



CY TECH
GRANDE ÉCOLE DES SCIENCES DE L'INGÉNIERIE

Programme de Monitoring : Surveillance de calcul

Rapport d'analyse

Lilian NARETTO
Ryan HOUSSENE
Paul LE DILY
Bilal TAALBI
Corentin BRILLANT

28 mars 2021

Table des matières

1	L'architecture adoptée / Diagrammes	1
1.1	Diagrammes	1
1.1.1	Diagrammes de cas d'utilisation	1
1.1.2	Diagrammes de séquence	2
1.2	Schémas explicatifs pour la réalisation	3
1.2.1	Schémas	3
1.2.2	Algorithmes	4
2	Moniteur & Calculateur	6
2.1	Moniteur	6
2.1.1	Squelette	6
2.1.2	Thread dédié à un calculateur	6
2.2	Calculateur	7
3	Evil Monkey	8
4	La politique de gestion de l'échec	9
5	Les tests envisagés et leur planification	10
6	Les ambiguïtés identifiées dans le sujet et les choix faits pour les lever	12
6.1	Le rapport des calculateurs	12
6.2	la synthèse du moniteur	12
6.3	Le choix du mode de communication entre un calculateur et le moniteur	12
6.4	La réception des rapports du côté moniteur	12
6.5	L'état d'une tâche	13
6.6	La gestion continue des calculateurs entrants	13
6.7	Le lancement des différents processus-calculateurs	13
6.8	La mort d'un calculateur	13
7	Organisation du code	14
7.1	Les fichiers	14
7.1.1	Les structures du projet	14
7.1.2	Launcher.c	14
7.1.3	Monitor.c	14
7.1.4	Processus.c	14
7.1.5	Thread_functions.c	14

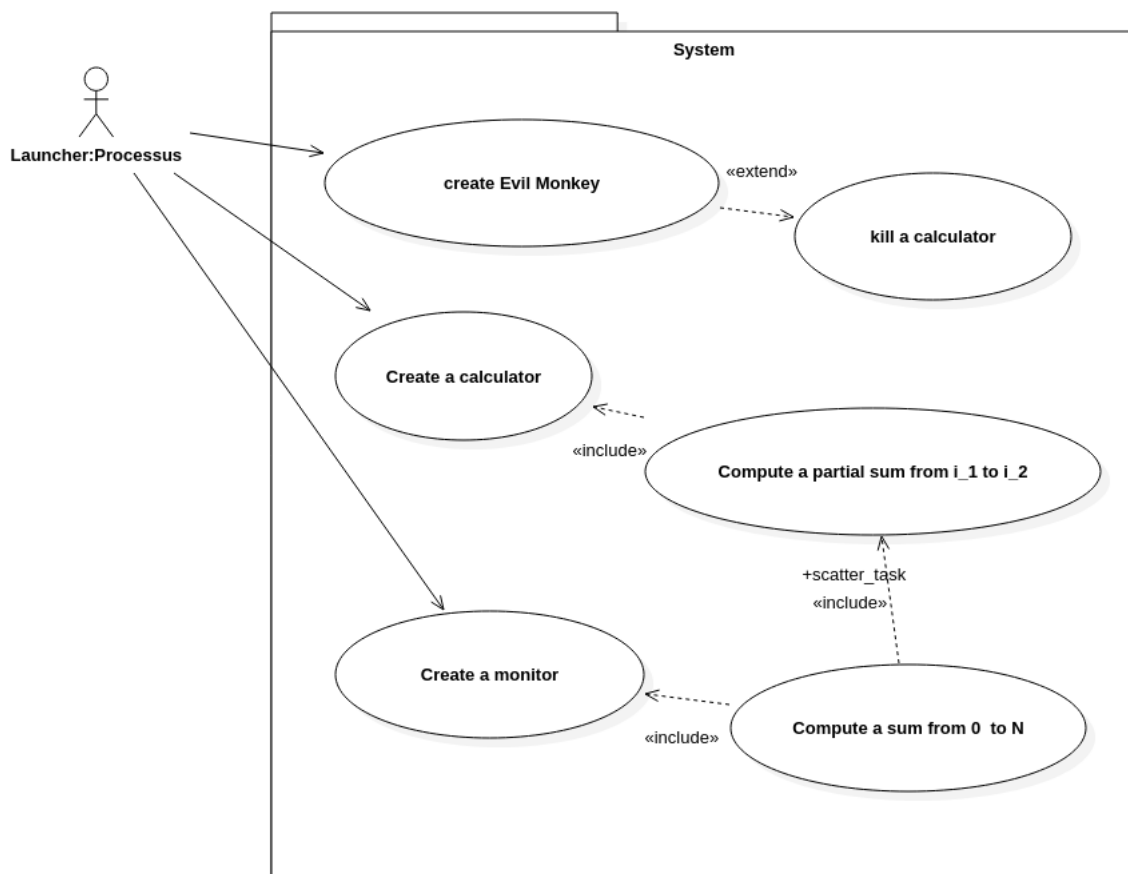
8	Manuel d'utilisation	15
9	La répartition du travail entre les membres du projet	18

