Vayel

Zeste de savoir

Introduction aux systèmes distribués

25 décembre 2018

Table des matières

1. Introduction 1
2. Préparation d'une salade de fruits 3
3. Distribuons les calculs 4
   3.1. Architecture du système 4
   3.2. Protocole de communication 6
4. L'exclusion mutuelle 7
   4.1. Des conflits d'écriture 8
   4.2. Régler les conflits avec des verrous 10
5. Une implémentation en Python 13
   5.1. L'architecture 13
   5.2. Un code basique 14
   5.3. Maintenant, on teste ! 18
6. Alerte générale ! Une panne ! 19
   6.1. Observation d'une panne 20
7. Un remède : le timeout 24
   7.1. Un code robuste aux pannes 24
8. Conclusion 28

Contenu masqué 29

1. Introduction

Afin de tester ma connexion Internet, j'ai pour habitude d'exécuter la commande `ping google.com`. Basiquement, je demande au moteur de recherche s'il reçoit mes messages et, s'il ne me répond pas, je considère que je n'ai pas accès à Internet. Pourtant, il se pourrait que ce soit Google le problème et non le réseau.

?

Tu plaisantes : un service indisponible chez Google ?

Effectivement, c'est plutôt improbable. Et la raison à cela est que leur architecture repose sur un monstrueux **système distribué**. Tout est répliqué, si bien que lorsqu'une machine tombe en panne, il y en a des dizaines d'autres pour prendre sa place, rendant les services hautement disponibles .

Une autre raison de connecter des machines et de les faire se coordonner est le **calcul réparti**. Quand on effectue de lourdes opérations, un moyen d'accélérer l'exécution est d'investir dans une machine plus puissante. Seulement, vos économies risquent de ne pas apprécier.[1] Qui plus est, la puissance convoitée est limitée par des lois physiques . Rien que ça !

1. Prenez l'exemple du Titan , d'une valeur de presque cent millions de dollars.

Pour pallier tout ça, on fait appel au **parallélisme** : on découpe les calculs en petits morceaux les plus indépendants possibles puis on fait exécuter ces opérations par des composants pouvant travailler en même temps. On les qualifie alors de **concurrents**. De la même manière que pour préparer une pizza, une personne peut découper les champignons pendant qu'une autre pétrit la pâte et une troisième râpe le fromage.

Il nous faut alors disposer de plusieurs unités de calcul, que ce soit les cœurs d'un processeur, plusieurs processeurs sur une seule machine ou plusieurs machines en réseau. On parle de **système distribué** ou **système réparti**. Un tel système est soumis à des problématiques de synchronisation et cohérence des données, de disponibilité, de tolérance aux pannes...

Ces systèmes se justifient par bien d'autres raisons, parmi lesquelles on peut citer :

— La **sécurité** : si une seule machine exposait `google.com`, le site se verrait très vulnérable aux attaques. Avec des dizaines de serveurs, on est plus serein.
— Le **passage à l'échelle** : supporter l'augmentation de la charge (par exemple du nombre de visiteurs d'un site web) n'est pas un problème avec un système réparti, il « suffit » d'ajouter des machines.
— Le **respect de la loi** : la réglementation de certains pays interdit le stockage d'informations sensibles à l'étranger. Par exemple, une banque internationale devra avoir des machines sur divers territoires pour héberger les données de ses clients tout en restant dans la légalité.
— ...

Ce tutoriel se veut une introduction pratique aux systèmes distribués. Au travers d'un exemple regorgeant de vitamines, nous verrons comment il est possible de **répartir ses calculs sur plusieurs machines** en vu d'augmenter les performances en temps d'exécution. L'objectif n'est pas d'obtenir un programme optimal ni de devenir un expert dans le domaine, seulement de se familiariser avec quelques notions sous-jacentes et de vous donner envie de creuser le sujet.

Pour profiter au mieux de ce tutoriel, il est préférable de satisfaire les prérequis suivants :

i Pré-requis

Bases en programmation. Le code sera écrit en Python. La clarté de sa syntaxe le rend accessible à quiconque ayant déjà programmé ; je ne crois donc pas qu'il soit nécessaire de connaître ce langage pour comprendre le contenu. En revanche, je n'introduirai que très brièvement l'écosystème Python donc il vous faudra vous débrouiller si vous souhaitez exécuter vous-même le code présenté. Tout est disponible sur GitHub .
Bases en réseau. Il sera nécessaire d'avoir des bases en réseau et notamment d'être familier avec les notions de protocole, adresse IP, port et requête. Par exemple, vous avez déjà mis en place un serveur web.

Ce tutoriel s'adresse donc principalement, mais pas exclusivement, à des programmeurs souhaitant s'initier aux systèmes distribués.

2. Préparation d'une salade de fruits

Pour introduire notre système distribué, nous partirons 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 alors ressembler à :

```python
import time


def prepare_seq(ingredients):
    for fruit, t in ingredients:
        print(f"1 {fruit} en préparation ({t}s)...")
        time.sleep(t)
    print("\nLa salade est prête ! Bonne dégustation !")


if __name__ == "__main__":
    # Une liste d'ingrédients. Chaque ingrédient est un couple
    # (nom, temps de préparation en secondes). Les temps inscrits ici ne
    # sont pas réalistes.
    INGREDIENTS = [
        ("banane", 2),
        ("pêche", 3),
        ("banane", 2),
        ("cerise", 1),
        ("cerise", 1),
        ("poire", 2),
        ("pêche", 3),
        ("pomme", 3),
        ("poire", 2),
        ("pastèque", 4),
        ("pomme", 3)
    ]
    start_time = time.time()
    prepare_seq(INGREDIENTS)
    end_time = time.time()
    print(f"Temps de préparation : {end_time - start_time:.1f}s")
```

Quand on exécute ce programme dans un terminal (`python seq.py`), on obtient :

```
1 banane en préparation (2s)...
1 pêche en préparation (3s)...
1 banane en préparation (2s)...
1 cerise en préparation (1s)...
```

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

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

Pour rappel le code est disponible sur GitHub .

