IUT LANNION 1ère Année BUT Réseaux informatiques et Télécoms	**R208 - Traitement de données structurées**
Programmation Objet Python : utilisation de classes Python	TD4 (2H)

L'objectif de ce TD est d'apprendre à utiliser des classes existantes de programmation réseaux, ce qui vous permettra de faire un lien avec les notions de réseau vues en R&T.

1. Présentation des Sockets Réseaux en Python (45'/60')

Le module socket permet de programmer des accès réseau bas niveau en Python. Dans ce chapitre, nous présenterons uniquement l'utilisation des sockets TCP/IP mais le module socket permet, en utilisant la même API, de créer et d'utiliser des sockets Unix, Bluetooth, etc.

La programmation réseau repose sur les sockets (couche 5)

Les **sockets** sont un modèle permettant la **communication inter processus** (IPC - Inter Process Communication) aussi bien sur une même machine qu'à travers un réseau TCP/IP.

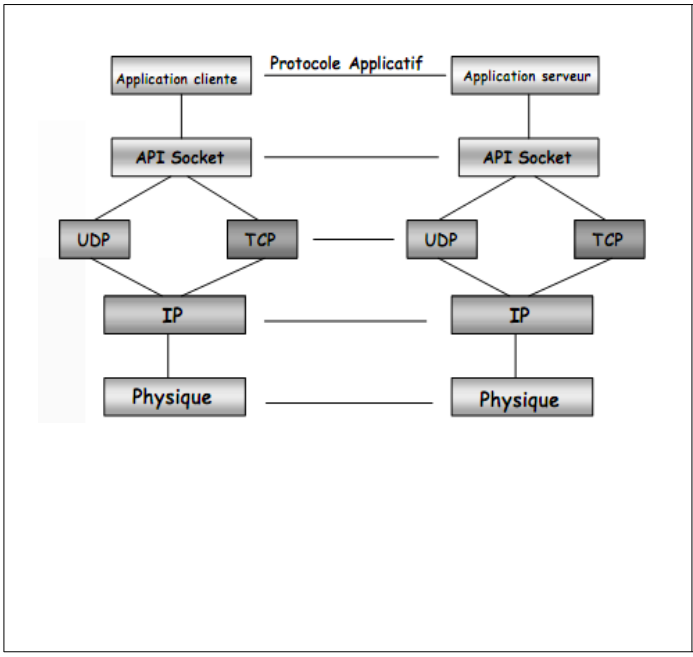
Les applications communicantes peuvent être développées dans différents langages de programmation pour des systèmes d'exploitation différents ; elles s'échangent en général des chaînes de caractères, ou des objets.

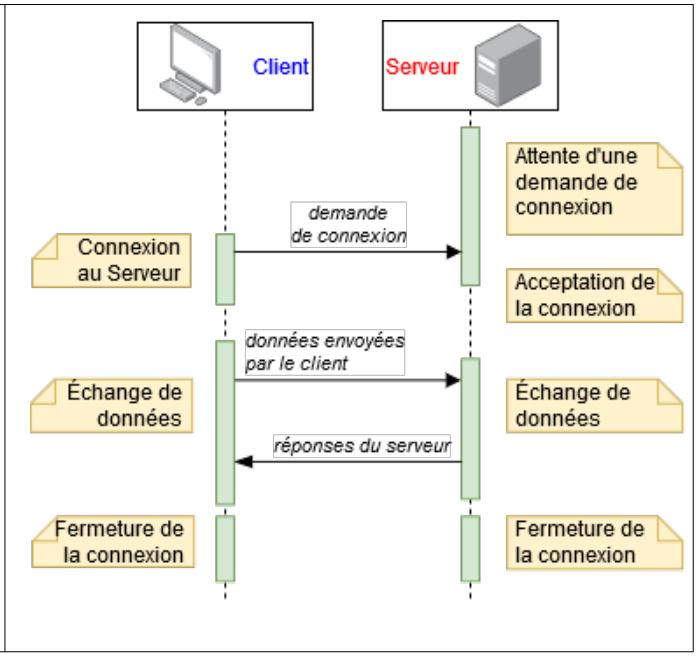
Il existe 2 modes de communications pour les sockets :

· le mode connecté : le protocole TCP (couche 4)

· le mode non connecté : le protocole UDP (couche 4)

L'API socket suppose une architecture client/serveur. Le serveur est en attente d'une connexion d'un client. Des données peuvent être lues et écrites aussi bien par le client que par le serveur.





