Master 1 Informatique - Programmation Répartie

Exclusion mutuelle basée sur l'algorithme de Naïmi-Tréhel

Hugo Maitre
Yuyang Wang
Adrien Linares

Enseignants :
Pr. Hinde Bouziane
Pr. Rodolphe Giroudeau

4 janvier 2022

Table des matières

1 Introduction

Nous avons pu mettre à profit durant ce projet les connaissances acquises au cours des différentes séances de TD et de TP afin d'implémenter un algorithme assurant l'exclusion mutuelle. Nous avions eu le choix entre trois algorithmes distribués et notre choix s'est fait avec consensus sur celui de Naïmi-Tréhel. [1]

L'exclusion mutuelle est un terme qu'on retrouve assez souvent dans le monde des systèmes répartis. Ce dernier établit que seul un processus puisse être dans une même section critique, afin d'éviter l'accès simultané à une ressource partagée. L'algorithme de Naïmi-Tréhel est basé sur les jetons. Il repose sur deux structures de données essentielles :

— **LAST :** On maintient un arbre dynamique qui fait que la racine de l'arbre est toujours le dernier noeud (site) qui aura fait une requête. Cette structure de données, implémentée dans notre cas par une variable présente sur chaque site, et souvent appelée father, last ou owner, sera appelée dans notre cas *Last*.

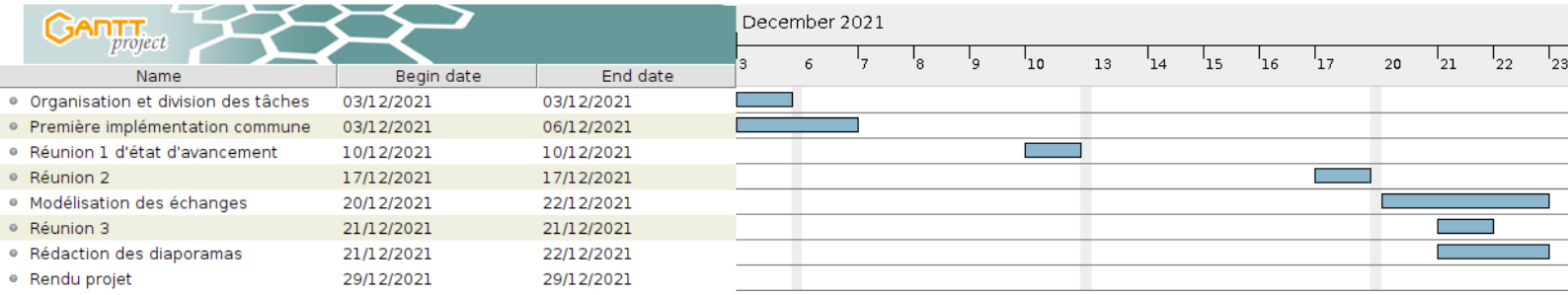
— **NEXT :** Aussi appelée file des next, et dans notre cas *Next*. Elle fonctionne comme une file d'attente, contenant les sites ayant fait une requête et en attente de section critique. Le début de cette file est le site possédant le jeton. Si aucune requête n'a été envoyée, il est le seul membre de cette file.

Cet algorithme garantit une réduction du nombre de messages en $\mathcal{O}(\log n)$, avec n le nombre de processus.

1. `https://fr.wikipedia.org/wiki/Algorithme_de_Naimi-Trehel`.

2 Organisation de notre travail

Nous nous sommes premièrement mis d'accord sur le planning à prendre en compte pour effectuer le projet. Le planning est répresenté sur le Diagramme de Gantt ci-dessous. Il n'a pas été respecté à la lettre mais il a pu nous donner une idée générale sur l'avancement du projet au cours du temps et vers quelle organisation se tourner.



Durant la phase developpement qui a duré environ 4 à 5 jours, nous avons eu de courtes réunions d'environ 30 min chaque jour, celles-ci se tenant en majorité sur Discord. Pour la phase de developpement, github [2] a été utilisé. La branche *xterm* est celle de notre version finale. Nous avons consacré les premiers jours à une reflexion chacun de notre coté et puis ensemble sur l'implémentation à mettre en oeuvre. Il à donc été decidé d'une première implementation qui gèrerait tous les sites en mêmes temps au travers de multiples threads, l' éxecution se ferait donc au sein d'un même terminal, on garderait une trace de l'algorithme et des échanges de messages en l'affichant directement dans le terminal. Puis pour notre version ultérieure, le programme serait lancé dans autant de xterm que l'on désirerait de sites. Nous nous sommes en réalité que très peu réparti les tâches et avons plutôt pris le parti d'implémenter ensemble et, lorque nécessaire, de débuger ensemble.

