Yueran ZHAO
Julie NOBLET-REVERBEL

Projet Programmation Parallèle et Concurrente
3TC - PPC INSA LYON

1. The code execution

This is everything you need to know about our code :

How to run the application :

- Download all the files in the same repository
- Install matplotlib and sysv-ipc libraries using “python3 -m pip install matplotlib --user” and “pip install sysv-ipc” in the terminal
- Then type “python3 Main.py” in a terminal opened in the directory of the files to start the simulation

The different classes of the program :

Homes : Each process has a unique id and randomly takes a trade policy (0 : Always sell, 1 : Always give, 2 : Sell if no takers). At the time of their creation and each day, the homes are assigned two random ints that define their production and consumption of energy, which vary according to the temperature of the day. The offer represents the difference between these two values : if it's negative, the home will be looking for a way to gain energy, if it is positive, the home will get rid of it according to its trade policy.

Weather : It's the first process to start each day, it fills a shared memory with today's temperature

External : This process is started by the Market, it sends signals to his parent (the Market) if an event occurs

Market : The energy price varies according to the events, and energy asks and sells from the Home classes. The energy price starts at 17 cents/kWatt. This process starts threads which take care of the transactions from the Homes to the Market.

Main : This process starts the other processes. Here you can change all the values of the variables and constants used in the program.

2. The project preparation

The goal of this programming project is to design and implement a *multi-process* and *multithread* simulation in Python with multiple ways of communication between them. The

program simulates an energy market where energy-producing and consuming homes, weather conditions and random events contribute to the evolution of energy price overtime :

- Homes can give away their surplus energy, sell it to or buy it from the market.

- If the temperature is very high or low, we assume that the Home needs to be heated up or cooled down, increasing the energy consumption of the Home

- The prices can go up because of homes asking for energy and can go down thanks to the sale of energy from the houses and a constant coefficient associated with the price. We chose this coefficient to be 0.92 to have the best simulation (the most realistic one).

- Other events, such as laws or fuel shortage, can impact energy prices.

The design of the project will contain four main **processes** :

- **Homes** : Energy producing and consuming home with initial rates and production as well as a specific trade policy (*Always sell, only give away or sell if no takers*).
- **Market** : Current energy price which will evolve each day according to the events happening this day. The market process is multi-threaded and receives energy selling and buying from homes in separate threads. There will be a limit on the number of simultaneous transactions.
- **Weather** : Simulation of weather temperature which will impact energy consumption.
- **External** : Simulation of external parameters impacting the market, such as hurricanes or fuel shortage.

The processes will communicate with each other in different ways :

- *Message queues* will be used by Homes to communicate with each other and exchange freely their energy before selling it to or buying it from the Market with *sockets*.
- There will be a *shared memory* updated by Weather with the temperature, and read by the Homes.
- External process, child of the Market process, will send *signal* events to its parent, each signal corresponding to an event.

Test that had been made :

1. Test the communication between the Homes and Market using MessageQueue. Verify that energy can be sold or bought according to the trade policy of the Homes and the energy price in the Market.

2. Test the communication between the Weather and Homes using shared memory. Verify that the temperature affects the energy consumption and production of the Homes.

3. Test the communication between the External and Market using signals. Verify that external events impact the energy price in the Market with the right proportion.

4. Test the multithreading capability of the Market. Verify that there is a limit on the number of simultaneous transactions and that each transaction is processed correctly.

