TP N°8 : Horloge logique de Lamport

1. Objectif

Dans ce TP, vous allez implémenter l'algorithme des horloges logiques de Lamport.

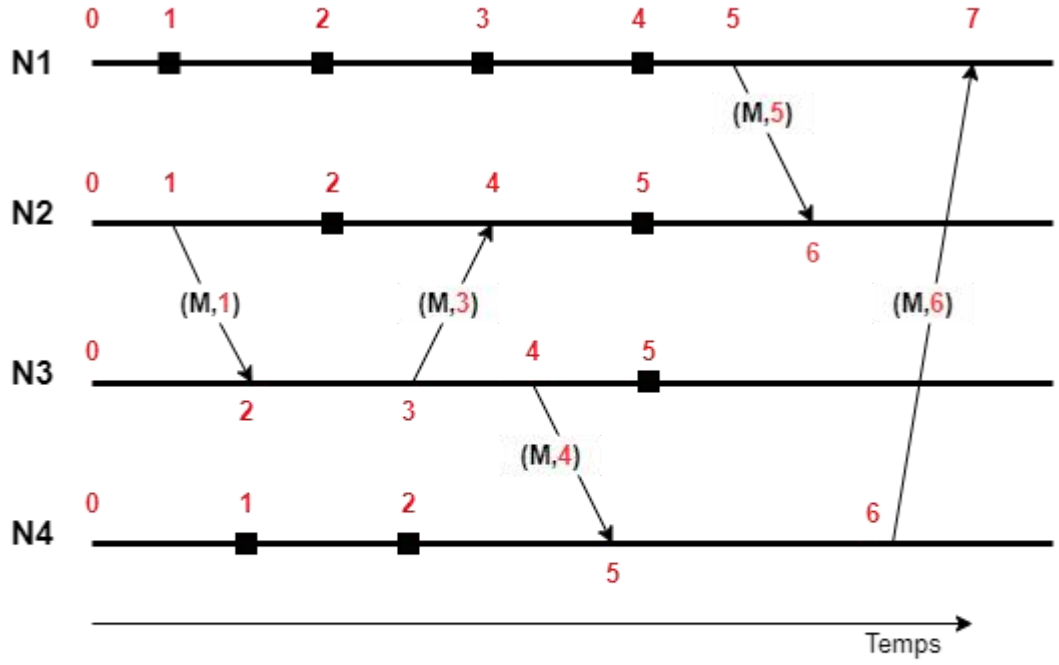
Le travail en question ce base sur la topologie en maille réalisée dans le TP N°7.

2. Principe de l'algorithme

Chaque processus **Pi** possède sa propre horloge **HLi** à valeurs entières. Un événement **e** qui se produit est daté par la valeur courante de **HLi** .

Les règles d'évolution des horloges au niveau du processus **Pi**:

- Initialisation : **HL =0**
- Événement local : **HL =HL +1** .
- Envoi d'un message **M** : envoyer un message **M** estampillé avec la valeur de l'horloge **(M, H)**, après avoir incrémenter la valeur de l'horloge.
- Réception d'un message **(M, H) : HL=max(HL, H) +1** .



Les valeurs en rouge représentent la valeur de l'horloge

■ : événement local

3. Structure du TP

En se basant sur le TP précédent, on ajoute un nouveau module **lamport** dans le dossier **component_node.**

4. Implémentation

4.1. Module lib:

On initialise une variable «HL» dans le module **lib**. HL contiendra la valeur de l'horloge locale.

```
HL= 0

class message():
    def __init__(self,t=0,c=0):
        self.clock=t
        self.content=c
```

Dans l'algorithme des horloges, les messages sont estampillés avec la valeur d'horloge. Il est possible d'envoyer le message concaténé avec la valeur d'horloge. Cependant, cette solution n'est pas scalable[1] (non évolutif).

Ainsi, il est plus judicieux de définir le message comme un objet dont les attributs sont «clock» et «content». «content» est le contenu du message.

À travers ce TP, on vas voir comment envoyer des messages de type **Object** instanciés depuis une classe.

Pour envoyer des objets à travers les sockets, il faut faire appel à la **sérialisation/déséria**blisation.

