Rapport Projet PPC

The Energy Market

BAGNOLI Eolia - VOINCHET Léna

INSA Lyon, TC, 2022-2023

Sommaire

Introduction

Dans le cadre de notre cours de programmation parallèle et concurrente, nous avons dû réaliser un projet en binôme : créer une simulation du marché de l'énergie. Nous devions créer plusieurs maisons, capables de se vendre et de s'acheter de l'énergie entre elles, ainsi que simuler des événements externes ou internes capables d'altérer les prix du marché. Dans notre code, nous avons choisi d'utiliser la température comme facteur interne, et de la faire varier aléatoirement. Pour les facteurs externes, nous avons sélectionné plusieurs scénarios qui influent tous sur le prix de l'énergie : un ouragan, une guerre, une pénurie de carburant ou encore une apocalypse.

Architecture et protocoles

Au vu du nombre conséquent de méthodes d'échange à gérer pour chaque processus, nous avons décidé de diviser notre code en 5 fichiers :

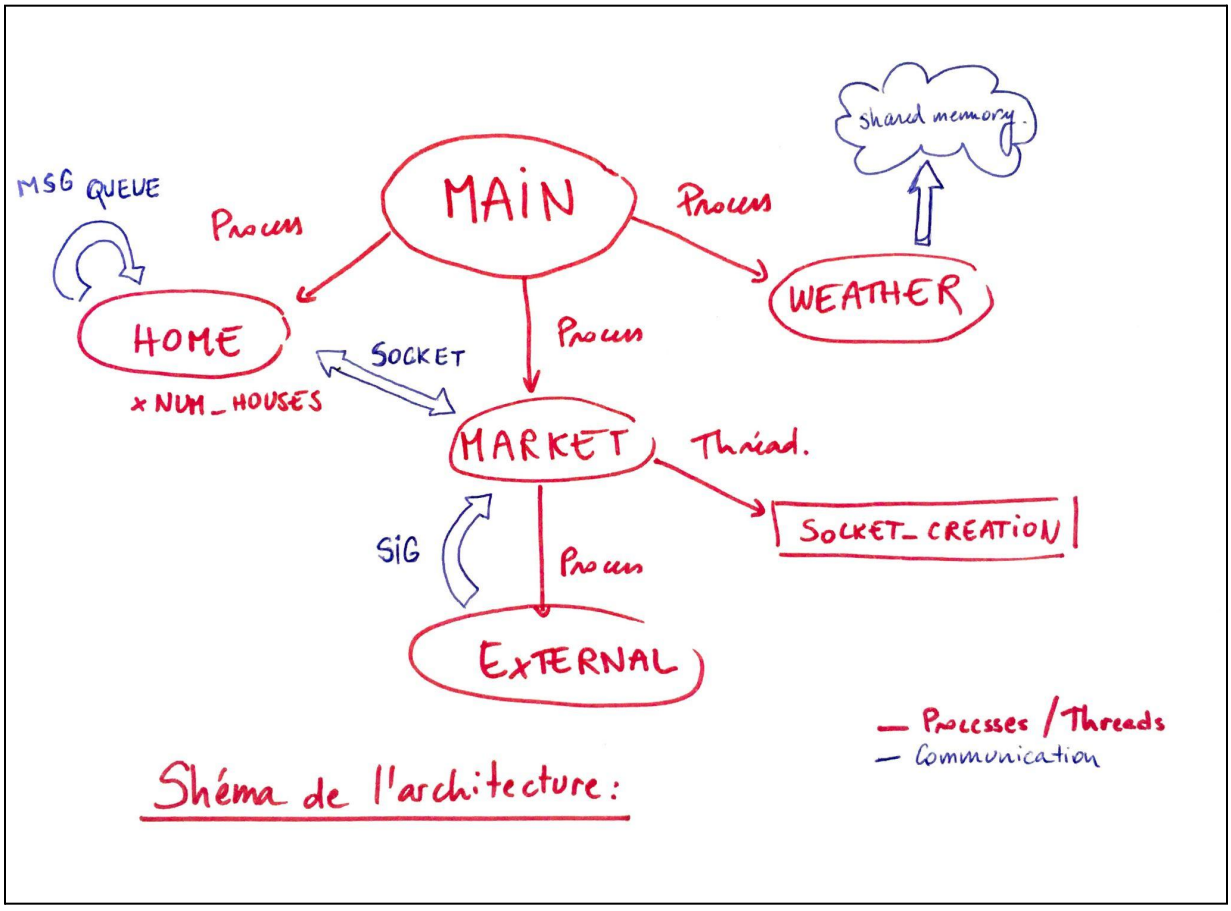
- Un **main.py**, simple, servant uniquement à lancer les processus les uns après les autres, ainsi qu'à mettre la Queue utilisée pour les communications entre les différentes maisons et la mémoire partagée.
- Un **home.py**, qui, en étant lancé comme processus plusieurs fois par le main, crée de multiples maisons, chacune caractérisée par sa politique d'échange, son taux de consommation et de production.
- Un **market.py** qui établit les fluctuations du prix de l'énergie en fonction de toutes les informations auxquelles il a accès (les changements de températures, les facteurs externes et les transactions entre les différentes maisons).
- Un **weather.py** qui envoie dans la mémoire partagée, la température actuelle à chaque seconde.
- Un **external.py** qui envoie des signaux à son processus-parent (Market) pour lui informer de la présence (ou non) d'un événement externe qui va impacter les prix.

