Travaux pratiques Systèmes distribués, services web et Blockchain

Format

4 TP de 3h. Chaque TP est noté à la fin de la séance. Vous pouvez travailler en binôme.

Objectif

L'objectif de ces travaux pratiques est de réaliser une application distribuée de surveillance d'un système électrique. L'application permet d'accéder à un système physique pour récupérer des données, de répartir et partager des tâches de calcul entre plusieurs participants, de synthétiser ces calculs et finalement publier les valeurs moyennes de grandeur mesurées dans un blockchain sous forme "sécurisé et traçable".

Nous disposons d'un système électrique avec plusieurs charges sur lequel il faut effectuer des mesures, et les mettre à disposition via un service web.

Séquencement

TP 1: préparation et partage des calculs dans une architecture maître-esclaves.

TP 2 : accès au système électrique pour collecter les données,

TP 3 : mise en œuvre d'un web service

TP4 : publication de données dans un Blockchain simplifié

Démarche

Nous suggérons une démarche en deux étapes, la première s'attache à fournir les traitements nécessaires, la seconde à les distribuer et les rendre robustes. Les étudiants réaliseront donc d'abord les traitements sous forme classique (centralisée, séquentielle) et puis une fois ceux-ci sont corrects, alors ils font leur passage sous une forme distribuée et parallèle/concurrente.

Nous avons volontairement restreint l'aide en se focalisant sur chaque étape sans se préoccuper (trop) de cohérence globale, la valeur ajoutée du travail des étudiants est la conception d'une architecture globale cohérente et l'intégration de ces connaissances techniques pour construire cette solution.

Il n'y a donc pas de solution unique attendue à cet ensemble de TP, en revanche il est de votre ressort de justifier le bien fondé de votre solution et de vos choix.

Architecture suggérée

Utilisation d'un git dédié.

Module « simulation du système »

Il s'agit de fournir un code qui modifie l'état du système électrique (éventuellement à faire de manière concurrente pour gérer l'exclusion mutuelle). L'objectif de ce module est d'avoir des valeurs mesurées qui varient en fonction du temps.

Module d'acquisition des valeurs

Il s'agit de lire les valeurs fournies par un automate de mesure (code fourni) toutes les x secondes et les rendre disponibles (à terme sous forme d'un service web). Les sorties sont donc un quintuplet, date, tension, courant puissance active et puissance réactive.

Ce module d'acquisition publiera les données sous forme de web service.

Module d'analyse

Il récupère les valeurs fournies et sous traite le calcul des valeurs moyennes à 4 sous modules (U, I PA, PR). Chacun de ces sous modules accumule les valeurs reçues et renvoie leur moyenne toutes les x valeurs.

Les sous-modules seront distribués sous forme de threads python et de RPC.

Module de publication

A partir des valeurs moyennes, il s'agit de fournir une suite authentifiée. Ce module devra donc attendre de recevoir les 4 valeurs puis les publier.

Architecture abstraite

- On dispose d'un processus électrique dont l'état varie au fil du temps
- Un processus P effectue des mesures remplit une file avec des 4-uplets (U, I, PA, PR)
- Un processus C lit ces valeurs et en extrait chaque élément qu'il stocke par paquet de x (on a donc 4 tableaux de x valeurs, un pour U, un pour I etc)
- Dès qu'un paquet est prêt, on lance un processus Ai d'analyse avec un paquet, le processus fait ses traitements et renvoie la moyenne
- Le processus C attend que tous les Ai aient fait les analyses et renvoie un 4uplet de ces moyennes
- On authentifie les différentes moyennes et on les publie

TP 1: préparation et partage des calculs dans une architecture maître-esclaves

Dans ce premier TP, vous allez réaliser une version séquentielle et ensuite distribuée de votre application. La version séquentielle réalise le calcul des moyennes séquentiellement et la version distribuée partage le calcul entre plusieurs processus séparés.

