RES 302
TP 2 : Réseau en Python

Dans ces exercices, nous allons utiliser l'interface *socket* en Python pour réaliser des échanges entre un programme client et un programme serveur, deux processus différents. Vous pourrez faire tourner les deux processus sur la même machine, ou si la configuration de la salle le permet, sur des machines différentes. Le code restera le même. Formellement, un socket est un point de communication bidirectionnel par lequel un processus pourra émettre ou recevoir des informations. Moins formellement, au lieu de parler de la communication entre processus, parlons de la communication entre personnes par courrier. Cette communication nécessite que les personnes disposent d'une boîte aux lettres. Une boîte aux lettres est un point de communication ayant une adresse connue du monde extérieur. Les sockets et les processus sont respectivement l'analogue de ces points de communications et des individus. Lorsque deux processus veulent s'échanger des données, il faut que chacun d'eux disposent d'au moins un socket. Les sockets ne sont certes pas le seul moyen de communication entre deux processus mais un grand nombre d'applications sont fondées sur les sockets.

Quand une communication entre deux sockets est établie, on ne se préoccupe pas de savoir comment est réalisée la liaison réelle. Si par exemple la communication se fait grâce à des machines intermédiaires, le programme utilisant les sockets l'ignore complètement. C'est le principe de la hiérarchisation en couches de service comme étudié en cours. On situe les sockets au niveau de la couche 4 (Transport). Précisément, les sockets sont les points d'accès aux services de la couche transport.

La primitive socket

Pour créer un socket en python, il suffit d'utiliser la primitive suivante :

import socket

socket.socket(*family*, *type*, *protocol*)

Paramètre *family*

Lors de la création d'un socket, le paramètre *family* définit le système d'adresse utilisé ainsi que le protocole de bas niveau :

- AF_INET utilisé par le protocole IPv4 (Internet Protocol version 4, l'actuel protocole Internet). Les adresses des machines sont des entiers sur 32 bits. On les représente souvent en base 256 sous la forme {\it n1.n2.n3.n4} . Par exemple l'adresse 130.79.7.13 est en fait représentée de manière interne $130 * 256^{3} + 79 * 256^{2} + 7 * 256 + 13$
- AF_INET6 protocole IPv6 qui remplacera le protocole IPv4. Les adresses IPv6 sont codées sur 128 bits.

Paramètre *type*

Il existe plusieurs types de socket, selon le mode d'utilisation :

- SOCK_DGRAM : la machine va recevoir et envoyer par paquets (appelés {*datagrammes*). Ces paquets peuvent être envoyés vers n'importe quelle machine et reçus de n'importe quelle machine (sachant que l'autre machine doit aussi utiliser ce type de transmission). Il n'y a pas de fiabilité et les paquets peuvent être modifiés pendant la transmission (erreurs). Ils peuvent aussi être perdus ou dupliqués et l'ordre n'est pas préservé. Par contre les paquets ne sont pas fragmentés ni regroupés ({\it analogie avec le courrier}).
- SOCK_STREAM : le socket doit, avant toute transmission, se *connecter* à la machine distante. Les données ne sont regroupées en trames. Il y a un flot continu de données d'émission et un flot continu de réception. La fiabilité est maximale : ni erreur, ni perte, ni duplication. Par contre comme il n'y a pas de trame, il faut faire attention que la lecture des données à la réception devra peut-être se faire plusieurs fois, même si l'envoi s'est fait en une seule fois. Des messages hors bandes peuvent être envoyés (*analogie avec le téléphone*).

Le tableau suivant résume les types de socket et leurs propriétés.

	SOCK_DGRAM	SOCK_STREAM
Fiabilité	NON	OUI
Préservation de l'ordre	NON	OUI
Non duplication	NON	OUI
Mode connecté	NON	OUI
Préservation des limites des messages	OUI	NON
Message hors-bande	NON	OUI

Le protocole

Ce champ identifie le protocole de transport utilisé pour mettre en oeuvre la transmission. Si on ne remplit pas ce champ, c'est le protocole par défaut qui est utilisé en fonction de la famille et du type de socket. Ce champ est généralement optionnel et n'est utilisé que pour préciser une variante d'un protocole pour un type et domaine donné. Pour le domaine AF_INET(6) et le type SOCK_DGRAM c'est le protocole UDP qui est utilisé par défaut. Pour le domaine AF_INET(6) et le type SOCK_STREAM c'est le protocole TCP qui est utilisé par défaut.