Une fois une bonne idée de la structure souhaitée pour notre programme, nous avons pu coder tous les mécanismes d'échanges de données, un par un. Il était alors plus simple de travailler dans des classes différentes, notamment afin de pouvoir diviser notre travail en parties claires et détaillées. Comme c'est un programme avec de nombreux process et thread

différents, parfois imbriqués, nous trouvions important de bien diviser les différentes instances pour ne pas finir avec un énorme bloc de code illisible.

Les process Market et Home communiquent à l'aide d'un socket. A ce sujet, nous avons décidé de lancer la création du socket dans un thread à part de Market. En effet, la création d'une socket est bloquante et nous voulions pouvoir continuer à effectuer des actions (par exemple commencer à calculer le prix de l'énergie) pendant ce laps de temps.

Tous les liens entre les différents process et les communications utilisées sont illustrées sur le schéma ci-dessous :



Conception et choix technologiques

Fonctions External et Weather

- La fonction External, qui est un process child de Market, fonctionne de la manière suivante : tous les intervalles de temps aléatoires (entre 2 et 15 secondes) un élément externe se produit. Il y a quatre évènements différents, avec des probabilités différentes pour chaque. A probabilités égales, il peut y avoir : un ouragan, une panne de gaz ou Poutine qui décide de réenvahir l'Ukraine pour la cinquantième fois. Ensuite, avec 2% de chances de se produire : l'apocalypse, qui provoque un arrêt forcé de la simulation.

- La fonction Weather incrémente ou décrémente aléatoirement de 1 la température de la simulation *(current_temp.value)* chaque seconde. Cette température est ensuite utilisée pour le calcul du prix de l'énergie. La fonction Weather implémente aussi d'autres fonctionnalités comme le calcul du temps de la simulation (voir explication en suite) ou l'affichage des mois.

Terminaison du programme

La simulation peut s'arrêter de deux manières différentes :

1) la manière "naturelle" : au bout du *TIME_MAX* défini dans le main. Dans ce cas, tous les process sortent de leur boucle *while*, terminent puis se *join* dans leur process parent pour terminer le code. La boucle *while* dans laquelle sont placés les process se termine quand la valeur de *current_temp*, qui est dans la mémoire partagée par tous les process, atteint 10 000 (c'est-à-dire que le temps de la simulation se termine). L'avantage d'utiliser *current_temp* plutôt que de créer une nouvelle variable *"temps_simulation"* est qu'on réutilise une variable qui est déjà dans la mémoire partagée : cela simplifie le code et évite de passer une énorme quantité de variables dans les appels de fonctions. C'est la fonction *weather* qui est chargée de compter le temps de la simulation et de passer *current_temp* à 10 000 quand le temps se termine, puisque c'est elle qui est chargée de modifier *current_temp* à la base.