Exercice 1: Calcul séquentiel des moyennes de valeurs mesurées

Dans ce premier TP, votre application n'est pas encore connectée au système électrique, mais elle collecte les valeurs des grandeurs mesurées depuis un module Python **acquisition.py** qui gènère des valeurs aléatoires pour chacune de grandeur. Ce module Python contient la liste de grandeurs mesurées avec leur temps d'acquisition et le nombre de valeurs à collecter. A titre d'exemple on mesure 4 valeurs pour chaque grandeur et les intervalles d'acquisition sont les suivantes :

```
MESURES = [
     ("U", 5,4),
     ("I", 5,4),
     ("PA", 6,4),
     ("PR", 6,4),
]
```

La fonction **getMesure** de ce module génère *n* valeurs aléatoires et attend *t* secondes avant de renvoyer le tableau de valeurs.

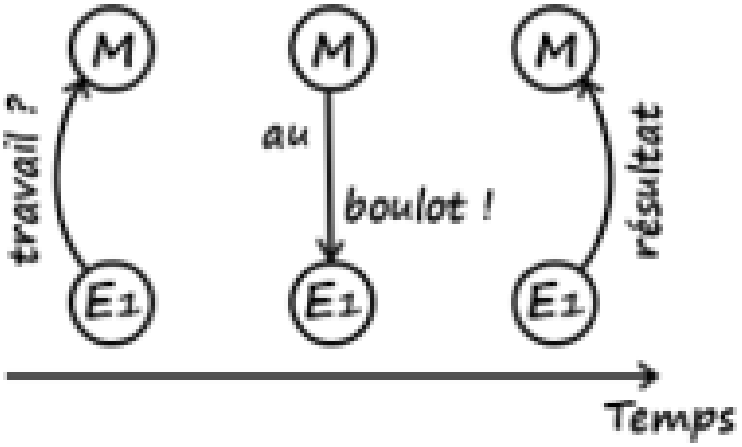
Nous testons un programme séquentiel du calcul des moyennes de chaque tableau de 4 valeurs de la grandeur mesurée. Le programme **seq.py** permet de réaliser ce calcul grâce à la fonction **moyennes_mesures**. Tester ce programme.

Quand on exécute ce programme, on obtient :

```
Collecte de 4 valeurs de PR toutes les (6s)...
Moyenne de PR est 2.99
Collecte de 4 valeurs de PA toutes les (6s)...
Moyenne de PA est 2.06
Collecte de 4 valeurs de U toutes les (5s)...
Moyenne de U est 5.02
Collecte de 4 valeurs de I toutes les (5s)...
Moyenne de I est 0.95
Temps de calcul: 22.0s
```

Un programme simple, mais le temps d'exécution est non négligeable ! Dans la suite, nous allons construire une application distribuée afin de paralléliser les opérations.

L'algorithme décrit précédemment est séquentiel. Tel quel, il ne peut être exécuté que par un seul acteur (une seule unité de calcul). Autant dire que ce dernier aura du pain sur la planche pour un ensemble important de grandeurs mesurées. Pourtant, les opérations sont indépendantes les unes des autres : la moyenne d'une grandeur n'est pas conditionnée par la présence ou l'absence de la moyenne d'une autre grandeur. Rien ne nous empêche donc d'effectuer ces tâches en parallèle. Dans cette partie du TP, nous profitons de cette propriété pour répartir le travail sur plusieurs acteurs (plusieurs calculateurs).



Supposons donc que nous disposons de *m machines* capables de communiquer. Une des machines, que nous noterons M (maître), reçoit la liste des ingrédients. Il lui faut alors distribuer les tâches entre les *m composants* (incluant elle-même). Nous utilisons une stratégie Maître-esclave : (le maître) découpe également le calcul, sauf qu'ici il attend qu'une machine (un esclave) le contacte pour lui donner du travail. On a donc un système à la demande, permettant d'éviter de confier des tâches à une machine en panne. Pour détecter les pannes après distribution du travail, on peut utiliser un délai (timeout). Dans un tel système, le nombre de travailleurs peut évoluer sans problème.