1.1. LES SOCKETS RÉSEAUX EN PYTHON

Pour manipuler des connections avec les sockets, il faut importer le module socket ***import socket.***
Ce module contient notamment une classe socket regroupant (comme toutes les classes) des propriétés et des méthodes.
On peut visualiser rapidement ce qui compose la classe socket avec ***print(dir(socket))***
La documentation est accessible ici :
https://docs.python.org/3/library/socket.html
https://github.com/python/cpython/blob/3.10/Lib/socket.py

Pour simplifier (en réalité c'est plus complexe) et pour faire l'analogie avec la création de classe vue au TD précédent, cette classe est codée ainsi :

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

Avec Spyder, on accède à la documentation d'une méthode avec CTRL I
Nous allons maintenant exploiter cette classe socket afin de créer une connexion réseau via TCP entre deux programmes (un client et un serveur), en utilisant certaines de ses méthodes qui seront documentées plus précisément après.

1.2. Exemple d'implémentation d'un client TCP de bas niveau

1.2.1. Documentation :

- ***Definition : socket(family: int=..., type: int=..., proto: int=..., fileno: Optional[int]=...)*** : le constructeur de l'objet socket attend deux paramètres :
  - la famille d'adresses : ici socket.AF_INET = adresses Internet de type IPv4 ;
  - le type du socket : ici socket.SOCK_STREAM = protocole TCP.

  Une socket doit être fermée après usage (***Definition : close() -> None***)
  Rq : il existe d'autres constantes pour d'autres types de sockets (IPV6, UDP)

- ***Definition : connect(address: Union[_Address, bytes]) -> None*** : la méthode connect((ipServeur,portServeur)) prend en paramètre un Tuple

À partir de la socket de connexion du client, il est possible de lire les données reçues grâce à la méthode `recv()` et de répondre avec la méthode `send()`. Les données lues et écrites par une socket sont des objets de type `bytes`. On utilisera les fonctions `encode` et `decode` pour passer d'un chaîne de caractères à des octets et inversement.

Definition : send(data: bytes, flags: int=...) -> int
Definition : encode(encoding: str=..., errors: str=...) -> bytes
Definition : recv(bufsize: int, flags: int=...) -> bytes
Definition : decode(encoding: str=..., errors: str=...) -> str

1.2.2. TRAVAIL À RÉALISER

Tester le programme python client.py qui se connecte à un serveur sur la boucle locale et le port 5000 puis lui envoie un message et attend la réponse du serveur.

```
# coding: utf-8
import socket

hote = "localhost"
portServeur = 5000

#print(dir(socket))
try :
    clientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    clientSocket.connect((hote, portServeur))
    print(f"Connexion sur serveur {hote}:{portServeur}")
    print(f"Socket du client : {clientSocket.getsockname()}")
    envoi : str = "Message Client vers Serveur\n"
    clientSocket.send(envoi.encode('utf-8'))
    print(f"Emission vers le serveur : {envoi}")
    reception : bytes = clientSocket.recv(500)
    print(f"Reception du message client : {reception.decode('utf-8')}")
    clientSocket.close()
    print(f"Deconnexion  {hote}:{portServeur}.")
except Exception as err :
    print(f"voici le message d'erreur {err}")
```

Vous pouvez le tester avec le programme serveur fourni à lancer ainsi dans un terminal : **java -jar serveurTCP_CM.jar** (ce qui prouve que deux programmes écrits avec de langages différents peuvent communiquer avec des sockets)

Vous pouvez vérifier que le port est bien ouvert avec la commande **ss – antp | grep 5000**

Résultat attendu :	**Connexion sur serveur localhost:5000** **Socket du client : ('127.0.0.1', 61200)** **Emission vers le serveur : Message Client vers Serveur** **Reception du message client : Message Serveur vers Client** **Deconnexion localhost:5000.**

Quelle exception principale peut se produire ?

Créer une classe Client qui dispose des méthodes suivantes :

- connect(),
- sendMessage(),
- receiveMessage(),
- close()

pour lesquelles, on devra définir les paramètres et le retour.

