Compte rendu TD3 - Option 2

Auteurs : Hugo Soleau, Roman Schwalek

Sommaire

Introduction

Afin de mener à bien ce TD de façon pratique, nous utiliserons à travers nos différents scripts les bibliothèques *requests*, *json* et *multiprocessing* afin d'importer nos données et de les stocker dans des espaces de mémoire partagée. Nous emploierons également les modules *time* et *datetime* afin de nous permettre une gestion du rythme des exécutions de nos programmes afin d'éviter entre autres les conflits lors des exécutions. Enfin, nous importerons le module *socket* afin de créer nos serveurs et d'y greffer nos requêtes.

```
# crypto_data_extract.py

import requests
import time
import json
from datetime import datetime, timedelta
from multiprocessing import shared_memory, Lock
...

# crypto_data_server.py

import socket
import json
import time
from multiprocessing import shared_memory, Lock
...

# TD3_serv.py

import multiprocessing
import crypto_data_server
import crypto_data_extract
...

# TD3_cli.py

import socket
import json
```

```
import time
...
```

Note sur l'exécution du programme

Le code fonctionne à l'exécution. Nous donnons ci-dessous un exemple de résultat d'exécution de celui-ci :

```
# Résultat serveur

>python TD3_serv.py
Ecoute serveur sur localhost:6789
Connecté à l'adresse : ('127.0.0.1', 50529)
Requête reçue: {'pair': 'ETHUSDT'}
```

```
# Résultat client

>python TD3_cli.py

Données reçues: [[1700567100000, "2004.80000000", "2006.10000000",
"2003.00000000", "2004.01000000", "1971.47430000", 1700567999999,
"3951678.51274900", 5087, "851.34060000", "1706418.39790500", "0"],
[1700568000000, "2004.00000000", "2008.34000000", "2003.00000000",
"2007.51000000", "2177.16350000", 1700568899999, "4367254.32287000", 5588,
"1015.58210000", "2037475.49439000", "0"], [1700568900000, "2007.51000000",
"2011.80000000", "2006.41000000", "2011.50000000", "1806.80500000", 1700569799999,
"3628369.09528700", 5031, "863.98660000", "1735212.81512900", "0"],
[1700569800000, "2011.50000000", "2012.43000000", "2008.87000000",
"2011.57000000", "1293.00590000", 1700570699999, "2600096.82215500", 4393,
"682.75280000", "1372915.33823700", "0"], ...]
```

Cependant, quelques soucis peuvent parfois se produire tels que l'incapacité à récupérer plusieurs fois d'affilée une paire ou bien un problème de mémoire insuffisante.
Nous n'avons malheureusement pas trouvé de solution au premier problème mais pour le second, il convient d'augmenter la taille de la mémoire partagée.

Partie 1

A. Récupération des données

Dans cet exercice, nous cherchons à récupérer l'ensemble des informations des bougies sur les paires de cryptomonnaies BTC/USDT, ETH/USDT, ADA/USDT, BNB/USDT, XRP/USDT à chaque minute et ce pour les dernières 24h.
Pour ce faire, nous commençons par définir une fonction *get_info* qui à partir d'un intervalle de temps (durée de la bougie e.g. nous choisissons 15min - afin de réduire le nombre de bougies importées) et d'une paire (sous forme d'un string) récupère les informations disponibles à l'adresse suivante : https://api.binance.com/api/v3/klines.

```
# crypto_data_extract.py

def get_info(pair, interval, limit=1000):
    """ Récupère les données OHLCV pour une paire de cryptomonnaies. """
    url = f"https://api.binance.com/api/v3/klines"
    params = {
        'symbol': pair,
        'interval': interval,
        'limit': limit
    }
    response = requests.get(url, params=params)
    if response.status_code == 200:
        return response.json()
    else:
        return None
```

Nous avons fixé arbitrairement le paramètre "limit" à 1000 pour accélerer nos tests mais laissons à l'utilisateur la possibilité de le changer en entrée de la fonction *get_info* s'il a besoin de davantage d'informations.
Nous avons également ajouté une condition permettant selon le code statut de distinguer la sortie de notre requête d'extraction afin de gérer d'éventuelles erreurs pouvant survenir lors de celle-ci et ainsi éviter l'arrêt de notre serveur lorsqu'aucune information n'est obtenue.