Exercice 1 : le protocole de communication

Nous connaissons nos acteurs et la façon dont ils sont connectés. Mais avec quelle langue communiquent-ils ? Autrement dit, il nous faut définir le protocole réseau utilisé. Dans ce TP, la communication entre le maître et les esclaves se fera par le protocole **RPC**[1].

Définition : RPC

RPC (*Remote Procedure Call*) est un protocole permettant d'appeler depuis une machine une fonction définie sur une autre machine du réseau.

Nous utiliserons la bibliothèque RPyC[2] . Pour qu'une machine client puisse exécuter une fonction *f* sur une machine server, il faut que server crée un service et expose sa méthode f.

Q1. Tester le code du **serveurRPC.py** pour lancer un service RPC.

```python
import rpyc
from rpyc.utils.server import ThreadedServer

class MyService(rpyc.Service):
    def exposed_f(self):
        # Cette méthode sera accessible sur le réseau du fait de
            son préfix "exposed_"
        return 42

    def g(self):
        # Cette méthode ne sera pas accessible sur le réseau
        return 43

def start():
    t = ThreadedServer(MyService, port=18861)
    t.start()

if __name__ == "__main__":
    start()
```

Q2. Dans une nouvelle console Python (Shell Python Pyzo), tester le code d'un client pour appeler la fonction f exposée par le serveur (remplacer *adresse_ip_du_serveur* par *localhost*):

```
>>> import rpyc
>>> conn = rpyc.connect("adresse_ip_du_serveur", 18861)
>>> conn.root.exposed_f()
42
>>> conn.root.f()  # Peut aussi être appelée sans le "exposed_"
42
>>> # Par contre, on n'a pas accès à g
>>> conn.root.g()
...
AttributeError: cannot access 'g'
```

Exercice 2 : les verrous

[1] https://fr.wikipedia.org/wiki/Remote_procedure_call
[2] https://rpyc.readthedocs.io/en/latest/

Pour exposer le service, c'est-à-dire pour le rendre accessible, le serveur a utilisé un *ThreadedServer*. Comme indiqué dans la documentation[3], ce serveur créera un thread (ou « fil » en français) pour chaque client. Pour éviter des conflits d'écriture entre ces threads, nous avons besoin de protéger les variables globales par des verrous (lock).

Q1. Coder dans un fichier python *inc_serveur.py* un simple serveur RPC qui expose une fonction qui incrément *N=1000000* fois une variable globale *i.* Après la boucle, la fonction affiche la valeur de *N – i*

```
import rpyc
from rpyc.utils.server import ThreadedServer
from threading import Lock
i = 0
N = 1000000
#lock = Lock()

class MyService(rpyc.Service):
    def exposed_inc(self):
        global i
        for _ in range(N):
            i = i+1
        print ("N - i = ", N - i)

def start():
    t = ThreadedServer(MyService, port=18861)
    t.start()

if __name__ == "__main__":
    start()
```

Q2. Coder dans un fichier python *inc_client.py* un client RPC qui fait appel à la fonction d'incrémentation (*exposed_inc*).

Q3. Dans un premier terminal (Linux ou Windows), démarrer le serveur. Dans un deuxième terminal, démarrer 3 clients en parallèle. Que constatez-vous ?

```
MacBook-Air-de-lahmadi:Sujet_TP AbdelkaderLahmadi$ python3 inc_client.py & python3 inc_client.py & python3 inc_client.py
[1] 85845
[2] 85846
[1]-  Done                    python3 inc_client.py
[2]+  Done                    python3 inc_client.py
```

Q4. Maintenant, modifier le code de la fonction d'incrémentation (*exposed_inc*) du serveur pour protéger la variable globale ***i*** avec un lock (verrou)[4].

Q5. Tester le nouveau programme du serveur avec 3 clients exécutés en parallèle.

Exercice 3 : Programmer le maître

Dans le cadre de ce TP, nous nous restreindrons à un maître et, disons, trois esclaves.
Utiliser un seul maître nous rend vulnérables en cas de panne, mais nous prenons ce risque au profit de la simplicité de notre architecture. Le nombre d'esclaves n'est pas très important dans le cadre de ce TP (du moment qu'il y en a au moins un et relativement peu pour que le maître ne soit pas surchargé).

