Python – les sockets

Rappel sur TCP

TCP (Transmission Control Protocol), est un protocole **orienté connexion**. Les applications sont connectées pour communiquer et TCP permet de s'assurer qu'une information envoyée au travers du réseau bien été réceptionnée par l'autre application. Si la connexion est rompue pour une raison quelconque, les applications doivent rétablir la connexion pour communiquer de nouveau.

Client et serveur

Il s'agit de créer deux applications :

- l'application serveur qui écoute sur le réseau en attendant des connexions
- l'application client qui se connecte au serveur.

Les différentes étapes

Voici dans un ordre simplifié les étapes du client et du serveur. Plus loin dans ce document vous programmerez un serveur pouvant communiquer avec plusieurs clients.

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.

Établir une connexion

Pour que le client se connecte au serveur, il lui faut deux informations :

- Le **nom d'hôte** (hostname) ou son adresse IP.
- Un **numéro de port**, qui est souvent propre au type de service utilisé. Le numéro de port est compris entre 0 et 65535 et les numéros entre 0 et 1023 sont réservés par le système. Il est préférable de ne pas les utiliser.

Les sockets

Les **sockets** sont des objets qui permettent d'ouvrir une connexion avec une machine locale ou distante et d'échanger avec elle.

Ces objets sont définis dans le module **socket** .

Les sockets

- Lancez l'interpréteur Python (Démarrer\Python 3.4\IDLE)
- Importez le module socket.

```
import socket
```

Créez le serveur

Il faut 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.SOCKSTREAM : le type du socket, SOCK_STREAM pour le protocole TCP. <code python> »> connexionprincipale = socket.socket(socket.AFINET,***