2) la manière "forcée" : afin de pouvoir stopper la simulation avant le temps imparti sans faire n'importe quoi avec les process, le code implémente un système de handlers qui s'assure que tous les process se *kill* bien proprement à l'appel d'un *SIGINT*. Le handler principal est placé dans le fichier Market. A l'appel d'un *SIGINT*, le handler regarde les *childs* de Market et *kill* ceux qui sont actifs. Il envoie ensuite un *SIGINT* au process Main, son process parent, puis se *kill*. Le process Main fait la même chose pour *kill* tous les process qui restent. En partant de Market, on

s'assure que External, qui est un child de Market, est tué aussi alors que si on partait du Main il y aurait un risque que Market se tue avant de tuer External. En résumé, on part du niveau de process le plus bas pour remonter et tuer les parents jusqu'à arriver au process principal.

Lancement de la simulation

Il y a deux temps d'arrêts au lancement du code, qui servent à s'assurer que tous les éléments sont bien en place pour la simulation.

Le premier, dans le Main, affiche des points pendant un seconde grâce à la fonction *time.sleep()*. Cela permet d'attendre que le serveur se mette en place avant de lui envoyer des clients. Ensuite, le Main lance ses Homes, qui se connectent une par une au serveur. Pour des soucis de propreté d'affichage, nous avons souhaité attendre que toutes les Homes soient connectés avant de commencer les échanges. C'est à cela que sert la variable *everybody_connected* : quand elle est passée à *True*, les éléments de la simulation se lancent.

Modèle économique

Enfin, nous avons décidé d'opter pour un modèle économique très simple, car nous préférions mettre l'accent sur le code. Nous avions peur que des protocoles économiques trop complexes nuisent à la propreté et à la bonne compréhension du code. Ainsi, nous avons décidé de ne pas donner de porte monnaie aux maisons. Celles-ci n'ayant pas de notion d'argent, la quantité d'énergie qu'elles vendent ou achètent ne dépend que de leurs besoins et les leurs taux de production et de consommation. Le prix de l'énergie affiché dans Market sert donc seulement comme un indicateur du bon fonctionnement de la simulation. Même si ce choix peut paraître étrange, nous l'avons pris car les maisons n'ont pas de moyens de gagner de l'argent, ce qui impliquerait que lorsque leur porte-monnaie est vide elles arrêtent simplement de participer à la simulation, et ce n'est pas ce que l'on recherche dans ce projet...

Détails de l'exécution

Nous allons, dans cette partie, détailler ce qu'il se passe quand on exécute notre code. Nous étudierons ensuite plus précisément certaines fonctions du programme par le biais de pseudo-code.

Après les différents imports et déclarations de variables dans le main, on demande à l'utilisateur de choisir l'état du boolean *full_simulation*. Si celui-ci est réglé à *True*, alors l'ensemble des transactions sera affiché sur le terminal. Dans le cas inverse, seul l'évolution du prix de l'énergie est *print* à chaque tour. Ensuite, on lance le process Market chargé du *socket* de communication avec les Home, le process Weather, puis on entre dans la boucle *for* chargée de créer l'ensemble des Home.

Nous allons nous pencher maintenant sur le fichier **home.py.** Quand un process Home est lancé, on voit que sa production et sa consommation d'énergie, ainsi que sa politique d'échange sont définies de manière aléatoire. Vient ensuite la connexion au *socket*, puis on va s'assurer que le serveur a bien reçu les requêtes de l'ensemble des Home lancées dans le Main (par le biais du *boolean everybody_connected*), avant de lancer la fonction *energy_gestion*.

