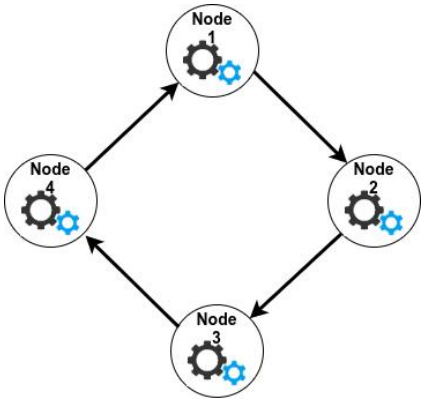
TP N°6 : Architecture Token Ring

1. Objectif



Dans ce TP, vous allez mettre en place un réseau en anneau (*Ring* avec jetons (*Token*) en utilisant l'API BSD sockets en langage python.

2. Principe de la topologie

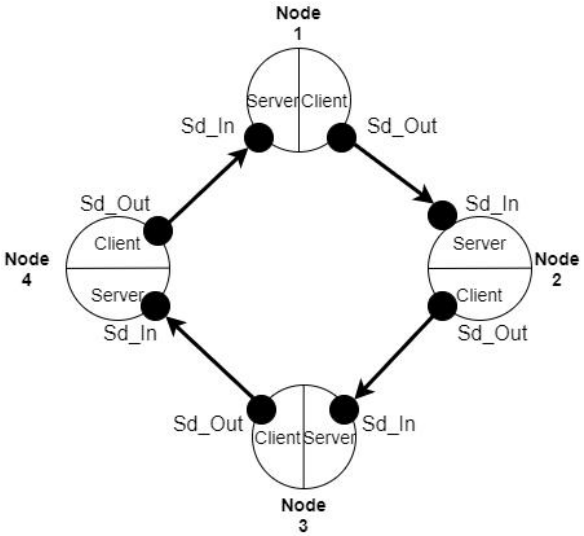
Dans un réseau en anneau à jetons (Token Ring), les nœuds sont connectés de manière unidirectionnelle et séquentielle. Par exemple, si le nœud 1 souhaite envoyer un message au nœud 3, il ne peut pas lui envoyer directement. Il doit d'abord envoyer le message au nœud 2 qui le transmettra ensuite au nœud 3.

Même si le flux est bidirectionnel avec TCP, il sera utilisé, dans ce TP, uniquement dans un seul sens

3. Conception

3.1. Définition:

Dans cette solution, l'architecture Token Ring suit le paradigme *Peer_to_Peer*. Ainsi, **chaque nœud doit contenir à la fois des fonctionnalités serveur et celles de client** afin de gérer respectivement les données entrantes et sortantes.



Un nœud est connecté au nœud précédent dans l'anneau en tant que serveur, et au nœud suivant en tant que client.

- La partie qui gère le socket serveur est nommée **Sd_In**
- La partie qui gère le socket client est nommée **Sd_Out**.

À savoir que le socket serveur sera utilisé pour réceptionner les données et celle du client sera utilisée pour l'envoie des données.

3.2. Création des threads

Dans la mesure où chaque nœud aura 02 fonctionnalités qui s'exécuteront en parallèles (l'envoie et le réception), chaque partie (**Sd_In** et **Sd_Out)** seront lancées chacune dans un thread.

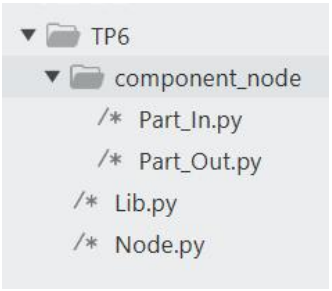
3.3. Attachement d'adresses aux sockets

Dans une architecture Token Ring, chaque nœud communique avec deux nœuds bien définis,

Donc il faut contrôlé l'attribution des adresses (adresse IP+ N°port) au socket : serveur.

Dans la mesure où la communication se fera dans une seul machine physique, l'adresse IP sera *localhost* pour tous les sockets. Seul le numéro de port qui sera manipulé.

3.4. Structure du TP



Dans ce TP, vous allez travailler avec des **modules** (scripts) que vous allez implémenter vous-même. Le but, est de séparer les fonctions et les classes dans des scripts à part et de les invoquer depuis un script principal. Pour une meilleur visibilité.

