Programmation réseau avec les sockets

Programmation réseau avec les sockets

Sommaire

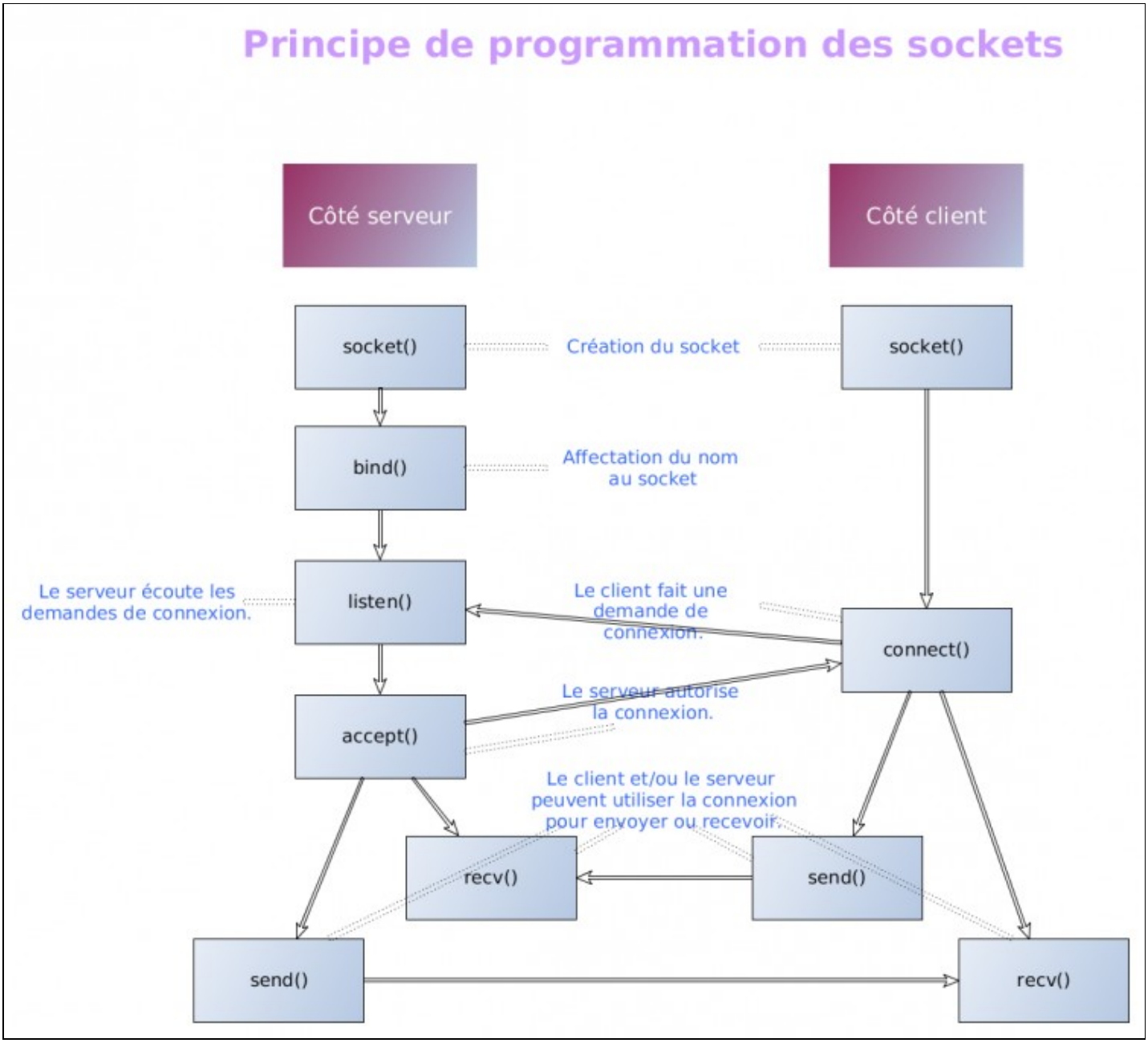
Rappels sur TCP/IP et concepts de base de l'API socket[modifier]

Rappels sur les protocoles TCP, IP, UDP[modifier]

Page sur les rappels réseau.