Chapitre 1

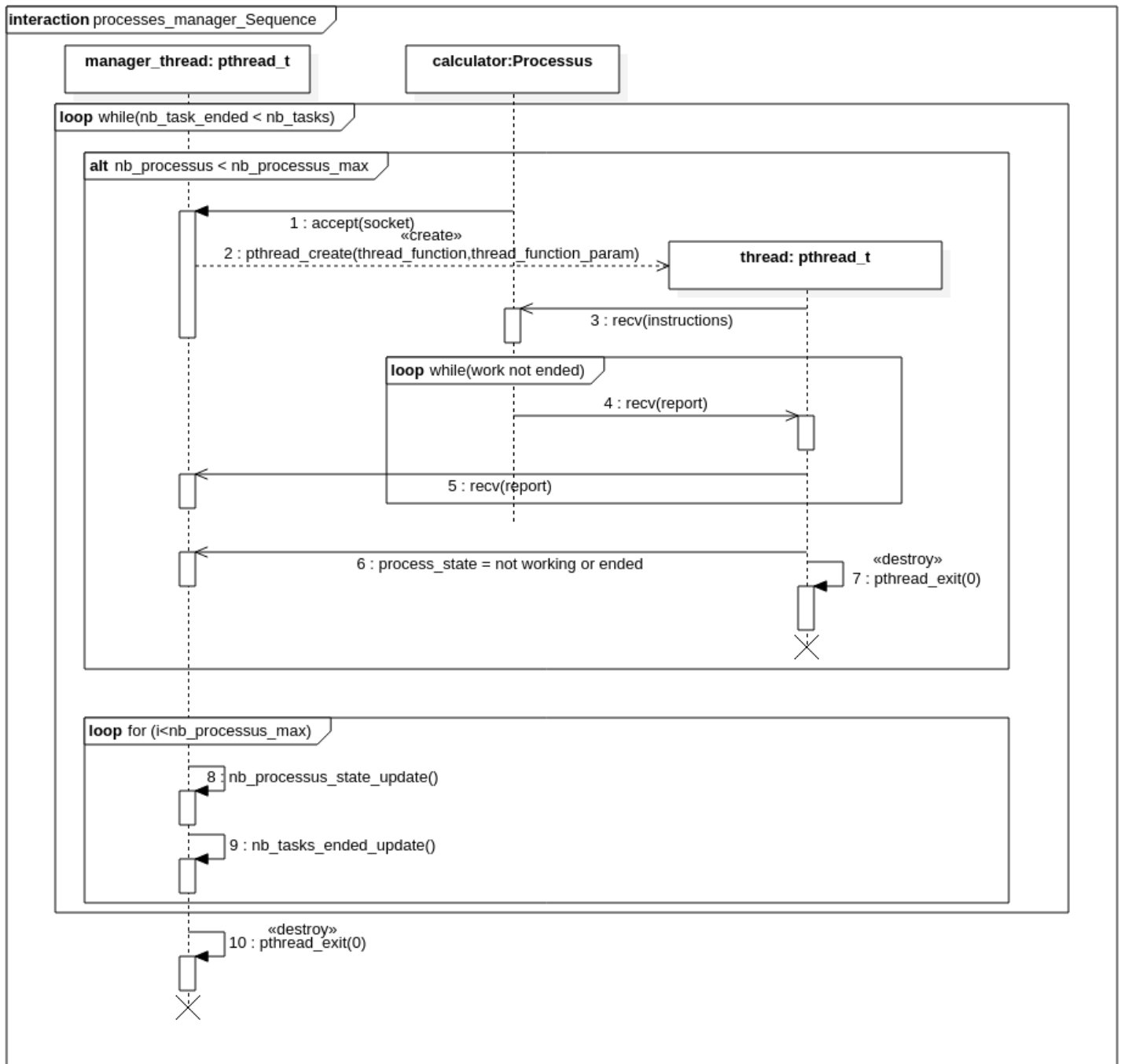
L'architecture adoptée / Diagrammes

1.1 Diagrammes

1.1.1 Diagrammes de cas d'utilisation

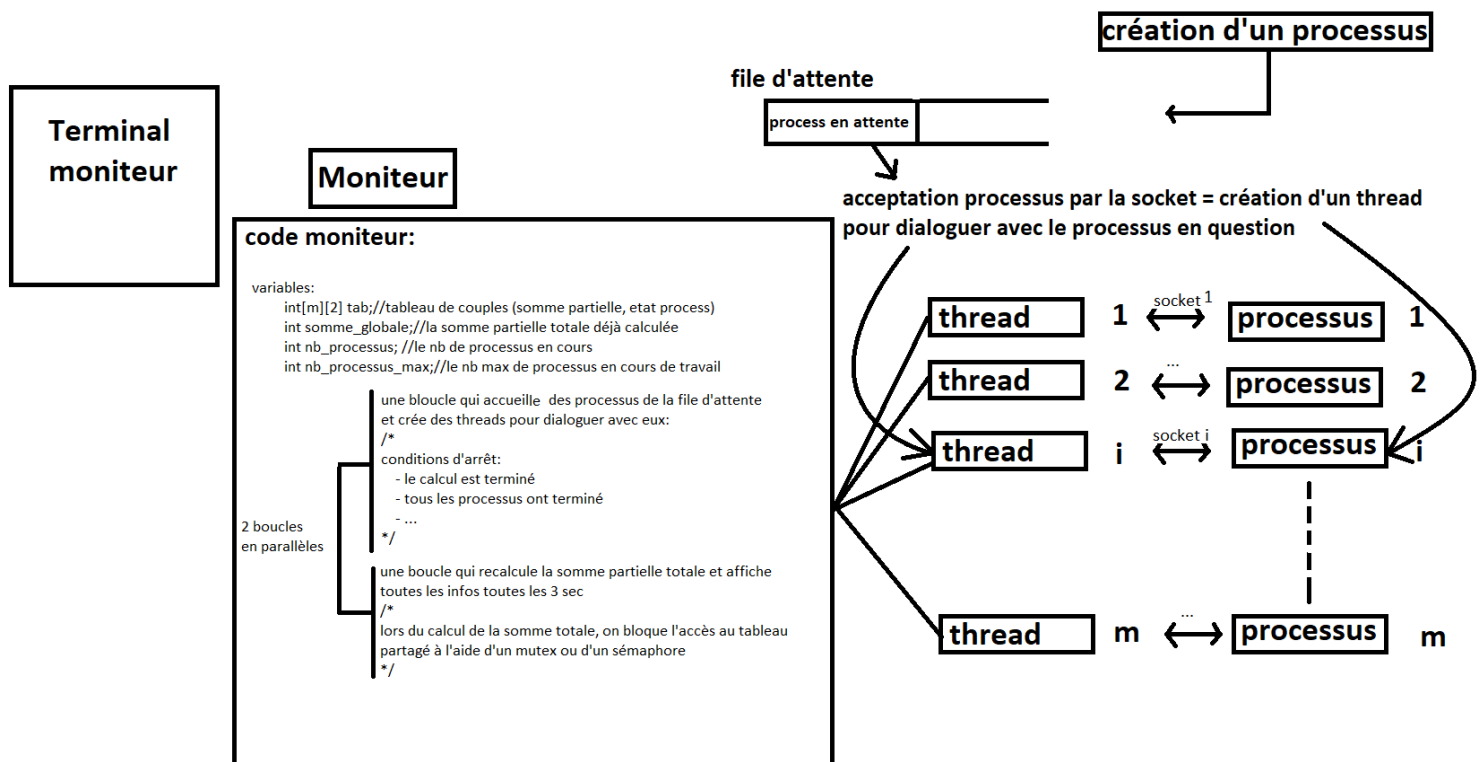


1.1.2 Diagrammes de séquence



1.2 Schémas explicatifs pour la réalisation

1.2.1 Schémas



Le moniteur et les calculateurs dialogueront sur le réseau interne de la machine **127.0.0.1** et sur le port **4206**

1.2.2 Algorithmes

Processus i

```
{
Ouverture d'une socket TCP
Tentative de connexion à 127.0.0.1 sur le port X
    Si la tentative échoue le processus termine
Sinon
On récupère les instructions de calcul envoyées par le moniteur via recv() et on les stocke dans une
variable
On lance ensuite 2 threads en parallèles
1 {
    Le premier thread récupère les instructions qu'on lui a passé lors de sa création
    Il calcule sans cesse jusqu'à ce qu'il termine le job (on le ralentit avec un sleep(1))
    Puis pthread_exit(0) pour terminer
}
2 {
    Le 2eme thread est chargé d'envoyer des rapports continument au moniteur toutes les 2 sec
    On envoie les rapports avec send() par exemple
}
Pour fermer le processus on attend que que le premier thread ait terminé le travail avec
pthread_join() par exemple
}
```