Certains modules seront organisés dans des dossiers. Un dossier qui contient des modules est appelé un **package** (paquetage en français). Le nom du package est le même que celui du dossier.

Exemple:

On crée un dossier **component_node** dans lequel on place les scripts **Part_In.py** et **Part_Out.py**.

- **component_node** est appelé «package»
- **Part_In.py** et **Part_Out.py** sont appelés «modules»

Pour importer les fonctions définies dans le script «Part_In.py», on écrit:

```
import component_node.Part_In
```

4. Implémentation

4.1. Module Lib:

Étape 1: Ouvrir un éditeur de texte et enregistrer le fichier sous le nom **Lib.py**
Étape 2: Importer les librairies nécessaire**s: socket, threading, sys**

```
1  import socket
2  import threading
3  import sys
```

4.2. Module Node

Ce module est le module principale qu'on lance au niveau de l'éditeur de commande:

```
>python Node.py 60001 1
```

- **60000:** représente le port pour le flux entrant, soit pour ***socket serverur***
- **1**: valeur booléenne. Si elle est égale à 1, alors le nœud possède le Token, sinon elle est égale à0.

Étape 1: Ouvrir un éditeur de texte et enregistrer le fichier sous le nom **Node.py**

Étape 2: Importer les modules :

```
1  from Lib import *
2  from component_node.Part_In import *
3  from component_node.Part_Out import *
```

Étape 3: Récupérer les paramètres d'entrée contenant les valeurs du numéro de port

Ajouter une instruction pour que l'utilisateur puisse saisir le numéro de port pour socket serveur. Le nom de la variable pour stocker le numéro de port est : **Port_In.**

Nous ajoutons un autre paramètre, celui de la possession du token. Seul le premier nœud possédera le token, ainsi la valeur de ce paramètre sera égale à 1, et 0 pour les autres nœuds.

```
5  PORT_In=int(sys.argv[1])
6  Have_Token=int(sys.argv[2]) #1: Have it, 0: Don't have it
```

Étape 4: Lancer un thread pour **Part_Out**

```
8  Sd_Out=Part_Out()
```

Étape 5: Lancer un thread pour **Part_In**:

Le but est d'instancier un objet «Sd_In» de la classe «Part_In». Cette classe est définie dans un module à part, appelé **Part_In.py**

```
9  Sd_In=Part_In(PORT_In, Have_Token)
10 Sd_In.start()
```

Étape 6: Activer le thread **Sd_Out**

- D'un côté, la primitive ***accept*** est bloquante. D'un autre côté, la primitive ***connect*** se déclenche automatiquement. Sauf que le socket serveur du nœud suivant devra être en écoute avant de lancer la demande de connexion depuis le nœud précédent. Ainsi, ajouter une instruction qui donnera à l'utilisateur la possibilité de déclencher la demande de connexion manuellement.

```
13 Sd_Out.port_next_Neighbor=int(input("Numéro de port du voisin: "))
14 Sd_Out.start()
```

4.3. Module Part_In:

Étape 1: Créer un dossier «component_node». Dans ce dossier nous allons créer deux modules: Part_In et Part_Out
Étape 2: Entrer dans le dossier «component_node»
Étape 3: Ouvrir un éditeur de texte et enregistrer le fichier sous le nom **Part_In.py**
Étape 4: Importer Lib

```
1  from Lib import *
```

Étape 5: Implémenter la classe :

Dans ce module **Part_In**, on défini la structure d'une classe **Part_In** dérivée d'un thread

```
class Part_In(threading.Thread):

    def __init__(self, port, T):
        threading.Thread.__init__(self)

        #Initiatialiser le port
        ...
        #Initiatialiser la valeur de T
        ...
        # créer socket appelée ss (self.ss)
        ...
        try:
            #attacher le socket déclaré un une adresse IP «localhost»
            #             et numéro de port récupéré dans «self.port»
            ...
        except:
            print("Le Sd_In n'arrive pas à s'attacher à l'adresse & numéro de port")
            sys.exit()

        # Mettre socket en mode écoute passive
        ...
```

Compléter le code en dessus.

