19

Communications à travers un réseau et multithreading

Le développement extraordinaire de l'Internet a amplement démontré que les ordinateurs peuvent être des outils de communication extrêmement efficaces. Dans ce chapitre, nous allons explorer les bases de cette technologie, en effectuant quelques expériences avec la plus fondamentale des méthodes d'interconnexion entre programmes, afin de mettre en évidence ce qui doit être mis en place pour assurer la transmission simultanée d'informations entre plusieurs partenaires.

Pour ce qui va suivre, nous supposerons donc que vous collaborez avec d'autres personnes, et que vos postes de travail Python sont connectés à un réseau local dont les communications utilisent le protocole TCP/IP. Le système d'exploitation n'a pas d'importance : vous pouvez par exemple installer l'un des scripts Python décrits ci-après sur un poste de travail fonctionnant sous Linux, et le faire dialoguer avec un autre script mis en œuvre sur un poste de travail confié aux bons soins d'un système d'exploitation différent, tel que Mac OS ou Windows.

Vous pouvez également expérimenter ce qui suit sur une seule et même machine, en mettant les différents scripts en œuvre dans des fenêtres indépendantes.

Les sockets

Le premier exercice qui va vous être proposé consistera à établir une communication entre deux machines seulement. L'une et l'autre pourront s'échanger des messages à tour de rôle, mais vous constaterez cependant que leurs configurations ne sont pas symétriques. Le script installé sur l'une de ces machines jouera en effet le rôle d'un logiciel *serveur*, alors que l'autre se comportera comme un logiciel *client*.

Le logiciel serveur fonctionne en continu, sur une machine dont l'identité est bien définie sur le réseau grâce à une *adresse IP* spécifique[116]. Il guette en permanence l'arrivée de requêtes expédiées par les clients potentiels en direction de cette adresse, par l'intermédiaire d'un *port de communication* bien

[116] Une machine particulière peut également être désignée par un nom plus explicite, mais à la condition qu'un mécanisme ait été mis en place sur le réseau (DNS) pour traduire automatiquement ce nom en adresse IP. Veuillez consulter un ouvrage sur les réseaux pour en savoir davantage.

déterminé. Pour ce faire, le script correspondant doit mettre en œuvre un objet logiciel associé à ce port, que l'on appelle un *socket*.

Depuis une autre machine, le logiciel client tente d'établir la connexion en émettant une *requête* appropriée. Cette requête est un message qui est confié au réseau, un peu comme on confie une lettre à la Poste. Le réseau pourrait en effet acheminer la requête vers n'importe quelle autre machine, mais une seule est visée : pour que la destination visée puisse être atteinte, la requête contient dans son en-tête l'indication de l'adresse IP et du port de communication destinataires.

Lorsque la connexion est établie avec le serveur, le client lui assigne lui-même l'un de ses propres ports de communication. À partir de ce moment, on peut considérer qu'*un canal privilégié relie les deux machines*, comme si on les avait connectées l'une à l'autre par l'intermédiaire d'un fil (les deux ports de communication respectifs jouant le rôle des deux extrémités de ce fil). L'échange d'informations proprement dit peut commencer.

Pour pouvoir utiliser les ports de communication réseau, les programmes font appel à un ensemble de procédures et de fonctions du système d'exploitation, par l'intermédiaire d'objets interfaces que l'on appelle donc des *sockets*. Ceux-ci peuvent mettre en œuvre deux techniques de communication différentes et complémentaires : celle des *paquets* (que l'on appelle aussi des *datagrammes*), très largement utilisée sur l'internet, et celle de la *connexion continue*, ou *stream socket*, qui est un peu plus simple.

Construction d'un serveur rudimentaire

Pour nos premières expériences, nous allons utiliser la technique des *stream sockets*.
Celle-ci est en effet parfaitement appropriée lorsqu'il s'agit de faire communiquer des ordinateurs interconnectés par l'intermédiaire d'un réseau local. C'est une technique particulièrement aisée à mettre en œuvre, et elle permet un débit élevé pour l'échange de données.

L'autre technologie (celle des *paquets*) serait préférable pour les communications expédiées *via* l'Internet, en raison de sa plus grande fiabilité (les mêmes paquets peuvent atteindre leur destination par différents chemins, être émis ou ré-émis en plusieurs exemplaires si cela se révèle nécessaire pour corriger les erreurs de transmission), mais sa mise en œuvre est un peu plus complexe. Nous ne l'étudierons pas dans ce livre.