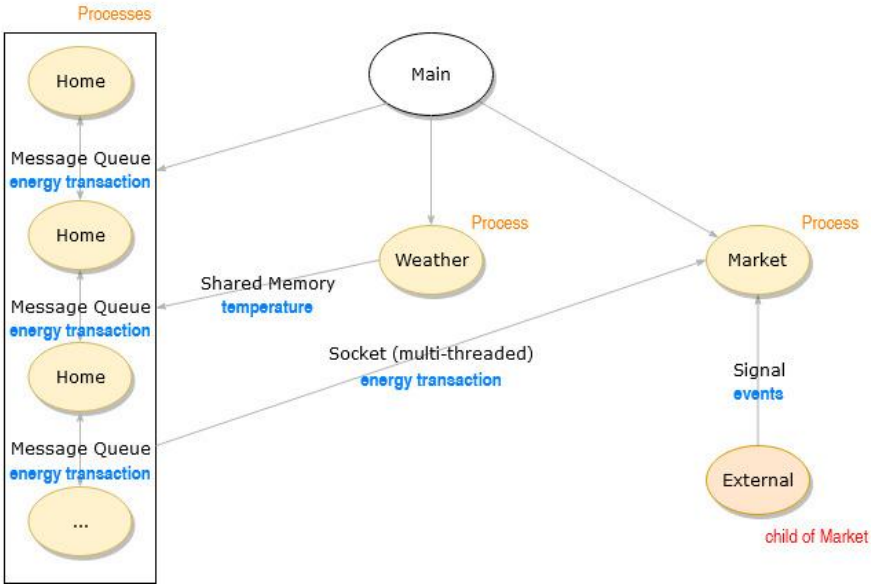
5. Test the overall simulation by running the program for a specified period of time and verifying that the energy price evolves correctly according to the actions of the Homes, the weather, and external events.

6. Test the user inputs in the Main process. Verify that the user can change all the values of the variables and constants used in the program.

For each test, we made a very simple program that was made just to test this functionality. For example, we first made the Home class, then a communication with messageQueue between two simple classes and a socket communication between two other simple programs and then we managed to combine them all.

3. The Solution

The purpose of this section is to explain our solution for the project and to present a development plan. The following graph shows how we see the relations between our processes and threads.



Main : Main process, controls most of the processes : the Market, the Weather and the Homes.

Market : Market is a process and multithreaded class. It calculates the price of the energy according to the values of transactions it receives from Home (with sockets, manage the

connections with a pool of threads) and the generated events from its child process External (with signals).

The Market's attribute will be :

- Price: initialized with 0.17 in the Main process, updated with energy transactions and external events
- Events: represented in an array, another array represents the probability for each event to happen, another one it actualized each day to say which event happened every day

Home : each Home will be a separate process representing a single house, consuming and producing electricity. When they need energy, the Homes will first take free energy from the Homes that give away the energy they produced and will then ask energy to the Market, which will increase the price of the energy. If there is energy left, depending on their trade policy, it will be throw away or sell to the Market (that will decrease the price of the energy)

The Home's attributes will be :

- Production: initialized with a random int between 40 and 80. It represents the quantity of energy produced by the home.
- Consumption: initialized with a random int between 6 and 60. It represents the quantity of energy consumed and it can evolve according to the temperature.
- Trade policy : initialized with a random int between 1 and 3. It represents the strategy of the house in case of energy surplus.

Weather : this class will be launched as an independent process but will remain connected to the Home by a shared memory.

The Weather's attribute will be :

- Temperature : initialized with a random int between 0 and 30. Its variation is a random int between -6 and 6.

External : this class will be launched as a child process of Market and will communicate to it by signals.

The External's attribute will be :

- Events : initialized with int corresponding to different events, gave by the Market

4. Algorithms

Main
Int num_homes
Start :

- WeatherSimulator (1 process)
- Market (1 process)
- Home (num_homes processes)

Join them

Market

```
Set signals = [1,2,3,4]
For each day
   Set nb_conn, sell, buy = 0
   Open a server socket
        Add clients to a ThreadPoolExecutor limited to 4
        If nb_conn < num_homes
                In a separate thread, receive message send by socket :
                        If val_received > 0
                                sell = sell + val_received
                        Else if val_received < 0
                                buy = buy + val_received
                Close connection
        Catch Signals (from External)
        price = formula price (depends on External Signals, sell and buy)
        If price > 1
                price = 1
        If price < 0.1
                price = 0.1
        Print (price, sell, buy, diagram)
        Wait for the other processes to finish their day (Barrier)
```

