

```

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-

tainement que vous pouvez exécuter d'autres applications entre-temps sur votre ordinateur, car les systèmes d'exploitation modernes sont *multi-tâches*.

Les pages qui suivent sont destinées à vous expliquer comment vous pouvez introduire cette fonctionnalité *multi-tâche* dans vos programmes, afin que vous puissiez développer de véritables applications réseau, capables de communiquer simultanément avec plusieurs partenaires.

Veuillez à présent considérer le script de la page précédente. Sa fonctionnalité essentielle réside dans la boucle **while** des lignes 23 à 29. Or, cette boucle s'interrompt à deux endroits :

- à la ligne 27, pour attendre les entrées clavier de l'utilisateur (fonction **input()**) ;
- à la ligne 29, pour attendre l'arrivée d'un message réseau.

Ces deux attentes sont donc successives, alors qu'il serait bien plus intéressant qu'elles soient *simultanées*. Si c'était le cas, l'utilisateur pourrait expédier des messages à tout moment, sans devoir attendre à chaque fois la réaction de son partenaire. Il pourrait également recevoir n'importe quel nombre de messages, sans l'obligation d'avoir à répondre à chacun d'eux pour recevoir les autres.

Nous pouvons arriver à ce résultat si nous apprenons à gérer plusieurs séquences d'instructions *en parallèle* au sein d'un même programme. Mais comment cela est-il possible ?

Au cours de l'histoire de l'informatique, plusieurs techniques ont été mises au point pour partager le temps de travail d'un processeur entre différentes tâches, de telle manière que celles-ci paraissent être effectuées en même temps (alors qu'en réalité le processeur s'occupe d'un petit bout de chacune d'elles à tour de rôle). Ces techniques sont implémentées dans le système d'exploitation, et il n'est pas nécessaire de les détailler ici, même s'il est possible d'accéder à chacune d'elles avec Python.

Dans les pages suivantes, nous allons apprendre à utiliser celle de ces techniques qui est à la fois la plus facile à mettre en œuvre et la seule qui soit véritablement *portable* (elle est en effet supportée par tous les grands systèmes d'exploitation) : on l'appelle la technique des processus légers ou **threads**¹¹⁸.

Dans un programme d'ordinateur, les threads sont des flux d'instructions qui sont menés en parallèle (quasi-simultanément), tout en partageant le même espace de noms global.

En fait, le flux d'instructions de n'importe quel programme Python suit toujours au moins un thread : le *thread principal*. À partir de celui-ci, d'autres *threads enfants* peuvent être amorcés, qui seront exécutés en parallèle. Chaque *thread enfant* se termine et disparaît sans autre forme de procès lorsque toutes les instructions qu'il contient ont été exécutées. Par contre, lorsque le *thread principal* se termine, il faut parfois s'assurer que tous ses *threads enfants* « meurent » avec lui.

Client réseau gérant l'émission et la réception simultanées

Nous allons maintenant mettre en pratique la technique des threads pour construire un système de *chat*¹¹⁹ simplifié. Ce système sera constitué d'un seul serveur et d'un nombre quelconque de clients. Contrairement à ce qui se passait dans notre premier exercice, personne n'utilisera le serveur lui-

¹¹⁸ Dans un système d'exploitation de type Unix (comme Linux), les différents *threads* d'un même programme font partie d'un seul *processus*. Il est également possible de gérer différents processus à l'aide d'un même script Python (opération *fork*), mais l'explication de cette technique dépasse largement le cadre de ce livre.

¹¹⁹ Le « *chat* » est l'occupation qui consiste à « papoter » par l'intermédiaire d'ordinateurs. Les canadiens francophones ont proposé le terme de *clavardage* pour désigner ce « bavardage par claviers interposés ».

même pour communiquer, mais lorsque celui-ci aura été mis en route, plusieurs clients pourront s'y connecter et commencer à s'échanger des messages.

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.

Le script ci-après définit le programme client. Le serveur sera décrit un peu plus loin. Vous constaterez que la partie principale du script (ligne 38 et suivantes) 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* (aux lignes 49 et 50), dont on démarre la fonctionnalité aux deux lignes suivantes. Ces objets *threads* sont créés par dérivation, à partir de la classe **Thread()** du module **threading**. Ils s'occuperont indépendamment de la réception et de l'émission des messages. Les deux *threads* *enfants* sont ainsi parfaitement encapsulés dans des objets distincts, ce qui facilite la compréhension du mécanisme.

