection.recv(2048))
            clients[numclient].close()
            clients.pop(numclient)
            nbclients-=1

if __name__== "__main__":
    main()
```

Code source du programme client « **multiThreadClientSocketTCP.py** »

```
#!/usr/bin/env python3

import socket
import os
from _thread import *

ClientSocket = None
host = '127.0.0.1'
port = 9090
myNumber = 0

def main():
    ClientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    ClientSocket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    try:
        ClientSocket.connect((host, port))
    except socket.error as e:
        print(str(e))
    finally:
        print("connected to Server !")
        myNumber = int(ClientSocket.recv(1024))
        print("myNumber client received : ", myNumber)
    start_new_thread(threaded_server, (ClientSocket, myNumber))
    while True:
        msg = input('')  # bloquant les retours => nécessite un thread
        ClientSocket.send(str.encode(msg))
        if msg == "quit":   # Bogue sur le quit !
            ClientSocket.send(str.encode(str(myNumber)))
            break

def threaded_server(connection, num):
    while True:
        response = connection.recv(1024)
        print(response.decode('utf-8'))


if __name__ == "__main__":
  main()
```