Home

```
For each day
        production = int random entre 20 et 80
        consommation = int random entre 6 et 60
        Recuperate temperature from shared_memory (shared with Weather)
        if temperature < 10 or > 38
                consommation = consommation + random(0,25)
        else if consomation > 50
                consommation = consommation - random.(0, 25)
        offer = production - consommation
        If offer > 0
                Put offer in MessageQueue_offer
        Else
                Put offer in MessageQueue_demand
        If offer > 0 and trade_policy = 1 or 3
                Recuperate its energy offer from MessageQueue_offer
                Take an energy demand from MessageQueue_demand
                left = demand - offer
                If left < 0
                        offer = left
                        Put offer in MessageQueue_offer
                Else
                        Put left in MessageQueue_demand
        Wait for all the homes to finish their energy exchanges (Barrier)
```

Send the offers and demands left in the MessageQueues to Market with a socket connexion
Wait for the other processes to finish their day (Barrier)

Weather
Initialize temperature_variation
Each each day
Add to temperature a random number between -temperature_variation and temperature_variation
If temperature > 40
temperature = 40
Else if temperature < -15
temperature = -15
Add temperature in the shared_memory (shared with Home)
Print (temperature)
Wait for other processes to finish their day (Barrier)

External
Set proba = [0.95, 0.75, 0.6, 0.5]
Receive signals from Market
Define an “External” class that inherits from "Process" with arguments nb_days and signals
For each event
If random(0,1) > proba[event]
Send signals[event]
Kill itself

5. conception/technos

Class:
We choose to write our program with different classes, not functions. It was easier that way to make them inherit from the Process class from the multiprocessing module and test them independently. In the class, we had to write several methods including the “run” methode.

- The classes can be reused in different parts of a program, which makes it easier to maintain and modify.
- The classes can inherit properties and behaviors from other classes, which reduces the amount of code needed to be written and provides a way to create new classes that are similar to existing ones.

Barrier:
To synchronize the different processes, we used barriers. Using a barrier in this code allows the process to wait for all the homes to finish exchanging their energy before sending their energy request or sell to the Market, and for the Market to finish receiving all the informations from the Homes before proceeding to the next day. This ensures that the Market has accurate informations on the demand and ask of energy for each day and can adjust the price accordingly. It also ensures that the simulation is synchronized and that the day and price changes occur at the same time for all the Homes and the Market.

Lock:
Locks are used in our code to ensure synchronization and mutual exclusion. In other words, the lock is used to prevent multiple processes from accessing and modifying a shared resource at the same time. By using locks, the code can ensure that only one process at a time can access the MessageQueue_demand, preventing infinite loop and ensuring data consistency.

Result presented in graph:
Our result of the price is presented in a real time graph. The advantage of using a graph in this program is that it provides a visual representation of the price fluctuation over time. This makes it easier to understand and analyze the changes in the price. The graph shows the price on the y-axis and the days on the x-axis, allowing the user to see the trends and patterns in the price over time. This information can be useful in predicting future prices and making informed decisions.

6. architecture/protocoles

Socket:
Socket is used to communicate the offer and demand of the Home processes to the Market process. The Market process listens for incoming connections and accepts requests from multiple Homes depending on the number of threads available in the pool, making the other processes waiting for the next thread available. This allows the Market to gather information on the energy demand and sell of the Homes while optimizing the performance of the computer. This enables the Market process to make informed decisions on the energy price for each day. The use of sockets also allows for real-time communication and updates, making the market process dynamic and responsive to the Homes' needs.