Principes de la programmation des sockets STREAM[modifier]

- un socket est un point de terminaison d'un communication réseau.



Programmation des sockets

socket()[modifier]

```
int socket(int domain, int type, int protocol);
```

- domain : AF_INET|AF_INET6
  - AF_INET : IPv4 Internet protocols ip
  - AF_INET6 : IPv6 Internet protocols ip
- type : SOCK_STREAM|SOCK_DGRAM
  - SOCK_STREAM Fournit des flux d'octets séquentiels, fiables, bidirectionnels et basés sur la connexion. Un transfert de données out-of-band (circuit virtuel).
  - SOCK_DGRAM Prend en charge les datagrammes (messages sans connexion, non fiables d'une longueur maximale fixe).
- protocol : SOCK_STREAM (flux full duplex).

bind()[modifier]

bind sert à lier un nom à une socket.

```
int bind(int sockfd, const struct sockaddr *addr,socklen_t addrlen);
```

- Lorsqu'un socket est créé avec la primitive **socket()**, il existe dans un espace de noms (famille d'adresses) mais aucune adresse ne lui est affectée.
- **bind()** assigne l'adresse spécifiée par **addr** au socket référencé par le descripteur de fichier **sockfd**.
- **addrlen** spécifie la taille, en octets, de la structure d'adresse pointée par addr.
- Traditionnellement, cette opération est appelé «attribuer un nom à une socket».
- Il est normalement nécessaire d'attribuer une adresse locale à l'aide de **bind()** avant qu'une socket SOCK_STREAM puisse recevoir une connexion avec la primitive **accept()**.

Détail de la structure sockaddr_in

```
/* Structure describing an Internet socket address.  */
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    in_port_t sin_port;                 /* Port number.  */
    struct in_addr sin_addr;            /* Internet address.  */

    /* Pad to size of `struct sockaddr'.  */
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           __SOCKADDR_COMMON_SIZE -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
  };
```

listen()[modifier]

```
int listen(int sockfd, int backlog);
```

La primitive **listen()** marque le socket désigné par **sockfd** comme un socket passif, c'est-à-dire comme un socket qui sera utilisé pour accepter les demandes de connexion entrantes en utilisant accept().

L'argument **sockfd** est un descripteur de fichier qui fait référence à un socket de type **SOCK_STREAM** ou **SOCK_SEQPACKET**.

L'argument **backlog** définit la longueur maximale de la file d'attente des connexions en attente pour **sockfd**.

- Si une demande de connexion arrive lorsque la file d'attente est pleine, le client peut recevoir une erreur avec une indication **ECONNREFUSED** ou, si le protocole sous-jacent prend en charge la retransmission, la demande peut être ignorée afin qu'une nouvelle tentative de connexion ultérieure réussisse.

accept()[modifier]

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

L'appel système accept () est utilisé avec les types de socket basés sur la connexion (SOCK_STREAM, SOCK_SEQPACKET). Il extrait la première demande de connexion dans la file d'attente des connexions en attente pour le socket d'écoute, sockfd, crée un nouveau socket connecté et renvoie un nouveau descripteur de fichier faisant référence à ce socket. Le socket nouvellement créé n'est pas à l'état d'écoute. Le socket d'origine sockfd n'est pas affecté par cet appel.

L'argument sockfd est un socket qui a été créé avec socket (2), lié à une adresse locale avec bind (2) et qui écoute les connexions après un listen (2).

send()[modifier]

```
ssize_t send(int sockfd, const void *buf, size_t len, int flags);
```

- sockfd : descripteur du socket.
- buf: adresse du buffer.
- len : longueur du buffer.

L'appel système send() est utilisés pour transmettre un message à une autre socket.

- L'appel send() ne peut être utilisé que lorsque le socket est dans un état **connecté** (afin que le destinataire prévu soit connu).

recv()[modifier]

```
ssize_t recv(int sockfd, void *buf, size_t len, int flags);
```

La primitive **recv()** est utilisée uniquement sur une socket connectée.