2. `https://github.com/hugomtr/NaimiTrehel`

3 Notre approche

3.1 Choix faits

Nous nous sommes tournés vers le C++ pour nous laisser la possibilité d'utiliser certaines librairies utiles pour des améliorations sur le code (gestion des durées avec simulation aléatoire, etc). Toutefois le code principal est rédigé intégralement avec des fonctions propres à C. Nous avons tous programmé sur Visual Studio Code, un IDE auquel nous étions tous le plus familier. Pour mener à bien ce projet en totalité il nous a fallu utiliser les notions retravaillées en cours, comme celles des threads et des sockets.

Nous avons séparé notre code en deux fichiers : l'header contenant tous les prototypes de nos fonctions ainsi que des constantes, et le fichier de code principal **main.cpp**.

Notre programme est organisé de sorte à ce que chaque site écoute sur un port donné (port choisi démarrant à 3000 par défaut puis incrémenté de 1 à chaque nouveau site). Chaque site peut de plus envoyer des messages à n'importe quel autre site (donc sur n'importe quel autre port correspondant au site voisin). Pour ce faire chaque site possède deux threads. Un thread chargé de la réception et du traitement des messages entrants, puis l'autre thread qui se charge de l'envoi de messages relatifs à l'entrée et la sortie de la section critique ainsi que de la terminaison du programme. Lorsque le thread de traitement de messages est exécuté, une socket qui joue le rôle du serveur est créée sur le port du site. Le socket accepte toutes demandes de connexions entrantes, les messages reçus sont traités et, en réponse, (selon les conditions de Naimi Trehel) un nouveau message est envoyé au site désiré par le biais d'une socket créée temporairement sur le port du site destinataire uniquement pour l'envoi de ce message. Donc chaque fois qu'un

message doit être envoyé pour les 2 threads, une socket est créée sur le port du destinataire uniquement pour l'envoi de ce message. Le message est envoyé selon une structure propre définie dans le programme.

Un *message*, dans le cadre de l'algorithme de Naïmi-Tréhel, peut-être de différents types, c'est pourquoi nous avons choisi de définir une énumération pour modéliser le type. Nous avons ainsi pu identifier 3 types de messages :

- **QUIT** : message annonçant que le site ne rentrera plus en section critique et qu'il a terminé tout son travail.
- **REQUEST** : message de demande pour la section critique.
- **TOKEN** : message représentant le jeton.

à cela s'ajoutent dans nos messages l'ID du site émetteur, celui du site destinataire et enfin celui du site ayant demandé la section critique. Le dernier paramètre peut sembler optionnel, mais il nous a été indispensable pour traiter les cas des messages du type REQUEST. Chaque thread possède aussi un verrou utilisé dès lors qu'il a besoin d'accéder à une zone mémoire auquel d'autres threads ont accès (pour la modification des tableaux communs last, next, has_token, etc).

En ce qui concerne la modélisation des sites, nous nous sommes tournés sur des xterm, chaque terminal simulant un site. On garde la trace de chaque message reçu et envoyé par un site en les affichant dans son terminal correspondant.

En théorie, l'algorithme Naïmi-Trehel nécessite à chaque site de mémoriser seulement deux variables : son parent dans la structure anti-arborescente ainsi que son next. En pratique, nous avons utilisé deux mutex pour chacun des threads d'un site pour garantir les bonnes modifications sur nos variables partagées entre les threads ainsi que les variables *requete SC* et *token present* nécessaires dans l'algorithme.

3.2 Nos principales difficultés

Notre difficulté principale a été de conceptualiser les idées au travers notre programme. Nous avons dû réfléchir différemment qu'en programmation classique où à partir d'un algorithme l'implémentation est plus aisée. Pour ce projet il fallait choisir à travers la multitude d'outils à notre disposition : socket, threads, multi-processing, sémaphores, etc, et justifier le choix de chacun, comprendre ce qu'ils permettaient de faire, leurs limites, pourquoi un tel outil est-il plus adapté, etc. Ensuite il a fallu s'accorder sur une structure pour représenter notre arbre de last, notre file de next ainsi que le choix de la structure de nos messages. Une fois tous ces détails en tête l'implementation s'est faite plus rapidement que prévu et le passage (d'un fichier général gérant tous les sites) à des xterms a été simple.