socket.SOCK_STREAM) </code> ==== Connecter le socket ==== Pour une connexion serveur, qui va attendre des connexions de clients, on utilise la méthode bind. Elle prend un paramètre : le tuple (nom_hote, port). Mais pour que le serveur écoute sur un port, il faut le configurer en conséquence en ne renseignant pas le nom de l'hôte sera vide mais en précisant un numéro de port 1024 et 65535. <code python> »> connexion_principale.bind[1] </code> Maintenant votre serveur et votre 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. Vous pouvez donc de nouveau saisir du code côté serveur : <code python> »> print(infos_connexion) ('127.0.0.1', 2901) </code> 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 (cette information peut être différente pour vous). ===== Faire communiquer les sockets ===== Pour faire communiquer les sockets, on utilise les méthodes send pour envoyer et recv pour recevoir. Les informations que vous transmettrez doivent être des chaînes de bytes et non pas de type str ! ==== Côté serveur : ==== <code python> »> connexionavecclient.send(b"Je viens d'accepter la connexion") 32 </code> La méthode send vous renvoie le nombre de caractères envoyés c'est à dire 32. Maintenant, 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 : <code python> »> msgrecu = connexionavecserveur.recv(1024) »> print(msgrecu) b"Je viens d'accepter la connexion" </code> Vous pouvez de cette manière faire communiquer des applications entre elles, applications pouvant être situées sur des ordinateurs différents. Le client peut également envoyer des informations au serveur et le serveur peut les réceptionner, tout cela grâce à ces méthodes send et recv. ===== Fermer la connexion ===== Pour fermer la connexion, il faut appeler la méthode close de notre socket. ==== Côté serveur : ==== <code python> »> connexionavecclient.close() </code> ==== Côté client : ==== <code python> »> connexionavecserveur.close() </code> ===== Premier programme serveur ===== Voici un premier programme serveur qui n'accepte qu'un seul client et qui fonctionne jusqu'à recevoir du client le message fin. Enregistrez ce programme dans le fichier serveur.py puis lancez-le (Menu Run / Run module ou bien F5). À chaque message reçu, le serveur envoie en retour le message 'OK'. <code python> import socket hote = port = 12800 connexionprincipale = socket.socket(socket.AFINET, socket.SOCKSTREAM) connexionprincipale.bind((hote, port)) connexionprincipale.listen(5) print("Le serveur écoute à présent sur le port {}".format(port)) connexionavecclient, infosconnexion = connexionprincipale.accept() msgrecu = b"" while msgrecu != b"fin": msgrecu = connexionavecclient.recv(1024) # L'instruction ci-dessous peut lever une exception si le message # Réceptionné comporte des accents print(msgrecu.decode()) connexionavecclient.send(b"OK") print("Fermeture de la connexion") connexionavecclient.close() connexionprincipale.close() </code> ===== Premier programme client ===== Le programme client 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. Enregistrez ce programme dans le fichier client.py puis lancez-le. <code python> import socket hote = "localhost" port = 12800 connexionavecserveur = socket.socket(socket.AFINET, socket.SOCKSTREAM) connexionavecserveur.connect[2] print("Connexion établie avec le serveur sur le port {}".format(port)) msgaenvoyer = b"" while msgaenvoyer != b"fin": msgaenvoyer = input("> ") # Peut planter si vous tapez des caractères spéciaux msgaenvoyer = msgaenvoyer.encode() # On envoie le message connexionavecserveur.send(msgaenvoyer) msgrecu = connexionavecserveur.recv(1024) print(msgrecu.decode()) # Là encore, peut planter s'il y a des accents print("Fermeture de la connexion") connexionavecserveur.close() </code> La méthode encode de str prend en paramètre un nom d'encodage et transforme une chaîne str en chaîne bytes. Cela est nécessaire car la méthode send n'accepte que le type de données bytes. Cela se fait en fonction d'un encodage précis (par défaut, Utf-8). La méthode 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). Sous Windows utilisez de préférence l'encodage Latin-1. ===== Deuxième programme serveur ===== Ce qui sera amélioré : * accepter plusieurs clients ; * pouvoir envoyer ou recevoir plusieurs messages sans attendre un accusé de réception ; * gérer les erreurs. ==== Le module select ==== Le module select permet d'interroger plusieurs clients dans l'attente d'un message à réceptionner, sans paralyser le programme. Select va écouter sur une liste de clients et retourner au bout d'un temps précisé. Ce que renvoie select, c'est la liste des clients qui ont un message à réceptionner. Il suffit de parcourir ces clients, de lire les messages en attente (grâce à recv). Utilisation de la fonction select du module select La fonction select prend trois ou quatre arguments et en renvoie trois. * rlist : la liste des sockets en attente d'être lus ; * wlist : la liste des sockets en attente d'être écrits ; * xlist : la liste des sockets en attente d'une erreur ; * timeout : le délai pendant lequel la fonction attend avant de retourner. Si vous précisez en timeout 0, la fonction retourne immédiatement. Si ce paramètre n'est pas précisé, la fonction retourne dès qu'un des sockets change d'état (est prêt à être lu s'il est dans rlist par exemple) mais pas avant. Pour ce programme vous allez utiliser rlist et timeout. Il s'agit de mettre des sockets dans une liste et que select les surveille, en retournant dès qu'un socket est prêt à être lu. Comme cela votre programme ne bloque pas et il peut recevoir des messages de plusieurs clients dans un ordre complètement inconnu. Si vous ne le précisez pas le timeout, select bloque jusqu'au moment où l'un des sockets que vous écoutez est prêt à être lu. Si vous précisez un timeout de 0, select retournera tout de suite. Sinon, select retournera au bout du temps que vous indiquez en secondes, ou plus tôt si un socket est prêt à être lu. Si vous précisez un timeout de 1, la fonction va bloquer pendant une seconde maximum. Mais si un des sockets en écoute est prêt à être lu dans l'intervalle (c'est-à-dire si un des clients envoie un message au serveur), la fonction retourne prématurément. select renvoie trois listes (rlist, wlist et xlist), sauf qu'il ne s'agit pas des listes fournies en entrée mais uniquement des sockets à lire dans le cas de rlist. Exemple : <code python> rlist, wlist, xlist = select.select(clients_connectes, [], [], 0.05) </code> Cette instruction va écouter les sockets contenus dans la liste clients_connectes. Elle retournera au plus tard dans 50 millisecondes. Mais elle retournera plus tôt si un client envoie un message. La liste des clients ayant envoyé un message se retrouve dans la variable rlist. On la parcourt ensuite et on peut appeler recv sur chacun des sockets. N'hésitez pas à voir la documentation du module select === Modification du programme du serveur (le programme du client ne change pas) === Vous allez créer un serveur