Thread i

Le thread i du moniteur a été créé pour dialoguer avec un nouveau calculateur qui doit se charger d'une partie du calcul, c'est un thread dédié qui terminera lorsque le calculateur sera tué ou aura terminé son calcul

```
{
- On envoie les instructions de calcul au calculateur avec send() par exemple
- Les instructions sont composées de (la somme partielle déjà calculée, le nombre de depart à
sommer, le dernier nombre à sommer)
- Lorsque les instructions sont envoyées, on démarre un compteur de temps pour savoir depuis
quand le processus travail
    On lance une boucle infinie pour recevoir les rapports réguliers du processus
    {
        - On recoit un rapport avec recv() par exemple (on utilise un appel non bloquant) et on met à
jour la case du tableau correspondante
        - Si il n'y a rien à lire dans le buffer on incrémente un compteur
        - Si le compteur atteint une certaine valeur (cad on a pas reçu de rapport depuis un moment)
on considère que le processus est mort
        - on ralentit la boucle avec des sleep() pour ne pas consommer trop de ressources
    }
    le thread termine avec pthread_exit(0)
}
```

Thread du moniteur qui accueille les calculateurs entrants et crée un thread i dédié pour dialoguer avec un calculateur i

```
{
  On lance une boucle infinie
  {
    -on ralentit la boucle avec un sleep()
    -on regarde dans le tableau s'il y a une place vacante pour un processus calculateur
    -->Si c'est le cas on attend d'accepter un nouveau processus avec accept()
    - lorsqu'on accepte un calculateur i on lance un thread i auquel on passe la socket pour
    continuer de dialoguer avec le calculateur (le comportement du thread i est décrit dans le slide 2)
    -->Sinon on vérifie si un calculateur s'est interrompu avant la fin de sa tâche et n'a pas
    libéré une place pour un autre calculateur qui reprendra sa suite
  }
}
```


Chapitre 2

Moniteur & Calculateur

2.1 Moniteur

2.1.1 Squelette

Le moniteur est chargé de calculer une somme de 0 à un certain nombre N . Pour ce faire, il segmente l'intervalle $[0, N]$ en autant de sous-intervalles que le moniteur peut accepter de calculateurs. Par exemple, si le calcul est une somme de 0 à 1000 et que le moniteur accepte 10 calculateurs, il divise le calcul en $[0, 99], [100, 199], \dots, [900, 1000]$. Le dernier intervalle prend le reste de la division euclidienne de N par le nombre de calculateurs. Le moniteur construit donc un tableau avec dans la case i les données(intervalle) de la tâche i . Chaque case est organisée selon la structure **tab_cell** décrite dans la sous-section 7.1.5 Les structures du chapitre Organisation du code de telle sorte que notre tableau de tâches est un tableau de **tab_cell**.

Pour distribuer les tâches de ce tableau aux différents calculateurs qui tentent de se connecter, on procède comme suit :

Si le maximum de connexions n'est pas atteint, le moniteur attend qu'un nouveau calculateur se connecte. Lorsqu'un nouveau calculateur s'est connecté, on lui dédie un nouveau thread du côté du moniteur. On regarde dans le tableau l'indice de la première case dont la tâche n'est pas terminée ou n'est pas en train d'être réalisée par un calculateur. On donne un pointeur vers cette case au thread dédié puis ce dernier envoie les instructions au calculateur et continue d'écouter les retours du calculateur qui auront lieu toutes les 2 secondes.

2.1.2 Thread dédié à un calculateur

Le thread s'occupait d'envoyer les informations au calculateur et de vérifier si le calcul s'effectuait bien. La fonction `thread_i_function()` était composée de deux parties. Une première partie qui s'occupait d'envoyer les informations au calculateur avec la fonction `send_info()` qui utilisait la fonction `send()` pour envoyer les informations. Et une seconde partie qui récupère le résultat pour le mettre à jour dans le tableau.

2.2 Calculateur

Nous avons délégué des sommes partielles à calculer à plusieurs processus. Pour ce faire chaque processus se doit de calculer une somme à partir d'un indice de début et jusqu'à un indice de fin. Nous récupérons en premier lieu un buffer sous forme de tableau d'entiers de taille 3 contenant respectivement le résultat de la somme partielle actuelle, l'indice auquel la somme partielle en est et l'indice de fin de la somme partielle.

Le processus lance alors 2 threads :

- le premier s'occupe de continuer à calculer la somme en incrémentant toutes les secondes et met à jour les données dans le buffer (somme partielle et indice de début).
- le second s'occupe d'envoyer un rapport au moniteur toutes les 2 secondes contenant la somme partielle calculée et l'indice auquel la somme se trouve au moment de l'envoi.

