Harry KALFON, Valentin LEMAIRE

Rapport de Projet - PPC

Ce rapport établit une synthèse de notre Projet de PPC en 3TC. Nous n'allons pas rappeler le sujet entièrement mais le survoler. En quelques mots, il s'agit de programmer une simulation d'un Marché de l'énergie en Python, en utilisant les techniques de multi-process et de multi-thread.

I. Description de la conception et de l'architecture

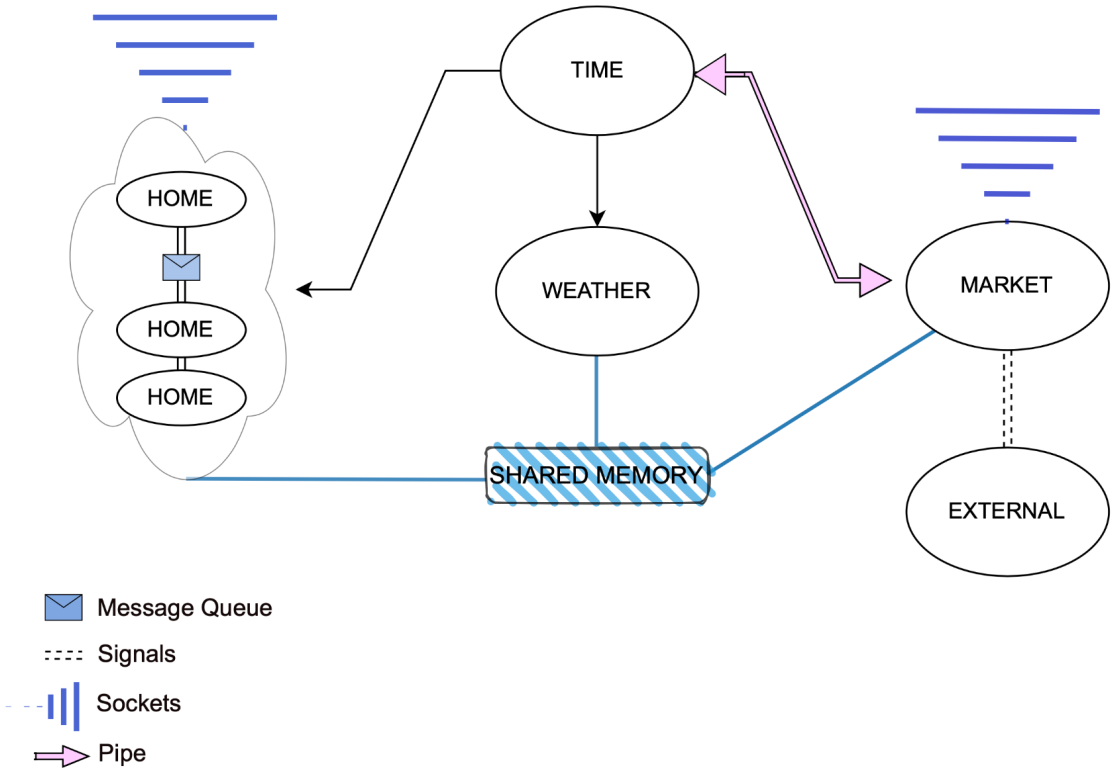


Figure 1 : Représentation schématique de notre solution

Comme indiqué sur la *figure 1*, un premier processus nommé Time gère la création des Maisons, de la Météo et du Marché. Ce processus est chargé de la synchronisation globale du temps. Ce dernier fonctionne en journée. Chaque processus : Maisons, Marché, Météo et celui de gestion des événements, exécutent leurs actions journalières puis attendent le début d'une nouvelle journée. Les prix de l'énergie, les événements et la Météo sont actualisés sur une base journalière.
Nous avons choisi de réaliser une simulation à temps fixe, c'est-à-dire une simulation sur un nombre de jours défini au préalable. Cela nous permet de faire terminer la simulation proprement en "joignant" nos processus à la fin. Le nombre de jours est précisé à la création du processus Time.