Le premier script ci-dessous met en place un serveur capable de communiquer avec un seul client. Nous verrons un peu plus loin ce qu'il faut lui ajouter afin qu'il puisse prendre en charge en parallèle les connexions de plusieurs clients.

```
 1#  # Définition d'un serveur réseau rudimentaire
 2#  # Ce serveur attend la connexion d'un client
 3#
 4#  import socket, sys
 5#
 6#  HOST = '192.168.1.168'
 7#  PORT = 50000
 8#  counter =0          # compteur de connexions actives
 9#
10#  # 1) création du socket :
11#  mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
12#
```

```
13#  # 2) liaison du socket à une adresse précise :
14#  try:
15#      mySocket.bind((HOST, PORT))
16#  except socket.error:
17#      print("La liaison du socket à l'adresse choisie a échoué.")
18#      sys.exit
19#
20#  while 1:
21#      # 3) Attente de la requête de connexion d'un client :
22#      print("Serveur prêt, en attente de requêtes ...")
23#      mySocket.listen(2)
24#
25#      # 4) Etablissement de la connexion :
26#      connexion, adresse = mySocket.accept()
27#      counter +=1
28#      print("Client connecté, adresse IP %s, port %s" % (adresse[0], adresse[1]))
29#
30#      # 5) Dialogue avec le client :
31#      msgServeur ="Vous êtes connecté au serveur Marcel. Envoyez vos messages."
32#      connexion.send(msgServeur.encode("Utf8"))
33#      msgClient = connexion.recv(1024).decode("Utf8")
34#      while 1:
35#          print("C>", msgClient)
36#          if msgClient.upper() == "FIN" or msgClient =="":
37#              break
38#          msgServeur = input("S> ")
39#          connexion.send(msgServeur.encode("Utf8"))
40#          msgClient = connexion.recv(1024).decode("Utf8")
41#
42#      # 6) Fermeture de la connexion :
43#      connexion.send("fin".encode("Utf8"))
44#      print("Connexion interrompue.")
45#      connexion.close()
46#
47#      ch = input("<R>ecommencer <T>erminer ? ")
48#      if ch.upper() =='T':
49#          break
```

Commentaires

- Ligne 4 : Le module **socket** contient toutes les fonctions et les classes nécessaires pour construire des programmes communicants. Comme nous allons le voir dans les lignes suivantes, l'établissement de la communication comporte six étapes.
- Lignes 6-7 : Ces deux variables définissent l'identité du serveur, telle qu'on l'intégrera au socket. **HOST** doit contenir une chaîne de caractères indiquant l'adresse IP du serveur sous la forme décimale habituelle, ou encore le nom DNS de ce même serveur (mais à la condition qu'un mécanisme de résolution des noms ait été mis en place sur le réseau). **PORT** doit contenir un entier, à savoir le numéro d'un port qui ne soit pas déjà utilisé pour un autre usage, et de préférence une valeur supérieure à 1024.
- Lignes 10-11 : Première étape du mécanisme d'interconnexion. On instancie un objet de la classe **socket()**, en précisant deux options qui indiquent le type d'adresses choisi (nous utiliserons des adresses de type « Internet ») ainsi que la technologie de transmission (datagrammes ou connexion continue (*stream*) : nous avons décidé d'utiliser cette dernière).
- Lignes 13 à 18 : Seconde étape. On tente d'établir la liaison entre le socket et le port de communication. Si cette liaison ne peut être établie (port de communication occupé, par exemple,

ou nom de machine incorrect), le programme se termine sur un message d'erreur. À propos de la ligne 15, remarquez que la méthode **bind()** du socket attend un argument du type tuple, raison pour laquelle nous devons enfermer nos deux variables dans une double paire de parenthèses.

- Ligne 20 : Notre programme serveur étant destiné à fonctionner en permanence dans l'attente des requêtes de clients potentiels, nous le lançons dans une boucle sans fin.
- Lignes 21 à 23 : Troisième étape. Le socket étant relié à un port de communication, il peut à présent se préparer à recevoir les requêtes envoyées par les clients. C'est le rôle de la méthode **listen()**. L'argument qu'on lui transmet indique le nombre maximum de connexions à accepter en parallèle. Nous verrons plus loin comment gérer celles-ci.
- Lignes 25 à 28 : Quatrième étape. Lorsqu'on fait appel à sa méthode **accept()**, le socket attend indéfiniment qu'une requête se présente. Le script est donc interrompu à cet endroit, un peu comme il le serait si nous faisions appel à une fonction **input()** pour attendre une entrée clavier. Si une requête est réceptionnée, la méthode **accept()** renvoie un tuple de deux éléments : le premier est la référence d'un nouvel objet de la classe **socket()**[117], qui sera la véritable interface de communication entre le client et le serveur, et le second un autre tuple contenant les coordonnées de ce client (son adresse IP et le n° de port qu'il utilise lui-même).
- Lignes 30 à 33 : Cinquième étape. La communication proprement dite est établie. Les méthodes **send()** et **recv()** du socket servent évidemment à l'émission et à la réception des messages, qui doivent impérativement être *des chaînes d'octets*. À l'émission, il faut donc prévoir explicitement la conversion des chaînes de caractères en données de type ***bytes***, et faire l'inverse à la réception.