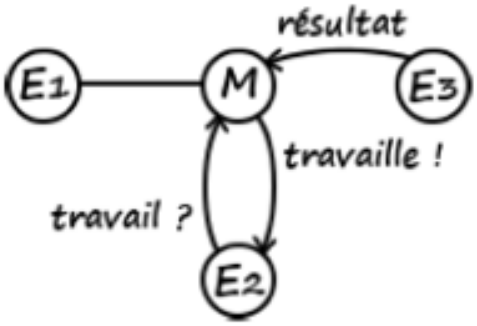
---

[3] https://rpyc.readthedocs.io/en/latest/api/utils_server.html#rpyc.utils.server.ThreadedServer
[4] https://docs.python.org/fr/3/library/threading.html#lock-objects

Dans notre cas, les communications ne se font qu'entre un esclave et le maître (pas entre les esclaves) et seuls les esclaves prennent l'initiative de la communication, soit pour demander du travail ou présenter le fruit de leur labeur.

Il nous faut donc héberger un service sur le maître uniquement, un service exposant deux méthodes :

- *give_task* () : reçoit une demande de travail d'un esclave et y répond
- *receive_result* () : reçoit le résultat d'une tâche effectuée par un esclave (dans ce cas calcul de la moyenne d'une grandeur mesurée).



Le code du maître consistera à créer le service, à le démarrer et à l'arrêter. Pour distribuer les tâches et déterminer quand tous les calculs sont terminés, le maître pourra garder en mémoire les fruits à préparer ainsi que ceux en cours de préparation.

Q1. Créer dans un fichier Python *maitre.py* une classe de service qui expose les deux méthodes *give_task* et *receive_result*.
La méthode *give_task* récupère une tâche depuis la liste de tâches et elle la retourne à l'esclave (utiliser la méthode *pop*[5] pour récupérer une tâche de la liste). Elle sauvegarde également dans une liste *tasks_being_done* la tâche envoyée.
La méthode *receive_result* prend en argument une tâche et le résultat. Elle enlève la tâche de la liste *tasks_being_done*. Elle vérifie s'il n'y pas plus de tâches à faire et que toutes les tâches ont été effectuées. Si c'est le cas, elle affiche « Les moyennes sont prêtes! ». **Un squelette du code de maître est disponible sur Arche.**

Exercice 4 : Programmer l'esclave
Le même code s'exécutera sur tous les esclaves et consistera basiquement en une unique boucle :

```
task = ask_for_task()
while task:
    res = work(task)
    send_results(task, res)
    task = ask_for_task()
```

Q1. Dans un fichier *esclave.py,* coder les méthodes suivantes :

- *create_connection()* : cette méthode initialise la connexion avec le serveur (le maître)

[5] https://docs.python.org/fr/3/tutorial/datastructures.html

- *calcul_moyenne(id,mesure,t,n)* : qui prend comme argument l'identifiant de la tâche (id), le nom de la mesure (mesure), son temps d'acquisition (t), et la taille du tableau de valeurs (n). Elle affiche ces informations, elle appelle la fonction *getMesure* pour récupérer le tableau de valeurs et calculer la moyenne. Elle retourne une chaîne de caractères qui contient le nom de la mesure et sa moyenne !
- *send_result()* : envoie le résultat au maître.
- *ask_task* : demande au maître une tâche.
- *run* : exécute une boucle *while* tant qu'il y a une tâche à exécuter pour calculer une moyenne et envoyer le résultat au maître.

Un squelette du code d'un esclave est disponible sur Arche.

Exercice 5 : Maintenant, on teste !
Pour tester votre programme, on démarre le maître (*maitre.py*) dans un terminal. Ensuite on démarre trois esclaves en parallèle dans un autre terminal.

On observe que le temps de préparation est environ divisé par 3 ! Nous avons désormais une version parallèle du calcul des moyennes de nos grandeurs mesurées .

