

## **Travaux pratiques : Distribution des tâches, exclusion mutuelle et blockchain**

**Durée : 8H**

Le but de ces travaux pratiques est de programmer un système distribué qui partage des tâches de calcul entre plusieurs participants et garantir une exclusion mutuelle lors de l'exécution de ces tâches. Le TP comporte trois parties : la première partie est consacrée à la réalisation du système de distribution des tâches, la deuxième partie est consacrée à la réalisation d'une exclusion mutuelle distribuée et la dernière partie est consacrée à la réalisation d'un blockchain simplifié.

**Une note est attribuée à la fin de la deuxième séance des TPs.**

### **Partie 1 : préparation d'une salade de fruits (10 points)**

Pour introduire notre système distribué, nous partons d'un programme simple chargé de préparer une salade de fruits. Notre salade sera simplement constituée d'un ensemble de fruits épluchés et découpés. Nous considérons que l'ordre d'ajout des fruits n'a pas d'importance. Notre programme pourrait ressembler au fichier `seq.py` disponible sur Arche. Tester ce programme.

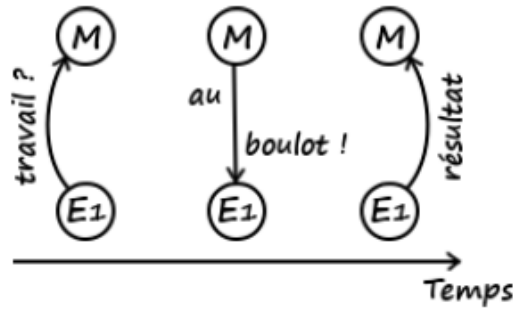
Quand on exécute ce programme, on obtient :

```
1 pêche en préparation (3s)...  
1 pomme en préparation (3s)...  
1 pomme en préparation (3s)...  
1 cerise en préparation (1s)...  
1 poire en préparation (2s)...  
1 banane en préparation (2s)...  
1 banane en préparation (2s)...  
1 pastèque en préparation (4s)...  
1 poire en préparation (2s)...  
1 pêche en préparation (3s)...  
1 cerise en préparation (1s)...
```

```
La salade est prête ! Bonne dégustation !  
Temps de préparation: 26.0s
```

Un programme simple, mais le temps d'exécution est non négligeable ! dans la suite, nous allons construire un système distribué 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 une grande salade de fruits. Pourtant, les opérations sont indépendantes les unes des autres : la préparation du kiwi n'est pas conditionnée par la présence ou l'absence de mangue dans la salade. 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 cuisiniers).



Supposons donc que nous disposons de  $n$  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  $n$  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**<sup>1</sup>.

#### 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**<sup>2</sup>. 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 suivant pour lancer un service RPC.

```

1 import rpyc
2 from rpyc.utils.server import ThreadedServer
3
4 class MyService(rpyc.Service):
5     def exposed_f(self):
6         # Cette méthode sera accessible sur le réseau du fait de
          son préfix "exposed_"
7         return 42
8
9     def g(self):
10        # Cette méthode ne sera pas accessible sur le réseau
11        return 43
12
13 def start():
14     t = ThreadedServer(MyService, port=18861)
15     t.start()
16
17 if __name__ == "__main__":
18     start()
  
```

<sup>1</sup> [https://fr.wikipedia.org/wiki/Remote\\_procedure\\_call](https://fr.wikipedia.org/wiki/Remote_procedure_call)

<sup>2</sup> <https://rpyc.readthedocs.io/en/latest/>

**Q2.** Dans une console Python, tester le serveur pour appeler la fonction *f* exposée par le serveur (remplacer *adresse\_ip\_du\_serveur* par *localhost*):

```
1 >>> import rpyc
2 >>> conn = rpyc.connect("adresse_ip_du_serveur", 18861)
3 >>> conn.root.exposed_f()
4 42
5 >>> conn.root.f() # Peut aussi être appelée sans le "exposed_"
6 42
7 >>> # Par contre, on n'a pas accès à g
8 >>> conn.root.g()
9 ...
10 AttributeError: cannot access 'g'
```

## Exercice 2 : les verrous

Pour exposer le service, c'est-à-dire pour le rendre accessible, le serveur a utilisé un *ThreadedServer*. Comme indiqué dans la documentation<sup>3</sup>, 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 incrémente  $N=1000000$  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'incrément