Un programme simple mais au temps d'exécution non négligeable ! Dans la section suivante, nous introduisons les briques pour construire un système distribué simple afin de paralléliser les opérations.

3. Distribuons les calculs

L'algorithme décrit dans la section précédente 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 section, nous profitons de cette propriété pour répartir le travail sur plusieurs acteurs (plusieurs cuisiniers).

3.1. Architecture du système

Supposons donc que nous disposons de n machines capables de communiquer. Une des machines, que nous noterons M, reçoit la liste des ingrédients. Il lui faut alors distribuer les tâches entre les n composants (incluant elle-même). Pour ce faire, il existe plusieurs stratégies, que je me contente de vous introduire :

— **Découpage statique :** M découpe le travail et le répartit entre les $n-1$ autres machines puis assemble les résultats qu'elle reçoit. Des tâches ne peuvent alors pas apparaître au cours du temps.

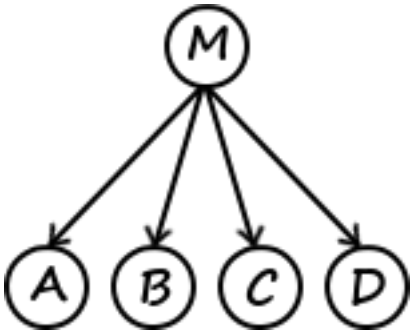

FIGURE 3. – Découpage statique

— **Maître-esclave :** M (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.

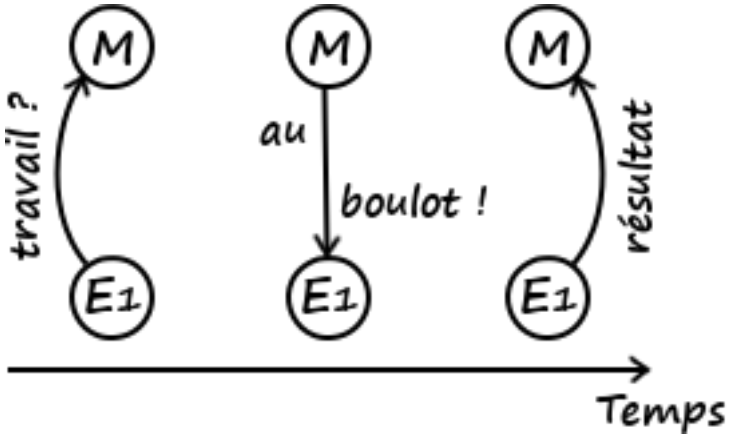

FIGURE 3. – Maître-esclave

— ***Work stealing* :** ici, pas de maître, tout le monde est au même niveau. On assigne à chacun une liste de tâches. Quand une machine a terminé son travail, elle en sélectionne une autre au hasard et lui vole des calculs à effectuer, et ce jusqu'à ce que plus personne n'ait rien à faire. Un avantage de cette méthode est que ce sont les chômeurs qui gèrent la répartition des tâches. Les machines à qui il reste du travail peuvent donc se concentrer sur celui-ci.

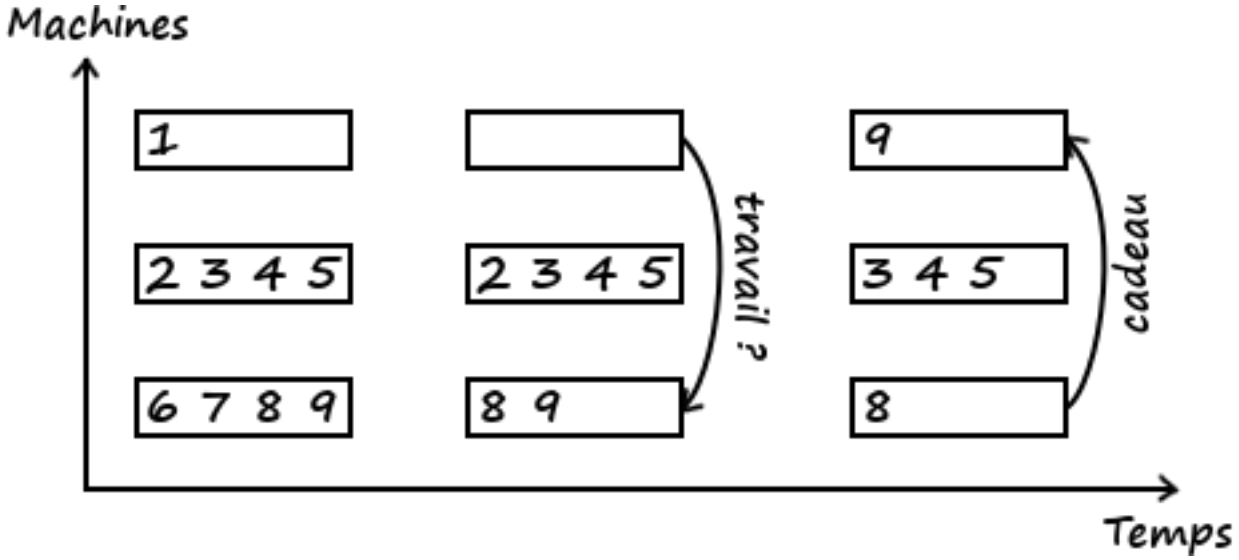

FIGURE 3. – *Work stealing*. La victime est bien aléatoire, on ne prend pas nécessairement celle à qui il reste le plus de tâches.

— ...

Dans le cadre de ce tutoriel, nous implémenterons la stratégie **maître-esclave** : elle est à la fois plutôt simple à appréhender et suffisamment élaborée pour introduire différentes notions de systèmes distribués. La découpe du travail sera très basique : une tâche consistera à préparer (éplucher et découper) un seul fruit.

3.2. 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 tutoriel, la communication entre le maître et les esclaves se fera arbitrairement[2] par le protocole RPC .