TP 2 : Accès au système électrique pour collecter les données

Le but de ce TP est de programmer en python la partie acquisition des données à partir du système physique réel, ainsi qu'une partie qui permet de modifier en temps réel la configuration du système physique pour avoir des données qui évoluent dans le temps.

Contexte matériel: internet des énergies

lien : http://internetdesenergies.blog.univ-lorraine.fr/

Le système électrique correspond à un objet physique appelé armoire. Une armoire électrique a été conçue pour simuler la mise en charge d'un réseau électrique afin de permettre de faire des mesures sur les charges résistives (R) Capacitive (RC) et Inductive (RL). Ces charges sont pilotées par un automate de type Wago 750-8202.

Un automate simple est équipé de cartes d'entrées/sorties :

- 750-1500 - output TOR : 16 sorties binaires
- 750-494 - input analog : module de mesures électriques sur 4 entrées sur 3 phases pour chacune tensions intensité 24 mots en tout

Une armoire correspond à 7 charges.
Les armoires utilisées sont dans la salle SAMI située en H4. Une documentation de mise en route est disponible dans la salle et sur la page du cours.

Première partie: prise en main du code fourni

Un ensemble code python est fourni. Le point d'entrée est le fichier ***docArmoirePython.md***

Lisez ce document et ensuite regardez les codes d'exemples fournis que vous pourrez faire fonctionner.
Identifiez les primitives qui serviront pour les parties suivantes

Deuxième partie: modification du système physique

Créer une fonction appelée simulation avec deux paramètres *sleepTime et duration*
qui toutes les sleepTime secondes

- tire un nombre c au hasard entre 0 et 6 (les numéros des charges)
- tire un nombre s au hasard entre 1 et 2 (le nombre de source)
- et modifie la charge c (si elle est on elle passe off et réciproquement) et positionne la source sur s

et s'arrête au bout de *duration* secondes d'exécution.

```
In[4]: simulation (2,30)
modification de la charge 5 mise sur source 2
modification de la charge 4 mise sur source 2
modification de la charge 5 mise sur source 2
modification de la charge 3 mise sur source 2
modification de la charge 7 mise sur source 2
modification de la charge 2 mise sur source 1
modification de la charge 4 mise sur source 2
modification de la charge 5 mise sur source 1
modification de la charge 5 mise sur source 2
```

Amélioration possible:

- *utiliser un temps de sommeil tiré au hasard à chaque itération*
- *utiliser plusieurs fonction en parallèle et gérer les concurrences d'accès à l'armoire*
-

Troisième partie: récupération et mise à disposition des données

Vous avez remarqué que plusieurs mesures du système électrique sont possibles. Nous considérerons par exemple les mesures sur la source n° 1.

Mise en route

Reprendre le problème producteur consommateur et le programmer.
On considère une file d'échange limitée à 10 valeurs implantée dans une classe *Tampon* munie de deux méthodes produire et consommer. La classe tampon gérera les problèmes de synchronisation et d'exclusion mutuelle.
Les classes *Producteurs* et *Consommateurs*, sont des Thread, et auront en paramètre de constructeur une instance de Tampon et le nombre de fois où elles doivent produire/consommer.

Exemple d'exécution avec une file de 4 places, 2 producteurs, 2 consommateurs faisant chacun 5 écritures/lectures.

```
Python Console>>> runfi
on produit 0
on produit 9
on produit 1
on produit 1
on consomme 0
on consomme 9
on consomme 1
on produit 8
fin producteur
on produit 1
on produit 7
on consomme 1
on consomme 8
on consomme 1
on consomme 7
on produit 8
on produit 1
on produit 1
fin producteur
on consomme 8
fin consommateur
on consomme 1
on consomme 1
fin consommateur
```

Mise en oeuvre sur le système physique

Ecrire une fonction python nommée acquisitionMesure qui se connecte sur l'armoire et effectue une mesure sur secteur 2 et la retourne