Si aucun message n'est disponible sur le socket, les appels de réception attendent l'arrivée d'un message, sauf si le socket n'est pas bloquant (voir **fcntl()**, auquel cas la valeur -1 est renvoyée et la variable externe errno est définie sur EAGAIN ou EWOULDBLOCK.

- Les appels reçus renvoient normalement toutes les données disponibles, jusqu'à concurrence du montant demandé, plutôt que d'attendre la réception du montant total demandé.

Utilisation du module socket[modifier]

le module socket[modifier]

Ce module permet d'accéder à l'interface de socket BSD.

- Il est disponible sur tous les systèmes Unix modernes, Windows, MacOS et probablement des plates-formes supplémentaires.

Remarque : Certains comportements peuvent dépendre de la plate-forme, car des appels sont effectués vers les API de socket du système d'exploitation.

L?interface Python est une adaptation simple de l?appel système Unix et de l?interface de bibliothèque pour les sockets vers le style orienté objet de Python:

- la fonction socket () renvoie un objet socket dont les méthodes implémentent les différents appels système socket.
- Les types de paramètres sont un peu plus évolués que dans l'interface C, comme pour les opérations read () et write () sur les fichiers Python.
- L'allocation de tampon sur les opérations de réception est automatique et la longueur du tampon est implicite sur les opérations d'envoi.

Les familles de socket[modifier]

Plusieurs type de socket sont supportés par ce module :

- AF_UNIX : socket mappé sur nom de fichier unix.
- AF_INET : le socket est identifé par une paire (adresse, n° de port).
- AF_INET6 : le socket est identifé par les arguments (host, port, flowinfo, scopeid).
    - flowinfo : Le champ **sin6_flowinfo** est nouveau dans Linux 2.4. Il est utilisé de manière transparente par le noyau lorsque la

longueur d'adresse passée le contient. Certains programmes qui passent un tampon d'adresse plus long, puis vérifient la longueur de l'adresse sortante peuvent s'arrêter.
- scopeid : **scopeid** est nécessaire pour déterminer de manière unique quel est le périphérique qui doit gérer le trafic, car il peut y avoir plusieurs interfaces, avec la même IP.

Il existe de nombreux autres types:

- AF_NETLINK
- AF_BLUETOOTH
- AF_CAN
- AF_ALG
- AF_PACKET

Pour plus d'informationis, consultez [la documentation python]

Les exemples[modifier]

Exemple ipv4[modifier]

Voici un exemple simple qui illustre le fonctionnement des sockets

Le programme helloServer.py

```
''' Echo serveur
    Ce programme créer un socket..socket()
    affecte un nom au socket......bind()
    écoute sur le port ...........listen()
    accepte la connexion .........accept()
    reçoit un message ............recv()
    re envoie le message .........sendall()
    puis s'arrête.
'''
import socket

HOST = ''                 # vide = toutes les interfaces réseau.
PORT = 3000               # Un port non privilégié
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((HOST, PORT))
print ("En attente de connexion sur le port "+ repr(PORT))
s.listen(1)
conn, addr = s.accept()
print ("Connecté sur le socket {0}:{1} ".format (addr[0],addr[1]))
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.sendall(data)
conn.close()
```

Le programme helloClient.py

```
''' Echo client
    Ce programme créer un socket..socket()
    ouvre une connexion ..........connect()
    envoie des données ...........sendall()
    relit les données envoyées ...recv()
    affiche les données ..........print()
    et s'arrête.
'''
import socket

HOST = 'localhost'    # l'hote distant (ou pas ...)
PORT = 3000           # Le meme port que le serveur
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))
s.sendall(bytes("Hello, world",'UTF-8'))

''' Les données reçues sont au format byte
    il faut les transormer en string avec la fonction decode.
'''
data = s.recv(1024)
s.close()
print ("Données reçues:{"+data.decode('utf-8')+"}")
```

Exemple ipv6[modifier]