Cette fonction (*energy_gestion)* est en quelque sorte le cœur de la simulation. En fonction de leurs *trade_policy*, chaque Home va avoir un comportement différent. Si une Home a trop d'énergie et qu'elle veut vendre alors elle envoie le message *'SELL'* au Market via le *socket*. Si sa politique est "toujours donner" elle met son surplus dans *selling_queue.* Si la maison a comme politique économique la no 3, elle met son surplus dans la selling_queue et attend 2 secondes. Si au bout de deux secondes personne n'a pris cette énergie, elle la récupère. On envoie des arrays : [id de la maison, quantité d'énergie] afin que les maisons puissent retrouver "leur" énergie dans la queue. Dans le cas où une Home est en manque d'énergie, elle vient vider la *queue* et si cela ne suffit pas, elle envoie le message *'BUY'* au Market.

Du côté du fichier **market.py**, il y a 5 fonctions qui permettent de faire fonctionner la simulation. La fonction *socket_creation* se charge uniquement de créer et maintenir le *socket* et la fonction Market quant à elle, est utilisée pour lancer ce dernier, lancer External et la fonction de calcul de l'énergie tant que la simulation tourne. Une autre fonction de cette partie du programme est *home_interaction* qui est chargé de décoder les informations envoyées sur le *socket* par Home et de modifier en fonction la variable *internal_event* servant à calculer le prix dans la fonction précédente (*True* si on achète, *False* si on vend). Enfin, la fonction *new_price* calcule, quand elle est appelée, le nouveau prix de l'énergie.

Pour finir la fonction *Weather* vient, à intervalle de temps régulier, changer la température en fonction d'une variable aléatoire.

Pseudo-codes des fonctions principales

- Fonction du calcul du prix dans la classe market.py

```
// Initialiser tous les coefficients
atenuation_coeff = 0.99
modulating_coeff_int = 0.001
modulating_coeff_ext = 0.01
modulating_coeff_buy = 0.2
modulating_coeff_sell = 0.1

// Initialiser les booléens pour le présence
// d'événements externes ou internes
global internal_event = false
global external_event = false
```

```
// Algorithme de calcul
SI il y a un événement EXT :
    SI il y a un événement INT :
        prix = prix*atenuation_coeff + modulating_coeff_int*current_temp.value
             + modulating_coeff_ext + modulating_coeff_buy
    SINON :
        prix = prix*atenuation_coeff + modulating_coeff_int*current_temp.value
             + modulating_coeff_ext - modulating_coeff_sell
SINON :
    SI il y a un événement INT :
        prix = prix*atenuation_coeff + modulating_coeff_int*current_temp.value
             + modulating_coeff_buy
    SINON :
        prix = prix*atenuation_coeff + modulating_coeff_int*current_temp.value
             - modulating_coeff_sell

SI prix<0.1 :
    prix = 0.1
```

- Gestion des achats et des ventes par la classe home.py

```
SI energy >= energy_needed :
    SI trade_policy 1 :
        La Home met toute son énergie en trop sur la queue
    SINON SI trade_policy 2 :
        La Home vend toute son énergie en trop via la socket
    SINON SI trade_policy 3 :
        La Home met toute son énergie en trop sur la queue
        Attente de 2 secondes
        SI son énergie n'a pas été prise :
            Elle la récupère
            Elle la revend sur la socket

SI energy < energy_needed :
    SI la queue n'est pas vide :
        energy = energy + ce qu'il y a en dernier dans la queue
    SINON :
        energy = energy + energy_needed
        La Home prévient de son achat sur la socket
```

Plan d'implantation

Pour créer ce projet, nous avons commencé par réfléchir sur la forme que nous voulions lui donner, combien de fichiers nous voulions créer et leur contenu. Nous avons aussi réfléchi aux protocoles de communication et au plan économique à appliquer. Une fois que nous avions l'idée générale, nous avons commencé à coder.