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

---

<sup>3</sup> [https://rpyc.readthedocs.io/en/latest/api/utils\\_server.html#rpyc.utils.server.ThreadedServer](https://rpyc.readthedocs.io/en/latest/api/utils_server.html#rpyc.utils.server.ThreadedServer)

```
[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 du serveur pour protéger la variable globale *i* avec un lock (verrou)<sup>4</sup>.

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

### Exercice 3 : Programmer le maître

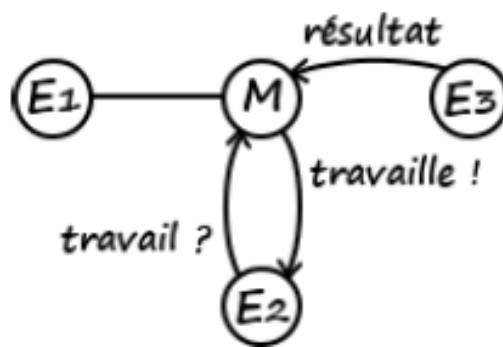
Dans le cadre de ce TP, nous nous restreindrons à un maître et, disons, trois esclaves.

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).

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 un fruit préparé).



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 *maître.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* pour récupérer une tâche de la liste). Elle sauvegarde également dans une liste *tasks\_being\_done* la tâche envoyée.

<sup>4</sup> <https://docs.python.org/fr/3/library/threading.html#lock-objects>

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 a plus de tâches à faire et que toutes les tâches ont été effectuées. Si c'est le cas, elle affiche « La salade est prête ! bonne dégustation ». **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 :

```
1 task = ask_for_task()
2 while task:
3     res = work(task)
4     send_results(task, res)
5     task = ask_for_task()
```

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

- `create_connection()` : cette méthode initialise la connexion avec le serveur
- `prepare_fruit(id,fruit,t)` : qui prend comme argument l'identifiant de la tâche, le nom du fruit et son temps de préparation. Elle affiche ces informations et elle simule avec un `time.sleep` le temps de préparation. Elle retourne une chaîne de caractères qui contient le nom du fruit est préparée !
- `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 préparer le fruit et envoyer le résultat.

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

#### Exercice 5 : Maintenant, on teste !

Pour tester votre programme, on démarre le maître dans un terminal. Ensuite on démarre trois esclaves en parallèle dans un autre terminal.

```
MacBook-Air-de-lahmadi:Sujet_TP AbdelkaderLahmadi$ python3 esclave.py & python3 esclave.py & python3 esclave.py
[1] 94956
[2] 94957
```