4 Présentation des résultats

4.1 Code

Nous allons présenter ci-dessous le code de la fonction main, qui indique que l'on peut saisir 2 paramètres, l'*id du site* et le *nombre de fois que le site souhaite rentrer en section critique* ou bien si l'on fait le choix de 4 paramètres on pourra cette fois également spécifier le *port par défaut* en plus du *nombre de sites participants*.

```
int main(int argc, char *argv[])
{
    if (argc < 3)
    {
        printf("Args : id_du_site nb_exec_site");
        exit(EXIT_FAILURE);
    } else if (argc == 5) {
        start_port = atoi(argv[1]);
        myID = atoi(argv[2]);
        NB_EXEC_SC = atoi(argv[3]);
        NB_NODES = atoi(argv[4]);
    } else {
        start_port = 3000;
        myID = atoi(argv[1]);
        NB_EXEC_SC = atoi(argv[2]);
        NB_NODES = 4;
    }
    run();
    return 0; }
```

4.2 Résultats

En exécutant l'algorithme avec 2 paramètres, dans ce cas le premier site est lancé avec 3 entrées en section, puis le site 1 et 2 avec 2 entrées en SC : on obtiendra le détail de chaque execution sur chacun des sites (captures ci-dessous) : il faudra donc ouvrir 3 xterm et exécuter respectivement les commandes suivantes :

```
$ ./main 3000 0 3 3
```

