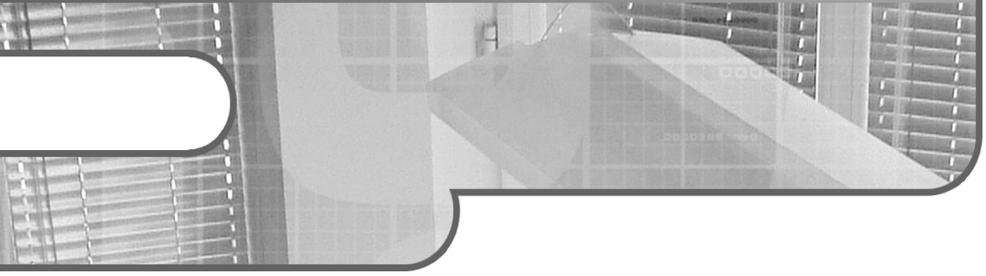

Chapitre 2.2
Programmation réseau

1. Écrire un serveur et un client

1.1 Utilisation d'une socket TCP

TCP est l'acronyme de *Transmission Control Protocol*, soit protocole de contrôle de transmission et a été développé en 1973 et documenté au sein de la RFC 793. Il est situé dans le modèle OSI au sein de la couche transport et est répandu de par sa fiabilité.

Il se caractérise par la manière de mettre en place la synchronisation entre client et serveur et par le découpage en segments des octets à transmettre, chaque segment étant parfaitement identifiable et disposant d'un système de contrôle d'intégrité qui fait que celui qui reçoit un paquet peut savoir que le paquet est corrompu et le redemander.

Le flux TCP utilise les sockets et c'est le module éponyme de Python qui va nous permettre de travailler avec TCP.

L'idée ici est de réaliser un mini serveur de données clé-valeur très basique et le plus simple possible de manière à voir comment créer un serveur puis un client, mais aussi comment manipuler les données qui sont échangées de l'un à l'autre.

Ce qu'il faut absolument comprendre est que les données qui sont transmises d'un serveur à un client ou d'un client à un serveur sont des octets, et rien d'autre. En Python 2, cela pouvait prêter à confusion, puisque le type **`str`** de Python 2 était assimilable à des octets. Le type Python 3 **`str`** représente une chaîne de caractères en unicode et c'est le type **`bytes`** qui permet de gérer des octets.

Il faut donc particulièrement être attentif à ce point, puisque la plupart des exemples présents sur le net, par ailleurs d'excellentes qualités, sont écrits pour Python 2 et génèrent une confusion sur le type de données réellement envoyées.

On va créer un serveur qui s'attend à recevoir un nombre et qui renvoie un code en fonction du fait que ce nombre soit premier ou non.

Voici le code ce cette fonction :

```
def isprime(n):
    '''check if integer n is a prime'''

    # negative numbers, 0 and 1 are not primes
    if n < 2:
        return False

    # 2 is the only even prime number
    if n == 2:
        return True

    # all other even numbers are not primes
    if not n & 1:
        return False

    # range starts with 3 and only needs to go up the squareroot
    # of n for all odd numbers
    for x in range(3, int(n**0.5)+1, 2):
        if n % x == 0:
            return False
    return True
```

Voici donc le code du serveur (disponible en téléchargement) :

```
#!/usr/bin/python3

import socket

params = ('127.0.0.1', 8809)
BUFFER_SIZE = 1024 # default

donnees = {}

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)#Internet, TCP
s.bind(params)
s.listen(1)
```

Le serveur est lancé et il écoute sur le port 8809. la méthode accept attend qu'une connexion se présente et dès qu'elle arrive, elle renvoie les éléments nécessaires pour que la connexion puisse être traitée. Ce qui est réalisé dans une boucle sans fin. Le principe est que tant que l'échange client serveur dure, il est traité.

```
conn, addr = s.accept()
print('Connexion acceptée: %s' % str(addr))

while True:
        data = conn.recv(BUFFER_SIZE)
        if not data:
            break
```

```
        print('Donnée reçue: %s' % data)
        try : number = int(data)
        except:
            response = b'E'
            phrase = "'%s' n'est pas un entier" % data
        else:
            if isprime(number):
                phrase = "'%s' est un nombre premier" % number
                response = b'T'
            else:
                phrase = "'%s' n'est pas est un nombre premier" % number
                response = b'F'
        finally:
            print(response)
            conn.send(response)
conn.close()
```

