Programmation Python
(adoptez un serpent !)

ADG

<dahirou.gueye@uadb.edu.sn>

Université Alioune Diop de Bambey

Master SI/SR

Octobre 2022

« Serpent scientifique et littéraire »

Chapitre 12 :

Programmation Réseau avec les Sockets

Le réseau

- Dans ce chapitre, nous allons donc apprendre à faire communiquer deux applications grâce aux sockets, des objets qui permettent de connecter un client à un serveur et de transmettre des données de l'un à l'autre.
- On va s'attacher ici à comprendre comment faire communiquer deux applications, qui peuvent être sur la même machine mais aussi sur des machines distantes
- Dans ce cas, elles se connectent grâce au réseau local ou à Internet.
- Il existe plusieurs protocoles de communication en réseau.
  - TCP
  - UDP

Clients et serveur

- Dans l'architecture que nous allons voir dans ce chapitre, on trouve en général un serveur et plusieurs clients. Le serveur, c'est une machine qui va traiter les requêtes du client.
- Si vous accédez par exemple à uadb.edu.sn, c'est parce que votre navigateur, faisant office de client, se connecte au serveur de uadb.edu.sn.
- Il lui envoie un message en lui demandant la page que vous souhaitez afficher et le serveur de uadb.edu.sn envoie la page demandée au client.
- Dans les exemples que nous allons voir, nous allons créer deux applications : l'application serveur et l'application client. Le serveur *écoute* donc en attendant des connexions et les clients se connectent au serveur.

Les différentes étapes

- **Le serveur :**

  1. attend une connexion de la part du client ;
  2. accepte la connexion quand le client se connecte ;
  3. échange des informations avec le client ;
  4. ferme la connexion.

- **Le client**

  1. se connecte au serveur ;
  2. échange des informations avec le serveur ;
  3. ferme la connexion.

Etablir une connexion

- **Un serveur peut cependant héberger plusieurs services. Par exemple un serveur peut héberger un serveur web mais également un serveur de messagerie, etc.**
- **Alors comment se connecter au bon service?**
  - En utilisant les ports . Les ports les plus connus sont 21 pour le FTP, 80 pour le HTTP, 443 pour le HTTPS, le 22 pour le SSH, 110 pour le service POP, etc.

Etablir une connexion

- **Pour que le client se connecte au serveur, il nous faut deux informations**
  - Le **nom d'hôte** (*host name* en anglais), qui identifie une machine sur Internet ou sur un réseau local. Les noms d'hôtes permettent de représenter des adresses IP de façon plus claire (on a un nom comme google.fr, plus facile à retenir que l'adresse IP correspondante 74.125.224.84).
  - Un **numéro de port**, qui est souvent propre au type d'information que l'on va échanger. Si on demande une connexion web, le navigateur va en général interroger le port 80 si c'est en http ou le port 443 si c'est en connexion sécurisée (https). Le numéro de port est compris entre 0 et 65535 (il y en a donc un certain nombre !) et les numéros entre 0 et 1023 sont réservés par le système. On peut les utiliser, mais c'est pas recommandé.

Les Sockets

- On commence par importer les modules socket et sys

```
1. import socket, sys
```

- Nous allons d'abord créer notre serveur puis, en parallèle, un client. Nous allons faire communiquer les deux.
- D'abord pour le **serveur**.

Construire notre socket

- **Nous allons pour cela faire appel au constructeur socket. Dans le cas d'une connexion TCP, il prend les deux paramètres suivants, dans l'ordre:**

  - socket.AF_INET : la famille d'adresses, ici ce sont des adresses Internet ;
  - socket.SOCK_STREAM : le type du socket, SOCK_STREAM pour le protocole TCP.

  ```
  >>> connexion_principale = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
  ```

Liaison du socket à une adresse précise à l'aide de : bind

```
try:
    connexion_principale.bind((HOST, PORT))
except socket.error:
    print("La liaison du socket à l'adresse choisie a échoué.")
    sys.exit
```