Nous avons commencé par créer nos fichiers, les lier, et créer les process. Nous avons ensuite mis en place les fonctionnements les plus simples de chaque process, comme la fonction d'incrémentation/décrémentation de la température, la création d'un évènement externe aléatoire etc.

Nous avons ensuite créé nos protocoles de communication et vérifié leur fonctionnement à l'aide de messages simples. Nous sommes allés du plus simple au plus complexe : commencé par la mémoire partagée et son accès par tous les différents process, continué avec la *message queue*, puis les signaux et enfin le *socket* entre Market et Home.

A ce stade là, nous avons rencontré un problème car nous avions mis la fonction "*home_interaction*" du Market dans une boucle infinie. Lorsque nous faisions une interruption clavier (ctrl-C), le process Market s'arrêtait mais pas son thread qui continuait de tourner en boucle. Nous étions obligées de faire des *kill* à partir d'un autre terminal et des PID toujours en marche. C'est là que nous avons eu l'idée de faire des *handlers* et que nous avons implémenté l'arrêt brutal propre.

A ce stade là, nous avions donc une simulation fonctionnelle, avec une communication inter-process terminée. Tout ce qui restait à faire tenait de l'implémentation du modèle économique et de l'affichage.

Nous avons soigné l'affichage pour le rendre plus agréable, ajouté le choix d'avoir une simulation détaillée ou non et amélioré le calcul de l'énergie pour le rendre plus complet.

Aspects à améliorer

Avec un temps plus long consacré à la réalisation de ce projet, il y a plusieurs éléments sur lesquels nous aurions pu travailler afin de le perfectionner.

Tout d'abord, l'aspect le moins abouti selon nous dans ce programme est le plan économique. En effet, les prix alloués au kilowatt-heure ont par exemple été choisis arbitrairement, uniquement pour tester les fonctionnalités de notre programme. Nous ne nous sommes pas vraiment penchées sur cette partie du projet, qui pour nous était secondaire. Les quantités d'énergies échangées entre les maisons sont elles aussi arbitraires, elles ont chacune le même "nombre" d'énergie au début et peuvent en vendre ou en acheter comme si l'énergie était une variable discrète dénombrable. De plus, les événements externes qui se produisent de manière aléatoire influent tous sur le marché de la même manière, sans distinction, car ils se contentent d'augmenter le prix de l'énergie avec un coefficient fixé, lui aussi, de manière arbitraire. Au vu des tests que nous avons réalisés, nous avons d'ailleurs constaté que les valeurs de nos coefficients pourraient être ajustées. Effectivement, nous voyons par exemple un changement de prix lors d'événements extérieurs, mais il est assez superficiel.

Ensuite, l'autre partie que nous aurions souhaité pouvoir développer davantage est l'aspect graphique. Nous avons réussi à proposer deux affichages lors de l'exécution de notre programme, un très lourd et un bien plus léger (et tout de même assez complet) qui permettent selon nous de comprendre les informations nécessaires et de vérifier le fonctionnement du code. Cela dit, nous aurions aimé pouvoir développer suffisamment le code pour avoir une interface graphique plus détaillée. Nous avions par exemple envisagé l'affichage de courbes qui

s'actualiseraient en temps réel afin de montrer l'évolution du prix du marché. Elles auraient également permis une lecture bien plus lisible de l'impact des événements externes, ou de la température.

Conclusion

Nous avons bien aimé travailler sur ce projet. La simulation était une façon intéressante de travailler sur la programmation parallèle et de mettre en commun tout ce que nous avions vu en TD. Comme nous l'avons dit plus haut, le seul bémol est que nous n'avons pas trop aimé le côté économie du projet, mais c'est une question de goûts personnels. Nous avons aussi trouvé pertinent de faire le projet en python, car c'est un langage que nous avions appris à utiliser mais jamais vraiment mis en pratique. Cela nous a permis de renforcer nos acquis. Au final, nous sommes plutôt contentes du résultat que nous avons rendu, même si il aurait pu être amélioré sur plusieurs aspects, surtout compte tenu du temps qui nous était imparti.