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 . 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` :

```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()
```

Une fois le service démarré (en exécutant le code Python ci-dessus : `python server.py`), il ne reste plus qu'à s'y connecter depuis un autre acteur (par exemple, depuis une autre machine ou depuis un autre terminal sur la même machine). Dans l'interpréteur Python (en exécutant simplement la commande `python` sans argument), on aurait :

2. La question du choix du protocole est hors de portée de ce tutoriel introductif.

```
>>> 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'
```

Il est important de comprendre que le service s'exécutant sur `server` permet de **recevoir** des requêtes puis d'y répondre. Uniquement de recevoir. Autrement dit, si le client ne contacte pas le serveur, ce dernier n'a aucun moyen (avec ce protocole) de lui transmettre des informations. On parle d'une **architecture client-serveur**.

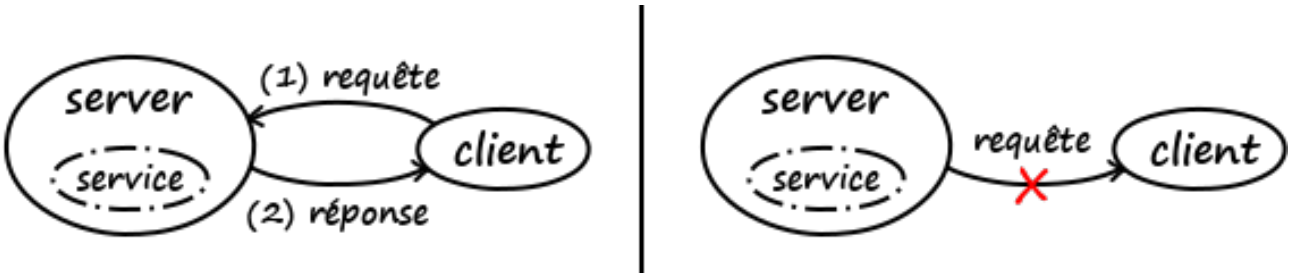

FIGURE 3. – Architecture client-serveur. Le client peut contacter le serveur et le serveur lui répond (à gauche).
Par contre, le serveur ne peut pas prendre l'initiative de la communication (à droite).

L'architecture de notre système est désormais définie. Avant de commencer l'implémentation, il nous faut introduire une notion importante pour limiter les erreurs incompréhensibles : l'**exclusion mutuelle**.

4. L'exclusion mutuelle

Pour exposer notre service, c'est-à-dire pour le rendre accessible, nous avons utilisé un `ThreadedServer`. Comme indiqué dans la documentation , ce serveur créera un *thread* (ou « fil » en français) pour chaque client.

Pour faire simple, un *thread* est un morceau de code exécuté en parallèle du programme principal. Dans notre cas, utiliser des fils permet de traiter plusieurs requêtes d'esclaves en même temps (à l'aide de plusieurs cœurs de processeur par exemple) afin de diminuer le temps d'attente.

4.1. Des conflits d'écriture

Une particularité des *threads* par rapport aux processus est la mémoire partagée : tous les fils créés peuvent lire et écrire les variables globales du programme principal. Cette caractéristique est très pratique pour partager des informations entre les *threads*, mais il faut la manier avec prudence pour éviter les conflits d'écriture. Quand cette précaution n'est pas prise, on peut se retrouver avec des comportements... inattendus. Illustrons ce genre de cas en incrémentant une variable globale `i` :

```
N = 1000000
i = 0


def incr():
    # Cette fonction incrémente `N fois` la variable globale `i`.
    global i
    for _ in range(N):
        i += 1


if __name__ == "__main__":
    print("Un seul thread :")
    incr()
    print("    N - i =", N-i)
```

On obtient sans surprise :

```
Un seul thread :
    N - i = 0
```

Maintenant, démarrons deux *threads* chargés d'incrémenter `i` en parallèle :

```
import time
from threading import Thread

N = 1000000
i = 0


def incr():
    # Cette fonction incrémente `N` fois la variable globale `i`.
    global i
    for _ in range(N):
        i += 1
```

```python


def run(t1, t2):
    start = time.time()
    # Les méthodes `t1.run()` et `t2.run()` sont exécutées en parallèle.
    # Elles vont chacune incrémenter `i` en parallèle.
    t1.start()
    t2.start()
    # On attend que `t1` et `t2` terminent, c'est-à-dire qu'ils aient chacun
    # incrémenté `i` `N` fois.
    t1.join()
    t2.join()
    end = time.time()
    # Comme `i` a en théorie été incrémentée `N` fois par chacun des threads,
    # on s'attend à ce qu'elle soit égale à `2N`.
    print(f"    {end - start:.2f} secondes")
    print(f"    2N - i = {2*N-i}")


if __name__ == "__main__":
    print("Un seul thread :")
    incr()
    print("    N - i =", N-i)

    print("\nDeux threads (sans exclusion mutuelle) :")
    i = 0
    run(Thread(target=incr), Thread(target=incr))
```

On obtient :

```
Un seul thread :
    N - i = 0

Deux threads (sans exclusion mutuelle) :
    0.19 secondes
    2N - i = 574629
```

Saperlipopette ! Mais que se passe-t-il ?

On constate effectivement que `2N - i` n'est pas nul, alors qu'en théorie `i` a été incrémentée `N` fois par chacun des deux *threads*. Pour comprendre les forces magiques à l'oeuvre, il faut se rendre compte que l'opération `i += 1` est un raccourci pour `i = i + 1` et se décompose (dans les grandes lignes) en trois étapes :

1. Lire la valeur de `i`

2. Y ajouter 1
3. Mettre à jour la valeur de `i`

Or cette séquence d'opérations **n'est pas atomique** en Python, c'est-à-dire que rien ne nous garantit qu'elle est exécutée en un seul morceau.

Quand il n'y a qu'un seul *thread* ce n'est pas problèmatique puisque cet acteur n'effectue qu'une opération à la fois et suit l'ordre du programme, comme on s'y attend. Par contre, quand plusieurs acteurs **travaillent en parallèle avec la même ressource**, les séquences non-atomiques peuvent s'imbriquer. Par exemple :

1. `i = 0`
2. `t1` lit `i` : `i = 0`
3. `t2` lit `i` : `i = 0`
4. `t1` incrémente `i` : `i = 0 + 1`
5. `t2` incrémente `i` : `i = 0 + 1`[3]
6. `i` vaut 1 au lieu de 2

4.2. Régler les conflits avec des verrous

Pour éviter ce conflit, on s'assure de l'**exclusion mutuelle** :

Définition : exclusion mutuelle

Faire en sorte que la mémoire partagée ne soit pas manipulée en écriture simultanément par plusieurs fils.