Faire écouter notre socket

- Notre socket est prêt à écouter sur un port mais il n'écoute pas encore.
- On va avant tout lui préciser le nombre maximum de connexions qu'il peut recevoir sur ce port sans les accepter.
- On utilise pour cela la méthode **listen**.
- On lui passe généralement **5** en paramètre.
- Est ce que notre serveur ne pourra dialoguer qu'avec 5 clients maximum ?
- Cela veut dire que si 5 clients se connectent et que le serveur n'accepte aucune de ces connexions, aucun autre client ne pourra se connecter. Mais généralement, très peu de temps après que le client ait demandé la connexion, le serveur l'accepte.
- Attente de la requête de connexion d'un client :

```
1.  while 1:
2.      print("Serveur prêt, en attente de requêtes ...")
3.      connexion_principale.listen(5)
```

Accepter une connexion venant du client

- Aucune connexion ne s'est encore présentée mais la méthode **accept** que nous allons utiliser va bloquer le programme tant qu'aucun client ne s'est connecté.
- Il est important de noter que la méthode **accept** renvoie deux informations :
  - le socket connecté qui vient de se créer, celui qui va nous permettre de dialoguer avec notre client tout juste connecté ;
  - un tuple représentant l'adresse IP et le port de connexion du client.

```
>>> connexion_avec_client, infos_connexion = connexion_principale.accept()
```

- Cette méthode bloque le programme. Elle attend qu'un client se connecte.
- Laissons cette fenêtre Python ouverte et, à présent, ouvrons-en une nouvelle pour construire notre client.

Création du client

- Construisons notre socket de la même façon

1. >>> import socket
2. >>> connexion_avec_serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

Connecter le client

- Pour se connecter à un serveur, on va utiliser la méthode **connect**. Elle prend en paramètre un tuple, comme bind, contenant le nom d'hôte et le numéro du port identifiant le serveur auquel on veut se connecter.

- Le numéro du port sur lequel on veut se connecter, vous le connaissez : c'est 11700. Vu que nos deux applications Python sont sur la même machine, le nom d'hôte va être localhost (c'est-à-dire la machine locale).

```
>>> connexion_avec_serveur.connect(('localhost', 11700))
```

- Notre serveur et notre client sont connectés !

- Si vous retournez dans la console Python abritant le serveur, vous pouvez constater que la méthode accept ne bloque plus, puisqu'elle vient d'accepter la connexion demandée par le client.

Connecter le client

- **Vous pouvez donc de nouveau saisir du code côté serveur :**

```
>>> print(infos_connexion)
('127.0.0.1', 2901)
>>>
```

- La première information, c'est l'adresse IP du client. Ici, elle vaut 127.0.0.1 c'est-à-dire l'IP de l'ordinateur local
- Le second est le port de sortie du client, qui ne nous intéresse pas ici.

Communication des sockets

- En utilisant les méthodes **send** pour envoyer et **recv** pour recevoir
- Les informations que vous transmettrez seront des chaînes de bytes, pas des str !
- côté serveur :

```
1. >>> connexion_avec_client.send(b"Je viens d'accepter la connexion")
2. 32
3. >>>
```

  - La méthode send vous renvoie le nombre de caractères envoyés.

Communication des sockets

côté client:

- on va réceptionner le message que l'on vient d'envoyer.
- La méthode recv prend en paramètre le nombre de caractères à lire.
- Généralement, on lui passe la valeur 1024. Si le message est plus grand que 1024 caractères, on récupérera le reste après.
- Dans la fenêtre Python côté client, donc :

```
1. >>> msg_recu = connexion_avec_serveur.recv(1024)
2. >>> msg_recu
3. b"Je viens d'accepter la connexion"
4. >>>
```

Communication des sockets

- Ce petit mécanisme peut servir à faire communiquer des applications entre elles non seulement sur la machine locale, mais aussi sur des machines distantes et reliées par Internet.
- Le client peut également envoyer des informations au serveur et le serveur peut les réceptionner, tout cela grâce aux méthodes send et recv que nous venons de voir.