Les 2 threads s'exécutent en parallèle et ont accès aux mêmes données.

Chapitre 3

Evil Monkey

Le processus Evil Monkey a pour rôle de simuler une perturbation dans le système. Ici, une perturbation est la mort inattendu d'un processus calculateur. Pour ce faire, on donne au processus Evil Monkey un accès à l'ensemble des calculateurs lancés par le programme lanceur et toujours en cours d'exécution, ainsi qu'un temps t en secondes. De cette manière le processus Evil Monkey va tuer aléatoirement (ici avec une loi uniforme) un des calculateurs toutes les t secondes.

Par défaut, le programme est réglé pour lancer le programme Evil Monkey 20 secondes à partir du lancement du système, puis il continue de tuer toutes les 20 secondes.

Chapitre 4

La politique de gestion de l'échec

Un échec est défini par la rupture de communication d'un calculateur avec le moniteur avant que le communicateur en question n'ait pu achever la tâche qui lui avait été confiée. Soit parce que le processus du calculateur a été tué, soit parce que la connexion a été rompue. Nous avons choisi de gérer les échecs de la manière suivante :

Le moniteur reçoit des rapports réguliers de la part du calculateur (toutes les 2 secondes exactement) mais lorsqu'un rapport n'a pas été reçu on incrémente un compteur qui comptabilise le nombre de rapports non reçus consécutifs. Finalement, si le compteur atteint une valeur prédéfinie, le moniteur considère que le calculateur est hors d'usage et il clôt la communication pour redistribuer le travail restant, associé à cette tâche, à un autre calculateur. Pour illustrer cette séquence, se référer au diagramme 1.1.2.

Chapitre 5

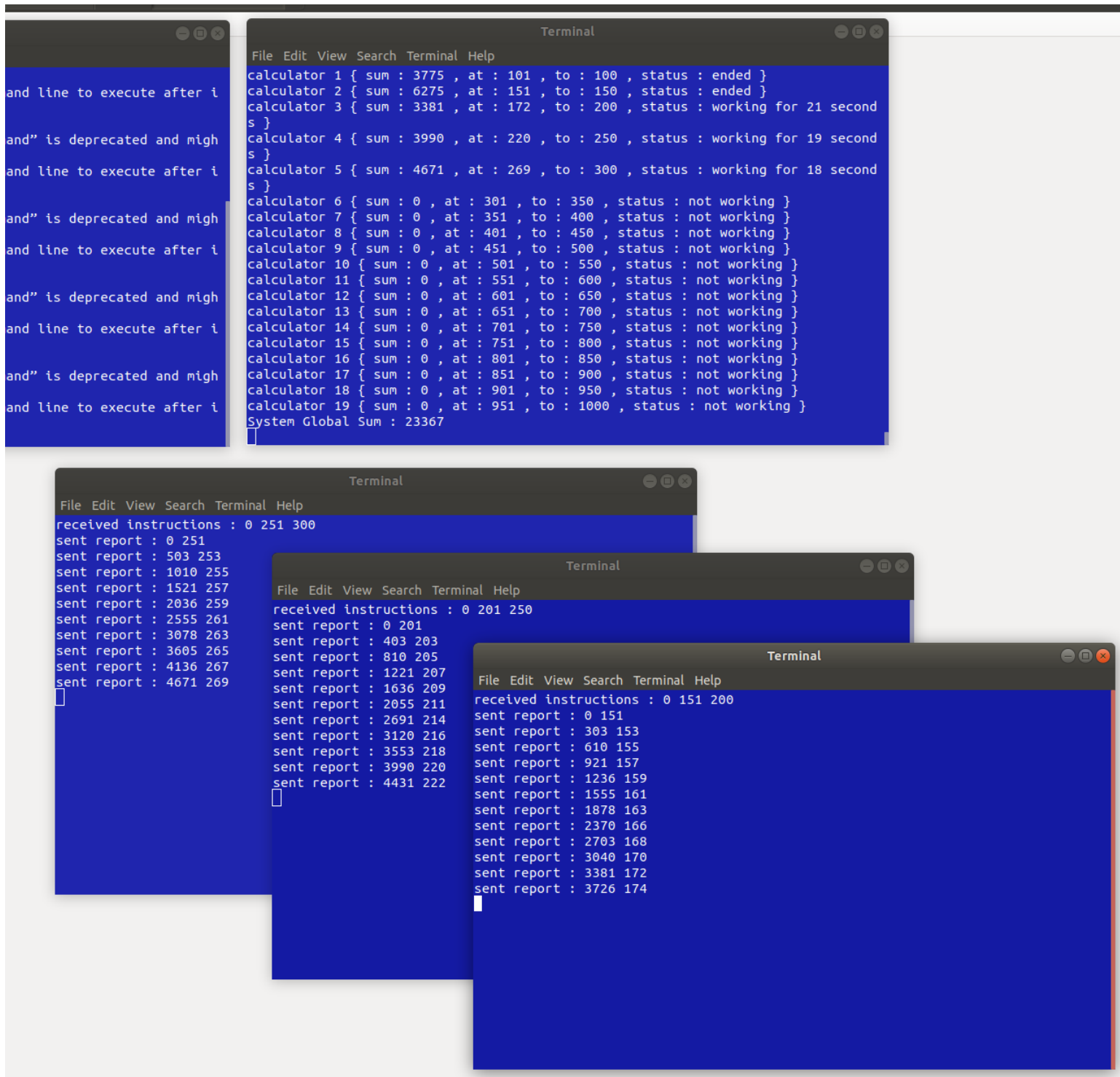
Les tests envisagés et leur planification