Créer une classe (Thread) Acquisition paramétrée par une période d'acquisition et une durée éventuelle pendant une durée effectue une mesure de source2 périodiquement (et pour le moment les affiche).
Exemple d'usage avec un processus chargé de la simulation.

```
Python Console
Acquisition de  U=242.43 I=0.10 PA=-16.00 PR=-6.00
Acquisition de  U=242.55 I=0.10 PA=-15.00 PR=-5.00
Acquisition de  U=241.43 I=0.10 PA=-15.00 PR=-11.00
Acquisition de  U=241.03 I=2.26 PA=513.00 PR=-2.00
Acquisition de  U=241.09 I=2.29 PA=552.00 PR=4.00
Acquisition de  U=241.12 I=2.28 PA=551.00 PR=0.00
Acquisition de  U=241.12 I=2.29 PA=552.00 PR=2.00
```

Il s'agit maintenant de regrouper les valeurs mesurées sous la forme de n-uplet. Modifier la classe précédente pour qu'elle regroupe ces valeurs par tableau de n valeurs (et les affiche).

```
Acquisition de  [(242.4399871826172, 0.09999999403953552, -16.0, -4.0), (242.45999145507812, 0.0999999940395
Acquisition de  [(240.76998901367188, 0.09999999403953552, -14.999999046325684, -12.0), (240.8300018310547,
Acquisition de  [(241.0, 0.09999999403953552, -14.999999046325684, -12.0), (239.37998962402344, 0.0999999940
Acquisition de  [(239.64999389648438, 0.09999999403953552, -14.0, -7.0), (239.59999084472656, 0.099999994039
```

Quatrième partie: mise en place dans le schéma global

Comme évoqué en introduction de l'énoncé, cette étape d'intégration est de votre ressort. Cette étape intermédiaire n'est pas obligatoire pour le résultat définitif (il va y avoir une communication sous forme de web service à insérer) mais peut vous aider dans la démarche d'intégration.

TP 3 : mise en oeuvre d'un web service

L'objectif du TP est de fournir les données acquises sous la forme d'un service web. Il s'agit d'une autre forme de communication à distance.
Différents exemples documentés sont fournis sur le site du cours.

Première partie: découverte des service web en interrogeant un service existant

A partir du fichier appli1.py fourni,

- Comprendre comment s'interroge un serveur rest (les exemples)
- Lire la documentation en ligne qui renseigne sur les options de requêtes possibles
- réaliser un client qui interroge le service météo pour obtenir (par exemple) les prévisions pour les heures qui viennent.

Cartographier les exemples fournis selon les mots clé suivants (que vous compléterez): client ; serveurs ; paramètres, Json, …..

L'objectif est de savoir face à un besoin quel(s) exemple(s) s'en rapproche (et donc de comprendre ce qui s'y fait).

Deuxième partie: programmation d'un serveur

1) Programmer un service web qui appelle une fonction générant une valeur aléatoire et renvoie cette valeur
2) testez ce service à partir d'un navigateur
3) programmer un client qui toutes les lit cett valeur et l'affiche

---

4) A partir des codes fait lors du TP2, faites de même mais cette fois la fonction lit une mesure sur l'armoire.
5) testez ce service via un navigateur

6) programmer un client utilisant ce service et qui regroupe les valeurs par paquet de N

Quatrième partie : mise en place dans le schéma global

Cette démarche d'intégration est sous votre responsabilité.

TP4 : publication de données dans un Blockchain simplifié

Le but de ce dernier TP est de programmer un blockchain simplifié pour que le maître stocke dans un bloc chaque tâche réalisée par un esclave. Ce blockchain réalise principalement les opérations suivantes :

- Construction d'un bloc et ses attributs
- Calcul de la valeur de hachage d'un bloc en utilisant l'algorithme SHA256
- Mining du bloc avec l'algorithme Preuve de travail (Proof of Work) pour résoudre la difficulté

Un blockchain est fondamentalement une liste de blocs. Chaque bloc contient sa valeur de hachage, la valeur de hachage du bloc précédent et des données de transactions. Si une valeur de hachage d'un bloc change cela affecte tous les blocs.