La partie complexe n'est pas tant la gestion de la connexion et des transferts réseau que le traitement des données. Ce qui compte est de savoir quoi faire des données et comment mettre en place un dialogue entre client et serveur, sachant que les seuls éléments échangeables sont des octets.

Dans cet exemple, on a choisi la convention suivante : si on ne comprend pas la donnée transmise par le client, on renvoie un code E (pour erreur). Si le nombre est premier, on renvoie un code T (pour True), sinon on renvoie un code F (pour False).

Les choix effectués dans cet exemple sont assez basiques et limités : on cherche à transmettre le moins de données possible, mais serveur et client doivent s'entendre pour se comprendre entre eux : le client devra connaître l'ensemble des codes susceptibles d'être envoyés par le serveur et les gérer.

Cela montre les problématiques typiques qui se posent lorsque l'on cherche à manipuler des données en utilisant des couches de bas niveau.

Voici le code du client, qu'il faut lire en faisant le lien avec celui du serveur :

```
#!/usr/bin/python3

import socket

params = ('127.0.0.1', 8809)
BUFFER_SIZE = 1024 # default

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(params)
```

Le parti a été pris d'envoyer une succession de messages de manière à couvrir tous les cas possibles de traitement côté serveur.

```
chiffres = [4j, 4, 5, -5, 17, 29, 2**50, 2**50-1]
```

En face de chaque message, qui est en réalité une sorte d'instruction donnée au serveur, est mis en commentaire le comportement attendu.
Voici le code qui effectue les requêtes vers le serveur et qui traite les réponses :

```
for chiffre in chiffres:
    s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect(params)
    print("Envoi d'un message %s" % chiffre)
    s.send(('%s\n' % chiffre).encode('utf8'))
    data = s.recv(BUFFER_SIZE)
    if len(data) == 0:
        print("\tPas de réponses")
    elif data == b'E\n':
        print("\tUne erreur est détectée par le serveur")
    elif data == b'T\n':
        print("\tLe nombre est bien un nombre premier")
    elif data == b'F\n':
        print("\tLe nombre n'est pas un nombre premier")
    else:
        print("\tLa donnée reçue n'est pas comprise : %s" % data)
    s.close()
```

TODO : tant que l'on est dans la boucle, on communique avec le serveur et donc le serveur reste dans la boucle infinie de communication. Le fait de fermer la connexion côté client envoie une trame vide qui va permettre de passer dans le break côté serveur (**`if not data: break`**) et ainsi libérer le serveur.
S'il y avait une boucle autour de la méthode **`accept`**, le serveur continuerait à tourner en attendant de nouvelles données qu'une nouvelle connexion lui parvienne et tournerait ainsi à l'infini, mais en l'occurrence, il n'y a pas cette boucle et il se stoppe donc.
Cet exemple ne permet donc que de parler à un seul client, puis d'arrêter le serveur. Il existe, dans la pratique, des cas d'utilisation tels que celui-ci.
Il existe, par exemple, un excellent outil nommé woof qui s'installe via le gestionnaire de paquet et se lance en ligne de commande :

```
$ sudo aptitude install woof
```

```
$ python path/to/woof.py path/to/filename
```

Un serveur est lancé et propose une URL qu'il suffit de donner à quelqu'un par IRC ou un autre moyen pour que cette personne, en cliquant dessus, télécharge le fichier. À la fin du téléchargement, le serveur se ferme. On peut donc transmettre un document par le réseau rapidement.