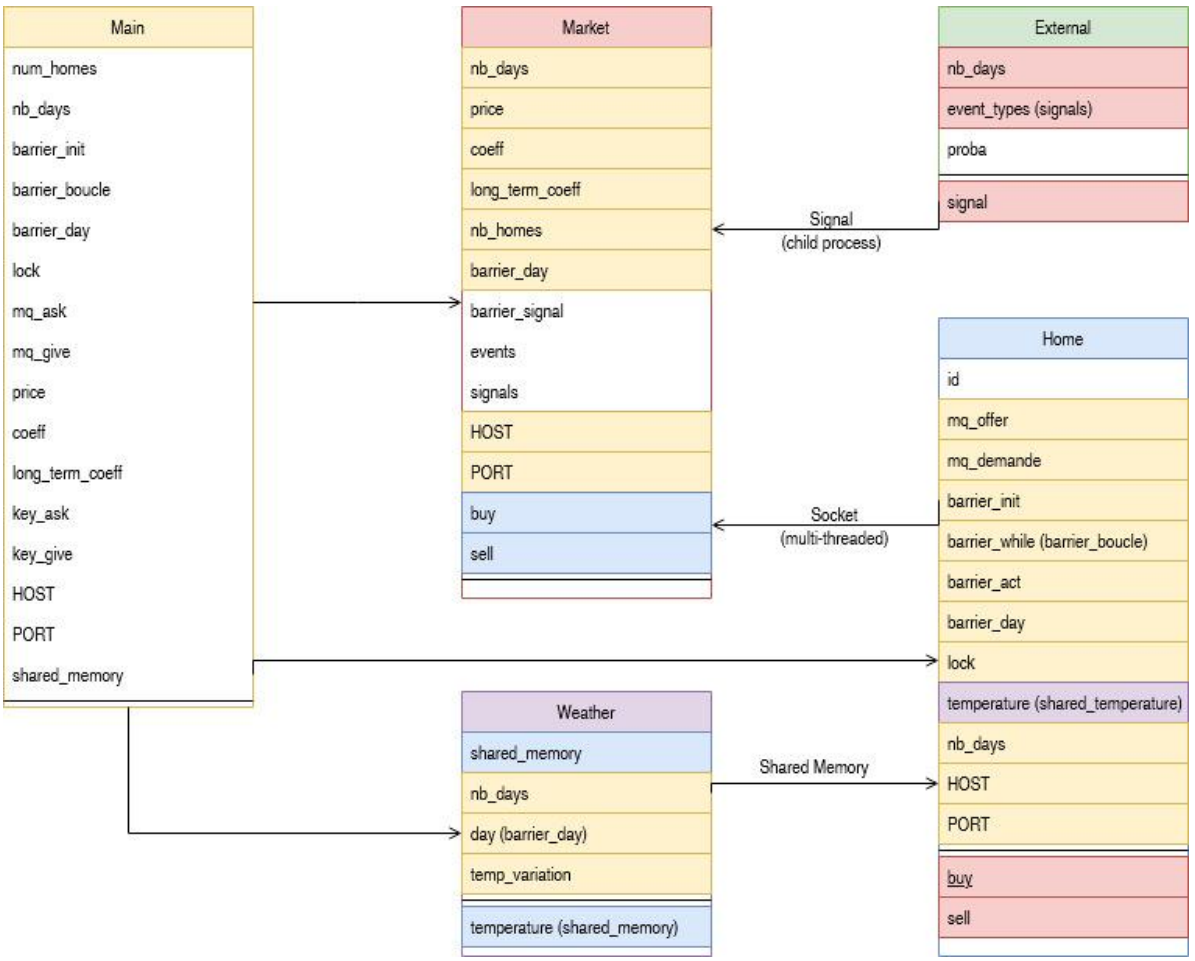
Pool of threads:
Using a ThreadPoolExecutor is better in this program because it allows multiple clients to connect to the server and handle their requests concurrently. This reduces the waiting time for clients and improves the overall performance of the program. The use of the ThreadPoolExecutor also allows the server to handle multiple clients simultaneously, which is more efficient than handling one client at a time.
In our program, the value of max_workers is set to 4, which is an appropriate value for our number of homes. So the executor will create a maximum of 4 threads to handle incoming connections, even if the number of homes (self.nb_homes) is greater than 4. This means that if there are more than 4 homes trying to connect to the server, only the first 4 will be handled by the executor, and the rest will wait until one of the previous threads finishes handling its connection.
Using a smaller number of threads can be better for the execution of a program in several ways. Indeed, having a smaller number of threads reduces the overall resource utilization, including memory usage and CPU cycles, allowing other processes to run efficiently on the same system. Moreover, too many threads can slow down the execution of a program, especially when the number of threads exceeds the number of available cores in the system.