II. Justification des choix techniques et algorithmiques

Nos processus sont créés en utilisant des classes. Cela nous permet de bien délimiter la portée de nos variables, et d'éviter que des processus essaient d'accéder à des variables qui ne leur appartiennent pas (ce qui ne poserait pas de problème avec des threads). Dans un choix de consistance, nous avons fait le choix de ne pas utiliser le processus main mais de déléguer au processus Time la création de la simulation.
Pour réaliser la synchronisation de nos différents processus, nous avons fait le choix d'utiliser des barrières (`multiprocessing.Barrier`). Cet objet permet d'attendre un certain nombre de processus en les retenant avant de les relâcher simultanément. Cela correspondait exactement à ce que nous recherchions, notamment pour s'assurer que tous nos processus aient bien terminé leur journée.

Gestion de la Météo

Comme précisé dans le sujet, les informations concernant la Météo sont transmises au travers d'une shared memory. Nous avons fait le choix d'utiliser un Array (`multiprocessing.Array`) initialisé avec 10 valeurs (la plupart sont inutilisées) de type `double`, de manière à pouvoir facilement ajouter des paramètres dans le futur.

Algorithm 1 Temperature determination

```
function updateSeason(self)
    if self.day ≡ 0 mod 10 and self.day ≠ 0 then
        self.season ← (self.season + 1) mod 4
    end if
end function

function calcWeather(self)
    updateSeason()
    pastTemp ← self.sharedMem[0]
    random ← random(0.2, 0.8)
    newTemp ← norm.invCDF(random, self.meanTemp[self.season], 3)
    if newTemp > pastTemp then
        self.sharedMem[0] ← min(newTemp, pastTemp + 3)
    else
        self.sharedMem[0] ← max(newTemp, pastTemp − 3)
    end if
end function
```

L'algorithme de gestion de la météo repose sur deux parties, la détermination de la saison et le calcul de la température en fonction de cette saison.
Nous avons fait le choix de modéliser les températures possibles d'un jour via une loi normale centrée sur la température moyenne de la saison et un écart type de 3. Un nombre aléatoire est tiré et va permettre la détermination de la température du jour grâce à l'inverse de la fonction de répartition. Pour limiter les grands écarts peu réalistes, nous avons limité à ±3°C, l'écart de température entre deux journées consécutives.

Algorithm 2 Market Socket Handle

```
1: function HANDLE_HOME(self, conn)
2:     with conn
3:     data ← conn.recv(1024)
4:     messages ← data.split(,)
5:     if messages[0] = buy then
6:         with self.marketLock
7:             self.dayResults ← self.dayResults − messages[1]
8:         conn.send(messages[1], self.price)
9:         conn.close()
10:    else if messages[0] = sell then
11:        with self.marketLock
12:            self.dayResults ← self.dayResults + messages[1]
13:        conn.close()
14:        conn.send(messages[1], self.price)
15:    else if messages[0] = end then
16:        self.serve ← false
17:        conn.close()
18:    end if
19: end function
```

Lorsque le Marché est prêt, la connexion par socket est établie. Les Maisons achètent ou vendent de l'énergie suivant la quantité qu'elles possèdent, grâce aux sockets dont la description précise sera explicitée ci-dessous. Lorsque les Maisons ont fini, la connexion est terminée.

Algorithm 3 Home transactions

```
1: function SELLTOHOMES(self,consumerMQ,producerMQ,transactionMQ)
2:     while self.energy > 0 and (consumerMQ.nbMessages >
   0 or consumerMQ.sendTime > now() − 1 or consumerMQ.recvTime >
   now() − 1) do
3:         messages ← consumerMQ.recv(block = false, type = 1)
4:         energySent ← min(self.energy, messages[1]
5:         self.energy ← self.energy − energySent
6:         transactionMQ.send(energysent, type = message[0])
7:     end while
8:     producerMQ.recv(type = self.pid)
9: end function
10:
11: function BUYFROMHOMES(self,consumerMQ,producerMQ,transactionMQ
12:     while self.energy < 0 and producerMQ.nbMessages > 0 do
13:         message ← transactionMQ.recv(block = false, type = self.pid)
14:         self.energy ← self.energy + message
15:         if self.energy < 0 then
16:             consumerMQ.send(self.pid + self.energy, type = 1)
17:         end if
18:     end while
19: end function
```

Cet algorithme est le détail du fonctionnement des transactions entre les Maisons. Les Maisons acheteuses qui ont besoin d'énergie s'enregistrent dans la queue des demandeurs avec un message de type 1 et envoie la quantité d'énergie voulue ainsi que leur PID. Les vendeurs eux, s'enregistrent avec leur PID dans la queue des vendeurs et leur quantité d'énergie. Ils envoient aux maisons dans le besoin de l'énergie via la queue de l'échange sur le PID de ces maisons. Les échanges ont lieu tant que les conditions explicitées dans le pseudo-code sont valides.

