Auteurs : HOANG Nghia Hieu et TRINH Duc Tuan

Programmation Parallèle et Concurrente

I. Introduction :

Dans le cadre du projet Programmation Parallèle et Concurrente, nous avons pu réaliser une simulation d'un marché de l'énergie. Le but de ce programme est de concevoir et implémenter les scripts multi-thread et multi-process en Python. Ce projet va créer une simulation d'un marché d'énergie qui gère le taux de production, consommation des maisons, l'influence de la météo et les évènements aléatoires. A la fin de cette simulation, on va déduire l'évolution du prix d'énergie en fonction du temps.

II. Conception et Techniques :

En général, on a construit l'ensemble du modèle basé sur 5 fichiers : home.py, homes.py, market.py, weather.py et external.py

Le fichier home.py représente le prototype d'une maison dans ce modèle : chaque maison a son propre id, sa production et sa consommation par jour, et sa politique commerciale. En plus, on leur donne également 2 attributs : la différence entre les 2 valeurs et la valeur d'échange avec market. Le but de ces 2 attributs est pour calculer le nombre d'échanges sur ce marché.

homes.py est l'endroit où on représente l'ensemble des maisons qui participent dans cette simulation. Ici on initialise la liste des maisons, ensuite cela se passe la communication entre les maisons, et aussi cela ouvre un serveur client pour la communication avec le market. On donne à l'utilisateur le droit de saisir un nombre de maisons qu'il veut car cela rend le modèle plus dynamique. Pour le Message Queue, on a choisi d'associer chaque message avec leur type, chaque type est un int dont valeur est calculée selon leur ordre de préférence (plus bas = prendre d'abord). En fin, on a aussi une boucle while pour représenter le modèle jour après jour. Parce que les maisons représentent l'un des deux composants clés cette simulation, on a décidé d'afficher toutes les informations nécessaires sur les maisons dans la fenêtre de terminal de ce fichier.

Le fichier market.py agit comme un intermédiaire entre les facteurs qui apparaissent sur le marché, il contient donc toutes les fonctions qui traitent des problèmes liés. Ici, il gère les problèmes liés à la gestion des échanges avec les maisons, ainsi qu'à l'obtention d'informations météorologiques, et il crée un processus enfant qui contient des informations sur les facteurs externes et gère ces facteurs.

weather.py agit comme un processus séparé, il s'exécute et se met à jour à la nouvelle température chaque seconde. On lui donne des coefficients différents pour chaque température actuelle pour que la température qu'on prend soit plus logique. Et il commence et se termine en même temps que le modèle nait et mort pour s'assurer que le problème de terminaison et de mémoire.

External.py contient le prototype des facteurs externes.

III. Architecture et Protocoles :

1. Architecture :

On va commencer par le processus homes qui peuvent choisir leurs propres politiques énergétiques : ils peuvent donner leur surplus d'énergie aux autres homes ou le vendent au marché et diminuent le prix d'énergie. Les homes qui sont en pénurie d'énergie, quelle que soit sa politique commerciale, doivent l'acheter au marché si les autres ne font pas les dons. Donc au début cela se passe l'échange entre les maisons ; quand cela finit, on affecte le montant d'énergie restant de chaque maison à son attribut exchange_market que on utilisera pour la communication avec le market. Normalement, le prix va augmenter quand le taux de consommation surpasse celui de production. Quand le market finit ses échanges avec les maisons, il commence à récupérer la température à partir du processus weather. Le fait qu'on peut la prendre plusieurs fois représente comme on prend la température à différents moments de la journée. En plus, les évènements aléatoires ont aussi des impacts directs sur le prix d'énergie. Ensuite, le changement de la température donne naissance au changement du taux de production/consommation, donc le prix. A ce moment-là, on peut choisir si on veut continuer la simulation ou pas en tapant 'y'. Si on continue, la simulation continue et passe au jour suivant, sinon on termine tous les 2 processus homes et market.