Étape 6: Méthode run ():

Dans la topologie proposée dans ce TP, le serveur de chaque nœud traitera uniquement avec le client du nœud précédent. Donc, le serveur sera de type itératif (i.e: pas besoin d'avoir une boucle pour accepter plusieurs clients)

```
    def run(self):
        #Dans la méthode run (), socket accept une seul demande de connexion
        self.connexion, self.add=self.ss.accept()

        #Appel de  la fonction «Handle_Neighbor» qui est défini en hautde ce module
        #Part_In.py
        Handle_Neighbor(self.connexion, self.add, self.T)
```

Cependant, la gestion du service de réception des message se fera dans une fonction à part: **`Handle_Neighbor`**. Cette fonction est définie en haut en dessus de la classe «Part_In».

Étape 7: Fonction Handle_Neighbor

1. On vérifie si le process ne possède pas le Token (si t==0)
2. Si t==0 alors il ne possède pas le Token, donc il se met en attente de réception de message.
3. Si le message == «TOKEN», alors il a reçu le Token et l'utilisateur a la main pour saisir un ou plusieurs messages.

```
def Handle_Neighbor (con, add, t):
    if t==0:
        msg=con.recv(1024)
        if msg.decode()=="TOKEN":
            print(" Vous avez reçu le token")
            print(" Vous avez le droit à la parole")
            print(" Pour libérer la parole, il faut saisir le mot -- TOKEN--")
            while True:
                expression=input("Vous pouvez vous exprimer : ")
                if expression =="TOKEN" :
                    break
```

4.4. Module Part_Out:

Étape 1: Entrer dans le dossier «component_node»
Étape 2: Ouvrir un éditeur de texte et enregistrer le fichier sous le nom **Part_Out.py**

Dans ce module Part_Out, on défini la structure d'une classe Part_Out dérivée d'un thread

```
from Lib import *
class Part_Out(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        #Initiatialiser le port du noeud voisin
        self.port_next_Neighbor=0

        # créer socket appelée s (self.s)
        ...

    def run(self):
        try:
            # Se connecter au socket sd_In du noeud suivant
            # avec le port "self.port_next_Neighbor"
            ...
        except:
            print("La partie OUT n'arrive pas à se connecter voisin")
            sys.exit()
```

Compléter le code en dessus.

5. Gestion du TOKEN

- Pour le moment seul la topologie en anneau est réalisée. Le Token n'est pas présent dans le ou les scripts.

Le Token peut être considérée comme un bit ou chaîne de caractère. Dans ce TP, le Token est une chaîne de caractère = «TOKEN»

L'idée est de :

- Mettre tous les nœuds en attente du Token à exception du premier nœud. Ce travail est fait grâce à la méthode «recv» dans la fonction **`Handle_Neighbor`** .

- Pour le premier nœud, il ne faut pas déclencher l'envoie du Token avant que tous les nœuds soient connectés entre eux. Ainsi, on devra mettre le thread de Sd_Out en **pause** et le déclencher suivant une condition.

Pour faire, on travaille avec l'objet Event qui permettra de bloquer et débloquer un thread.

Event Object

Nous allons faire une petite pause dans ce TP, pour que je vous explique ce que c'est l'Objet Event.

En programmation concurrente avec les threads, nous avons parfois besoin de coordonner les threads avec une variable booléenne. Il peut s'agir de déclencher une action ou de signaler un résultat.

Cela peut être réalisé avec un verrou d'exclusion mutuelle (mutex) défini comme variable booléenne, mais cela ne permet pas aux threads d'attendre que la variable soit définie sur True. Au lieu de cela, on utilise un **objet événement**.

C'est quoi objet Event?

Python fournit un objet événement via la classe **threading.Event**.

Un objet **threading.Event** encapsule une variable booléenne qui peut être définie «*set*» (True) ou (False). Le **threading.Event** fournit un moyen simple de partager cette variable entre les threads qui peuvent servir de déclencheur pour une action. C'est l'un des mécanismes les plus simples de communication entre les threads.

Comment utiliser un objet Event?