Pour ce faire, on utilise un **verrou** : avant de manipuler la ressource, on la réserve pour s'assurer que personne d'autre ne s'en sert en même temps que nous. L'exemple précédent devient :

1. `i = 0`
2. `t1` pose un verrou sur `i`
3. `t1` lit `i` : `i = 0`
4. `t2` ne peut lire `i` à cause du verrou, il attend
5. `t1` incrémente `i` : `i = 1`
6. `t1` libère le verrou
7. `t2` détecte la libération du verrou et en pose un
8. `t2` lit `i` : `i = 1`
9. `t2` incrémente `i` : `i = 2`
10. `t2` libère le verrou
11. `i` vaut 2 (youpi !)

En Python, on a :

3. Effectivement, `t2` a lu la valeur de `i` en 3, soit avant que `t1` écrive la valeur incrémentée (en 4).

```python
import time
from threading import Thread, Lock

N = 1000000
i = 0


def incr():
    # Cette fonction incrémente `N` fois la variable globale `i`.
    global i
    for _ in range(N):
        i += 1


def incr_with_lock(lock):
    global i
    for _ in range(N):
        # On utilise le verrou le moins longtemps possible pour ne pas bloquer
        # excessivement les autres threads. C'est pourquoi la ressource est
        # réservée dans la boucle et non pas à l'extérieur.
        with lock:
            i += 1


def run(t1, t2):
    start = time.time()
    # Les méthodes `t1.run()` et `t2.run()` sont exécutées en parallèle.
    # Elles vont chacune incrémenter `i` en parallèle.
    t1.start()
    t2.start()
    # On attend que `t1` et `t2` terminent, c'est-à-dire qu'ils aient chacun
    # incrémenté `i` `N` fois.
    t1.join()
    t2.join()
    end = time.time()
    # Comme `i` a en théorie été incrémentée `N` fois par chacun des threads,
    # on s'attend à ce qu'elle soit égale à `2N`.
    print(f"    {end - start:.2f} secondes")
    print(f"    2N - i = {2*N-i}")


if __name__ == "__main__":
    print("Un seul thread :")
    incr()
```

```
45      print("    N - i =", N-i)
46
47      print("\nDeux threads (sans exclusion mutuelle) :")
48      i = 0
49      run(Thread(target=incr), Thread(target=incr))
50
51      print("\nDeux threads (avec exclusion mutuelle) :")
52      # Nous définissons un seul verrou que nous partageons entre les
            threads.
53      lock = Lock()
54      i = 0
55      run(
56          Thread(target=incr_with_lock, args=(lock,)),
57          Thread(target=incr_with_lock, args=(lock,)),
58      )
```

Ce code affiche :

```
1  Un seul thread :
2      N - i = 0
3
4  Deux threads (sans exclusion mutuelle) :
5      0.16 secondes
6      2N - i = 556016
7
8  Deux threads (avec exclusion mutuelle) :
9      3.04 secondes
10     2N - i = 0
```

Ici, plus de conflits : `i` a la valeur attendue. On remarque également que le temps d'exécution est bien supérieur : c'est normal puisque le verrou induit de l'attente de la part des *threads*.

En résumé, retenez que si plusieurs *threads* peuvent utiliser la même ressource en écriture, cet usage doit faire au préalable l'objet d'une réservation de la ressource en question.

Vous vous demandez peut-être comment on évite les conflits lors de la pose de verrou. Après tout, les deux *threads* pouvaient bien incrémenter la variable `i` chacun de leur côté en effaçant le travail de l'autre ; il pourrait se passer la même chose avec la pose de verrou. Je vous invite à consulter la page Wikipédia de l'exclusion mutuelle si le sujet vous intéresse.

Nous avons désormais tous les outils en main pour implémenter notre multiplication matricielle répartie. Je vous invite à mettre votre lecture en pause et vos mains dans le cambouis. Expérimenter par vous-même vous permettra d'assimiler les notions bien plus efficacement.

Une solution vous est présentée dans la section suivante. Je veillerai à introduire les éléments progressivement de sorte que vous n'ayez pas toute la réponse d'un coup et puissiez reprendre vos expériences en cours de lecture.

5. Une implémentation en Python

5.1. L'architecture

La première question que nous nous posons est la suivante :

? Combien d'acteurs nous faut-il et de quels types ?

Dans le cadre de ce tutoriel, nous nous restreindrons à un maître et, disons, trois esclaves. Nous reposer sur un seul maître nous rend vulnérables en cas de panne de cette machine (nous reviendrons plus tard là-dessus), 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 tutoriel (du moment qu'il y en a au moins un et relativement peu pour que le maître ne soit pas surchargé).

FIGURE 5. – Les acteurs de notre système : un maître et trois esclaves.

Maintenant que nous avons nos acteurs, demandons-nous comment les faire interagir.

? Quelle est la nature et le sens des communications entre les acteurs ?

Nous l'avons vu plus haut, le protocole RPC fonctionne par requêtes-réponses. Pour qu'un acteur puisse recevoir des requêtes, il lui faut exposer un service. 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 notre cas, un fruit préparé).

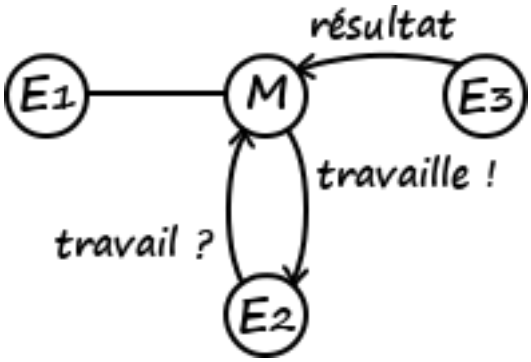

FIGURE 5. – Seuls les esclaves font des requêtes, lesquelles peuvent être de deux types : demander du travail ou retourner un résultat.