Nb : L'url donnée nous retournant toutes les informations de notre bougie, il nous suffit ensuite pour récupérer les informations à chaque minute d'exécuter la commande suivante :

```
# crypto_data_extract.py

pairs = ["BTCUSDT", "ETHUSDT", "ADAUSDT", "BNBUSDT", "XRPUSDT"]
interval = '15m'
unit = convert_time_unit(interval[-1:])
nb_data = int(24*unit/int(interval[:-1])) # Nombre de bougies dans les 24
dernières heures

while True :
    market_data = {}
    for pair in pairs :
        data = get_info(pair, interval)
        if data:
                    market_data[pair] = data[-nb_data:]  # Stocke les entrées des
dernières 24h
                else:
                    print(f"Impossible de récupérer les données de la paire
{pair}")
    time.sleep(60)
```

On précise que la fonction *convert_time_unit* donne la conversion par rapport à l'heure de l'unité de l'intervalle comme suit :

```
# crypto_data_extract.py

def convert_time_unit(s):
    return (s=='s')*3600 + (s=='m')*60 + (s=='h') + (s=='d')/24 + (s=='w')/(7*24)
+ (s=='M')/(30.5*24)
```

B. Stockage dans la mémoire partagée

Afin de permettre l'accès à un utilisateur local de nos données extraites, on exploite un segment de donnée partagée afin de les y stocker sur les 24 dernières heures.
Pour ceci, nous commençons par définir la fonction *update_shared_memory* permettant de mettre à jour le segment de mémoire "shm" à partir des nouvelles données "new_data".

```
# crypto_data_extract.py

def update_shared_memory(shm, new_data, lock):
    with lock:
        # Lit les données actuelles si possible sinon initialise à vide en cas
d'échec
        try:
            current_data = json.loads(shm.buf.tobytes().split(b'\0', 1)[0])
        except json.JSONDecodeError:
            current_data = {}

        # Met à jour et filtre les données pour chaque paire
        for pair, klines in new_data.items():
            if pair in current_data:
                # Concatène les anciennes et nouvelles données
                merged_data = current_data[pair] + klines
                # Elimine les données au-delà des dernières 24h
                cutoff_time = datetime.now() - timedelta(days=1)
                filtered_data = [kline for kline in merged_data if
datetime.fromtimestamp(kline[0]/1000) > cutoff_time]
                current_data[pair] = filtered_data
            else:
                current_data[pair] = klines

        # Écriture des données mises à jour dans la mémoire partagée
        serialized_data = json.dumps(current_data).encode('utf-8')
        shm.buf[:len(serialized_data)] = serialized_data
        shm.buf[len(serialized_data)] = 0  # Marqueur de fin
```

Le verrou "lock" permet de protéger notre processus en ne mélangeant pas leurs actions sur la mémoire partagée et donc de s'assurer que nous réécrivons la mémoire partagée pour le bon processus en début de mise à jour.
Le reste de la fonction se charge d'ajouter les nouvelles données à notre mémoire puis de supprimer ceux au-delà des dernières 24h en prenant comme point de repère la date d'ouverture de nos bougies.

Enfin, après avoir filtré nos données, nous inscrivons nos données dans notre mémoire partagée et nous ajoutons un marqueur à la fin de notre mémoire pour signaler la fin de notre processus d'écriture.

Une fois notre fonction de mise à jour conçue, il ne nous reste qu'à modifier notre protocole principal qui en plus de récupérer nos données, les inscrit dans notre mémoire principale à chaque étape.

```
# crypto_data_extract.py

pairs = ["BTCUSDT", "ETHUSDT", "ADAUSDT", "BNBUSDT", "XRPUSDT"]
interval = '15m'
unit = convert_time_unit(interval[-1:])
nb_data = int(24*unit/int(interval[:-1])) # Nombre de bougies dans les 24
dernières heures
shm = shared_memory.SharedMemory(create=True, size=1024 * 1024)
lock = Lock()

try:
        while True:
            market_data = {}
            for pair in pairs:
                data = get_info(pair, interval)
                if data:
                    market_data[pair] = data[-nb_data:]  # Stocke les entrées des
dernières 24h
                else:
                    print(f"Impossible de récupérer les données de la paire
{pair}")

            update_shared_memory(shm, market_data, lock)
            # Vérifie que la mémoire partagée contient les bonnes données
            # print(read_shared_memory(shm, lock))

            time.sleep(60)
    finally:
        shm.close()
        shm.unlink()
```