```
1# # Définition d'un client réseau gérant en parallèle l'émission
2# # et la réception des messages (utilisation de 2 THREADS).
3#
4# host = '192.168.1.168'
5# port = 46000
6#
7# import socket, sys, threading
8#
9# class ThreadReception(threading.Thread):
10#     """Objet thread gérant la réception des messages"""
11#     def __init__(self, conn):
12#         threading.Thread.__init__(self)
13#         self.connexion = conn           # réf. du socket de connexion
14#
15#     def run(self):
16#         while 1:
17#             message_recu = self.connexion.recv(1024).decode("Utf8")
18#             print("*" + message_recu + "*")
19#             if not message_recu or message_recu.upper() == "FIN":
20#                 break
21#             # Le thread <réception> se termine ici.
22#             # On force la fermeture du thread <émission> :
23#             th_E._stop()
24#             print("Client arrêté. Connexion interrompue.")
25#             self.connexion.close()
26#
27# class ThreadEmission(threading.Thread):
28#     """Objet thread gérant l'émission des messages"""
29#     def __init__(self, conn):
30#         threading.Thread.__init__(self)
31#         self.connexion = conn           # réf. du socket de connexion
32#
33#     def run(self):
34#         while 1:
35#             message_emis = input()
36#             self.connexion.send(message_emis.encode("Utf8"))
37#
38# # Programme principal - Établissement de la connexion :
39# connexion = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
40# try:
41#     connexion.connect((host, port))
```

```

42# except socket.error:
43#     print("La connexion a échoué.")
44#     sys.exit()
45# print("Connexion établie avec le serveur.")
46#
47# # Dialogue avec le serveur : on lance deux threads pour gérer
48# # indépendamment l'émission et la réception des messages :
49# th_E = ThreadEmission(connexion)
50# th_R = ThreadReception(connexion)
51# th_E.start()
52# th_R.start()

```

Commentaires

- Remarque générale: Dans cet exemple, nous avons décidé de créer deux objets *threads* indépendants du thread principal, afin de bien mettre en évidence les mécanismes. Notre programme utilise donc trois threads en tout, alors que le lecteur attentif aura remarqué que deux pourraient suffire. En effet : le thread principal ne sert en définitive qu'à lancer les deux autres ! Il n'y a cependant aucun intérêt à limiter le nombre de threads. Au contraire : à partir du moment où l'on décide d'utiliser cette technique, il faut en profiter pour compartimenter l'application en unités bien distinctes.
- Ligne 7 : Le module **threading** contient la définition de toute une série de classes intéressantes pour gérer les threads. Nous n'utiliserons ici que la seule classe **Thread()**, mais une autre sera exploitée plus loin (la classe **Lock()**), lorsque nous devrons nous préoccuper de problèmes de synchronisation entre différents threads concurrents.
- Lignes 9 à 25 : Les classes dérivées de la classe **Thread()** contiendront essentiellement une méthode **run()**. C'est dans celle-ci que l'on placera la portion de programme spécifiquement confiée au thread. Il s'agira souvent d'une boucle répétitive, comme ici. *Vous pouvez parfaitement considérer le contenu de cette méthode comme un script indépendant, qui s'exécute en parallèle avec les autres composants de votre application.* Lorsque ce code a été complètement exécuté, le thread se referme.
- Lignes 16 à 20 : Cette boucle gère la réception des messages. À chaque itération, le flux d'instructions s'interrompt à la ligne 17 dans l'attente d'un nouveau message, mais le reste du programme n'est pas figé pour autant : les autres threads continuent leur travail indépendamment.
- Ligne 19 : La sortie de boucle est provoquée par la réception d'un message '**fin**' (en majuscules ou en minuscules), ou encore d'un message vide (c'est notamment le cas si la connexion est coupée par le partenaire). Quelques instructions de « nettoyage » sont alors exécutées, et puis le thread se termine.
- Ligne 23 : Lorsque la réception des messages est terminée, nous souhaitons que le reste du programme se termine lui aussi. Il nous faut donc forcer la fermeture de l'autre objet thread, celui que nous avons mis en place pour gérer l'émission des messages. Cette fermeture forcée peut être obtenue à l'aide de la méthode **_stop()**¹²⁰.

¹²⁰ Que les puristes veuillent bien me pardonner : j'admet volontiers que cette astuce pour forcer l'arrêt d'un thread n'est pas vraiment recommandable. Je me suis autorisé ce raccourci afin de ne pas trop alourdir ce texte, qui se veut seulement une initiation. Le lecteur exigeant pourra approfondir cette question en consultant l'un ou l'autre des ouvrages de référence mentionnés dans la bibliographie.

- Lignes 27 à 36 : Cette classe définit donc un autre objet thread, qui contient cette fois une boucle de répétition perpétuelle. Il ne pourra donc se terminer que contraint et forcé par la méthode décrite au paragraphe précédent. À chaque itération de cette boucle, le flux d'instructions s'interrompt à la ligne 35 dans l'attente d'une entrée clavier, mais cela n'empêche en aucune manière les autres threads de faire leur travail.
- Lignes 38 à 45 : Ces lignes sont reprises à l'identique des scripts précédents.
- Lignes 47 à 52 : Instanciation et démarrage des deux objets *threads enfants*. Veuillez noter qu'il est recommandé de provoquer ce démarrage en invoquant la méthode intégrée **start()**, plutôt qu'en faisant appel directement à la méthode **run()** que vous aurez définie vous-même. Sachez également que vous ne pouvez invoquer **start()** qu'une seule fois (une fois arrêté, un objet thread ne peut pas être redémarré).

Serveur réseau gérant les connexions de plusieurs clients en parallèle

Le script ci-après crée un serveur capable de prendre en charge les connexions d'un certain nombre de clients du même type que ce que nous avons décrit dans les pages précédentes.

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.