Exercice 1 : Coder dans un fichier python *bloc.py*, une classe Bloc avec les attributs suivants : hash, previousHash, data, timeStamp. Son constructeur prend les arguments data et previousHash. L'attribut timestamp est initialisée avec la valeur actuelle de l'horloge en utilisant la méthode *time()*.

Exercice 2 : Créer un deuxième fichier *utilitaires.py*. Dans ce fichier créer la méthode applySha256 qui prend en argument *input* et retourne sa valeur de hachage sous format d'une chaîne de caractères. Cette méthode calcule le HASH de la valeur *input* en utilisant l'algorithme de hachage SHA256. Les algorithmes de hachage en Python sont disponibles dans la bibliothèque hashlib[6].

Exercice 3 : Dans la classe Bloc, coder la méthode calculateHash() qui calcule la valeur de hachage de la concaténation de trois attributs : previousHash, timeStamp et data et retourne la valeur calculée.

[6] https://docs.python.org/fr/3/library/hashlib.html

Exercice 4 : Dans le constructeur de la classe Bloc, initialiser l'attribut *hash* avec la valeur retournée par la méthode *calculateHash*.

Exercice 5 : Dans le fichier bloc.py (après la définition de la classe Bloc), rajouter cette ligne

```
if __name__ == '__main__':
```

pour tester dedans la création de quelques blocs et le calcul de leurs valeurs de hachage.

Créer une liste maChaine qui contient 3 objets de type Bloc. Chaque bloc utilise l'une des chaînes de caractères suivantes pour data et la valeur de previousHash est celle du bloc précédent (le premier bloc on lui donne comme valeur de previousHash la chaîne ''0'') :

- ''Bonjour, je suis le premier bloc''
- ''Hello, je suis le deuxième bloc''
- ''Yo, je suis le troisième bloc''

Afficher pour chaque bloc sa valeur de hachage (la valeur de l'attribut hash). Exécuter le programme plusieurs fois. Que constatez-vous ? Pourquoi les valeurs de hachages de blocs changent à chaque exécution ?.

Exercice 6 : Dans *utilitaires.py*, coder la fonction isChaineValid(blockchain) qui vérifie que votre chaîne est valide, c'est à dire, la valeur de hachage actuelle est correcte (comparer la valeur de hash avec celle retournée par la fonction calculateHash) et que la valeur de previousHash est celle du bloc précédent. Cette fonction retourne un booléen. Dans le main de *bloc.py* tester cette fonction.

Exercice 7 : Vous allez maintenant coder l'algorithme de minage de type Proof of Work. Dans la classe Bloc ajouter un attribut nonce (un entier). Dans la même classe, coder la fonction *mineBlock(difficulty)*. Cette fonction utilise une variable *target* qui contient autant de '0' que la valeur de *difficulty*. Ensuite, d'une façon itérative (boucle while) et en augmentant à chaque itération la valeur de de l'attribut nonce, la fonction essaye de trouver une valeur de hachage du bloc (attribut hash) qui commence par autant de '0' que la variable target. Dès que cette valeur est trouvée, la boucle s'arrête et le minage du bloc est fait. Pensez à rajouter l'attribut nonce dans le calcul de la valeur de hachage par la fonction calculateHash().

Exercice 8 : Tester dans le main du fichier bloc.py le mining de 3 blocs. Déclarer dans ce fichier une variable *difficulty* qui vaut 1 initialement. Utilisez la méthode time pour afficher aussi en secondes la durée de minage de chaque bloc. Ensuite augmenter la valeur de *difficulty*. Que constatez-vous ?

Exercice 9 : Vous reprenez maintenant le code de maitre.py pour stocker dans un bloc chaque tâche (numéro, mesure) exécutée par un esclave. Vous modifiez la fonction *exposed_receive_result* pour créer un bloc, miner le bloc, vérifier que la chaîne est valide et afficher la liste de blocs. Tester votre programme avec les esclaves programmés dans le TP1.