Nb : Pour nous aider au débuggage et vérifier la bonne inscription de nos données dans la mémoire, nous avons conçu une fonction *read_shared_memory* qui lit la mémoire partagée.

```
# crypto_data_extract.py

def read_shared_memory(shm, lock):
    with lock:
        data = shm.buf.tobytes().split(b'\0', 1)[0]
    return json.loads(data)
```

Partie 2

A. Mise en place du serveur Socket

Dans cette partie, nous tâchons de concevoir un serveur Socket avec lequel notre client pourra communiquer. Pour cela on renseigne une adresse locale et un port depuis lequel notre client se connectera afin d'effectuer ses requêtes.
Afin de gérer la mutltitude d'erreurs pouvant survenir, on filtre selon le remplissage de notre table d'extraction, le formatage de nos données, etc. Et à chaque cas particulier présentant une anomalie, nous générons une erreur ou un message d'avertissement à minima.

```
# crypto_data_server.py

def run(shm, lock):
    HOST = 'localhost'  # Adresse locale
    PORT = 6789         # Port à utiliser

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.bind((HOST, PORT))
        s.listen()
        print(f"Ecoute serveur sur {HOST}:{PORT}")

        while True:
            try:
                conn, addr = s.accept()
                print(f"Connecté à l'adresse : {addr}")

                with conn:
                    data = conn.recv(1024)
                    if not data:
                        print("Pas de données reçues. Fermeture des connexions.")
                        break

                    try:
                        request = json.loads(data.decode('utf-8'))
                        print(f"Requête reçue: {request}")
                        pair = request.get("pair")
                    except json.JSONDecodeError:
                        response = "Invalid JSON format"
                        conn.sendall(response.encode('utf-8'))
                        print("Objet JSON reçu de format incorrect.")
                        continue

                    market_data = read_shared_memory(shm, lock)
                    if pair and pair in market_data:
                        response = json.dumps(market_data[pair])
                    else:
                        response = f"No data available for {pair}"

                    conn.sendall(response.encode('utf-8'))
                    time.sleep(1)  # Attend avant de fermer la connexion

            except Exception as e:
```

```
                print(f"Error occurred: {e}")
                continue
```

Ensuite, nous élaborons un script client qui via une fonction *request_pair_data* va se connecter à notre serveur puis récupérer les informations de la pair donnée en arguments.

```
# TD3_cli.py

def request_pair_data(pair):
    HOST = 'localhost' # L'adresse du serveur
    PORT = 6789        # Le port utilisé par le serveur
    request_data = json.dumps({"pair": pair})  # Création de la requête JSON

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((HOST, PORT))
        s.sendall(request_data.encode('utf-8'))

        response = s.recv(1024)  # Attendre la réponse
        print(f"Données reçues: {response.decode('utf-8')}")
        time.sleep(1)
```

Il est important de noter que dans nos deux fonctions précédentes, nous forçons la mise en veille temporaire de notre processus afin d'éviter le crash de notre serveur après chaque requête à cause d'une fermeture trop rapide.

Nb : La fonction *read_shared_memory* du script **crypto_data_server.py** fonctionne de la même façon que celle de **crypto_data_extract.py** à ceci prêt que son but étant de permettre la lecture par le client, on gère ici le cas où aucune donnée n'a pu être récupérée.

B. Traitement des requêtes

Finalement, afin de récupérer et traiter nos données, nous exploitons le stockage dans un fichier JSON de nos informations.
Nous procédons alors en créant notre mémoire partagée et le verrou qui identifie le processus. Puis, après avoir récupéré nos données, nous exécutons notre processus en créant notre serveur depuis lequel notre client exécuteras ses requêtes.

```
# TD3_serv.py

shm = multiprocessing.shared_memory.SharedMemory(create=True, size=1024 * 1024)
lock = multiprocessing.Lock()

# Récupère les données dans le processus data_process
data_process = multiprocessing.Process(target=crypto_data_extract.main, args=(shm,
lock))
data_process.start()

# Démarre le serveur socket
```

```
    crypto_data_server.run(shm, lock)

    # Attend que le processus de récupération des données se termine
    data_process.join()
```