Le programme helloServer6.py

```
# Echo server
import socket
import sys

HOST = None               # vide = toutes les interfaces réseau.
PORT = 3000               # Un port non privilégié
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC,
                              socket.SOCK_STREAM, 0, socket.AI_PASSIVE):
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.bind(sa)
        print ("En attente de connexion sur le port "+ repr(PORT))
        ''' listen(1) : 1 = nombre de connexions non acceptées que le système
            avant de refuser de nouvelles connexions
        '''
        s.listen(1)
    except socket.error as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print ("Impossible d'ouvrir le socket")
    sys.exit(1)
```

```
conn, addr = s.accept()
#print 'Connected by', addr
print ("Connecté sur le socket {0}:{1} ".format (addr[0],addr[1]))
while 1:
    data = conn.recv(1024)
    if not data: break
    conn.send(data)
conn.close()
```

Le programme helloClient6.py

```
# Echo client
import socket
import sys

HOST = 'localhost'    # l'hote distant (ou pas ...)
PORT = 3000           # Le meme port que le serveur
s = None
for res in socket.getaddrinfo(HOST, PORT, socket.AF_UNSPEC, socket.SOCK_STREA
    af, socktype, proto, canonname, sa = res
    try:
        s = socket.socket(af, socktype, proto)
    except socket.error as msg:
        s = None
        continue
    try:
        s.connect(sa)
    except socket.error as msg:
        s.close()
        s = None
        continue
    break
if s is None:
    print ("Impossible d'ouvrir le socket")
    sys.exit(1)
s.sendall(bytes("Hello, world",'UTF-8'))
data = s.recv(1024)
s.close()
print ("Données reçues:{"+data.decode('utf-8')+"}")
```

Socket en mode connecté : TCP ou stream[modifier]

Socket en mode non connecté : UDP ou datagram[modifier]

UDP[modifier]

UDP : **User Datagramme Protocole**

- UDP est utile pour transmettre rapidement de petites quantités de données, depuis un serveur vers de nombreux clients.

- pas de détection d'erreur
- Adapté aux aux applications requête/réponse (DNS, NFS, voix sur IP , jeu en ligne ...)

Propriétés de UDP[modifier]

- L'**UDP** ne garantit pas la livraison des paquets de messages.
- Si pour un problème dans un réseau, si un paquet est perdu, il pourrait être perdu pour toujours.
- Puisqu'il n'y a aucune garantie de livraison assurée des messages, UDP est considéré comme un *protocole peu fiable*.
- Les mécanismes sous-jacents qui implémentent UDP n'impliquent aucune communication basée sur la connexion.
- Il n'y a **pas de streaming** de données entre un serveur UDP ou et un client UDP.
- Un client **UDP** peut envoyer plusieurs paquets distincts à un serveur UDP et il peut également recevoir plusieurs paquets distincts comme réponses du serveur UDP.
- Comme **UDP** est un protocole sans connexion, la **surcharge** impliquée dans UDP est **moindre** par rapport à un protocole basé sur une connexion comme TCP.

Attention: Le terme **non fiable** ne veut pas dire que le *réseau* n'est pas fiable. Il veut dire simplement que UDP ne gère pas le contrôle de flux à son niveau. Donc si on veut fiabiliser une liaison via UDP , il faut gérer le contrôle de flux au niveau de l'application.

intégrité des données[modifier]

L'intégrité des données est assurée par une somme de contrôle sur l'en-tête.

- L'utilisation de cette somme est cependant facultative en IPv4 mais obligatoire avec IPv6.
- Si un hôte n'a pas calculé la somme de contrôle d'un datagramme émis, la valeur de celle-ci est fixée à zéro.
- La somme de contrôle inclut également les adresses IP de la source et de la destination.