Faire un programme de test (main) pour utiliser cette classe (n'oubliez pas de lancer le serveur!)

1.3. Exemple d'implémentation d'un serveur TCP de bas niveau

1.3.1. Documentation

- **socket() :** pour implémenter un serveur, on crée aussi un objet socket() du module socket en spécifiant le type identique au client.

On appelle ensuite les méthodes suivantes :

- **Definition : setsockopt(level: int, optname: int, value: Union[int, bytes]) -> None** : pour spécifier les paramètres pour la socket. Ici, on passe 1 pour la valeur socket.SO_REUSEADDR afin d'indiquer au système que le port utilisé par la socket peut être immédiatement réutilisé dès que cette dernière est fermée. Sinon, le système temporise la réutilisation et la ré-exécution du programme peut entraîner un échec.
- **Definition : bind(address: Union[_Address, bytes]) -> None : p**our spécifier sous la forme d'un n-uplet le nom ou l'adresse de l'hôte et le port utilisés par cette socket serveur.
- **Definition : listen(backlog: int=..., /) -> None** : pour spécifier qu'il s'agit d'une socket serveur et le nombre de connexions en attente qui peuvent être tolérées avant rejet par le système.
- **accept()** : Cette méthode attend qu'une connexion entrante se produise pour cette socket. Elle méthode retourne une socket de communication avec le client ainsi que l'adresse du client.

À partir de la socket de connexion du client, il est possible de lire les données reçues grâce à la méthode `recv()` et de répondre avec la méthode `send()`. Les données lues et écrites par une socket sont des objets de type `bytes`. On utilisera les fonctions encode et decode pour passer d'un chaîne de caractères à des octets et inversement.

1.3.2. TRAVAIL À RÉALISER

Tester le programme python qui crée un serveur TCP sur la boucle locale sur le port 5000.
Il attend indéfiniment la connexion d'un client TCP puis attend la réception d'un message du client et lui retourne.

```
# coding: utf-8

import socket

hote = 'localhost'
port = 5000
try :
    #Configuration du socket en serveur
    serveur = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    serveur.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    serveur.bind((hote, port))
    serveur.listen(1)

    #Attente Connexion
    print(f"Attente Connexion client {hote}:{port}")
    clientSocket, adresse = serveur.accept()

    #Reception
    buff :bytes
    buff = clientSocket.recv(100)
    message : str
    message = buff.decode('utf-8')
    print(f"Reception du message client : {message} Socket du client {adresse}")

    #Envoi
    envoi : str = "Message Serveur vers Client\n"
    buff = envoi.encode('utf-8')
    clientSocket.send(envoi.encode('utf-8'))
    print(f"Emission vers le client {envoi}")

    #Deconnexion
    print(f"Deconnexion  {hote}:{port}.")
    clientSocket.shutdown(socket.SHUT_RDWR)
    clientSocket.close()
    #serveur.shutdown(socket.SHUT_RDWR)
    serveur.close()
except Exception as err:
        print(f"voici le message d'erreur {err}")
```

Vous pouvez le tester votre programme avec votre client précédent.

Résultat attendu :	Attente Connexion client localhost:5000 Reception du message client : Message Client vers Serveur Socket du client ('127.0.0.1', 61200) Emission vers le client Message Serveur vers Client Deconnexion localhost:5000.

Quelle exception principale peut se produire ?

Créer une classe Serveur qui dispose des méthodes suivantes :

- demarre(), elle comportera une boucle qui permet d'attendre puis d'échanger avec un client
- stop(),

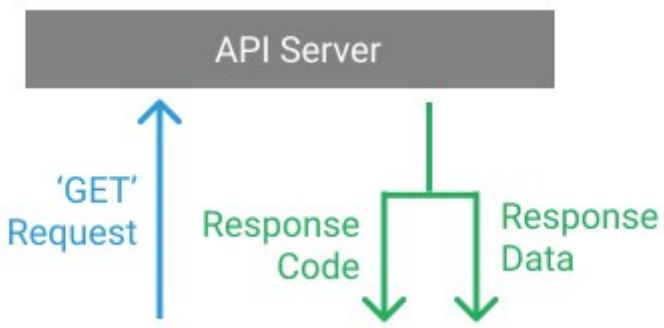
On utilisera une propriété pour compter le nombre de connexions reçues
pour lesquelles, on devra définir les paramètres et le retour.
Faire un programme de test (main) pour utiliser cette classe avec votre client précédent