> *La méthode **send()** renvoie le nombre d'octets expédiés. L'appel de la méthode **recv()** doit comporter un argument entier indiquant le nombre maximum d'octets à réceptionner en une fois. Les octets surnuméraires sont mis en attente dans un tampon, ils sont transmis lorsque la même méthode **recv()** est appelée à nouveau.*

- Lignes 34 à 40 : Cette nouvelle boucle sans fin maintient le dialogue jusqu'à ce que le client décide d'envoyer le mot « fin » ou une simple chaîne vide. Les écrans des deux machines afficheront chacune l'évolution de ce dialogue.
- Lignes 42 à 45 : Sixième étape. Fermeture de la connexion.

Construction d'un client rudimentaire

Le script ci-dessous définit un logiciel client complémentaire du serveur décrit dans les pages précédentes. On notera sa grande simplicité.

```
1#  # Définition d'un client réseau rudimentaire
2#  # Ce client dialogue avec un serveur ad hoc
3#
4#  import socket, sys
5#
6#  HOST = '192.168.1.168'
```

[117] Nous verrons plus loin l'utilité de créer ainsi un nouvel objet socket pour prendre en charge la communication, plutôt que d'utiliser celui qui a déjà créé à la ligne 10. En bref, si nous voulons que notre serveur puisse prendre en charge simultanément les connexions de plusieurs clients, il nous faudra disposer d'un socket distinct pour chacun d'eux, indépendamment du premier que l'on laissera fonctionner en permanence pour réceptionner les requêtes qui continuent à arriver en provenance de nouveaux clients.

```
 7#  PORT = 50000
 8#
 9#  # 1) création du socket :
10#  mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
11#
12#  # 2) envoi d'une requête de connexion au serveur :
13#  try:
14#      mySocket.connect((HOST, PORT))
15#  except socket.error:
16#      print("La connexion a échoué.")
17#      sys.exit()
18#  print("Connexion établie avec le serveur.")
19#
20#  # 3) Dialogue avec le serveur :
21#  msgServeur = mySocket.recv(1024).decode("Utf8")
22#
23#  while 1:
24#      if msgServeur.upper() == "FIN" or msgServeur =="":
25#          break
26#      print("S>", msgServeur)
27#      msgClient = input("C> ")
28#      mySocket.send(msgClient.encode("Utf8"))
29#      msgServeur = mySocket.recv(1024).decode("Utf8")
30#
31#  # 4) Fermeture de la connexion :
32#  print("Connexion interrompue.")
33#  mySocket.close()
```

Commentaires

- Le début du script est similaire à celui du serveur. L'adresse IP et le port de communication doivent être ceux du serveur.
- Lignes 12 à 18 : On ne crée cette fois qu'un seul objet socket, dont on utilise la méthode **connect()** pour envoyer la requête de connexion.
- Lignes 20 à 33 : Une fois la connexion établie, on peut dialoguer avec le serveur en utilisant les méthodes **send()** et **recv()** déjà décrites plus haut pour celui-ci.

Gestion de plusieurs tâches en parallèle à l'aide de threads

Le système de communication que nous avons élaboré dans les pages précédentes est vraiment très rudimentaire : d'une part il ne met en relation que deux machines, et d'autre part il limite la liberté d'expression des deux interlocuteurs. Ceux-ci 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. Lorsqu'il vient de recevoir une telle réponse, son système reste incapable d'en réceptionner une autre, tant qu'il n'a pas entré lui-même un nouveau message, et ainsi de suite.

Tous ces problèmes proviennent du fait que nos scripts habituels ne peuvent s'occuper que d'une seule chose à la fois. Lorsque le flux d'instructions rencontre une fonction **input()**, par exemple, il ne se passe plus rien tant que l'utilisateur n'a pas introduit la donnée attendue. Et même si cette attente dure très longtemps, il n'est habituellement pas possible que le programme effectue d'autres tâches pendant ce temps. Ceci n'est toutefois vrai qu'au sein d'un seul et même programme : vous savez cer-