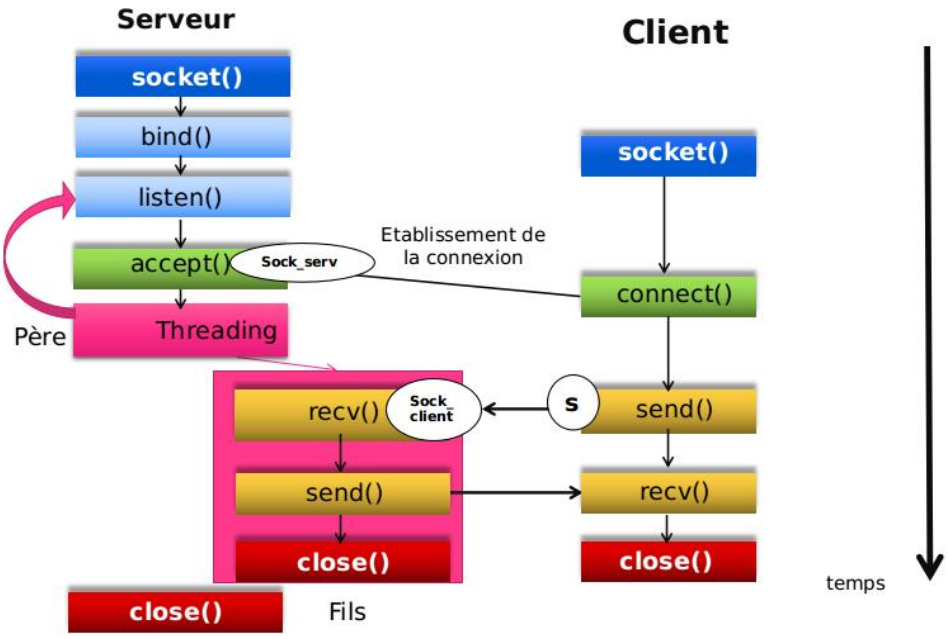
TP N°5 : Serveur concurrent

1. Objectif

Implémenter un serveur concurrent qui gère plusieurs clients simultanément en utilisant l'API socket en mode TCP.

2. Implémentation

Dans ce TP, nous allons implémenter un seul script, représentant le serveur concurrent. En ce qui concerne la partie client nous allons utiliser le script développé dans la fiche TP N°2 ou TP N°3.
On procède en plusieurs étapes, décrites schématiquement ci-après :



1) Ouvrir un éditeur de texte et enregistrer le fichier sous le nom **serveur_concurrent.py**
2) Importer les modules nécessaires
3) Création du socket
4) Associer le socket à une adresse IP et un numéro de port
5) Mettre le socket en mode écoute
6) Boucle While:
   Dans le script du serveur, une fois il a lancé un thread pour gérer la communication avec un client , il revient pour accepter de nouvelle demande de connexion et lancer de nouveau un autre thread pour gérer la communication avec un autre client. Ce travail se répète à l'infini.

   Selon-vous, est il nécessaire de faire appel à la jointure dans le cas d'un serveur concurrent?
7) Créer la classe de type thread
8) Surcharger le constructeur sans oublier d'appeler le constructeur threading.Thread.__init__
9) Surcharger la méthode run():
   C'est le code que devra s'occuper du transfert des données avec le client.

10) Exécuter le premier client
11) Exécuter un deuxième client:
    Ouvrez un 3ème terminal, lancez le même script du client et exécuter le sans pour autant interrompre l'exécution du premier client.
    Que se passe-t-il ?

3. Améliorer le script

3.1. Modifier le script client pour qu'il puisse maintenir un dialogue avec le serveur (envoyer plusieurs messages)

3.2. Modifier la boucle sans fin (au niveau du serveur) pour maintenir 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.

Utiliser la méthode lower() pour la comparaison,exemple:

```
s1 = 'STRIng'
if s1.lower()=='string'
    print('They are equal'
```

3.3. Travailler avec un dictionnaire

Modifier le script serveur, pour lui permettre d'enregistrer toutes les connexions des clients.
En python, il existe divers types composites : chaînes, listes, tuples et dictionnaire.

Les chaînes, listes et , tuples sont tous des séquences, c'est-à-dire des suites ordonnées d'éléments. Dans une séquence, il est facile d'accéder à un élément quelconque à l'aide d'un index (un nombre entier), mais à la condition expresse de connaître son emplacement.

Les dictionnaires que nous découvrons ici constituent un autre type composite. Ils ressemblent aux listes dans une certaine mesure (ils sont modifiables comme elles, les éléments peuvent être de n'importe quel type[1]), mais ce ne sont pas des séquences. Il est possible d'accéder à n'importe lequel d'entre eux à l'aide d'un index spécifique que l'on appellera une **clé**, laquelle pourra être a*lphabétique, numérique,* ou même d'*un type composite sous certaines conditions*.

Création d'un dictionnaire :
Pour créer un dictionnaire (nommé «étudiant») vide :

```
Étudiant ={}
```

Pour remplir un dictionnaire:

```
Étudiant["Nom"]="Mohamed"
Étudiant["age"]=20
```

Nom et age représentent les clés du dictionnaire.

Pour afficher un dictionnaire:

```
print(Étudiant)
```

[1] peuvent être des valeurs numériques, des chaînes, des listes, des tuples, des dictionnaires, mais aussi des fonctions, des classes ou des instances.