Fermeture de la connexion

- **Côté serveur :**

1. >>> connexion_avec_client.close()
2. >>>

- **Côté client :**

1. >>> connexion_avec_serveur.close()
2. >>>

Le serveur

- Ci-dessous le code du serveur. Il n'accepte qu'un seul client (nous verrons plus bas comment en accepter plusieurs) et il tourne jusqu'à recevoir du client le message ‘FIN’ ou ‘espace’
- Le serveur et le client pourront s'échanger des messages à tour de rôle

Le serveur

```
1. import socket,sys
2. hote = ''
3. port = 12800
4. counter=0
5. connexion_principale = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
6. try:
7.         connexion_principale.bind((hote, port))
8. except socket.error:
9.       print("La liaison du socket à l'adresse choisie a échoué.")
10.      sys.exit
11. connexion_principale.listen(5)
12. print("Le serveur écoute à présent sur le port {}".format(port))
```

Le serveur

```
13. # Etablissement de la connexion :
14. connexion, adresse = connexion_principale.accept()
15. counter +=1
16. print("Client connecté, adresse IP %s, port %s" % (adresse[0], adresse[1]))
```

Le serveur (dialogue avec le client)

```
17. # Dialogue avec le client
18. msgServeur ="Vous êtes connectés au serveur uadb. SVP; envoyez vos
    messages."
19. connexion.send(msgServeur.encode("Utf8"))
20. msgClient = connexion.recv(1024).decode("Utf8")
21. while 1:
22.         print("C>", msgClient)
23.         if msgClient.upper() == "FIN" or msgClient =="":
24.             break
25.         msgServeur = input("S> ")
26.         connexion.send(msgServeur.encode("Utf8"))
27.         msgClient = connexion.recv(1024).decode("Utf8")
```

Le serveur (dialogue avec le client)

```
28. # Fermeture de la connexion :
29. connexion.send("fin".encode("Utf8"))
30. print("Connexion interrompue.")
31. Connexion.close()
32. ch = input("<R>ecommencer <T>erminer ? ")
33. if ch.upper() =='T':
34.             break
```

Le client

- Il va tenter de se connecter sur le port 12800 de la machine locale.
- Il demande à l'utilisateur de saisir quelque chose au clavier et envoie ce quelque chose au serveur, puis attend sa réponse.

```
1.  import socket,sys
2.  hote  =  "
3.  port = 12800
4.  connexion_avec_serveur = socket.socket(socket.AF_INET,
    socket.SOCK_STREAM)
5.  try:
6.          connexion_avec_serveur.connect((hote, port))
7.  except socket.error:
8.          print("La connexion a échoué.")
9.          sys.exit()
10. print("Connexion établie avec le serveur.")
```

Le client (dialogue avec le serveur)

- Dialogue avec le serveur

```
1.  msgServeur = connexion_avec_serveur.recv(1024).decode("Utf8")
2.  while 1:
3.      if msgServeur.upper() == "FIN" or msgServeur =="":
4.          break
5.      print("S>", msgServeur)
6.      msgClient = input("C> ")
7.      connexion_avec_serveur.send(msgClient.encode("Utf8"))
8.      msgServeur = connexion_avec_serveur.recv(1024).decode("Utf8")
9.  # Fermeture de la connexion :
10. print("Connexion interrompue.")
11. connexion_avec_serveur.close()
```

Les méthodes encode et décode

- **encode** est une méthode de str. Elle peut prendre en paramètre un nom d'encodage et permet de passer un str en chaîne **bytes**. C'est, comme vous le savez, ce type de chaîne que send accepte. En fait, encodeencode la chaîne str en fonction d'un encodage précis (par défaut, **Utf-8)**.
- **decode**, à l'inverse, est une méthode de **bytes**. Elle aussi peut prendre en paramètre un encodage et elle renvoie une chaîne str décodée grâce à l'encodage (par défaut **Utf-8**).
- Si l'encodage de votre console est différent d'**Utf-8** (ce sera souvent le cas sur Windows), des erreurs peuvent se produire si les messages que vous encodez ou décodez comportent des accents.