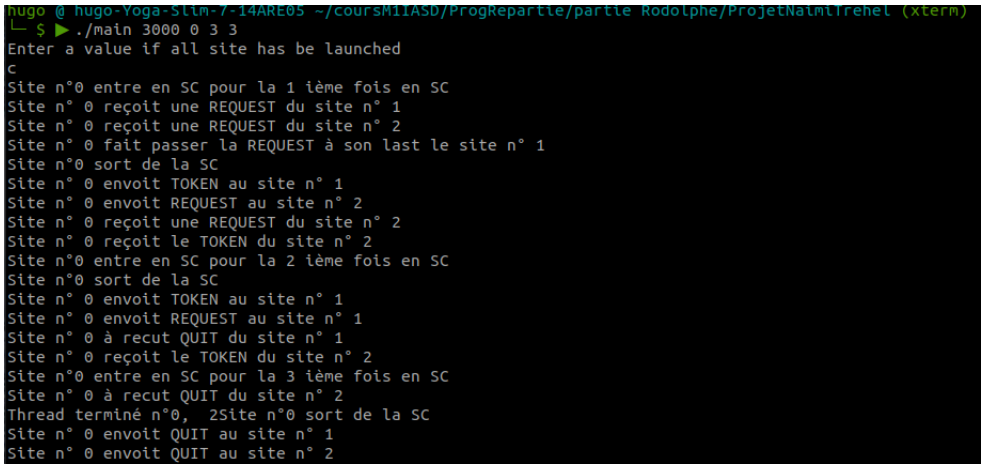


FIGURE 1 – Site 0

```
$ ./main 3000 1 2 3
```

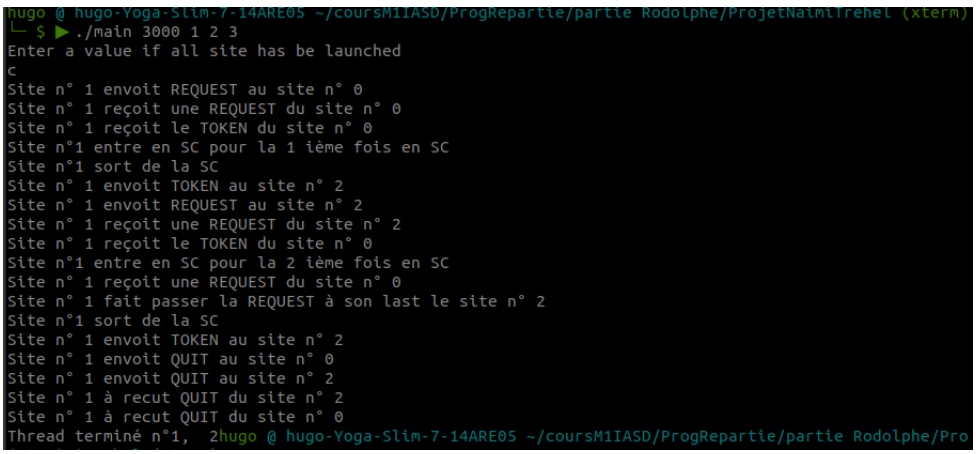


FIGURE 2 – Site 1

```
$ ./main 3000 2 2 3
```

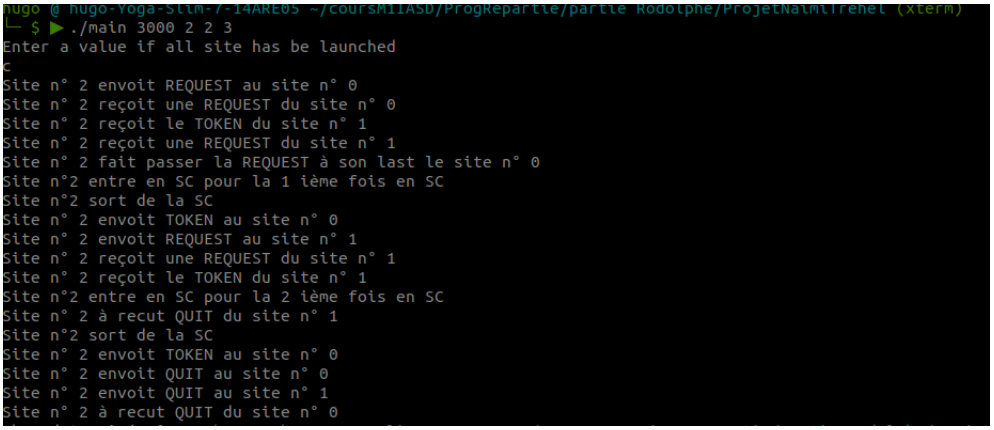


FIGURE 3 – Site 2

Note : Chaque site est rendu actif après que l'utilisateur ait rentré un terme dans le xterm correspondant. Pour un bon déroulement il faudra enclencher le site 0 en dernier. En effet le risque est que le site 0 possésseur du jeton exécute toutes ses sections critiques (si le temps en SC pour le premier site est vrament faible) avant même que l'utilisateur ait eu le temps de déclencher les autres sites et donc de permettre leurs connexions.

5 Conclusion

5.1 Perspectives d'amélioration

Nous aurions pu utiliser un système de TimeOut pour mieux simuler le fait que des requêtes ne mettent pas toutes forcémenent le même temps à arriver. Nous aurions également pu utiliser la norme OpenMP, dont certains de nous maîtrisent les bases, afin de mieux paralléliser notre code. Une autre idée aurait été celle d'une interface graphique modélisant les échanges de manière intuitive et simple. Enfin, nous avons pensé à mesurer le temps d'exécution de notre algorithme, premièrement par la commande Unix *time* mais notre algorithme est fait de telle sorte à ce qu'il soit borné par le temps d'attente de saisie de l'utilisateur pour l'exécution d'un site en question. Afin de nous abstraire de ce biais et d'autres évenements, nous aurions pu donc directement intégrer dans notre code des fonctionnalités permettant de faire ce travail. La question aurait été donc de savoir qu'est ce que l'on mesure exactement, nous avons en conséquence pensé au temps d'arrivée d'un message entre deux processus.

De même, il aurait été possible d'étendre notre programme pour qu'il soit résistant aux pannes. Il se pourrait qu'un message échoue à être délivré ou qu'un message réussisse à être reçu mais ait été erroné. Afin de se préparer à ces pannes de message, nous pourrions imaginer un mécanisme de surveillance qui relance l'envoi du message au cas où un message est susceptible d'être perdu ou erroné. Un autre type de panne pourrait apparaître quand un site se termine d'une façon imprévue. Cela risquerait d'isoler les descendants du site en panne du réseau, ou même de détruire le jeton si le site en panne le possède. Une résolution des pannes sur les sites peut être également réalisée par un mécanisme de TimeOut pour détecter un site en panne, suivi d'un broadcast pour notifier tous les sites si une panne se produit.

5.2 Ce qu'on a appris

La réalisation de ce projet nous a premièrement permis d'améliorer de manière assez conséquente notre connaissance du C. Utiliser des notions à première vue assez différentes, réussir à les mettre toutes en symbiose afin d'arriver au fonctionnement souhaité n'a en effet pas été chose facile mais aura été un défi très formateur à notre sens. Ces notions, celles des sockets et des threads au service de l'algorithme d'exclusion mutuelle de Naïmi-Tréhel, nous ont également permis d'en savoir plus sur les systèmes répartis et comment tout ce petit monde fonctionne. Nous avons également appris des choses côté hardware, étroitement lié à certains points traités.

Le travail d'équipe, tant dans les échanges humains qu'à travers des outils collaboratifs mentionnés précédemment dans le rapport, nous a aidés dans la conduite de projet, et dans cette compétence, considérés de « soft » par beaucoup. Nous avons également, de manière plus globale, beaucoup appris sur la programmation répartie, thème de cet UE, nous-semblent-ils, essentielle en Informatique.