This is because the operating system must context switch between threads, leading to overhead.
Therefore, having a number of threads that is less than the number of homes can optimize the performance, resource utilization, and ease of debugging of the program.

MessageQueue:
The advantage of using message queue in this implementation is that it enables the Home processes to communicate and exchange energy among themselves efficiently. The message queue provides a mechanism for the Homes to send and receive energy offers and demands in a secure and organized manner. This ensures that there is no loss of data during the exchange and that each Home gets the exact amount of energy it needs. Additionally, the lock mechanism used when the Homes empty the MessageQueue_demand, ensures that only one Home can access the message queue at a time. This eliminates the possibility of data corruption and ensures that the energy exchange process is done in an orderly and efficient manner.

Shared memory:
The shared memory allows each home process to access the data of the temperature of the Weather. Each Home process can take into account the current temperature when deciding its energy production and consumption. The temperature been calculated by the Weather process and shared by the shared memory to each Home. By having access to the same temperature data, all processes can make more informed decisions and coordinate their energy usage more efficiently. Additionally, using shared memory improves the overall performance of the code as it eliminates the need for communication between processes using slow interprocess communication mechanisms such as sockets or pipes.

Signal:
When an event occurs in Weather, there is a signal sent from Weather to the Market and then different signals affect the price of energy in different proportions. The signal module is used to capture specific signals that are sent to the program, such as war, snowstorm, law, and fuel shortage, and react to them by changing the market price. This allows the market price to be dynamically adjusted based on real-world events, making the simulation more realistic. The handler function is called when a signal is received, and it updates the market price accordingly. This signal advantage makes the market simulation more interactive and responsive to external factors.



7. Conclusion

We had some troubles making the socket communication between the Homes and the Market processes. Indeed, when we started the execution of the Market, this process was continuously waiting for messages sent by sockets. We had to add the num_conn to calculate the number of numbers received from the Homes to close the socket of the Market and let the Home processes run. Then another issue came. The program was running too fast so the socket address was released fast enough. The Market process wasn't able to reopen the socket connection and the Homes weren't able to connect to the socket connection. We resolved that issue by adding a timer (time.sleep(1)).

During the coding of the signals between External and Market, we first tried to use SIGUSR1 and SIGUSR2 to send 2 types of events. Then we decided to use a list of events to have more types of events. When the Market receives the signals, it finds the type of the signal and changes the price of the energy. We had an issue of infinite running of the External processes. At the beginning, we forgot to kill the child process so the events were still generated after sending the signals and were taken into account the next day (asynchronous

execution). Indeed, we create a new child External each day of simulation, so we can kill it after its execution.

As prospects for improvement, we think that making some events lasting longer could be more realistic. We can also find better coefficients to make the variation of the energy price more realistic. Indeed, the price of energy is often at the maximum (1€/kWh), at the minimum (0.1€/kWh) or too stable, even if there are events during one particular day.
It could also be interesting to have a graph showing the variation of the temperature during the simulation and the transaction amounts.
Making a bank account for each Home could also be an improvement for our program. This would give us more visibility about energy consumption and production and would help making more realistic intervals of production or consumption for each Home.

Overall, it was a very interesting project to discover the different ways of communication between processes.