Pour afficher une valeur du dictionnaire:

```
print(Étudiant["Nom"])
```

Pour supprimer une entrée dans le dictionnaire:

```
del Étudiant["age"]
```

Pour afficher la liste des clés utilisées dans le dictionnaire :

```
print (Étudiant.keys())
```

La méthode values() renvoie la liste des valeurs mémorisées dans le dictionnaire :

```
print (Étudiant.values())
```

Pour savoir si un dictionnaire comprend une clé déterminée. La méthode renvoie une valeur 'True' ou 'False' (en fait, 1 ou 0).

```
print ("Dîplome" in Étudiant)
```

Création d'un dictionnaire imbriqué

En Python, un dictionnaire imbriqué est un dictionnaire à l'intérieur d'un autre dictionnaire.

```
Étudiant["Activités"]={}
Étudiant["Activités"][1]="Sport"
Étudiant["Activités"][2]="Dessin"
print(Étudiant)
```

3.4. Interface Graphique

Améliorer ce TP, en ajoutant une interface graphique.

Tkinter: Il existe beaucoup de modules pour construire des applications graphiques. Par exemple : Tkinter, wxpython, PyQt, PyGObject, etc. Tkinter est un module de base intégré dans Python , normalement vous n'avez rien à faire pour pouvoir l'utiliser. Tkinter permet de piloter la bibliothèque graphique Tk (Tool Kit), Tkinter signifiant tk interface.

1) Créer une Fenêtre

Nous allons importer le package Tkinter, créer une fenêtre et lui attribuée une dimension un titre.

```
from tkinter import *
fenetre = Tk()
fenetre.geometry('370x400')
fenetre.title("Client")
# Placer les widgets
#....
fenetre.mainloop()
```

La méthode mainloop() est la plus importante. Cette fonction fait appelle à une boucle infinie. Une boucle d'événements indique essentiellement au code de continuer à afficher la fenêtre jusqu'à ce que nous la fermions manuellement.

De plus, aucun code ne s'exécutera après la boucle, jusqu'à ce que la fenêtre sur laquelle elle est appelée soit fermée.

2) Les widget Tkinter

Pour créer un logiciel graphique vous devez ajouter dans une fenêtre des éléments graphiques que l'on nomme widget. Ce widget peut être tout aussi bien une liste déroulante que du texte.

2.1) Les labels

Les labels sont des espaces prévus pour afficher du texte.

```
label = Label(fenetre, text="PORT_Server")
label.place(x=10,y=5)
```

Le widget label possède des options en plus du text: *bg* (background color)*, font, justify*, ...etc

En créant des interfaces graphiques avec Tkinter, vous aurez certainement besoin de définir les emplacements de vos widgets. En effet, il existe trois méthodes principales : pack(), grid() et place().

Method place() : Ce gestionnaire de géométrie organise les widgets en les plaçant dans une position spécifique dans le widget parent.

Syntaxe : `widget.place( place_options )`

Il existe une liste des options possibles. Ce qui nous intéresse au décalage horizontal et vertical en pixels (x, y).

2.2) Entrée / input

```
W_Port = Entry(fenetre)
W_Port.place(x=100,y=5)
```

Récupérer la valeur d'un input : Pour récupérer la valeur d'un input il vous faudra utiliser la méthode **get()** :

2.3) Text

Les widgets de texte offrent des fonctionnalités avancées qui vous permettent de modifier un texte multiligne.

Syntaxe: `w = Text ( master, option, ... )`

master : Ceci représente la fenêtre

En plus des mêmes options du label (citées en haut), le widget text possède d'autres options comme *highlightcolor* (couleur de surbrillance)

```
W_Message = Text(fenetre, width=30, height=18, font=("Arial", 12))
W_Message.place(x=10,y=50)
```

Récupérer la valeur d'un input :

Le widget Text possède aussi la méthode **get()** pour récupérer la valeur d'un Texte saisi par l'utilisateur, qui a un argument de position de départ, et un argument de fin facultatif pour spécifier la position de fin du texte à récupérer.

Syntaxe : `get(start, end=None)`

Si 'end' n'est pas indiqué, un seul caractère spécifié à la position de départ 'start' sera renvoyé.

```
msg = W_Message.get("1.4","end")
```

Récupérer le texte de la première ligne, à partir du 4ème caractère j'usqu'à la fin.

Insérer une valeur dans un widget text :
Syntaxe : `insert(index, [,string]…)`
Les chaînes passées en deuxième argument sont insérées à la position spécifiée par l'index passé en premier argument.

```
msg=s.recv(1024)
W_Message.tag_config('r', background="lightsteelblue", foreground="royalblue")
W_Message.insert(END,"\n")
W_Message.insert(END,msg.decode()+"\n", 'r')
```

2.4) Les boutons

Les boutons permettent de proposer une action à l'utilisateur. Dans l'exemple ci-dessous, on lui propose de fermer la fenêtre.

```
B_connect = Button(fenetre, text="connect", command=lambda: get_connect(W_Port))
B_connect.place(x=300,y=5)
```

Ajouter un bouton fonctionnel
Il est possible d'attribuer une action au bouton grâce à l'option **command**. Dans l'exemple, on fait appel à la fonction **get_connect()**.

Passage de paramètre lors de clic d'un bouton
Pour faire passer des arguments à la fonction associée à l'option **command**, on ajout la fonction ***lambda***.

Désactiver un bouton Tkinter en Python.
Un bouton Tkinter a trois états: ***active, normal, disabled***.
disabled : pour griser le bouton et le rendre insensible.

```
        def get_connect(str):
            s.connect(('127.0.0.1', int(str.get())))
            B_connect['state'] = DISABLED
```

Attribuer une image à un bouton
Il est possible d'attribuer une image à un bouton ou un label, en utilisant la fonction **PhotoImage()** :

```
        send_img = PhotoImage(file="image_send.png")
        B_send.config(image=send_img)
```

Il est possible que le type de l'image en question ne soit pas reconnu par interpréteur. Il faut s'assurer que l'image est vraiment ***png.***