```

1# # Définition d'un serveur réseau gérant un système de CHAT simplifié.
2# # Utilise les threads pour gérer les connexions clientes en parallèle.
3#
4# HOST = '192.168.1.168'
5# PORT = 46000
6#
7# import socket, sys, threading
8#
9# class ThreadClient(threading.Thread):
10#     '''dérivation d'un objet thread pour gérer la connexion avec un client'''
11#     def __init__(self, conn):
12#         threading.Thread.__init__(self)
13#         self.connexion = conn
14#
15#     def run(self):
16#         # Dialogue avec le client :
17#         nom = self.getName()           # Chaque thread possède un nom
18#         while 1:
19#             msgClient = self.connexion.recv(1024).decode("Utf8")
20#             if not msgClient or msgClient.upper() == "FIN":
21#                 break
22#             message = "%s> %s" % (nom, msgClient)
23#             print(message)
24#             # Faire suivre le message à tous les autres clients :
25#             for cle in conn_client:
26#                 if cle != nom:        # ne pas le renvoyer à l'émetteur
27#                     conn_client[cle].send(message.encode("Utf8"))
28#
29#         # Fermeture de la connexion :
30#         self.connexion.close()       # couper la connexion côté serveur

```

```

31#         del conn_client[nom]      # supprimer son entrée dans le dictionnaire
32#         print("Client %s déconnecté." % nom)
33#         # Le thread se termine ici
34#
35# # Initialisation du serveur - Mise en place du socket :
36# mySocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
37# try:
38#     mySocket.bind((HOST, PORT))
39# except socket.error:
40#     print("La liaison du socket à l'adresse choisie a échoué.")
41#     sys.exit()
42# print("Serveur prêt, en attente de requêtes ...")
43# mySocket.listen(5)
44#
45# # Attente et prise en charge des connexions demandées par les clients :
46# conn_client = {}           # dictionnaire des connexions clients
47# while 1:
48#     connexion, adresse = mySocket.accept()
49#     # Créer un nouvel objet thread pour gérer la connexion :
50#     th = ThreadClient(connexion)
51#     th.start()
52#     # Mémoriser la connexion dans le dictionnaire :
53#     it = th.getName()        # identifiant du thread
54#     conn_client[it] = connexion
55#     print("Client %s connecté, adresse IP %s, port %s." %\
56#           (it, adresse[0], adresse[1]))
57#     # Dialogue avec le client :
58#     msg ="Vous êtes connecté. Envoyez vos messages."
59#     connexion.send(msg.encode("Utf8"))

```

Commentaires

- Lignes 35 à 43 : L'initialisation de ce serveur est identique à celle du serveur rudimentaire décrit au début du présent chapitre.
- Ligne 46 : Les références des différentes connexions doivent être mémorisées. Nous pourrions les placer dans une liste, mais il est plus judicieux de les placer dans un dictionnaire, pour deux raisons : la première est que nous devrons pouvoir ajouter ou enlever ces références dans n'importe quel ordre, puisque les clients se connecteront et se déconnecteront à leur guise. La seconde est que nous pouvons disposer aisément d'un identifiant unique pour chaque connexion, lequel pourra servir de clé d'accès dans un dictionnaire. Cet identifiant nous sera en effet fourni automatiquement par la classe **Thread()**.