Socket et threading

- Le système précédent met en relation que deux machines
- Les deux interlocuteurs ne peuvent en effet envoyer des messages que chacun à leur tour: Par exemple, lorsque l'un d'eux vient d'émettre un message, son système reste bloqué tant que son partenaire ne lui a pas envoyé une réponse
- Problématique: Boucle while s'interrompt à deux niveaux:
  - attendre les entrées clavier de l'utilisateur:
    (fonction input())
  - attendre l'arrivée d'un message réseau:
    msgServeur = mySocket.recv(1024).decode("Utf8")
- Si pour un socket maison cela peut être acceptable, pour des projets ambitieux il est plutôt conseillé de passer par des threading.
- Nous allons maintenant mettre en pratique la technique des threads pour construire un système de chat simplifié: Ce système sera constitué d'un seul serveur et d'un nombre quelconque de clients.

Socket et threading

- Chaque client enverra tous ses messages au serveur, mais celui-ci les réexpédiera immédiatement à tous les autres clients connectés, de telle sorte que chacun puisse voir l'ensemble du trafic.
- Chacun pourra à tout moment envoyer ses messages, et recevoir ceux des autres, dans n'importe quel ordre, la réception et l'émission étant gérées simultanément, dans des threads séparés.
- La réception et l'émission étant gérées simultanément, dans des threads séparés.

Server.py

```
# coding: utf-8

import socket, sys, threading

class ThreadClient(threading.Thread):

    def __init__(self, ip, port, clientsocket):
        threading.Thread.__init__(self)
        self.ip = ip
        self.port = port
        self.clientsocket = clientsocket
        print("[+] Nouveau thread pour %s %s" % (self.ip, self.port, ))
```

Server.py

```
def run(self):
    # Dialogue avec le client :
    nom = self.getName() # Nom thread
    while 1:
        msgClient = self.clientsocket.recv(1024).decode("Utf8")
        if not msgClient or msgClient.upper() =="FIN":
            break
        message = "%s> %s" % (nom, msgClient)
        print(message)
        # Faire suivre le message à tous les autres clients :
        for cle in conn_client:
            if cle != nom: # ne pas le renvoyer à l'émetteur
                conn_client[cle].send(message.encode("Utf8"))
        # Fermeture de la connexion :
        self.connexion.close() # couper la connexion côté serveur
        del conn_client[nom] # supprimer son entrée dans le dictionnaire
        print("Client %s déconnecté." % nom)  # Le thread se termine ici
```

Server.py

```
# Initialisation du serveur - Mise en place du socket :
mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
        mySocket.bind((HOST, PORT))
except socket.error:
      print("La liaison du socket à l'adresse choisie a échoué.")
       sys.exit()
print("Serveur prêt, en attente de requêtes ...")
mySocket.listen(5)
```

Server.py

```
# Attente et prise en charge des connexions demandées par les clients:
conn_client = {} # dictionnaire des connexions clients
while 1:
    connexion, adresse = mySocket.accept()
    # Créer un nouvel objet thread pour gérer la connexion :
    th = ThreadClient(connexion)
    th.start()
    # Mémoriser la connexion dans le dictionnaire :
    it = th.getName() # identifiant du thread
    conn_client[it] = connexion
    print("Client %s connecté, adresse IP %s, port %s." %\ (it, adresse[0], adresse[1]))
    # Dialogue avec le client :
    msg ="Vous êtes connectés. SVP, envoyez vos messages. »
    connexion.send(msg.encode("Utf8"))
```

Server.py

- En résumé:
  - Ce serveur n'est pas utilisé lui-même pour communiquer : ce sont les clients qui communiquent les uns avec les autres, par l'intermédiaire du serveur.
  - Celui-ci joue donc le rôle d'un relais : il accepte les connexions des clients, puis attend l'arrivée de leurs messages.
  - Lorsqu'un message arrive en provenance d'un client particulier, le serveur le réexpédie à tous les autres, en lui ajoutant au passage une chaîne d'identification spécifique du client émetteur, afin que chacun puisse voir tous les messages, et savoir de qui ils proviennent.