Ces deux protocoles utilisent des numéros de port pour permettre plusieurs connexions depuis une machine. Chaque machine ayant une adresse IP a ainsi 65535 ports numérotés de 1 à 65535. En fait, par la carte réseau de la machine on peut ainsi avoir plusieurs connexions *simultanées* (on parle de multiplexage : faire passer plusieurs connexions sur une même ligne).

Certains ports sont réservés à des services particuliers : echo(7), telnet(23), ftp(21), ssh(22), http(80). Ce sont les ports côté serveur. Du côté client, le numéro de port n'est pas fixe. Le système prend en général un port libre supérieur à 1024.

Conclusion

On utilise principalement les deux primitives suivantes pour créer un socket :

```
import socket

s = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)
s = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)
```

Adressage

Avec python, les adresses pour le domaine AF_INET sont représentées par un \textbf{tuple} de la forme (host, port) dans lequel :

- **host** est une chaîne de caractères qui représente soit un URL du type *www.google.fr* ou une adresse IPv4 de la forme *130.79.200.1* ;
- **port** est un entier qui représente le numéro de port ;

Concernant les adresses pour le domaine AF_INET6, elles sont représentées par un **tuple** de la forme (host, port, flowinfo, scopeid) dans lequel :

- **host** est une chaîne de caractère qui représente soit un URL du type *www.google.fr* ou une adresse IPv6 de la forme *2001:660:4701:1001::1*
- **port** est un entier qui représente le numéro de port ;
- **flowinfo** est un entier qui correspond au champ *flowlabel* de l'en-tête IPv6 ;
- **scopeid** est un entier qui désigne un ensemble d'interfaces en adéquation avec la portée (locale ou globale) de l'adresse IPv6 précisée dans **host**

Pour les adresses IPv4, on peut stipuler toutes les adresses disponibles sur la machine en renseignant une chaîne vide dans *host*. Cette syntaxe est également disponible pour IPv6.

Juste après l'appel à la fonction *socket()*, on peut lui attacher une adresse locale à l'aide de la fonction *bind().* Cette étape est nécessaire du côté serveur pour préciser sur quelle adresse et quel port notre programme est en écoute. Du côté client, on peut laisser le système d'exploitation choisir une adresse et un numéro de port adéquat. La primitive de cette fonction est la suivante :

```
import socket

  socket.bind(sockaddr)
```

Le paramètre *sockaddr* doit correspondre à une structure d'adresse valide en fonction du domaine de socket (AF_INET ou AF_INET6).

Envoyer et recevoir en UDP

On utilise directement des envois et réceptions de datagrammes en spécifiant à chaque fois l'adresse destination.

import socket

int socket.sendto(string, address)
(string,address) socket.recvfrom(buffsize)

La valeur de retour de la fonction *recvfrom* est un **tuple** correspondant aux données reçues (*string*) et à l'adresse (*address*) de la socket émettrice. On rappelle qu'une *address* est également un **tuple** au format ("a.b.c.d", num_port) pour la famille d'adresse AF_INET.

Connexions TCP

Ce mode de communication doit passer par une connexion entre le client et le serveur. Ensuite, lors d'envoi/réception de segments dans la socket, il n'est plus utile de spécifier l'adresse destination/source.

import socket

socket.listen(backlog)
(socket, addr) s.accept()
socket.connect(address)

int socket.send(string)
string socket.recv(bufsize)

EXERCICES

Dans la suite, il vous faudra coder deux programmes (dans deux fichiers différents). L'un par exemple client.py et l'autre serveur.py.