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()
```

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.

5.2. Un code basique

Tout d'abord, du code partagé entre maître et esclaves (`shared.py`) et que nous aurons l'occasion de réutiliser plus loin :

```
import datetime as dt
from random import shuffle
import os


## Une liste d'ingrédients. Chaque ingrédient est un couple
# (nom, temps de préparation en secondes). Les temps inscrits ici ne
## sont pas réalistes.
INGREDIENTS = [
    ("pomme", 3),
    ("pomme", 3),
    ("poire", 2),
    ("poire", 2),
    ("banane", 2),
```

```
15      ("banane", 2),
16      ("cerise", 1),
17      ("cerise", 1),
18      ("pêche", 3),
19      ("pêche", 3),
20      ("pastèque", 4),
21  ]
22  shuffle(INGREDIENTS)
23
24
25  def log(agent, msg, task, out=None):
26      now = dt.datetime.now().timestamp()
27      if out is None:
28          direction = ""
29      elif out:
30          direction = "OUT"
31      else:
32          direction = "IN"
33      print(f"[{now}][{agent}][T-{task:02d}][{direction}] {msg}")
34
35
36  def log_master(*args, **kwargs):
37      log("MAITRE", *args, **kwargs)
38
39
40  def log_slave(*args, **kwargs):
41      log(f"E-{os.getpid()}", *args, **kwargs)
42
43
44  def run_master(service_factory):
45      import sys
46      from rpyc.utils.server import ThreadedServer
47
48      service = service_factory(INGREDIENTS)
49
50      port = int(sys.argv[1])
51      server = ThreadedServer(service, port=port)
52
            print(f"Le maître est accessible à {server.host}:{server.port}.",
            end="\n\n")
53      server.start()
```

5.2.1. Les esclaves

```python
import os
import sys
import time

import rpyc

from shared import *


def create_connection():
    # sys.argv contient les arguments passés en ligne de commande
    # sys.argv[0] est le nom du script Python exécuté. L'indice
    # du premier argument est donc 1.
    # Un esclave s'exécute de cette manière : `python3 slave.py
        master_ip master_service_port`
    # Par exemple : `python3 slave.py 192.168.168.1 18861`
    master_addr = sys.argv[1]
    master_port = int(sys.argv[2])
    return rpyc.connect(master_addr, master_port)


def prepare_fruit(id_, fruit, t):
    log_slave(f"1 {fruit} en préparation ({t}s)...", id_)
    time.sleep(t)
    return f"1 {fruit} préparée"


def send_result(conn, task, result):
    conn.root.receive_result(task, result)


def ask_task(conn):
    return conn.root.give_task()


def run(conn):
    # `task` est un couple `(fruit, temps de préparation)`.
    task = ask_task(conn)

    while task:
        id_, fruit, t = task

        log_slave(f"1 {fruit} à préparer reçue", id_, out=False)
        prepared_fruit = prepare_fruit(id_, fruit, t)

        log_slave(f"1 {fruit} prête envoyée", id_, out=True)
        send_result(conn, task, prepared_fruit)
```

```
47
48          task = ask_task(conn)
49
50
51  if __name__ == "__main__":
52      run(create_connection())
```

5.2.2. Le maître

```
1   from threading import Lock
2   import time
3
4   import rpyc
5
6   from shared import *
7
8
9   def prepare_distributed(ingredients):
10
            """Retourne un service RPC appelé par les esclaves pour demander du tra
11      et notifier des résultats.
12      """
13
14      # We add ids to tasks as we may have the same fruit multiple
            times.
15      # It will make it easier to debug.
16      tasks_to_do = [(i, *ingredient) for i, ingredient in
            enumerate(ingredients)]
17      tasks_being_done = []
18
19      # On crée un verrou commun pour tous les threads attribués
20      # aux clients.
21      lock = Lock()
22
23      start_time = None
24
25      class MasterService(rpyc.Service):
26          def exposed_receive_result(self, task, result):
27              log_master(f"{result} reçue", task[0], out=False)
28
29              with lock:
30                  tasks_being_done.remove(task)
31                  # Le résultat n'est pas très important dans notre
                        exemple, mais
32                  # en pratique il faudrait bien entendu le stocker.
33
```

```
            if not tasks_to_do and not tasks_being_done:
                end_time = time.time()

                    print("\nLa salade est prête ! Bonne dégustation !")

                    print(f"Temps de préparation : {end_time - start_time:.1f}s

        def exposed_give_task(self):
            nonlocal start_time
            if start_time is None:
                start_time = time.time()

            try:
                with lock:
                    task = tasks_to_do.pop()
            except IndexError:
                # Il n'y a plus de tâche en attente, on avertit
                    l'esclave.
                task = None
            else:
                with lock:
                    tasks_being_done.append(task)
                id_, fruit, _ = task
                log_master(f"1 {fruit} envoyée à la préparation",
                    id_, out=True)
            return task
    return MasterService


if __name__ == "__main__":
    run_master(prepare_distributed)
```

5.3. Maintenant, on teste !

Pour tester ce code, on peut se passer de plusieurs machines et se contenter de plusieurs processus. Commençons par démarrer le maître :

```
python master.py 18861
```

Notez qu'il vous faudra avoir installé RPyC. Maintenant, démarrons trois esclaves en parallèle dans un autre terminal :