1.2 Utilisation d'une socket UDP

Comme TCP, UDP appartient à la couche transport du modèle OSI, mais contrairement à lui, il ne réalise pas de connexion préliminaire, de synchronisation et ne garantit pas la bonne réception des données. Par contre, il intègre un système qui assure l'intégrité de la donnée transmise, comme TCP.

UDP est moins fiable que TCP, car il peut perdre des paquets, mais il est beaucoup plus rapide car il nécessite beaucoup moins d'allers-retours. Il est donc préférable pour des technologies pour lesquelles la fiabilité n'est pas le principal besoin par rapport à la rapidité, comme la voix sur IP, le streaming ou les jeux en réseau.

Ceci est vrai dans la mesure où les applications concernées sont capables soit de se passer d'une petite partie des données qui auraient été perdues, soit de les reconstruire (à partir du moment où cela est plus rapide que les redemander), soit de les substituer.

D'autre part, TCP est utilisé pour établir un dialogue entre un client et un serveur alors qu'UDP est utilisé pour envoyer des données du client vers le serveur sans que le client attende un retour.

Là encore, on va utiliser les sockets, mais comme les protocoles TCP et UDP diffèrent dans leur fonctionnement, cette différence est visible dans la manière d'écrire un serveur et un client.

Ainsi, le serveur n'attend pas de synchroniser une connexion avec un client et le client ne se connecte pas au serveur avant de lui envoyer des données.

En dehors de cela, la manière de traiter les données est identique à ce que l'on a vu pour TCP. Des exemples plus simples que ceux montrés dans cet ouvrage permettent de mettre en évidence les différences entre les protocoles TCP (http://wiki.python.org/moin/TcpCommunication) et UDP (http://wiki.python.org/moin/UdpCommunication). Dans les deux cas, les serveurs affichent la donnée reçue et pour TCP, la donnée est également renvoyée au client.

L'exemple suivant est dans la lignée de celui pour TCP, le client donnant des informations au serveur. Les lignes inutiles sont laissées en commentaire afin de mettre en évidence les différences avec TCP.

```
#!/usr/bin/python3

import socket

params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default

donnees = {}

s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #SOCK_STREAM
s.bind(params)
# Les lignes suivantes n'ont pas de sens en UDP:
#s.listen(1)
#conn, addr = s.accept()
#print('Connexion acceptée: %s' % str(addr))
```

On voit clairement dans la première partie qu'il n'est pas nécessaire de se mettre à l'écoute et d'initier une connexion.
Dans la partie qui suit, la différence de sémantique entre **recv** et **recvfrom** est révélatrice du fait que la méthode **recvfrom** reçoit à la fois la donnée et l'adresse du client.

```
while True:
        #data = conn.recv(BUFFER_SIZE)
        data, addr = s.recvfrom(BUFFER_SIZE)
        print('Connexion reçue: %s' % str(addr))
        print('Donnée reçue: %s' % data)
        if not data:
            break
        print('Donnée reçue: %s' % data)
        try:
            number = int(data)
        except:
            response = b'E'
            phrase = "'%s' n'est pas un entier" % data
        else:
            if isprime(number):
                phrase = "'%s' est un nombre premier" % number
                response = b'T'
            else:
                phrase = "'%s' n'est pas est un nombre premier" % number
                response = b'F'
        finally:
            print(response)
#conn.close()
```

À noter que si pour TCP il faut une boucle pour gérer chaque connexion et une autre pour gérer les échanges au sein de la connexion, en UDP une seule boucle suffit. Pour que l'exemple puisse se terminer proprement, il a été rajouté un cas d'utilisation particulier permettant de sortir de la boucle.
Voici la partie cliente correspondant :

```
#!/usr/bin/python3

import socket
params = ('127.0.0.1', 8808)
BUFFER_SIZE = 1024 # default

s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#s.connect(params)
```

Là encore, pas besoin de connexion. C'est au moment où l'on envoie les données que l'on donne les informations sur le serveur à atteindre.

```
chiffres = [4j, 4, 5, -5, 17, 29, 2**50, 2**50-1]
```