```
1 pastèque envoyée en préparation 10
1 pêche envoyée en préparation 9
1 pêche envoyée en préparation 8
1 pêche préparée reçue
1 cerise envoyée en préparation 7
1 pêche préparée reçue
1 cerise envoyée en préparation 6
1 pastèque préparée reçue
1 banane envoyée en préparation 5
1 cerise préparée reçue
1 banane envoyée en préparation 4
1 cerise préparée reçue
1 poire envoyée en préparation 3
1 banane préparée reçue
1 poire envoyée en préparation 2
1 banane préparée reçue
1 pomme envoyée en préparation 1
1 poire préparée reçue
1 pomme envoyée en préparation 0
1 poire préparée reçue
1 pomme préparée reçue
1 pomme préparée reçue
La salade est prête! bonne dégustation
Temps de préparation : 9.0s
```

On observe que le temps de préparation est environ divisé par 3 ! Nous avons désormais une version parallèle de la préparation de notre salade de fruits.

## Partie 2 : une exclusion mutuelle distribuée (5 points)

Dans cette deuxième partie du TP, vous allez coordonner les esclaves pour que l'envoi du résultat au maître soit synchronisé en exclusion mutuelle, c'est à dire que chaque esclave prépare le fruit mais un à la fois dépose ce fruit dans le saladier. La partie distribution des tâches par le maître ne change pas. Seulement dans la méthode *send\_result*, l'esclave demande l'autorisation avant d'envoyer son résultat. Nous utilisons l'algorithme de Lamport d'exclusion mutuelle pour que les esclaves gèrent cet accès.

### Exercice 1. Implémentation de l'algorithme de Lamport

Au début, vous allez implémenter l'algorithme de Lamport en utilisant le broker RabbitMQ<sup>5</sup> pour échanger les messages entre les nœuds avec le protocole MQTT. Ce protocole utilise un mécanisme de type inscription/publication pour échanger les données entre deux ou plusieurs nœuds. Le nœud publie les données dans un broker (un serveur) et un autre s'inscrit au broker pour récupérer les données. Les échanges s'effectuent en spécifiant un ou plusieurs topics communs entre les nœuds.

**Quick reminder: What is MQTT?** – “MQTT stands for MQ Telemetry Transport. It is a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimise network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging “machine-to-machine” (M2M) or “Internet of Things” world of connected devices, and for mobile applications where bandwidth and battery power are at a premium.” – source <http://mqtt.org/faq>

En Python la bibliothèque Pika permet d'interagir avec le broker RabbitMQ à installer sur les machines. Utiliser la version 0.12.0 de pika : *pip3 install pika==0.12.0*

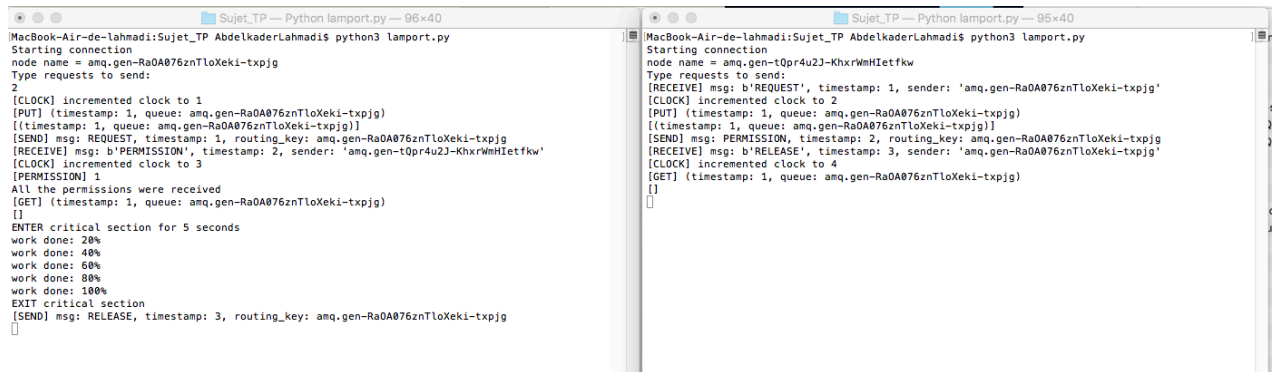
**Q1.** Depuis Arche, importer le fichier *request.py* qui définit une classe pour représenter une requête de l'algorithme de Lamport. La classe possède un constructeur avec les arguments suivants : *timestamp* et *queue\_name*. La classe redéfinit la fonction *\_\_lt\_\_* pour comparer les timestamp de deux requêtes à placer dans une file de priorité (*queue.PriorityQueue()*).

**Q2.** Le code de l'algorithme de Lamport est **disponible sous Arche dans le fichier *lamport.py***. Importer ce fichier dans le répertoire du projet. Étudier ce programme, notamment la fonction *process\_received\_messages* qui permet de traiter chaque message reçue. Il s'agit ici de la version de l'algorithme de Lamport sans optimisation, c'est à dire pour entrer en section critique un processus doit recevoir *n-1* permissions et il est à la tête de la file d'attente.

**Q3.** Tester le programme *lamport.py* dans deux terminaux pour simuler une exécution de cet algorithme.

---

<sup>5</sup> <https://www.rabbitmq.com/>



```

MacBook-Air-de-lahmadi:Sujet_TP AbdelkaderLahmadi$ python3 lamport.py
Starting connection
node name = amq.gen-Ra0A076znTloXeki-tpjg
Type requests to send:
2
[CLOCK] incremented clock to 1
[PUT] (timestamp: 1, queue: amq.gen-Ra0A076znTloXeki-tpjg)
[[timestamp: 1, queue: amq.gen-Ra0A076znTloXeki-tpjg]]
[SEND] msg: REQUEST, timestamp: 1, routing_key: amq.gen-Ra0A076znTloXeki-tpjg
[RECEIVE] msg: b'PERMISSION', timestamp: 2, sender: 'amq.gen-t0pr4u2J-KhxrWmHietfw'
[CLOCK] incremented clock to 3
[PERMISSION] 1
All the permissions were received
[GET] (timestamp: 1, queue: amq.gen-Ra0A076znTloXeki-tpjg)
[]
ENTER critical section for 5 seconds
work done: 20%
work done: 40%
work done: 60%
work done: 80%
work done: 100%
EXIT critical section
[SEND] msg: RELEASE, timestamp: 3, routing_key: amq.gen-Ra0A076znTloXeki-tpjg
[]