```
##!/bin/bash

```

```
MASTER_HOST=localhost
MASTER_PORT=18861

python slave.py $MASTER_HOST $MASTER_PORT &
python slave.py $MASTER_HOST $MASTER_PORT &
python slave.py $MASTER_HOST $MASTER_PORT &
wait
```

Je vous présente ci-dessous le résultat mis en forme :

!(https ://jsfiddle.net/0r1nc35v/4/)

On remarque que le temps de préparation est environ le tiers du temps de préparation de la version séquentielle. Ça coïncide bien avec le fait que toutes les tâches sont indépendantes et réparties entre trois fois plus d'acteurs. Le surplus ($9 > 26/3$) est dû aux échanges réseau et aux opérations supplémentaires (gestion des listes `task_to_do` et `task_being_done` par exemple).

i

Pour effectuer une comparaison rigoureuse, il faudrait probablement inclure le démarrage du serveur RPC puisque c'est un coût supplémentaire réel de la version distribuée. Ici, les mesures de temps d'exécution servent juste à *illustrer* le gain de temps.

Nous avons désormais une version parallèle de notre salade de fruits. Fonctionnelle certes, mais très peu robuste. Dans la section suivante nous nous posons une question essentielle : que se passe-t-il en cas de panne ?

6. Alerte générale ! Une panne !

?

Que se passe-t-il en cas de panne ?

Cette question est centrale dans le domaine des systèmes distribués. Tout d'abord, il faut se demander ce que signifie « tomber en panne ». Vous vous doutez qu'on ne gérera pas de la même façon une machine qui part en fumée pour de bon et une qui reste en vie mais se met à dire n'importe quoi.

Dans le cadre de ce tutoriel, nous nous restreindrons au premier cas : les **pannes franches** (*crash*), qui sont les plus simples à gérer. Pour un aperçu des autres types, vous pouvez vous référer à l'article « Tolérance aux pannes » de Wikipédia.

Définition : panne franche

En cas de panne franche, l'acteur ne fait rien du tout.

Autrement dit, soit l'acteur se comporte correctement (pas de panne), soit il ne fait rien du tout (panne). En particulier, on fait l'hypothèse que l'acteur ne peut se comporter d'une manière inattendue : soit il ne répond pas à nos messages, soit ses réponses sont correctes. Par définition, quand un esclave subit une panne franche, il ne peut plus revenir à son état normal.

En pratique, il est très important de faire clairement ses hypothèses. Durant cette phase de spécification de son algorithme distribué, on sacrifie souvent de la robustesse (gérer le plus de types de pannes possible) au profit de la simplicité du système mis en place (implémentation, coût...). Bien évidemment, le compromis dépend du contexte : le niveau d'exigence n'est pas le même pour un jeu vidéo que pour une banque.

6.1. Observation d'une panne

Regardons un peu comment réagit notre programme distribué à une panne franche d'un esclave. Pour simuler une telle situation, nous sortons simplement de la boucle avec une certaine probabilité. Nous ajoutons également un paramètre au script pour déterminer si un esclave peut tomber en panne ou non :

```
1  import os
2  import random
3  import sys
4  import time
5
6  import rpyc
7
8  from shared import *
9
10 CRASH_PROB = 0.6
11
12
13 def create_connection():
14     master_addr = sys.argv[1]
15     master_port = int(sys.argv[2])
16     return rpyc.connect(master_addr, master_port)
17
18
19 def prepare_fruit(id_, fruit, t):
20     log_slave(f"1 {fruit} en préparation ({t}s)", id_,
           WORKING_LABEL)
21     time.sleep(t)
22     return f"1 {fruit} préparée"
23
24
25 def send_result(conn, task, result):
26     conn.root.receive_result(task, result)
27
28
29 def ask_task(conn):
```

```
    return conn.root.give_task()


def may_crash():
    try:
        return bool(sys.argv[3])
    except IndexError:
        return False


def run(conn):
    # `task` est un couple `(fruit, temps de préparation)`.
    task = ask_task(conn)
    may_crash_ = may_crash()

    while task:
        id_, fruit, t = task
        log_slave(f"1 {fruit} à préparer reçue", id_, IN_LABEL)
        prepared_fruit = prepare_fruit(id_, fruit, t)

        if may_crash_ and random.random() < CRASH_PROB:

                log_slave(f"alerte, une panne ! 1 {fruit} en préparation",
                id_, ERROR_LABEL)
            break

        log_slave(f"1 {fruit} prête envoyée", id_, OUT_LABEL)
        send_result(conn, task, prepared_fruit)
        task = ask_task(conn)


if __name__ == "__main__":
    run(create_connection())
```

Côté maître, on change juste un petit peu l'affichage :

```
## ...

def prepare_distributed(ingredients):

    # ...

    class MasterService(rpyc.Service):
        def exposed_receive_result(self, task, result):
            with lock:
                tasks_being_done.remove(task)
                # Le résultat n'est pas très important dans notre
                    exemple, mais
```

```
                # en pratique il faudrait bien entendu le stocker.

            tasks_being_done_formatted = [
                f"{task[1]} (T-{task[0]})"
                for task in tasks_being_done
            ]
            log_master(

                    f"{result} reçue. En cours : {', '.join(tasks_being_done_f
                task[0],
                IN_LABEL,
            )

            if not tasks_to_do and not tasks_being_done:
                end_time = time.time()

                    print("\nLa salade est prête ! Bonne dégustation !")

                    print(f"Temps de préparation : {end_time - start_time:.1f}

        # ...
```

Il est bien question ici d'une panne franche puisque l'esclave ou bien se comporte correctement ou bien ne fait rien du tout (quitter la boucle termine le processus par la même occasion). Pour démarrer les esclaves, on utilise ce script :