Etant donné que le projet s'est réalisé sur une relativement courte période et que le code a été implémenté en collaboration, les tests n'ont pas eu de réelle planification. On peut seulement distinguer une phase de début des tests. En effet, comme le projet est basé sur une communication Client/Serveur, nous avons dû attendre d'avoir une version assez aboutie du moniteur (qui joue le rôle de serveur) et du calculateur (qui joue le rôle de client) pour pouvoir tester les codes. Les principaux tests étaient les suivants :

- vérifier que les messages envoyés étaient correctement reçus
- vérifier que le calcul était juste
- vérifier la synchronisation entre les threads
- vérifier qu'il n'y ait pas de threads zombies

Notre système devait être lancé manuellement. Nous avons fait le choix de lancer un terminal par calculateur pour pouvoir suivre plus aisément les affichages de chacun et du moniteur. Ensuite, une fois l'implémentation du moniteur et du calculateur terminée, nous avons pu rassembler les processus dans un unique terminal et n'afficher que la synthèse du moniteur (comme si nous n'avions pas accès aux calculateurs distants). Nous avons entamé l'implémentation du Evil Monkey et du programme lanceur du projet. Dans la version finale du projet, le programme Evil Monkey affiche donc son activité dans le même terminal que le moniteur.

Voici une capture d'écran du projet lancé manuellement :



Chapitre 6

Les ambiguïtés identifiées dans le sujet et les choix faits pour les lever

6.1 Le rapport des calculateurs

Le sujet nous indique que le rapport des calculateurs doit être constitué de la somme partielle calculée et de son temps d'exécution. Ce qui ne donne pas d'indications sur l'état de la tâche en cours. On ne sait pas s'il s'agit de la somme partielle finale ou d'une somme partielle intermédiaire de l'intervalle à sommer. Nous avons décidé de rajouter dans le rapport du calculateur l'indice du dernier nombre ajouté à la somme partielle. Ceci permet de redistribuer le travail restant sur cette tâche à un autre calculateur si celui-ci était tué.

6.2 la synthèse du moniteur

Le sujet précise que le moniteur doit faire une synthèse toutes les 3 secondes. Dans notre architecture, chaque thread dédié à son calculateur est chargé de mettre à jour la case mémoire pour cette tâche (on rappelle que celle-ci est stockée sous la structure `tab_cell`). Chaque thread dédié fera donc sa propre synthèse. L'affichage global sera effectué par un thread en parallèle auquel on aura passé un pointeur vers le tableau des tâches, et ce toutes les 3 secondes.

6.3 Le choix du mode de communication entre un calculateur et le moniteur

Le sujet nous laissant libres de choisir un mode de communication nous avons choisi d'utiliser une communication de type TCP pour les raisons suivantes. Le calculateur étant censé envoyer des rapports réguliers il doit rester en liaison permanente avec le moniteur, il s'agit d'un mode connecté.

6.4 La réception des rapports du côté moniteur

Le sujet précisait qu'un rapport devait être envoyé toutes les 2 secondes mais il ne précisait pas la fréquence de lecture du buffer du côté du moniteur. Nous avons donc choisi que le thread

dédié à un calculateur regarderait s'il a reçu un rapport toutes les deux secondes. Pour garder une cohérence dans le buffer, il était impératif de le lire plus souvent qu'on n'y écrit.