Un objet event doit être créé et l'événement sera dans l'état "non défini":

```
# Create an instance of an event
event = threading.Event()
```

L'événement peut être défini via la fonction **set():**

```
# Mark the event as True => le thread n'est plus bloqué
event.set()
```

L'événement peut être marqué comme "False" via la fonction **clear()**:

```
# Mark the event as False => le thread est bloqué
event.clear()
```

Enfin, les threads peuvent attendre, via la fonction wait(), que l'événement soit défini. L'appel de cette fonction bloquera jusqu'à ce que l'événement soit marqué comme défini True (par exemple, un autre thread appelant la fonction set()):

```
# Wait for the event to be set to True
# Bloque le thread jusqu'à ce que le event soit True
# Débloque le thread lorsque le event devient True
event.wait()
```

Étape 1: Créer un objet event:

Dans ce TP, on déclare un objet event nommé: **__flag** dans le constructeur de la classe «Part_Out» de type Event:

```
self.__flag = threading.Event()
self.__flag.clear() # Set to False
```

On le défini par défaut à False, pour qu'on puisse directement le bloqué

Étape 2: Bloquer l'envoie du Token:

Une fois le socket s'est connecté au socket suivant, on bloque le thread en question pour qu'il n'envoie pas systématiquement le token.

```
    def run(self):
        try:
            # Se connecter au socket sd_In du noeud suivant
            # avec le port "self.port_next_Neighbor"
            ...
        except:
            print("La partie OUT n'arrive pas à se connecter voisin")
            sys.exit()

        self.__flag.wait()
        self.s.send(b"TOKEN")
```

Étape 3: Méthode **resume()**

On défini une méthode à l'intérieur de la classe Part_Out qui défini la valeur de **__flag** à True. Ainsi, débloquer le thread pour qu'il continu sont exécution

```
    def resume(self):
        self.__flag.set()
```

Étape 4: Débloquer l'envoie du token:

Maintenant, on doit définir à quel moment le thread responsable de l'envoie du «Token» se débloque. On sait qu'une fois le thread de la réception du thread a terminé de la tache de saisir les données, le thread repsonsable de l'envoie du thread devra l'envoyer. Ainsi, on appelle la méthode «resume()» de l'objet «Sd_Out» dans la focntion «Handle_Neighbor()».

```
def Handle_Neighbor (con, add, t, Sortie):
    if t==0:
        msg=con.recv(1024)
        if msg.decode()=="TOKEN":
            print(" Vous avez reçu le token")
            print(" Vous avez le droit à la parole")
            print(" Pour libérer la parole, il faut saisir le mot -- TOKEN--")
            while True:
                expression=input("Vous pouvez vous exprimer : ")
                if expression =="TOKEN" :
                    break
    Sortie.resume()
```

Sortie: représente le thread **Sd_Out**. Ainsi, **ne pas oublier** de faire passer le **Sd_Out** comme paramètre lors:

1. L'instantiation de l'Objet **Sd_In** du module **Node.py**:

```
Sd_In=Part_In(PORT_In, Have_Token, Sd_Out)
```

2. Dans le constructeur de la classe **Part_In**:

```
    def __init__(self, port, T, S):
        threading.Thread.__init__(self)

        #Initialiser le port
        ...
        #Initialiser la valeur de T
        ...
        #Thread Sd_Out
        self.Sortie=S
        #Créer socket appelée ss (self.ss)
        ...
```

3. L'appel de la fonction **Handle_Neighbor()**:

```
#Appel de  la fonction «Handle_Neighbor» qui est défini en hautde ce module
#Part_In.py
Handle_Neighbor(self.connexion, self.add, self.T, self.Sortie)
```

Étape 5: Cependant, on devra (comme utilisateur) avoir la main de déclencher la libération du Token au niveau du premier thread.

```
    if t==1:
        input("Vous êtes l'intiateur du token tapez Entrer pour le libérer")
    Sortie.resume()
    t=0 # Si il était l'intiateur, il ne l'est plus
```

Le «t» reçoit zéro, veut dire le nœud n'est plus l'initiateur du Token.

Étape 6: Répéter à l'infini:

Maintenant, on devra répéter à l'infini l'attente du Token et sa libération.

1. Ajouter la boucle while dans la fonction Handle_Neighbor():

```
1  from Lib import *
2  def Handle_Neighbor (con, add, t, Sortie):
3      while True:
4          if t==0:
```

2. Dans la méthode **run()**, entrer bloquer le thread, envoyer le Token, définir le **Event** à False pour qu'il soit bloqué de nouveau lors de la prochaine itération:

```
        while True:
            self.__flag.wait()
            self.s.send(b"TOKEN")
            self.__flag.clear()
```

6. Exécution

Ouvrir 03 terminaux, lancer le programme dans chaque terminal.

1. Donner comme numéro de port:

   >python Node.py 1001 1

   >python Node.py 1002 0

   >python Node.py 1003 0

2. Établir la connexion en donnant le Numéro de port distance.