```
##!/bin/bash

CRASH=1
MASTER_HOST=localhost
MASTER_PORT=18861

python slave_crash.py $MASTER_HOST $MASTER_PORT $CRASH &
python slave_crash.py $MASTER_HOST $MASTER_PORT $CRASH &
python slave_crash.py $MASTER_HOST $MASTER_PORT $CRASH &
wait
```

Quand on exécute cette version du code, on obtient :

!(https ://jsfiddle.net/gyzeuowx/2/)

Comme la panne advient entre la réception de la tâche par l'esclave et l'envoi du résultat, le maître se retrouve à attendre ce dernier indéfiniment : le processus ne termine jamais. Notons que tant qu'il reste des esclaves en vie, le maître continue à distribuer des tâches sans souci. Par contre, dès que tous les esclaves ont péri, le maître devient un zombi.

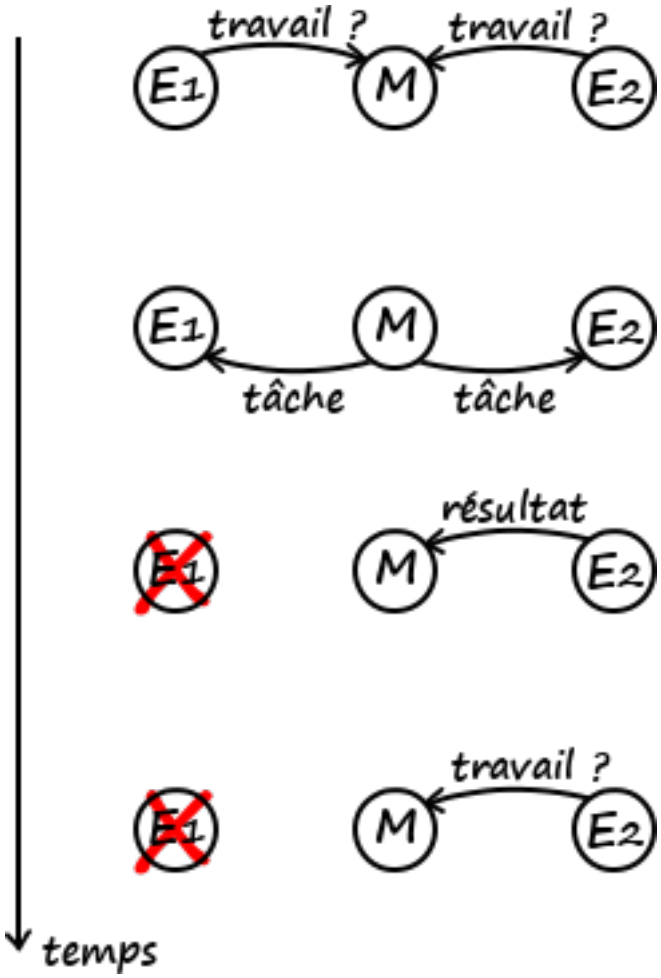

FIGURE 6. – E_1 reçoit une tâche mais meurt avant d'en retourner le résultat.
Avec notre code basique, le maître ne détecte pas cette disparition et attendra indéfiniment la réponse.
Notons que la panne de E_1 n'impacte pas la communication avec E_2.
Par contre, au moment où E_2 meurt et qu'il n'y a plus d'esclaves, le maître devient inactif.

Regardons ce qu'il se passe quand un esclave ne tombe jamais en panne :

```
##!/bin/bash

CRASH=1
MASTER_HOST=localhost
MASTER_PORT=18861

python slave_crash.py $MASTER_HOST $MASTER_PORT $CRASH &
python slave_crash.py $MASTER_HOST $MASTER_PORT $CRASH &
python slave_crash.py $MASTER_HOST $MASTER_PORT &
wait
```

On obtient :

!(https ://jsfiddle.net/okxezfjd/2/)

Autrement dit, l'esclave en vie continue à préparer des fruits jusqu'à ce qu'il n'en reste plus. Par contre, le maître ne pourra jamais servir la salade parce que les tâches distribuées aux esclaves en panne resteront à jamais marquées comme en cours (pour rappel, un esclave ayant subi une panne franche ne peut pas revenir à son état normal).

Mince alors ! Tout allait pour le mieux avant que ces pannes ne soient de la partie. Dans la section suivante, nous verrons une manière basique de régler le problème.

7. Un remède : le timeout

Une manière simple de régler ce problème est de définir un *timeout* pour les réponses : si l'esclave n'a pas donné signe de vie avant N unités de temps, on considère qu'il a eu un problème et déplace sa tâche dans la pile « en cours » à la pile « à faire ».

Mais combien de temps faut-il attendre ? La communication entre le maître et un esclave fait intervenir un troisième acteur : le canal. Et la performance de ce dernier est impactée notamment par la charge qu'il subit. Un exemple est le texto de bonne année reçu quelques heures voire jours plus tard. Autrement dit, un long délai de réponse ne découle pas nécessairement d'une panne de l'esclave et peut être dû à un canal défaillant.

Et, là encore, il nous faut définir ce que nous entendons par « défaillant ». Comme pour les pannes, il existe plusieurs modèles de canaux, plus ou moins faciles à gérer. La représentation la plus simple, qui fait également le plus d'hypothèses et est donc la moins générique, est celle du canal parfait :

Définition : canal parfait

Un canal est parfait s'il transmet correctement (sans corruption) les messages dans un délai fini connu.

Dans la suite, nous ferons cette hypothèse forte de canal parfait. Ainsi, nous supposons que tous les messages parviennent à leur destinataire tels qu'envoyés dans un délai maximal connu T.

Je vous encourage à étendre le code pour gérer les pannes franches des esclaves dans le cadre d'un canal parfait. Je vous présente une manière de faire ci-dessous.

7.1. Un code robuste aux pannes

Pour les esclaves, le code est très similaire au précédent. Le seul changement est l'ajout d'un état d'attente : quand le maître n'a pas de travail pour nous mais que des tâches sont en cours de traitement par d'autres esclaves, nous attendons au cas où ces esclaves tombent en panne et que leur tâche devienne de nouveau disponible.

```python
## ...

WAITING_DELAY = 3

## ...

def run(conn):
    # `task` est un couple `(fruit, temps de préparation)`.
    task = ask_task(conn)
    may_crash_ = may_crash()

    while task is not None:
        # Quand `task` est un tuple vide, on n'a pas de fruit à préparer tout
        # de suite mais il est possible qu'on en ait dans le futur.
        if not task:
            # Pour éviter de surcharger le serveur avec des demandes, on attend
            # un peu avant de redemander une tâche.
            time.sleep(WAITING_DELAY)
            task = ask_task(conn)
            continue

        id_, fruit, t = task
        log_slave(f"1 {fruit} à préparer reçue", id_, IN_LABEL)
        prepared_fruit = prepare_fruit(id_, fruit, t)

        if may_crash_ and random.random() < CRASH_PROB:
            log_slave(f"alerte, une panne ! 1 {fruit} en préparation", id_, ERROR_LABEL)
            break

        log_slave(f"1 {fruit} prête envoyée", id_, OUT_LABEL)
        send_result(conn, task, prepared_fruit)
        task = ask_task(conn)

## ...
```

Côté maître, on a :

```python
## ...

def pending_to_free(task, pending_list, free_list, lock):
    """Déplace une tâche en cours d'exécution dans la pile des tâches à exe
    Cette fonction est appelée quand l'esclave en charge de la tâche n'a pas do
```