- Lignes 47 à 51: Le programme commence ici une boucle de répétition perpétuelle, qui va constamment attendre l'arrivée de nouvelles connexions. Pour chacune de celles-ci, un nouvel objet **ThreadClient()** est créé, lequel pourra s'occuper d'elle indépendamment de toutes les autres.
- Lignes 52 à 54 : Obtention d'un identifiant unique à l'aide de la méthode **getName()**. Nous pouvons profiter ici du fait que Python attribue automatiquement un nom unique à chaque nouveau thread : ce nom convient bien comme identifiant (ou clé) pour retrouver la connexion correspondante dans notre dictionnaire. Vous pourrez constater qu'il s'agit d'une chaîne de caractères, de la forme « **Thread-N** » (N étant le numéro d'ordre du thread).
- Lignes 15 à 17: Gardez bien à l'esprit qu'il se créera autant d'objets **ThreadClient()** que de connexions, et que tous ces objets fonctionneront en parallèle. La méthode **getName()** peut alors

être utilisée au sein d'un quelconque de ces objets pour retrouver son identité particulière. Nous utiliserons cette information pour distinguer la connexion courante de toutes les autres (voir ligne 26).

- Lignes 18 à 23 : L'utilité du thread est de réceptionner tous les messages provenant d'un client particulier. Il faut donc pour cela une boucle de répétition perpétuelle, qui ne s'interrompra qu'à la réception du message spécifique : « fin », ou encore à la réception d'un message vide (cas où la connexion est coupée par le partenaire).
- Lignes 24 à 27 : Chaque message reçu d'un client doit être réexpédié à tous les autres. Nous utilisons ici une boucle **for** pour parcourir l'ensemble des clés du dictionnaire des connexions, lesquelles nous permettent ensuite de retrouver les connexions elles-mêmes. Un simple test (à la ligne 26) nous évite de réexpédier le message au client d'où il provient.
- Ligne 31 : Lorsque nous fermons un socket de connexion, il est préférable de supprimer sa référence dans le dictionnaire, puisque cette référence ne peut plus servir. Et nous pouvons faire cela sans précaution particulière, car les éléments d'un dictionnaire ne sont pas ordonnés (nous pouvons en ajouter ou en enlever dans n'importe quel ordre).

Jeu des bombardes, version réseau

Au chapitre 15, nous avons commenté le développement d'un petit jeu de combat dans lequel des joueurs s'affrontaient à l'aide de bombardes. L'intérêt de ce jeu reste toutefois fort limité, tant qu'il se pratique sur un seul et même ordinateur. Nous allons donc le perfectionner, en y intégrant les techniques que nous venons d'apprendre. Comme le système de « chat » décrit dans les pages précédentes, l'application complète se composera désormais de deux programmes distincts : un logiciel serveur qui ne sera mis en fonctionnement que sur une seule machine, et un logiciel client qui pourra être lancé sur toute une série d'autres.

Du fait du caractère portable de Python, il vous sera même possible d'organiser des combats de bombardes entre ordinateurs gérés par des systèmes d'exploitation différents (*Linux ↔ Windows ↔ Mac OS*).