1.4. CONNECTION À UN SERVEUR WEB

Modifier le programme client afin de vous connecter au serveur web de l'IUT pour lire la page d'accueil. Il faut envoyer : GET /index.php HTTP/1.1 \n afin de mettre en œuvre le protocole HTTP.
Quelle conclusion pouvons nous tirer de l'utilisation des connexions TCP de bas niveau ?

2. Utilisation des classes Python d'accès HTTP

Une API, ou **Application Programming Interface**, est un serveur qui vous permet de récupérer et d'envoyer des données à l'aide de code.



2.1. Différentes méthodes HTTP et codes d'état

Il existe différentes méthodes HTTP pour les API REST .
REST (representational state transfer) est un style d'architecture logicielle définissant un ensemble de contraintes à utiliser pour créer des services web. ... Dans un service web REST, les requêtes effectuées sur l'URI d'une ressource produisent une réponse dont le corps est formaté en HTML, XML, **JSON** ou un autre format.
Ces méthodes indiquent à l'API quelles opérations doivent être effectuées sur les données. Bien qu'il existe de nombreuses méthodes HTTP, les cinq méthodes répertoriées ci-dessous sont les plus couramment utilisées avec les API REST :

MÉTHODE HTTP	LA DESCRIPTION
GET	Récupérer les données existantes
POST	Ajouter de nouvelles données
PUT	Mettre à jour les données existantes
PATCH	Mettre à jour partiellement les données existantes
DELETE	Supprimer les données

Une fois qu'une API REST reçoit et traite une requête HTTP, elle renvoie une réponse avec un code d'état HTTP. Ce code d'état fournit des informations sur la réponse et aide l'application cliente à savoir de quel type de réponse il s'agit.
Les codes d'état sont numérotés en fonction de la catégorie du résultat :

PLAGE DE CODES	CATÉGORIE
1xx	Réponse informative
2xx	Opération réussie
3xx	Redirection
4xx	Erreur client
5xx	Erreur du serveur

Accès à l'API avec un point de terminaison = URL publiques exposée par le serveur pour qu'une application cliente puissent accéder aux ressources et aux données.

On va utiliser ici une API REST factice https://fakestoreapi.com/ qui propose plusieurs méthodes :

- GET/products
- GET/products/1
- GET/products/categories
- GET/products/category/jewelery
- GET/cart?userId=1
- GET/products?limit=5
- POST/products
- PUT/products/1
- PATCH/products/1
- DELETE/products/1

La bibliothèque Python `requests` permet d'interagir avec les services Web.
Vous pouvez (sur votre machine) installer cette bibliothèque en utilisant la commande pip comme ceci : $ pip install requests

2.2. TRAVAIL À RÉALISER

Tester le programme Python suivant qui interroge l'API REST

```
import requests
BASE_URL = 'https://fakestoreapi.com'
response = requests.get(f"{BASE_URL}/products")
print(response.status_code)
print(response.json())
```

Modifier le programme afin de n'afficher que la *description* des produits et la valeur *count*

Comme il y a beaucoup de résultats, il est possible de limiter l'affichage en ajoutant des paramètres à la demande : (Remarque : c'est équivalent à `products?`)

Tester la limitation	`query_params = {`
	`  "limit": 3`
	`}`
	`response = requests.get(f"{BASE_URL}/products", params=query_params)`

Tester l'ajout d'un produit qui utilise une autre méthode et qui transmet le produit au format JSON	`new_product = {`
	`  "title": 'test product',`
	`  "price": 13.5,`
	`  "description": 'lorem ipsum set',`
	`  "image": 'https://i.pravatar.cc',`
	`  "category": 'electronic'`
	`}`

```
response = requests.post(f"{BASE_URL}/products", json=new_product)
```

Vérifier l'insertion du produit en affichant la réponse et en interrogeant à nouveau l'API pour ce produit en particulier (https://fakestoreapi.com/)

Que pensez vous cette API ?

To be continued in BUT2 parcours DevClouds...