MacBook-Air-de-lahmadi:Sujet_TP AbdelkaderLahmadi$ python3 lamport.py
Starting connection
node name = amq.gen-t0pr4u2J-KhxrWmHietfw
Type requests to send:
[RECEIVE] msg: b'REQUEST', timestamp: 1, sender: 'amq.gen-Ra0A076znTloXeki-tpjg'
[CLOCK] incremented clock to 2
[PUT] (timestamp: 1, queue: amq.gen-Ra0A076znTloXeki-tpjg)
[[timestamp: 1, queue: amq.gen-Ra0A076znTloXeki-tpjg]]
[SEND] msg: PERMISSION, timestamp: 2, routing_key: amq.gen-Ra0A076znTloXeki-tpjg
[RECEIVE] msg: b'RELEASE', timestamp: 3, sender: 'amq.gen-Ra0A076znTloXeki-tpjg'
[CLOCK] incremented clock to 4
[GET] (timestamp: 1, queue: amq.gen-Ra0A076znTloXeki-tpjg)
[]

```

## Exercice 2. Le saladier en section critique

Maintenant, vous allez modifier le code du programme esclave.py pour qu'un seul esclave accède au saladier pour préparer et déposer le fruit après obtenir la permission en utilisant l'algorithme de Lamport.

**Q1.** Dupliquer le code de *lamport.py* dans un fichier nommé *lamport\_saladier.py*. Nous allons maintenant adapter le code de la classe Lamport pour pouvoir l'utiliser par un esclave. Les étapes à suivre pour cette adaptation sont les suivantes :

- Nous avons besoin que l'algorithme permet à un esclave d'attendre la permission d'entrée en section critique. Nous rajoutons ainsi dans la classe Lamport une méthode *wait\_critical\_section* qui bloque sur un sémaphore. Utiliser pour cela *threading.Semaphore* (*sem = threading.Semaphore(0)*) pour créer un sémaphore initialisé à 0. Dans *wait\_critical\_section* appeler sa méthode *acquire*.
- Dans la même classe Lamport, rajouter une méthode *allow\_critical\_section* qui permet de libérer le sémaphore avec la méthode *release*.
- Rajouter une méthode *release\_critical\_section* qui fait les mêmes opérations que la méthode *enter\_critical\_section* : mettre à zéro la variable *received\_permissions*, et envoyer un message de type RELEASE aux autres noeuds.
- Modifier le code de la méthode *process\_received\_messages* pour utiliser la nouvelle méthode *allow\_critical\_section* lorsqu'un nœud a le droit d'entrer en section critique.

**Q2.** Dupliquer le code d'esclave.py dans un fichier nommé *esclave\_lamport.py*. Créer un objet de la classe Lamport (version du fichier *lamport\_saladier*). Modifier la méthode *run* pour que la préparation et l'envoi d'un fruit soit synchronisée entre les esclaves avec une exclusion mutuelle, c'est à dire : avant de préparer le fruit, chaque esclave envoie une requête (*create\_request*) et attend la permission (*wait\_critical\_section*). Après l'envoi du résultat au maître, il libère la section critique (*release\_critical\_section*). Le constructeur de la classe Lamport, prend en argument le nombre de nœuds esclaves dans votre système.

**Q3.** Testons maintenant votre application avec un maître et 2 esclaves utilisant une section critique distribuée. Vous pouvez observer que le temps de préparation est environ le même que la version séquentielle. Vous pouvez également noter que le système nécessite au moins 2 processus lancés en parallèle pour fonctionner.