2. Protocoles :

Cette simulation prend 4 types de processus :

- Home : représente les maisons avec leurs propres taux de production et consommation initialisé au début du lancement du programme. En plus, une politique énergétique sera aussi ajoutée pour indiquer quelle stratégie ils vont utiliser en face de l'excès d 'énergie ou pénurie d'énergie : 1. Toujours donner aux autres homes, 2. Toujours vendre au marché et 3. Vendre au marché si les autres n'en ont pas besoin.
- Market : représente le marché avec un prix d'énergie au début qui va changer de temps en temps avec les transactions entre lui-même avec les homes (vente ou achat), le changement de la température et les évènements aléatoires. En général, ce processus sera multi-threaded et prend toutes les transactions avec les processus de home avec les différents threads. En plus, il peut limiter le nombre de transactions en même temps avec les homes.
- Weather : simuler le changement de la température qui va influencer le taux de production et consommation.
- External : simuler les évènements aléatoires qui peuvent donner les impacts sur le prix d'énergie.

Concernant la communication entre les processus :

Les processus home vont communiquer entre eux avec un message queue, et avec market via sockets. Toutes les communications entre les homes se faites avec un Pool de process, et le market utilise des threads séparés pour traiter chaque connexion depuis chaque maison. Le processus weather va mettre à jour le changement de la température, et ce processus se roule tout seul. En plus, weather communique avec le market via un mémoire partagé de type Pipe. Le processus external, un processus fils de market, va envoyer les signaux quand il y aurait un évènement survenant qui va changer le prix d'énergie. Homes et market vont donner aux utilisateurs la mise à jour de cette simulation via ses terminaux.

IV. Algorithmes :

Il y a quelques algorithmes importants dans notre programme qu'on veut exprimer ici :

1. Algorithme pour l'échange entre les maisons :

```
with pool :
    pool.map(handle_first_send, homes_list)
with pool :
    for result in pool.map_async(handle_first_receive, homes_list).get():
        homes_list[result[0]-1].exchange_market = result[1]
with pool :
    for result in pool.map_async(handle_last_receive, homes_list).get():
        homes_list[result[0]-1].exchange_market = result[1]
```

```
def handle_first_send (home) :
    # Homes that are willing to give away their excess energy advertise their amount on the Message Queue, they send with their type associated in way that type_trading_policy_1 < type_trading_policy_3 so they have higher preference when other homes receive message.
    if home_trading policy == 1 or home_trading policy == 3 :
        messageQueue.send(home.gap, type)
```

```
def handle_first_receive (home) :
    # Homes that need energy, no matter what kind of trading policy, take energy from the Message Queue
    if home.gap < 0 :
        # Continue taking until they have enough or there's no energy left
        while home.gap + mq.recv() < 0 and type != type_eof :
            home.gap = home.gap + mq.recv()
        home.gap = home.gap + mq.recv()
        # They take enough so they send back the extra amount that they don't need back to the Message Queue
        if home.gap > 0 :
            mq.send(home.gap)
        # They reach the end of file so they stop taking and send back the eof message to the Queue
        else :
            mq.send(eof)
    # Homes that don't need energy ignore this handle and send back the message that they received
    else :
        mq.send(message)
```

```
def handle_last_receive (home) :
    # After the take, if there's energy left from home with trading policy 3, send them back to their home so that they send sell it to the market, if no then ignore this handle
    If home.trading_policy != 3 :
        Ignore
    else :
        if type == type_trading_policy_3 :
            home.exchange_market = mq.recv()
        else :
            ignore
```

2. Algorithme pour la mise à jour de la température :

```
def update_weather (child_conn, temperature):
    while True :
        change = random.randint # based on the current temperature
        temperature += change
        if temperature >= max :
            temperature -= 6
        elif temperature <= min :
            temperature += 8
        if child_conn.recv() == 'get' :
            child_conn.send(temperature)
        time.sleep(1)
```

3. Algorithme pour le market:

```
loop0 = True
loop1 = False
while True
    while loop0:
        initialize ThreadPool Executor to handle multiples connections from homes
        initialize the connection via socket
        wait till all threads to end
    loop1 = True

    while loop1:
```

```
begin of weather process
update of temperature with process
begin of external process
wait for external process to end and send a signal to announce the event
update the new price.
send theses new changes to homes or end this simulation based on decision of users
```

V. Plan :

1. Plan d'implantation :

On a commencé par réaliser un UML pour facilement visualiser les communications entre les processus et threads. Il comprend les relations entre les processus (parent-fils ou non liés), les communications interprocessus telles que messages queues entre les homes, socket entre market et homes, la structure dans le shared memory et comment l'accéder, les signaux utilisés entre les processus, les pipes entre les processus parent-fils.