La sérialisation est le processus de conversion d'un objet en flux d'octets (*bytes*) pour stocker l'objet ou le transmettre à la mémoire, une base de données ou un fichier. Son principal objectif est d'enregistrer l'état d'un objet afin de pouvoir le recréer si nécessaire. Le processus inverse est appelé désérialisation.

Sérialisation d'objets Python

Le module **pickle** implémente des protocoles binaires de sérialisation et dé-sérialisation d'objets Python.

Sérialisation (et *désérialisation*) sont aussi connus sous les termes de *pickling*, de "*marshalling*" ou encore de "*flattening*".

[1] La scalabilité est un terme employé dans le domaine de l'informatique matérielle et logicielle, pour définir la faculté d'un produit informatique à s'adapter aux fluctuations de la demande en conservant ses différentes fonctionnalités.

Comparaison avec JSON:

- pickle est un format binaire, tandis que JSON est un format textuel (constitué de caractères Unicode et généralement encodé en UTF-8) ;
- JSON peut être lu par une personne, contrairement à pickle ;
- JSON offre l'interopérabilité avec de nombreux outils en dehors de l'écosystème Python, alors que pickle est propre à Python ;
- Par défaut, JSON n'est capable de sérialiser qu'un nombre limité de types natifs Python, contrairement à pickle.

Ainsi, ne pas oublier d'importer le module **pickle** dans **lib.py**

```
import pickle
```

4.2. Module node

Dans ce module, on vas rien apporter comme modification par rapport à ce qui a été fait dans TP N°7.

4.3. Module mesh_work:

Dans ce module, plusieurs modifications seront effectuées :

Étape 1: Définir la fonction **introduce_self :**

```
def introduce_self(s):
    msg=message()
    my_addr=s.getsockname()
    msg.content="New:"+str(my_addr[1])
    for i in range (len(my_list)):
        if my_list[i]!='':
            s.sendto(pickle.dumps(msg), ('localhost',int(my_list[i])))
```

pickle.**dumpS:** Renvoie la représentation sérialisée de l'objet sous forme de bytes. À ne pas confondre avec la méthode pickle.**dump** qui permet d'écrire la représentation sérialisée de l'objet dans un fichier

Étape 2: Définir la fonction **handle_reception:**

```
def handle_reception(s):
    msgD=message()
    while True:
        msg, addr=s.recvfrom(1024)
        msgD = pickle.loads(msg)
        if "New" in msgD.content:
            discover(msgD.content)
        else:
            print("Received : ",msgD.content," from : ",addr)
            lamport_when_receive(msgD)
```

pickle.**loadS** : renvoie l'objet reconstitué à partir de la représentation sérialisée data. À ne pas confondre avec la méthode pickle.**load.**

Étape 3: Définir la fonction **work_to_do :**

```
def work_to_do(s):
    msg=message()
    while True:
        print("----------------------------------------")
        print("Vous avez la main de faire quelque chose:")
        print("   1: Envoyer un message:")
        print("   2: Afficher la liste de vos voisins")
        choice=input("Choix ? : ")
        if choice== "1":
            msg.content=input("Entrer le message : ")
            msg.clock=lamport_when_send()
            y=input("Entrer le destinataire : ")
            s.sendto(pickle.dumps(msg),('localhost',int(y)))
        if choice== "2":
            lamport_local_event()
            print("My List = ",my_list)
            print("My HL = ",lib.HL)
```

4.4. Module lamport:

Étape 1: Ouvrir un éditeur de texte et enregistrer le fichier sous le nom **lamport.py** dans le répertoir **component_node**

```
1   import lib
2
3   def lamport_when_receive(msgD):
4       if lib.HL>msgD.clock:
5           lib.HL=lib.HL+1
6       else:
7           lib.HL=msgD.clock+1
8
9   def lamport_when_send():
10      lib.HL=lib.HL+1
11      return lib.HL
12
13  def lamport_local_event():
14      lib.HL=lib.HL+1
```

5. Exécution

Ouvrir 03 terminaux, lancer le programme dans chaque terminal.

>python node.py 1001

>python node.py 1002

>python node.py 1003