6.5 L'état d'une tâche

L'ajout du problème Evil Monkey nous oblige à trouver des solutions de secours. En effet, Evil Monkey a comme tâche de tuer les calculateurs aléatoirement un par un. Pour éviter que le programme Evil Monkey ne détruise toutes les tâches nous avons ajouté à la structure `tab_cell` une variable `status` qui représente le status de chaque tâche :

- 2 : définit une tâche qui a été terminée correctement.
- 1 : définit une tâche qui est en cours.
- 0 : définit une tâche en pause qui n'est affectée à aucun processus. On peut ainsi savoir quelle tâche donner à un nouveau calculateur.

6.6 La gestion continue des calculateurs entrants

Le sujet ne précisait pas si nous devions accepter des calculateurs jusqu'à un nombre maximum ou si nous devions accepter des calculateurs jusqu'à ce que le calcul total soit terminé. Dans la version finale, nous avons opté pour la deuxième solution. Une boucle reste en permanence à l'écoute d'un calculateur en demande de connexion, ainsi on pioche un processus en attente dans la file d'attente de la socket du moniteur.

6.7 Le lancement des différents processus-calculateurs

Le sujet ne décrivait pas comment lancer les calculateurs qui sont supposés être sur des machines distantes. Nous avons donc implémenté dans le programme lanceur un moyen de lancer les calculateurs. Dans la version finale du projet le lancement d'un calculateur se fait en cliquant sur Enter.

6.8 La mort d'un calculateur

Le sujet n'expliquait pas clairement comment gérer la mort d'un calculateur. Nous avons décidé de faire comme ce qui est expliqué dans la politique de gestion de l'échec(4)

Chapitre 7

Organisation du code

7.1 Les fichiers

7.1.1 Les structures du projet

Le fichier "structures.h" définit les structures nécessaires au projet. La structure `tab_cell` qui est composée de la somme globale et partielle. Et d'autres éléments permettant d'effectuer les calculs en parallèle.

7.1.2 Launcher.c

Lanceur qui exécute le moniteur et les différents calculateurs.

7.1.3 Monitor.c

Crée les différents processus et affiche l'état des différents calculateurs. Crée plusieurs threads dont un qui s'occupe de plusieurs processus et un qui montre un rapport de l'état du système.

7.1.4 Processus.c

Calcule la somme partielle et envoie sa valeur au moniteur.

