Université Côte d'Azur
Faculté des Sciences - Département d'Informatique

Licence d'Informatique L2
Introduction aux Systèmes et Réseaux

TD n°7 : Applications client/serveur

L'objectif de ce TD est d'explorer l'utilisation combinée des sockets et de la primitive `select`.

1 Un premier serveur

Expliquez ce que fait le programme suivant et comment on pourrait le tester à partir de la ligne de commande du shell Unix :

```
import socket

HOST = '127.0.0.1'  # or 'localhost' or '' - Standard loopback interface address
PORT = 2000         # Port to listen on (non-privileged ports are > 1023)

MAXBYTES = 4096

# create socket
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s: # AF_INET: IPv4
                                                             # SOCK_STREAM: TCP
    s.bind((HOST, PORT)) # bind this socket to specific port on host
    s.listen() # make the socket a listening one

    conn, (client_addr, client_port) = s.accept() # blocking. returns if a client connects.
    with conn:
        print('Connected by application: %d on machine: %s' % (client_port, client_addr))
        data = conn.recv(MAXBYTES)
        while len(data) > 0: # otherwise means a disconnection from the client side.
            conn.sendall(data)
            data = conn.recv(MAXBYTES)
```

Comment faire en sorte qu'une fois un client déconnecté, le serveur ne se termine pas et puisse attendre d'autres clients ?

Quel défauts comporte cette approche ?

2 Client réseau

Expliquez ce que fait le code suivant :

```
import os, socket, sys

MAXBYTES = 4096

if len(sys.argv) != 3:
    print('Usage:', sys.argv[0], 'hote pdrt')
```

```
    sys.exit(1)

HOST = sys.argv[1]
PORT = int(sys.argv[2])

af, socktype, proto, canonname, sockaddr = socket.getaddrinfo(HOST, PORT)[0]
# get protocol informations of server
print("family:", af) # IPv4 or IPv6
print("socket type:", socktype) # TCP or UDP
s = socket.socket(af, socktype) # create same type of socket as server
s.connect(sockaddr)
print('connected to:', sockaddr)

while True: # Client synchrone !! On alterne lecture et ecriture.
# Le serveur doit donc lui aussi alterner
    line = os.read(0, MAXBYTES)
    if len(line) == 0:
        s.shutdown(socket.SHUT_WR)
        break
    s.send(line)

    data = s.recv(MAXBYTES)
    if len(data) == 0:
        break
    os.write(1, data)
s.close()
```

3 La solution `select`

Expliquez ce que fait le programme suivant :

```
import select, socket

HOST = '127.0.0.1'  # or 'localhost' or '' - Standard loopback interface address
PORT = 2003         # Port to listen on (non-privileged ports are > 1023)

MAXBYTES = 4096

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind((HOST, PORT))
serversocket.listen()
socket_list = [serversocket]

while len(socket_list) > 0:
    readable, _, _ = select.select(socket_list, [], [])
    for s in readable:
        print(s)
        if s == serversocket: # serversocket receives a connection
            connection, (client_addr, client_port) = s.accept()
            socket_list.append(connection)
        else: # data is sent from given client
```

```
            data = s.recv(MAXBYTES)
            if len(data) > 0:
                s.sendall(data)
            else: # client has disconnected
                s.close()
                socket_list.remove(s)
serversocket.close()
```

Comment le modifier pour pouvoir terminer le serveur à la frappe de la touche 'Entrée' ?

4 Un deuxième serveur

Expliquez ce que fait ce programme.

```
import os, select, socket, sys

HOST = '127.0.0.1'  # or 'localhost' or '' - Standard loopback interface address
PORT = 2005         # Port to listen on (non-privileged ports are > 1023)

MAXBYTES = 4096

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((HOST, PORT))
server_socket.listen()
nb_open = 0
# Create list of potential active_sockets and place server_socket socket in
# first position
socket_list = [server_socket]
first = True
while first or nb_open > 0:
    first = False
    active_sockets, _, _ = select.select(socket_list, [], [], 60)
    for s in active_sockets:
        if s == server_socket:
            conn, (addr, port) = server_socket.accept()
            socket_list.append(conn)
            print("Incoming connexion from %s on port %d..." % (addr, port))
            nb_open += 1
        else:
            msg = s.recv(MAXBYTES)
            if len(msg) == 0:
                print("NULL message. Closing connection...")
                s.close()
                # Remove the closed connection from potential active_sockets
                socket_list.remove(s)
                nb_open -= 1
            else:
                os.write(1, msg)
server_socket.close()
print("Last connection closed. Bye!")
sys.exit(0)
```

Une fois que vous avez bien compris son fonctionnement, modifiez-le de façon à le transformer en serveur `rsh`, c'est-à-dire pour permettre l'exécution d'une commande distante et en récupérer le résultat. Il sera donc nécessaire de créer un fils qui se recouvrira. Avant cela il faudra bien entendu rediriger ses deux sorties vers le descripteur de fichier de la socket. Ce dernier est accessible via un appel à `ma_socket.fileno()`.

5 Fonctionnement d'un serveur web

Dans les exemples précédents, lorsque le processus est occupé à lire un socket de transfert, il n'est pas disponible pour accepter une connection, et vice-versa. Dans le cas d'un serveur web, le nombre de connections simultanées peut être très élevé. Cela se traduit par une lenteur désagréable, voire inacceptable du serveur. Pour parer à ce problème, une idée classique consiste à construire un serveur multi-processus : au lieu de faire faire tout le travail à un seul processus, on fait appel à plusieurs processus. Par exemple, on peut faire en sorte que seul le processus père accepte les connections, puis qu'à chaque connexion, il crée une socket de communication client qu'il passe à un nouveau processus fils qui traitera spécifiquement cette connection...

Une version plus élaborée consisterait à créer au départ un *pool* de processus de même taille que le nombre de coeurs de votre ordinateur, et à faire en sorte que chaque fils gère un nombre à peu près équilibré de connexions. Cependant, il est impossible de transférer le contrôle de sockets entre processus (lourds). La méthode actuelle est de passer par la création d'un *pool* de processus légers (*threads*) qui ont pour principale caractéristique de pouvoir communiquer par mémoire partagée. Cependant, il est nécessaire de connaître la programmation orientée-objets pour manipuler les threads en python, vous verrez donc ça plus tard...