pouvant accepter plusieurs clients, réceptionner leurs messages et leur envoyer une confirmation à chaque réception. Vous allez rajouter l'utilisation de select pour travailler avec plusieurs clients. select va permettre : * d'écouter plusieurs clients connectés * de savoir si un (ou plusieurs) clients sont connectés au serveur. Rappel : la méthode accept est aussi une fonction bloquante. <code python> import socket import select hote = '' port = 12800 connexionprincipale = socket.socket(socket.AFINET, socket.SOCKSTREAM) connexionprincipale.bind[3] connexionprincipale.listen(5) print("Le serveur écoute à présent sur le port {}".format(port)) serveurlance = True clientsconnectes = [] while serveurlance: # vérifier que de nouveaux clients ne demandent pas à se connecter # Pour cela, on écoute la connexionprincipale en lecture # On attend maximum 50ms connexionsdemandees, wlist, xlist = select.select([connexionprincipale], [], [], 0.05)
for connexion in connexionsdemandees: connexionavecclient, infosconnexion = connexion.accept() # On ajoute le socket connecté à la liste des clients clientsconnectes.append(connexionavecclient)
 # Maintenant, on écoute la liste des clients connectés # Les clients renvoyés par select sont ceux devant être lus (recv) # On attend là encore 50ms maximum # On enferme l'appel à select.select dans un bloc try # En effet, si la liste de clients connectés est vide, une exception # Peut être levée clientsalire = [] try: clientsalire, wlist, xlist = select.select(clientsconnectes, [], [], 0.05) except select.error: pass else: # On parcourt la liste des clients à lire for client in clientsalire: # Client est de type socket msgrecu = client.recv(1024) # Peut planter si le message contient des caractères spéciaux msgrecu = msgrecu.decode() print("Reçu {}".format(msgrecu)) client.send(b"OK") if msgrecu == "fin": serveurlance = False print("Fermeture des connexions") for client in clientsconnectes: client.close() connexion_principale.close() </code> Maintenant le serveur peut accepter des connexions de plus d'un client et ne se bloque pas dans l'attente d'un message, du moins pas plus de 50 millisecondes. Les déconnexions fortuites ne sont pas gérées non plus. === Pour aller plus loin : === Regarder la documentation du module socket, de select et de socketserver. Le module socketserver, propose une alternative pour monter vos applications serveur. Il en existe d'autres. Vous pouvez utiliser des sockets non bloquants (c'est-à-dire qui ne bloquent pas le programme quand vous utilisez leur méthode accept ou recv) ou des threads pour exécuter différentes portions de votre programme en parallèle.

[1]
, 12800)) </code> ==== Faire écouter le socket ==== Le socket est prêt à écouter sur le port 12800 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. Cela ne veut pas dire que le 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. Vous pouvez donc avoir bien plus de clients connectés. <code python> »» connexion_principale.listen(5) </code> ==== Accepter une connexion venant du client ==== Dernière étape, accepter une connexion. Aucune connexion ne s'est encore présentée mais la méthode accept qui va être utilisée va bloquer le programme tant qu'aucun client ne s'est connecté. 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. Le port de connexion du client n'est pas le même que celui du serveur, car le client, en ouvrant une connexion, passe par un port client qui va être choisi par le système parmi les ports disponibles. Il y a donc deux ports mis en œuvre dans une connexion TCP. <code python> »» connexion*avec*client, infos*connexion* = *connexion*principale.accept() </code>

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

===== Création du client ===== Créez votre socket de la même façon : <code python> »» import socket »» connexion*avec*serveur = socket.socket(socket.AF*INET, socket.SOCK*STREAM) </code> ==== Connecter le client ==== Pour se connecter à un serveur, on utilise la méthode **connect** qui prend en paramètre un tuple contenant le nom d'hôte (actuellement **localhost**) et le numéro du port identifiant le serveur (actuellement 12800) auquel on veut se connecter. <code python> »» connexion*avec*serveur.connect(('localhost', 12800
[2] [3]
hote, port