```
MacBook-Air-de-lahmadi:Sujet_TP AbdelkaderLahmadi$ python3 esclave_lamport.py & python3 esclave_lamport.py
[1] 49879
Starting connection
Starting connection
node name = amq.gen-sfWE1v8KGvNMJSH7JV7szw
node name = amq.gen-CG2ev2JfmIlzFCU2LXsU-w
1 pastèque à préparer reçue 10
[CLOCK] incremented clock to 1
[PUT] (timestamp: 1, queue: amq.gen-sfWE1v8KGvNMJSH7JV7szw)
[(timestamp: 1, queue: amq.gen-sfWE1v8KGvNMJSH7JV7szw)]
[SEND] msg: REQUEST, timestamp: 1, routing_key: amq.gen-sfWE1v8KGvNMJSH7JV7szw
1 pêche à préparer reçue 9
[CLOCK] incremented clock to 1
[PUT] (timestamp: 1, queue: amq.gen-CG2ev2JfmIlzFCU2LXsU-w)
[(timestamp: 1, queue: amq.gen-CG2ev2JfmIlzFCU2LXsU-w)]
[SEND] msg: REQUEST, timestamp: 1, routing_key: amq.gen-CG2ev2JfmIlzFCU2LXsU-w
[RECEIVE] msg: b'REQUEST', timestamp: 1, sender: 'amq.gen-sfWE1v8KGvNMJSH7JV7szw'
[CLOCK] incremented clock to 2
[PUT] (timestamp: 1, queue: amq.gen-sfWE1v8KGvNMJSH7JV7szw)
[(timestamp: 1, queue: amq.gen-CG2ev2JfmIlzFCU2LXsU-w), (timestamp: 1, queue: amq.gen-sfWE1v8KGvNMJSH7JV7szw)]
[RECEIVE] msg: b'REQUEST', timestamp: 1, sender: 'amq.gen-CG2ev2JfmIlzFCU2LXsU-w'
[CLOCK] incremented clock to 2
[PUT] (timestamp: 1, queue: amq.gen-CG2ev2JfmIlzFCU2LXsU-w)
[(timestamp: 1, queue: amq.gen-sfWE1v8KGvNMJSH7JV7szw), (timestamp: 1, queue: amq.gen-CG2ev2JfmIlzFCU2LXsU-w)]
[SEND] msg: PERMISSION, timestamp: 2, routing_key: amq.gen-CG2ev2JfmIlzFCU2LXsU-w
[RECEIVE] msg: b'PERMISSION', timestamp: 2, sender: 'amq.gen-CG2ev2JfmIlzFCU2LXsU-w'
[CLOCK] incremented clock to 3
[PERMISSION] 1
```

Running script: `"/Users/AbdelkaderLahmadi/Desktop/Enseignements/2021-2022/3A/Algo_distribués_ISN/Sujet_TP/maitre.py"`

```
1 pastèque envoyée en préparation 10
1 pêche envoyée en préparation 9
1 pêche préparée reçue
1 pêche envoyée en préparation 8
1 pastèque préparée reçue
1 cerise envoyée en préparation 7
1 pêche préparée reçue
1 cerise envoyée en préparation 6
1 cerise préparée reçue
1 banane envoyée en préparation 5
1 cerise préparée reçue
1 banane envoyée en préparation 4
1 banane préparée reçue
1 poire envoyée en préparation 3
1 banane préparée reçue
1 poire envoyée en préparation 2
1 poire préparée reçue
1 pomme envoyée en préparation 1
1 poire préparée reçue
1 pomme envoyée en préparation 0
1 pomme préparée reçue
1 pomme préparée reçue
La salade est prête! bonne dégustation
Temps de préparation : 23.1s
```

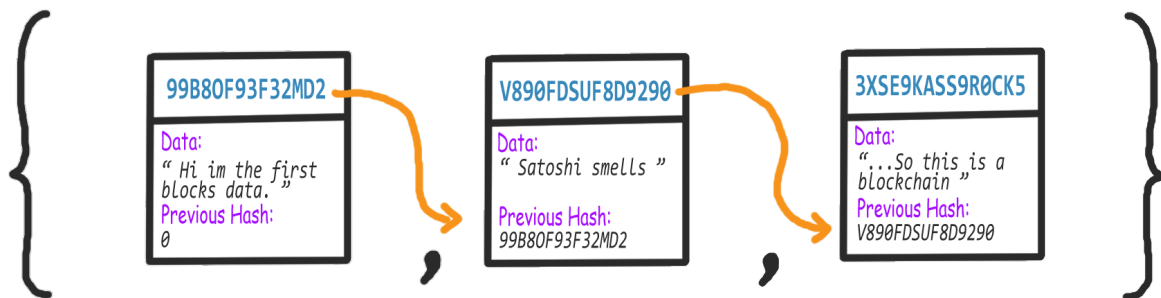


### Partie 3 : un blockchain simplifié (5 points)

Le but de cette troisième partie 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 (*hexdigest*). 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<sup>6</sup>.

**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 hachage calculée.

**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.

<sup>6</sup> <https://docs.python.org/fr/3/library/hashlib.html>

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 :** 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 7 :** 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 8 :** Vous reprenez maintenant le code de `maitre.py` pour ajouter le stockage dans un bloc une chaîne de caractère qui représente chaque tâche (numéro, ingrédient) exécutée par un esclave. Vous modifiez la fonction `exposed_receive_result` pour créer et miner un bloc, ensuite l'insérer dans la chaîne et afficher la liste de blocs. Tester votre programme avec les esclaves de la partie 1 (sans exclusion mutuelle distribuée).