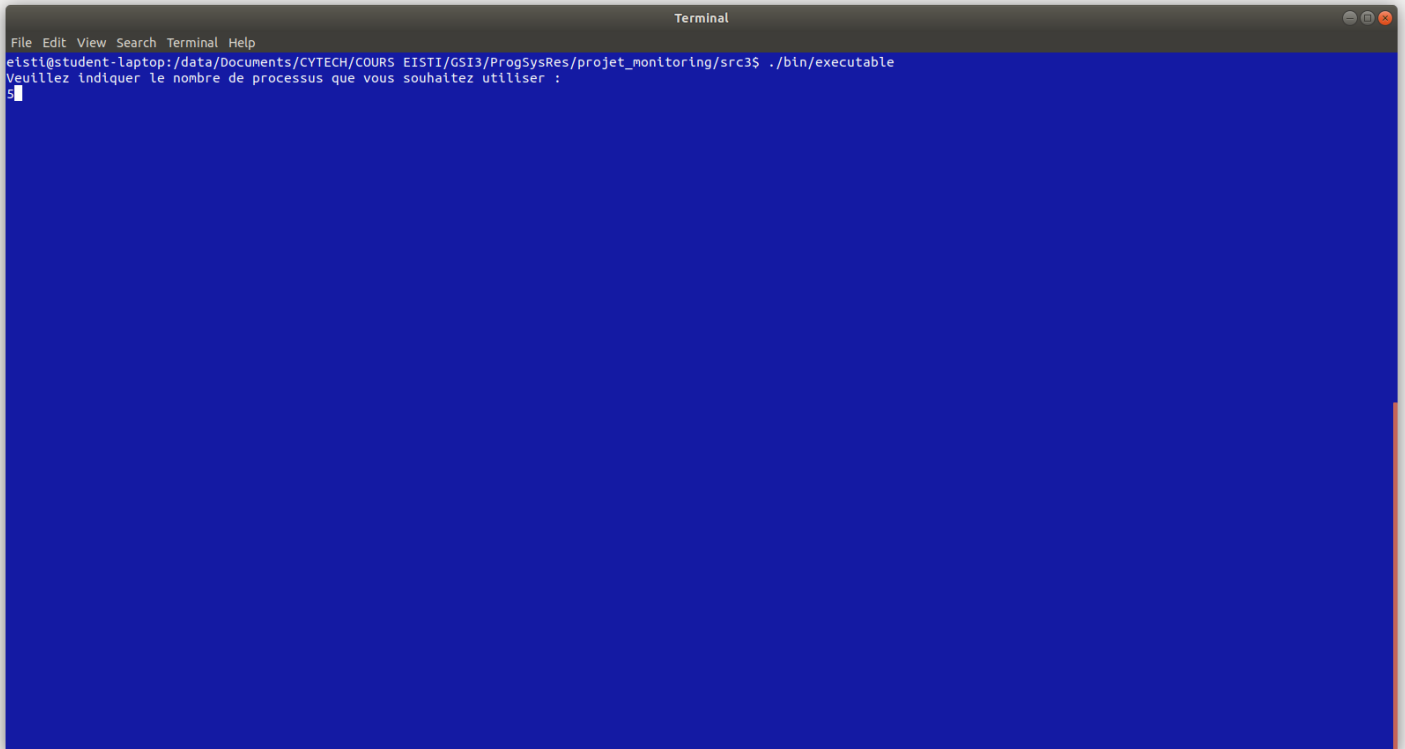
7.1.5 Thread_functions.c

Chapitre 8

Manuel d'utilisation

Le mode d'emploi pour lancer le projet est indiqué dans fichier README.txt. Ici nous présentons l'affichage obtenu dans la console et l'utilisation une fois que le programme est lancé.

La première étape après avoir lancé le programme est de saisir le nombre maximum de calculateurs que le moniteur pourra utiliser pour réaliser son calcul. Comme ci-dessous :

A screenshot of a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Search", "Terminal", and "Help". The terminal shows the command `eisti@student-laptop:/data/Documents/CYTECH/COURS EISTI/GSI3/ProgSysRes/projet_monitoring/src3$./bin/executable` being executed. Below the command, the text "Veuillez indiquer le nombre de processus que vous souhaitez utiliser :" is displayed, followed by a cursor on a new line. The rest of the terminal area is filled with a solid blue color, likely representing a full-screen display or a large output area.

Ensuite vous verrez le moniteur afficher sa synthèse toutes les 3 secondes :

```
Terminal
File Edit View Search Terminal Help
eisti@student-laptop:/data/Documents/CYTECH/COURS EISTI/GSI3/ProgSysRes/projet_monitoring/src3$ ./bin/executable
Veuillez indiquer le nombre de processus que vous souhaitez utiliser :
5
Nombre de processus calculateurs : 5
Somme totale partielle calculée : 0
--PROCESSUS 0
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 1
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 2
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 3
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 4
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
█
```

Après 20 secondes, le programme Evil Monkey démarrera et commencera à tuer des calculateurs et nous en informera dans la console :

```
Terminal
File Edit View Search Terminal Help
--Temps : 0
--Etat : 0
--PROCESSUS 3
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 4
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
-----Evil Monkey--looking for a victim among 2 processes
-----Evil Monkey--I've killed 1 since monitor started
Nombre de processus calculateurs : 5
Somme totale partielle calculée : 3258
--PROCESSUS 0
--somme partielle calculée : 153
--Temps : 18
--Etat : 1
--PROCESSUS 1
--somme partielle calculée : 3105
--Temps : 14
--Etat : 1
--PROCESSUS 2
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 3
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 4
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
█
```

Pour lancer de nouveaux calculateurs, il suffit de cliquer sur Enter, le processus créé tentera automatiquement de communiquer avec le moniteur via sa socket TCP :

```
Terminal
File Edit View Search Terminal Help
--Etat : 0
--PROCESSUS 2
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 3
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 4
--somme partielle calculée : 0
--Temps : 0
--Etat : 0

Nombre de processus calculateurs : 5
Somme totale partielle calculée : 66
--PROCESSUS 0
--somme partielle calculée : 66
--Temps : 12
--Etat : 1
--PROCESSUS 1
--somme partielle calculée : 0
--Temps : 0
--Etat : 1
--PROCESSUS 2
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 3
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
--PROCESSUS 4
--somme partielle calculée : 0
--Temps : 0
--Etat : 0
```

here we launched 3 calculators

Attention : Si l'on interrompt le programme manuellement à l'aide de controle+C, le programme n'aura pas le temps de fermer la socket utilisée. Il faudra attendre que le système ferme de lui-même la socket avant de pouvoir relancer le programme.

Chapitre 9

La répartition du travail entre les membres du projet

Architectes : Corentin Brillant, Paul Le Dily, Bilal Taalbi

Développeurs : Corentin Brillant, Paul Le Dily, Bilal Taalbi, Ryan Houssene, Lilian Naretto

Rédaction du rapport : Corentin Brillant, Paul Le Dily, Bilal Taalbi, Ryan Houssene, Lilian Naretto

Documentation(Diagrammes) :Corentin Brillant, Lilian Naretto