Algorithm 4 Signal transmission

```
1: function SendExternalEvent(self, eventID, duration, factor)
2:     convertBinary(eventID, duration, factor)
3:     for all eventID, duration, factor do
4:         for bit in Binary do
5:             if bit = 0 then
6:                 sendSignal(SIGUSR1)
7:             else
8:                 sendSignal(SIGUSR2)
9:             end if
10:        end for
11:        sendSignal(SIGALRM)
12:    end for
13: end function
```

Cet algorithme détaille l'envoi des signaux. Il est intéressant de remarquer ici que nous avons utilisé l'encodage en binaire.

III. Description du plan d'implémentation et des tests effectués

L'implémentation des éléments décrits ci-dessus s'est déroulée en plusieurs étapes :

MessageQueue (MQ) entre les Maisons

La première fonctionnalité que nous avons implémentée a été les MQs entre les Maisons. La première itération présentait deux MQs, une pour les Maisons consommatrices, et une utilisée pour les transactions. En examinant les détails des transactions, nous nous sommes aperçus que cette méthode présentait plusieurs défauts : certaines Maisons consommatrices ne recevaient pas d'énergie, et des Maisons productrices restaient bloquées dans la boucle. Pour remédier à ces problèmes, nous avons décidé d'utiliser une troisième MQ, utilisée par les Maisons productrices pour se déclarer. Cette MQ nous a permis de rajouter une condition dans la boucle des Maisons consommatrices, permettant de tester s'il reste des Maisons productrices avec lesquelles échanger (cf).

Signaux entre le Marché et le processus External

Nous nous sommes ensuite intéressés à l'envoi de signaux entre le Marché et le processus External. Nous nous sommes rendus compte que les signaux présentaient une capacité de transport d'information très limitée, ce qui compromettait nos plans de pouvoir définir une infinité d'événements, avec des durées et des impacts variables. De plus, dans une volonté

de ne pas interférer avec les signaux systèmes, nous voulions nous limiter aux signaux `SIGUSR1`, `SIGUSR2` et `SIGALRM`. Cela nous a donc motivé à développer la solution présentée plus haut, à savoir encoder les informations sous un format binaire. Nous les transmettons ensuite en utilisant une combinaison des trois signaux ci-dessus, le signal `SIGALRM` servant de délimiteur entre les différentes informations. En augmentant la probabilité des événements, il a été possible de tester la robustesse de cette méthode en simulant une dizaine de jours et en constatant que le Marché était à même de recevoir et déchiffrer correctement les informations transmises par le processus External.

Socket entre le Marché et les Maisons

La phase d'échange inter-maison étant terminée, les Maisons devaient échanger avec le Marché. Nous avons donc mis en place une socket serveur côté Marché qui accepte les connexions des Maisons. Nous avons fait le choix d'utiliser une socket non bloquante, afin de pouvoir aisément signaler au Marché que la phase d'échange était terminée. La gestion des connexions clients a été multithreadée, ce qui posait potentiellement des problèmes au niveau de l'accès à la variable des résultats du jour. L'utilisation d'un Mutex a permis de résoudre ce problème.

Gestion de la Météo

Pour la Météo, il nous a fallu définir une shared memory entre les différents processus pouvant en avoir besoin, ie les Maisons, le Marché, et la Météo. Pour cela, la shared memory est créée par le parent Time, et est passée aux enfants. Dans une volonté d'un certain réalisme sur les températures, nous voulions que ces dernières ne varient pas trop brutalement et reflètent la saison actuelle dans la simulation. Nous avons pris la décision arbitraire de fixer le nombre de jours par saison à 10. Chaque saison possède une température moyenne utilisée par la suite dans une loi normale, comme décrit plus haut.

Affichage

Afin d'avoir une vision simplifiée sur l'évolution de la simulation et de pouvoir régler les coefficients des divers facteurs, nous avons fait le choix d'afficher le prix, la température, la saison et la demande actuelle sur une interface graphique. Ainsi nous avons pu régler la valeur des coefficients afin d'avoir un Marché plus ou moins stable sans événement extérieur.