Les exemples[modifier]

Le serveur UDP[modifier]

le fichier **udpServer.py**

```
import socket
```

```
localIP     = "127.0.0.1"
localPort   = 3000
bufferSize  = 1024

msgFromServer       = "Hello cher client UDP"
bytesToSend         = str.encode(msgFromServer)

# Création d'un socket datagramme
UDPServerSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM
# Affectation d'un nom au socket (affectation d'une adresse)
UDPServerSocket.bind((localIP, localPort))
msg = "Le serveur UDP écoute sur le port {}".format(localPort)
print(msg)
# Le socket est en attente de messages entrant.
while(True):
    bytesAddressPair = UDPServerSocket.recvfrom(bufferSize)
    message = bytesAddressPair[0]
    address = bytesAddressPair[1]
    clientMsg = "Message reçu depuis le client:{}".format(message)
    clientIP  = "Addresse IP du client:{}".format(address)
    print(clientMsg)
    print(clientIP)
    # Envoie de la réponse au client.
    UDPServerSocket.sendto(bytesToSend, address)
```

Le client UDP[modifier]

le fichier **udpClient.py**

```
import socket

msgFromClient       = "Hello UDP Server"
bytesToSend         = str.encode(msgFromClient)
serverAddressPort   = ("127.0.0.1", 3000)
bufferSize          = 1024

# Création d'un socket UDP coté client.
UDPClientSocket = socket.socket(family=socket.AF_INET, type=socket.SOCK_DGRAM

# Envoie un message au serveur via la socket UDP.
UDPClientSocket.sendto(bytesToSend, serverAddressPort)
msgFromServer = UDPClientSocket.recvfrom(bufferSize)
msg = "Message reçu du serveur {}".format(msgFromServer[0])
print(msg)
```

Les sockets et la Programmation Orientée Objets[modifier]

Le module python SocketServer[modifier]

Le module **socketserver** simplifie la tâche d'écriture des serveurs réseau.

Il existe quatre classes de base:

classe **SocketServer.TCPServer** (adresse_serveur, RequestHandlerClass, bind_and_activate = True)

```
Celui-ci utilise le protocole Internet TCP, qui fournit des flux de données c
```

classe **SocketServer.UDPServer** (adresse_serveur, RequestHandlerClass, bind_and_activate = True)

```
Cela utilise des datagrammes, qui sont des paquets discrets d'informations qu
```

classe **SocketServer.UnixStreamServer** (adresse_serveur, RequestHandlerClass, bind_and_activate = True) classe **SocketServer.UnixDatagramServer** (adresse_serveur, RequestHandlerClass, bind_and_activate = True)

```
Ces classes plus rarement utilisées sont similaires aux classes TCP et UDP, m
```

Ces quatre classes traitent les demandes de manière **synchrone**.

- Chaque demande doit être terminée avant que la prochaine demande puisse être lancée.
- Cela ne convient pas si chaque demande prend beaucoup de temps, car elle nécessite beaucoup de calculs ou parce qu'elle renvoie beaucoup de données que le client est lent à traiter.
- La solution consiste à créer un processus ou un thread distinct pour gérer chaque demande.
- Les classes mixtes **ForkingMixIn** et **ThreadingMixIn** peuvent être utilisées pour prendre en charge un comportement **asynchrone**.

Création d'un serveur[modifier]

La création d'un serveur nécessite plusieurs étapes.

- Tout d'abord, vous devez créer une classe de gestionnaire de demandes en sous-classant la classe **BaseRequestHandler** et en remplaçant sa méthode **handle()**.
- Cette méthode traitera les demandes entrantes.
- Deuxièmement, vous devez instancier l'une des classes de serveur, en lui passant l'adresse du serveur et la classe de gestionnaire de requêtes.
- Appelez ensuite la méthode **handle_request()** ***ou*** *serve_forever()'* de l'objet serveur pour traiter une ou plusieurs requêtes.
- Enfin, appelez **server_close()** pour fermer le socket.

Lorsque vous héritez de **ThreadingMixIn** pour le comportement de connexion par thread, vous devez déclarer explicitement comment vous souhaitez que vos threads se comportent lors d'un arrêt brutal.

- La classe **ThreadingMixIn** définit un attribut **daemon_threads**, qui indique si le serveur doit ou non attendre la fin du thread.
- Vous devez définir le flag explicitement si vous souhaitez que les threads se comportent de manière autonome.
- La valeur par défaut est **False**, ce qui signifie que Python ne se fermera pas tant que tous les threads créés par **ThreadingMixIn** ne seront pas terminés.