client.py

- **Le script ci-après définit le programme client**
- **Vous constaterez que la partie principale du script est similaire à celle de l'exemple précédent.**
- **Seule la partie « Dialogue avec le serveur » a été remplacée.**
- **Au lieu d'une boucle while, vous y trouvez à présent les instructions de création de deux objets threads, dont on démarre la fonctionnalité aux deux lignes suivantes.**
- **Dans ce qui suit:**
  - Définition d'un client réseau gérant en parallèle l'émission
  - et la réception des messages (utilisation de 2 THREADS)

client.py

```
import socket, sys, threading
class ThreadReception(threading.Thread):
    """objet thread gérant la réception des messages"""
        def __init__(self, conn):
                threading.Thread.__init__(self)
                self.connexion = conn # réf. du socket de connexion
        def run(self):
                while 1:
                    message_recu = self.connexion.recv(1024).decode("Utf8")
                    print("*" + message_recu + "*")
                    if not message_recu or message_recu.upper() =="FIN":
                          break
                    # Le thread <réception> se termine ici.
                    # On force la fermeture du thread <émission> :
                    th_E._stop()
                    print("Client arrêté. Connexion interrompue.")
                    self.connexion.close()
```

client.py

```
class ThreadEmission(threading.Thread):
    """objet thread gérant l'émission des messages"""
        def __init__(self, conn):
            threading.Thread.__init__(self)
            self.connexion = conn # réf. du socket de connexion

        def run(self):
            while 1:
                message_emis = input()
                self.connexion.send(message_emis.encode("Utf8"))
```

client.py

```
# Programme principal - Établissement de la connexion :
connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
try:
        connexion.connect((host, port))
except socket.error:
        print("La connexion a échoué.")
        sys.exit()
print("Connexion établie avec le serveur.")

# Dialogue avec le serveur : on lance deux threads pour gérer
# indépendamment l'émission et la réception des messages :
th_E = ThreadEmission(connexion)
th_R = ThreadReception(connexion)
th_E.start()
th_R.start()
```

Quelques méthodes de l'objet socket

```
accept()                              : accepte une connexion, retourne un nouveau socket et une adresse c
bind(addr)                            : associe le socket à une adresse locale
close()                               : ferme le socket
connect(addr)                         : connecte le socket à une adresse distante
connect_ex(addr)                      : connect, retourne un code erreur au lieu d'une exception
dup()                                 : retourne un nouveau objet socket identique à celui en cours
fileno()                              : retourne une description du fichier
getpeername()                         : retourne l'adresse distante
getsockname()                         : retourne l'adresse locale
getsockopt(level, optname[, buflen])  : retourne les options du socket
gettimeout()                          : retourne le timeout ou none
listen(n)                             : commence à écouter les connexions entrantes
makefile([mode, [bufsize]])           : retourne un fichier objet pour le socket
recv(buflen[, flags])                 : recoit des données
recv_into(buffer[, nbytes[, flags]])  : recoit des données (dans un buffer)
recvfrom(buflen[, flags])             : reçoit des données et l'adresse de l'envoyeur
recvfrom_into(buffer[,nbytes,[,flags]) : reçoit des données et l'adresse de l'envoyeur (dans un buffer)
sendall(data[, flags])                : envoye toutes les données
send(data[, flags])                   : envoye des données mais il se peut que pas toutes le soit
sendto(data[, flags], addr)           : envoye des données à une adresse donnée
setblocking(0 | 1)                    : active ou désactive le blocage le flag I/O
setsockopt(level, optname, value)     : définit les options du socket
settimeout(None | float)              : active ou désactive le timeout
shutdown(how)                         : fermer les connexions dans un ou les deux sens.
```