IV. Description des problèmes rencontrés

Le premier problème que nous avons rencontré a été la synchronisation entre nos process. Nous ne connaissions pas l'existence des barrières, et l'utilisation de Mutex ou de Sémaphore ne nous permettait pas de remplir nos objectifs. Après un peu de recherche dans la documentation, nous sommes tombés sur les barrières que nous avons utilisées de façon majeure dans tout notre programme.
Nous avons aussi rencontré de nombreuses fois des problèmes avec les sockets, notamment celle coté Marché. En effet, de par notre choix de la fermer et de la rouvrir chaque jour, nous avons dû rajouter une option lors de la déclaration de la socket pour pouvoir la réutiliser.
Nous avons aussi passé du temps sur les MessageQueues (MQ), où nous ne comprenions pas pourquoi les messages avaient les mauvais PID. En réalité, il s'agissait d'anciens

messages qui demeuraient dans les MQ entre deux exécutions. L'ajout d'une fonction pour vider les queues au début du programme et à la fin de chaque journée a permis de résoudre ce problème.
Concernant les PID, nous avons remarqué que la méthode os.getppid() ne retournait pas le bon parent process ID pour le transfert de signaux entre External et le Marché. Nous avons contourné le problème en passant en argument le PID du Marché à la création de External.

Nous avons aussi relevé un comportement assez particulier avec l'affichage. Pour des raisons qui nous sont toujours inconnues à ce jour, la vitesse d'exécution de la simulation est assez aléatoire quand bien même les valeurs d'initialisation sont identiques.

V. Description de l'exécution du programme

Initialisation du programme

Le processus Time est créé. Il va créer les process Weather, Market, et les différentes Homes. Il initialise aussi la fenêtre graphique.

Déroulement d'une journée

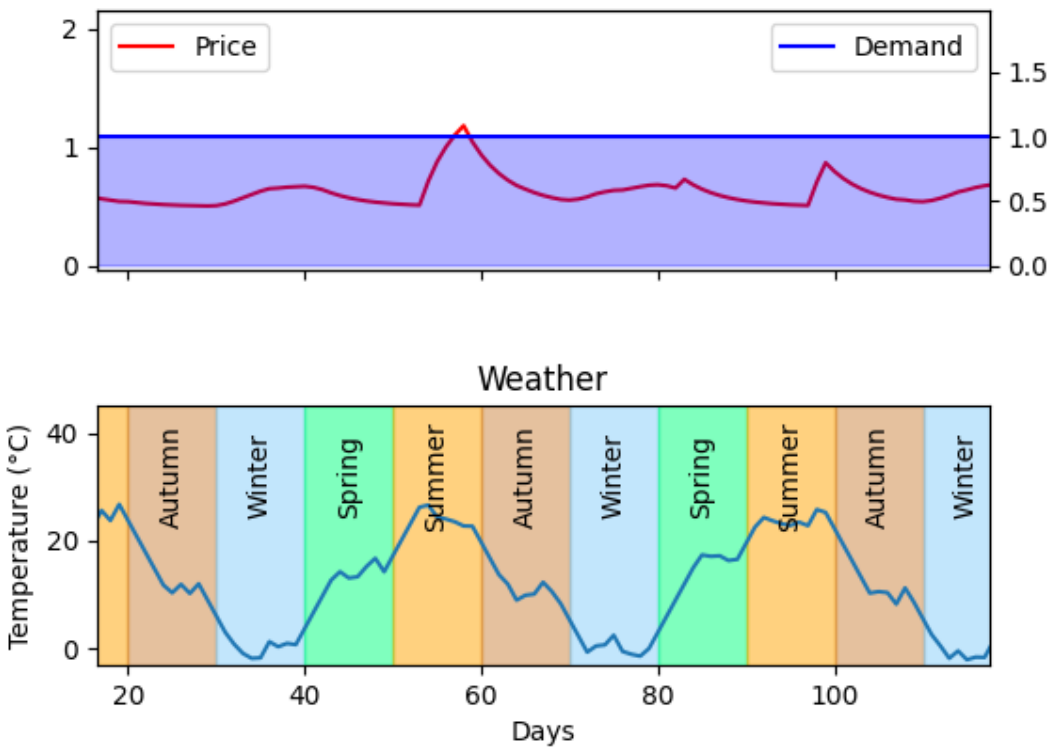
La journée débute par le calcul de la météo. Puis les informations sont envoyées du Marché à la fenêtre graphique pour y être affichées. Après passage de la 3e barrière `start_new_day_barrier`, plusieurs actions se réalisent en parallèle :

- Le Marché reçoit les potentiels événements du jour générés par le processus External, puis effectue des calculs pour déterminer le prix du jour, puis démarre la socket serveur.
- Les Maisons calculent leur énergie du jour, déterminant si elles sont en surplus ou en déficit. Elles commencent à réaliser leurs échanges entre elles selon la procédure décrite plus haut.