Ensuite, pour le code, on a commencé par homes et market. Pour homes, on est allé du Message Queue car c'était indépendant des autres process, en même temps le market a eu ses fonctions de traitement et le serveur socket. Suivant, on implémente la partie de client socket du côté des maisons et assurer la connexion entre client-serveur. Continuer, on complète la section weather et le partage de du valeur de la température avec le Pipe. On intègre la partie des facteurs externes, et en fin quand tout est bien, on met les boucle while pour simuler les jours, parfaire les affichages et tester les cas pour en déduire les coefficients les plus logiques.

2. Tests :

Pendant notre travail, il y a plusieurs tests ont été mis en place :

- Test des informations sur l'énergie après les messages queues. En effet, nous avons des différentes politiques énergétiques pour nos propres maisons. Donc, il faut vérifier, après les échanges d'énergie, les valeurs correctes pour les communiquer avec market.
- Test de connexion via socket entre homes et market (multithreaded). En réalité, on a commencé par tester les connexions avec un seul client pour visualiser comment fonctionnent les sockets et de même temps, gérer l'exécution de notre programme en général. Et à la fin, on a bien testé avec ThreadPool pour pouvoir recevoir les connexions de plusieurs homes.
- Test des coefficients associés à chaque facteur pour avoir des changements du prix cohérents
- Test de la valeur partagé de la température.
- Tests de la terminaison des processus.
- On a aussi des fichiers main (déjà supprimé) et homes_farsighted pour effectuer des tests.

VI. Execution :

Au début du lancement de notre programme, il faut d'abord exécuter market.py avec la commande python3 market.py. Une fois lancée, ce script va attendre les connexions de homes pour réaliser les transactions entre eux. Ensuite, on lance homes.py pour initialiser les homes. Après le lancement de ce programme, les homes vont communiquer entre eux-mêmes via les messages queues pour annoncer leurs politiques d'énergie, demander aux autres s'ils en ont besoin et puis le

faire ou demander le don auprès des autres s'ils sont en pénurie d'énergie. A la fin des échanges entre les homes, ils vont utiliser un socket à envoyer les informations sur l'énergie qu'ils ont et aussi commencer les transactions vente/achat entre les homes et market. Au côté de market, après la réception des valeurs à vendre/acheter des homes, il y aura des changements au niveau du prix et aussi du stockage d'énergie du marché. Et puis, le processus de weather sera lancé pour mettre à jour la température chaque fois qu'on tape 'get'. En basant sur cette température, on va déduire les changements sur la production/consommation chez les homes et aussi le prix d'énergie. Après, le processus des facteurs externes va créer un événement aléatoire qui impacte le prix. Enfin, l'utilisateur va décider s'il veut envoyer ces changements aux homes ou terminer cette simulation en fermant toutes les connexions.

VII. Conclusion et points à améliorer :

1. Conclusion :

A travers ce projet, on a acquis de nouvelles connaissances et aussi révisé celles que l'on a vues pendant les cours. On a la chance de synthétiser les méthodes de communications entre les processus, les multithreads dans un projet. C'est aussi une bonne occasion de réviser et revoir ces cours d'une manière plus pratique. En plus pour nous ce projet est vraiment intéressant, bien que si on avait eu plus de temps on aurait pu faire mieux, car la durée de ce projet est assez courte et nous avons aussi d'autres projets à compléter.

2. Points à améliorer :

- Fermeture des sockets : Après le lancement de cette simulation, le socket reste encore à l'écoute sur l'adresse déclarée au début du script. Quand on décide de terminer cette simulation en tapant Ctrl+C, cette adresse reste encore occupée et cela va lancer l'erreur OSError: [Errno 98] Address already in use. Proposition pour le corriger : Donner le droit à l'utilisateur de choisir le port en ajoutant un argument dans la commande de lancement.
- Budget pour chaque maison et pour market : On a une idée de donner à chaque maison un budget, et elle l'utilisera pour des échanges. Une maison va « mourir » si elle n'a plus d'argent. Le market a aussi un budget, si ce budget est plus ou moins d'un seuil, le modèle s'effondrera également. Néanmoins ces valeurs prennent beaucoup de temps pour déterminer des valeurs cohérentes, par exemple pour que le modèle dure plus de quelques jours.
- On peut aussi rendre l'affichage plus beau en faisant le GUI, ou bien utiliser matplotlib pour générer des graphes à partir des données, de cette façon on peut mieux poursuivre l'évolution du modèle (qu'en voyant les terminaux).