Les exemples[modifier]

Le fichier tcpHandler.py (serveur)[modifier]

```
import socketserver

class TCPHandler(socketserver.BaseRequestHandler):
    """
    Démonstration d'une classe Server TCP.

    Il faut implémenter la méthode handle() pour
    échanger des données avec le client.

    """

    def handle(self):
        # self.request -  socket TCP connecté au client
        self.data = self.request.recv(1024).strip()
        print("{} sent:".format(self.client_address[0]))
        print(self.data)
        # Envoie un accusé de réception lors de  l'arrivée des données.
        self.request.sendall("ACK from TCP Server".encode())

if __name__ == "__main__":
    HOST, PORT = "localhost", 3000

    # Initialise l'objet TCPServer et effectue la liaison avec le port 3000
    tcp_server = socketserver.TCPServer((HOST, PORT), TCPHandler)
    print("Le serveur écoute sur le port {}".format(PORT))
    # Active le serveur TCP.
    # Pour interrompre le serveur appuyez sur Ctrl-C.
    tcp_server.serve_forever()
```

Le fichier tcpHandlerClient.py[modifier]

```
import socket

host_ip, server_port = "127.0.0.1", 3000
data = " Hello wild world... best regards ..!\n"

# Initialise le client TCP avec SOCK_STREAM
tcp_client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

try:
    # Etablit la connexion avec le serveur TCP
    tcp_client.connect((host_ip, server_port))
    # Envoie des données au serveur
```

```
    tcp_client.sendall(data.encode())

    # Lit les données du serveur et ferme la connexion.
    received = tcp_client.recv(1024)
finally:
    tcp_client.close()

print ("Bytes Sent:     {}".format(data))
print ("Bytes Received: {}".format(received.decode()))
```

Combinaison des sockets et des threads[modifier]

Les deux programmes suivants illustrent le fonctionnement d'un serveur multithread.

- Il met en oeuvre un **pool de thread** pour gérer les sockets (un socket pour chaque client)
- Il utilise une **classe** qui implémente le code du thread serveur.

Le serveur multithread[modifier]

```
import socket
from threading import Thread

''' Serveur Python multithread:
    Créer un pool de thread pour gérer des socket serveur.
'''
class ClientThread(Thread):

    def __init__(self,ip,port):
        Thread.__init__(self)
        self.ip = ip
        self.port = port
        print ("\n[+] Nouveau client " + ip + ":" + str(port) )

    def run(self):
        while True :
            data = conn.recv(2048)
            print ("Serveur à reçu :{"+ data.decode('utf-8')+"}")
            MESSAGE = input("Serveur : Entrez une reponse depuis le serveur (
            if MESSAGE == 'exit':
                print("\nFin du thread "+ ip + ":" + str(port))
                break
            print ("Le serveur envoie {0} au client {1}:{2}".format(MESSAGE,s
            conn.send(bytes(MESSAGE,'UTF-8'))  # echo


HOST = '0.0.0.0'
PORT = 3000
BUFFER_SIZE = 20

tcpServer = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcpServer.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
```

```
tcpServer.bind((HOST, PORT))
threads = []

while True:
    tcpServer.listen(4)
    print ("Serveur multithread : En attente de connexion..." )
    (conn, (ip,port)) = tcpServer.accept()
    newthread = ClientThread(ip,port)
    newthread.start()
    threads.append(newthread)

for t in threads:
    t.join()
```

Le client multithread[modifier]

```
# Python TCP Client A
import socket

host = socket.gethostname()
port = 3000
BUFFER_SIZE = 2000
MESSAGE = input("tcpClient: Entrez un message (exit=fin):")

tcpClient = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
tcpClient.connect((host, port))

while MESSAGE != 'exit':
    tcpClient.send(bytes(MESSAGE,'UTF-8'))
    data = tcpClient.recv(BUFFER_SIZE)
    print (" Client a reçu:"+data.decode('utf-8'))
    MESSAGE = input("tcpClient: Entrez un message pour continuer (exit=fin):"
print (" Fin du programme client.")
tcpClient.close()
```