Le Marché et les Maisons s'attendent mutuellement pour débuter les échanges via socket. Une fois leurs échanges terminés, les Maisons s'attendent, puis la maison avec l'id 0 signale au Marché la fin des transactions suivant la procédure décrite plus haut. Elle vide aussi les messageQueues. Finalement les Maisons attendent contre la barrière de la nouvelle journée. Le marché calcule la demande pour le jour suivant et met à jour la durée des événements en cours. Enfin il va attendre contre la barrière de la nouvelle journée. Quand tous les processus attendent contre cette barrière, une nouvelle journée démarre.

Terminaison du programme

Les différents processus créés par Time sont join et l'affichage graphique est gardé ouvert tant que l'utilisateur ne le ferme pas, permettant une inspection des données de la simulation.



Graphique 1 : Affichage des résultats lors du fonctionnement de notre programme

Voici l'affichage obtenu suite au fonctionnement de notre programme. Nous pouvons voir la corrélation entre la température liée à la saison et le prix de l'énergie. Les augmentations brusques du prix de l'énergie représenté en rouge correspondent aux impacts des événements externes.

VI. Améliorations possibles

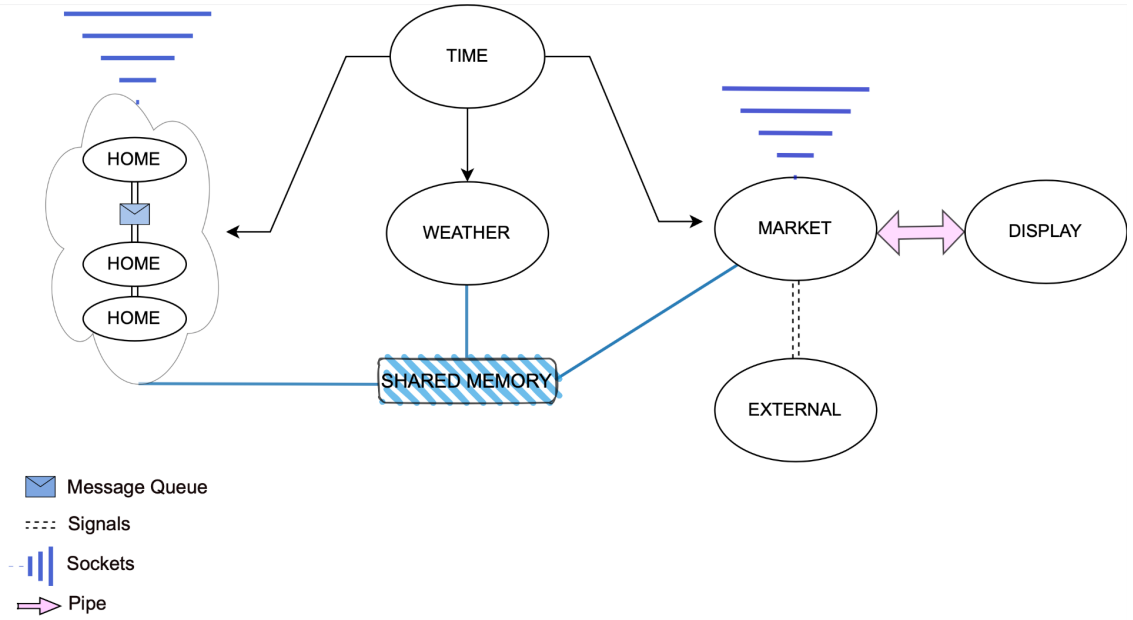


Figure 2 : Représentation schématique évolutive de notre solution

Comme le suggère la figure 2, une amélioration possible du programme consisterait à faire l'affichage graphique dans un processus séparé, communiquant via des pipes avec les autres processus. Cela permettrait de ne pas ralentir la simulation par l'affichage.

Nous voudrions aussi changer l'impact des événements en fonction de leur durée, en suivant une loi normale par exemple, pour éviter un impact "tout ou rien".

Il serait aussi intéressant de trouver la cause des ralentissements de la simulation.