```
    signe de vie pendant un certain temps.
    """
    try:
        # En Python, les listes ne sont pas copiées quand elles sont passées en
        # paramètre, donc l'instruction ci-dessous modifie bien la liste lue par
        # le maître.
        with lock:
            pending_list.remove(task)
        log_master(
            f"remet la {task[1]} dans le panier à préparer",
            task[0],
            WORKING_LABEL
        )
    except ValueError:
        # La tâche n'est déjà plus dans la liste, on ne fait rien.
        pass
    else:
        with lock:
            free_list.append(task)


def prepare_distributed(ingredients):
    # ...

    class MasterService(rpyc.Service):
        def exposed_receive_result(self, task, result):
            with lock:
                try:
                    tasks_being_done.remove(task)
                except ValueError:
                    # On rentre ici si le délai a été atteint alors que
                    # l'esclave est toujours en vie et répond en retard. Dans ce
                    # cas, sa tâche a été redistribuée donc on ne fait rien.
                    return

            tasks_being_done_formatted = [
                f"{task[1]} (T-{task[0]})"
                for task in tasks_being_done
            ]
            if not tasks_being_done_formatted:
                tasks_being_done_formatted = ["rien"]
            log_master(

                    f"{result} reçue. En cours : {', '.join(tasks_being_done_f
                task[0],
```

```
50                  IN_LABEL,
51              )
52
53              if not tasks_to_do and not tasks_being_done:
54                  end_time = time.time()
55
                        print("\nLa salade est prête ! Bonne dégustation !")
56
                        print(f"Temps de préparation : {end_time - start_time:.1f}s
57
58          def exposed_give_task(self):
59              nonlocal start_time
60              if start_time is None:
61                  start_time = time.time()
62
63              try:
64                  with lock:
65                      task = tasks_to_do.pop()
66              except IndexError:
67                  if not tasks_being_done:
68                      # Il n'y aura plus de tâche à faire, l'esclave
                            peut terminer.
69                      task = None
70                  else:
71                      # Il n'y a pas de tâche disponible pour le
                            moment mais il
72                      # pourra y en avoir dans le futur.
73                      task = tuple()
74              else:
75                  with lock:
76                      tasks_being_done.append(task)
77                  id_, fruit, _ = task
78                  log_master(f"1 {fruit} envoyée à la préparation",
                        id_, OUT_LABEL)
79                  # On lance un décompte en parallèle. La fonction
                        `pending_to_free`
80                  # sera appelée avec les arguments `args` dans
                        `timeout` secondes.
81                  # Si l'esclave a retourné le résultat d'ici là, la
                        tâche aura déjà été
82                  # enlevée de la liste par `exposed_receive_result`
                        et l'appel
83                  # à la fonction n'aura aucun effet.
84                  Timer(
85                      timeout,
86                      pending_to_free,
87                      args=(task, tasks_being_done, tasks_to_do,
                            lock)
88                  ).start()
89
```

```
            return task

    return MasterService

## ...
```

Quand on exécute le code avec un esclave ne tombant jamais en panne, on obtient cela :

!(https ://jsfiddle.net/m0yxw2z4/2/)

Notre système distribué est donc robuste à des pannes franches des esclaves dans le cadre d'un canal parfait. Notons qu'en pratique, il faudrait tester notre code plus rigoureusement pour s'assurer que c'est bien le cas, c'est-à-dire qu'il répond aux spécifications. Remarquons que si le maître tombe, le système réparti s'effondre.

En outre, ce code ne gère pas le cas où tous les esclaves tombent en panne (le maître se retrouve alors à attendre indéfiniment). Vous pouvez remédier à cela à titre d'exercice. Une manière de faire :

Indice 1

Indice 1

Indice 2

Contenu masqué n°2

8. Conclusion

Ce tutoriel est terminé. Avec un peu de chance, il aura attisé votre intérêt pour les systèmes distribués. Gardez en tête qu'il ne s'agit que d'une introduction informelle, n'ayant pas pour objectif de faire de vous un expert du domaine.

Notre implémentation de la salade de fruits répartie est réduite à sa plus simple forme puisque nous n'avons fait que des hypothèses peu contraignantes :

— Les pannes franches sont les plus faciles à gérer ;
— Le canal parfait est tel qu'il ne nous pose aucun problème.

En pratique, il faut s'assurer que ces hypothèses soient réalistes vis-à-vis du contexte. Par exemple, on ne peut raisonnablement pas supposer un canal parfait dans le cadre d'objets connectés en pleine cambrousse.

Pour poursuivre votre apprentissage, je vous recommande ces vidéos de Vivien Quema sur la plateforme Wandida de l'EPFL. Cet article , en anglais, est aussi très accessible et introduit

des concepts importants. Je vous invite enfin à étendre l'architecture et le code ci-dessus pour gérer :

1. Des tâches dynamiques. Par exemple, concocter une recette plus élaborée dans laquelle l'ordre des tâches importe (préparer la pâte et découper les pommes *avant* de répartir les secondes sur la première puis de saupoudrer de farine mélangée à du beurre)[4].
2. Plusieurs maîtres (au cas où un tombe).

Je suis ouvert à tout retour, de préférence constructif, dans les commentaires ou par message privé. Aussi, n'hésitez évidemment pas à poser les questions que vous pourriez avoir.

Je remercie vivement @informaticienzero pour la validation de ce contenu et @nohar pour les retours de qualité.

Contenu masqué

Contenu masqué n°1 : Indice 1

Définir un délai maximal pour les demandes de tâche : si du travail n'a pas été sollicité avant N unités de temps, considérer qu'il n'y a plus d'esclave disponible, afficher un message et éteindre le serveur RPC. Retourner au texte.

Contenu masqué n°2

Au démarrage du maître, lancer un *thread* chargé d'effectuer le décompte avant l'extinction du serveur RPC. Dès qu'un esclave se manifeste, reprendre le décompte du début. Attention à l'exclusion mutuelle. Retourner au texte.

4. Vous aurez reconnu la recette du crumble aux pommes.