Exercice 1 : un client – un serveur

Dans ce premier exercice, nous allons développer un serveur capable de gérer un seul client. Le principe est qu'un serveur démarre et attend une connexion d'un client. Lorsqu'un client se connecte, ce dernier envoie une chaine de caractères au serveur (par exemple « Hello »). Ce dernier affiche la chaîne de caractère reçu sur la sortie standard.

1.1 Socket TCP

Nous allons d'abord réaliser ce travail avec des sockets TCP, qui requièrent donc une connexion TCP entre le client et le serveur avant tout échange possible de données. Suivre la démarche suivante :

- Développer le programme serveur, qui instancie une socket TCP et attend une connexion. Dans un premier temps, vous pouvez créer la socket sur l'adresse IP locale de la machine (« localhost ») et sur le numéro de port 1212. Appeler la fonction qui permet au serveur d'attendre une connexion cliente.
- Développer un programme client, qui crée une socket sur « localhost » et le numéro de port 1313. Lancer une connexion depuis le client vers le serveur
- Reprendre le programme client, et le compléter en ajoutant l'émission d'une chaîne de caractères vers le serveur
- Reprendre le programme serveur, appeler la fonction de réception de message, et afficher la chaîne de caractères reçu
- Reprendre le programme serveur et passer en paramètre du programme le numéro de port sur lequel vous désirez lancer le serveur, en conservant un numéro de port par défaut dans votre programme.
- Reprendre le programme client et passer en paramètre du programme l'adresse IP et le numéro de port **du serveur** sur lequel le client devra se connecter.
- Sur le même principe, mais pour la chaîne de caractères côté client, au lieu de coder la chaîne dans le programme, modifier le client pour qu'en cours d'exécution le programme demande à l'utilisateur d'entrer une chaîne de caractères, que vous prendrez le soin d'envoyer
- Dans le programme client, ajouter une boucle à l'infini qui demande une chaîne de caractères, puis l'envoie.

1.2 Socket UDP

Reprendre les programmes client et serveur avec les dernières fonctionnalités, et changer l'usage des socket TCP en socket UDP.

Exercice 2 : Un serveur multi-client

Dans cet exercice nous allons développer un serveur multi-utilisateurs (cependant à capacité réduite). Le serveur attend la connexion d'un client, puis un deuxième, et ensuite attend un message du premier client pour le retransmettre vers le deuxième client. Nous allons donc écrire 3 programmes différentes : un programme serveur, un programme client émetteur, et un programme client récepteur. Ce travail est à réaliser en TCP dans un premier temps, et en UDP dans un deuxième temps.

2.1 Socket TCP.

- Reprendre le programme serveur de l'exercice 1.1, et ajouter l'attente d'une seconde connexion de client.
- Reprendre le programme client de l'exercice 1.1 et le dupliquer en deux fichiers (par exemple client_em.py et client_rec.py).
- Modifier le programme client_rec.py en remplaçant la boucle d'attente sur l'entrée standard et l'émission du message par une attente de message du serveur. Sur réception d'un message, le client devra afficher la chaîne de caractères reçu.

2.2 Socket UDP

Reprendre le même programme qu'en 2.2, en utilisant des sockets UDP

2.3 Optimisation et semblant de protocole de signalisation

Dans cet exercice, nous allons chercher à écrire le code client dans un seul fichier. Le programme devra se connecter au serveur, puis selon qu'il est exécuté en premier ou deuxième, ce même programme devra soit envoyer une chaîne de caractères, soit en attendre une du serveur. Pour cela, c'est le serveur qui va indiquer au client s'il est le premier ou le deuxième.
Ainsi, il faudra suivre le mode opératoire suivant : le serveur est lancé et attends 2 connexions de clients à suivre. Lors de la première connexion, le serveur accepte la connexion, et envoie '1' au client qui vient de se connecter, lui indiquant que c'est à lui d'envoyer les messages. Lors de la deuxième connexion, le serveur enverra '2' immédiatement après avoir accepté la connexion. Il faudra donc lancer deux instances du même programme client.

Réaliser les modifications nécessaires sur le programme que vous aviez réalisé en 2.1, en utilisant des sockets TCP uniquement.