7. Optimiser le travail

Sachant que le même programme sera exécuté N fois pour représenter les N nœuds et le numéro de port devra être unique pour chaque socket.
Améliorer ce TP, en ajoutant des instructions pour vérifier si le numéro de port n'est pris par un autre socket. En utilisant des fichiers texte.
Pour faire, ajouter un autre Module «Checking». Dans ce module, on défini plusieurs fonctions:

- **verif_nb_arg** : pour vérifier le nombre d'arguments;
- **verif_type_PORT** : vérifier que le numéro de port saisit est un entier;
- **verif_PORT_In_Liste** : vérifier dans un fichier texte si le numéro de port n'est pas déjà enregistré;
- **Add_PORT_Lis**t : enregistrer le numéro de port dans un fichier texte.

7.1. Gestion de fichier texte:

1) Ouvrir un fichier:

```
fileobject = open(filename, mode)
```

fileobject : descripteur;
filename : le nom ou le chemin du fichier, de type chaîne de caractère.

Mode	Description
'r'	Ouvrir un fichier en lecture. (par défaut)
'w'	Ouvrir un fichier pour l'écriture. Dans ce mode, si le fichier spécifié n'existe pas, il sera créé. Si le fichier existe, alors ses données sont détruites.
'x'	Ouvrir un fichier pour une création exclusive. Si le fichier existe déjà, l'opération échoue.
'a'	Ouvrir un fichier en mode ajout. Si le fichier n'existe pas, ce mode le créera. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier.
'+'	Ouvrir un fichier pour la mise à jour (lecture et écriture)

Il ne faut pas compter sur le codage par défaut, sans quoi le code se comportera différemment selon les plateformes. Sous Windows, il s'agit de "cp1252" mais de "utf-8" sous Linux.

```
f = open("etudiants.txt", "r", encoding = 'utf-8')
```

2) Fermer le fichier: `f.close()`

3) Utilisant with pour l'ouverture

La meilleure façon de faire est d'utiliser l'instruction **with**. Cela garantit que le fichier est fermé lors de la sortie du bloc à l'intérieur de **with**.

```
with open("etudiants.txt", "a", encoding = 'utf-8') as f:
    # traitments sur
    # le fichier f
```

4) Écrire dans un fichier

```
with open("etudiants.txt", "a", encoding = 'utf-8') as f:
    f.write("Mostafa \n ")
    f.write("Ismail \n ")
    f.write("Dounia \n ")
    # autres instruction
```

5) Lire un fichier

Pour lire un fichier, il faut l'ouvrir au moins en mode "r". En plus de cela, il faut également vous assurer que le fichier existe déjà car, en mode "r", la fonction **open()** génère une erreur ***FileNotFoundError*** si elle ne parvient pas à trouver un fichier.

Pour tester si un fichier existe ou non, nous pouvons utiliser la fonction **isfile()** du module **os.path.**

```
isfile(path)
```

La méthode **read(size)** permet de lire un nombre défini(size) de données . Si le paramètre size n'est pas spécifié, il lit et retourne jusqu'à la fin du fichier.

```
with open("etudiants.txt", "r", encoding='utf-8') as f:
    print("4 premières données : ", f.read(4))
    print("4 données suivantes : ", f.read(4))
    print("le reste du fichier : ", f.read())
    print("lecture supplémentaire ", f.read())
```

La méthode **read()** renvoie newline sous la forme '\ n'.

Il est possible de changer le curseur de fichier actuel (position) en utilisant la méthode **seek()**. De même, la méthode **tell()** renvoie la position actuelle (en nombre d'octets).

```
with open("etudiants.txt", "r", encoding='utf-8') as f:
    print("4 premières données ", f.read(4))
    print("position actuelle : ", f.tell())
    print("4 données suivantes : ", f.read(4))
    f.seek(0)  # position le curseur au début
    print("4 données : ", f.read(4))
```

On peut lire un fichier ligne par ligne en utilisant une boucle **for**. C'est à la fois efficace et rapide.

```
with open("etudiants.txt", "r", encoding='utf-8') as f:
    for ligne in f:
        print(ligne)
```

Dans le cas d'une ligne de string avec **\n,** on utilise la méthode **strip()** enlève le "\n"

```
line.strip()
```