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

Licence d'Informatique L2
Introduction aux Systèmes et Réseaux

TD n ° 7 (Corrigé) : 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 serversocket: # AF_INET: IPv4
                                                                        # SOCK_STREAM: TCP
    serversocket.bind((HOST, PORT)) # bind this socket to specific port on host
    serversocket.listen() # make the socket a listening one

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

Il est possible de le tester grâce à la commande : `telnet localhost port`. Ce programme crée un server communiquant en TCP, et attend la connexion d'un client. Une fois celui-ci connecté, le server attend de recevoir des données. Celles-ci sont systématiquement récrites dans le socket, à destination du client. Si ce dernier se déconnecte, alors le serveur se termine.

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

Nous pouvons rajouter une boucle while, cf. code ci-dessous :

```
import socket

HOST = '127.0.0.1'  # or 'localhost' or '' - Standard loopback interface address
PORT = 2002         # Port to listen on (non-privileged ports are > 1023)
```

```
MAXBYTES = 4096

# create socket
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as serversocket: # AF_INET: IPv4
                                                                        # SOCK_STREAM: TCP
    serversocket.bind((HOST, PORT)) # bind this socket to specific port on host
    serversocket.listen() # make the socket a listening one
    while True:
        clientsocket, (addr, port) = serversocket.accept() # blocking. returns if a client clients
        with clientsocket:
            print('connected by application: %d on machine: %s' % (port, addr))
            data = clientsocket.recv(MAXBYTES)
            while len(data) > 0: # otherwise means a disclientsocketection from the client side.
                clientsocket.sendall(data)
                data = clientsocket.recv(MAXBYTES)
```

Quel défauts comporte cette approche ?

D'une part le serveur gère les connexions client de façon séquentielle et non en parallèle. En outre, la seule façon d'interrompre le serveur est via un signal externe, type CTRL-C, ce qui n'est pas très propre. Résoudre le premier point est difficile car `accept()` *et* `recv()` *sont bloquants. Le serveur ne peut pas en être même temps à l'écoute de nouvelles connections entrantes et à l'écoute de messages sur l'un des sockets client. Le second problème est du même acabit que le premier. L'idée serait d'écouter le clavier, et de pouvoir terminer le serveur si on lit la frappe d'une touche donnée. Mais il faudrait que notre processus puisse être à la fois en train d'écouter le clavier, le socket client, et les connexions entrantes.*

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 port')
    sys.exit(1)

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

sockaddr = (HOST, PORT)

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # IPv4, TCP
s.connect(sockaddr)
print('connected to:', sockaddr)
while True: # Client synchrone !! On alterne écriture vers serveur
            # et lecture depuis serveur. 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) # attention, si le serveur n'envoie rien on est bloqué.
    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()
socketlist = [serversocket]

while len(socketlist) > 0:
    readable, _, _ = select.select(socketlist, [], [])
    for s in readable:
        if s == serversocket: # serversocket receives a connection
            clientsocket, (addr, port) = s.accept()
            print("connection from:", addr, port)
            socketlist.append(clientsocket)
        else: # data is sent from given client
            data = s.recv(MAXBYTES)
            if len(data) > 0:
                s.sendall(data)
            else: # client has disconnected
                s.close()
                socketlist.remove(s)
serversocket.close()
```

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

Nous pouvons rajouter `sys.stdin` dans la liste des connexions à écouter : notre application sera donc également à l'écoute de l'entrée standard. Notons ici que du point de vue applicatif, un socket est vu comme un fichier. L'entrée standard également. Un `readline()` sur l'entrée standard permet de lire ce qui a été entré au clavier, auquel cas on sortira de la boucle. Une fois sorti, il est nécessaire de fermer les connexions clients et serveur.

```
import select, socket, sys

HOST = '127.0.0.1'  # or 'localhost' or '' - Standard loopback interface address
```

```
PORT = 2004         # 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()
socketlist = [serversocket, sys.stdin]
running = True
while running:
    readable, _, _ = select.select(socketlist, [], [])
    for s in readable:
        if s == serversocket: # serversocket receives a connection
            clientsocket, (addr, port) = s.accept()
            print("connection from:", addr, port)
            socketlist.append(clientsocket)
        elif s == sys.stdin: # data is sent from keyboard
            _ = s.readline()
            running = False
        else: # data is sent from given client
            data = s.recv(MAXBYTES)
            if len(data) > 0:
                s.sendall(data)
            else: # client has disconnected
                s.close()
                socketlist.remove(s)
for conn in socketlist:
    conn.close()
serversocket.close()
```

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

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # IPv4, TCP
serversocket.bind((HOST, PORT))
serversocket.listen()
print("server listening on port:", PORT)
nb_open = 0
# Create list of potential active sockets and place server socket in
# first position
socketlist = [serversocket]
first = True
while first or nb_open > 0:
```

```
    first = False
    activesockets, _, _ = select.select(socketlist, [], [])
    for s in activesockets:
        if s == serversocket:
            clientsocket, (addr, port) = serversocket.accept()
            socketlist.append(clientsocket)
            print("Incoming connection 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
                socketlist.remove(s)
                nb_open -= 1
            else:
                os.write(1, msg)
serversocket.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()`.

```
import os, select, socket, sys

HOST = '127.0.0.1' # or 'localhost' or '' - Standard loopback interface address
PORT = 2006        # Port to listen on (non-privileged ports are > 1023)

MAXBYTES = 4096

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM) # IPv4, TCP
serversocket.bind((HOST, PORT))
serversocket.listen()
print("server listening on port:", PORT)
nb_open = 0
# Create list of potential active sockets and place server socket in
# first position
socketlist = [serversocket]
first = True
while first or nb_open > 0:
    first = False
    activesockets, _, _ = select.select(socketlist, [], [])
    for s in activesockets:
```

```
        if s == serversocket:
            clientsocket, (addr, port) = serversocket.accept()
            socketlist.append(clientsocket)
            print("Incoming connection 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
                socketlist.remove(s)
                nb_open -= 1
            else:
                argv = msg.split()
                if len(argv) >= 1:
                    if os.fork() == 0:
                        os.dup2(s.fileno(), 1)
                        os.dup2(s.fileno(), 2)
                        os.close(s.fileno())
                        try:
                            os.execvp(argv[0], argv)
                        except OSError:
                            os.write(2, argv[0] + b': no such command\n')
                            sys.exit(1)
                        os.wait()
                else:
                    s.send(b': no such command\n') # the server must always send smth to th
serversocket.close()
print("Last connection closed. Bye!")
sys.exit(0)
```

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...