Université de Montpellier

Master Informatique Génie Logiciel

HAI721I Algorithme Naimi-Tréhel

par

Martin SANCHEZ
Matthieu TRINQUART

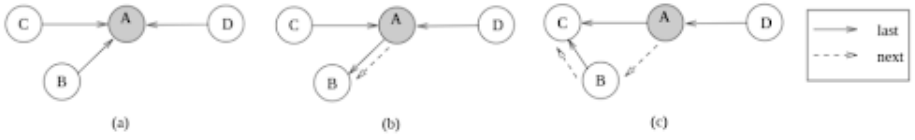
Encadrants universitaires : Hinde BOUZIANE , Rodolphe GIROUDEAU

Table des matières

Introduction

1 Présentation de l'algorithme

L'algorithme de Naimi-Tréhel est un algorithme qui se charge de l'exclusion mutuelle dans un système distribué.
L'algorithme peut-être représenté comme un arbre où chaque nœud est un processus qui peut demander une section critique. Et une pile, qui représente l'ordre de passage en section critique. Un nœud a un père, et un suivant, il ne peut avoir qu'un père mais plusieurs fils.
Quand un processus accède à la racine il entre dans de la file d'attente pour passer en section critique et attend le jeton.
Un processus pour demander à avoir la section critique envoie un message à son père, qui le fait remonter jusqu'à la racine actuelle, chaque nœud est alors modifié pour faire du demandeur la nouvelle racine de l'arbre.
Quand un noeud reçoit une demande de section critique et qu'il n'est pas la racine, il transmet à son propre père jusqu'à ce qu'on tombe sur la racine de l'arbre. Du coup le demandeur devient le Next de la racine et attend que le noeud ait fini sa section critique pour recevoir le jeton.
La racine après avoir fini sa section critique peut libérer le jeton et le passer au next.



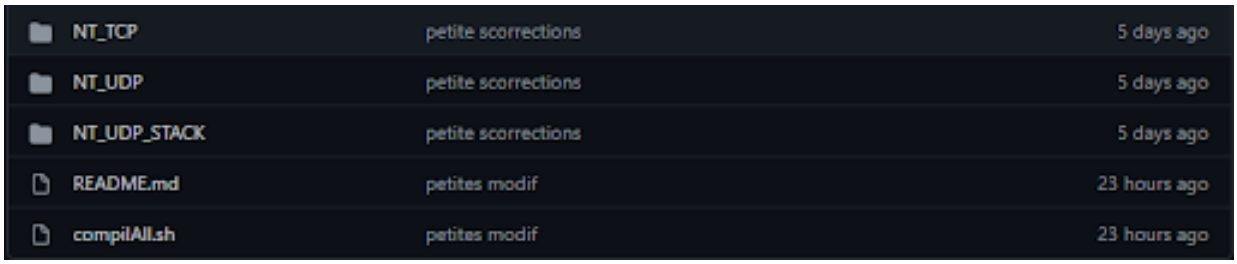
Ainsi l'algorithme de Naimi-Tréhel dans le pire des cas aura N-1 (N étant le nombre de noeud) messages de demande de jeton +1 message pour l'envoi du jeton ce qui fait au maximum N messages échangés lors d'une demande pour passer en section critique. Mais en moyenne le nombre de messages envoyés pour une demande de section critique est de O(log((n)).

Notre implémentation

2 Organisation du projet

Nous avons utilisé Github, cela nous a permis d'organiser les différentes versions du projet. Voici le lien du dépôt : https://github.com/MartinSanchezIut/NaimiTrehel

2.1 Organisation générale



Organisation du dépôt

Pour ce projet, nous avons décidé de faire plusieurs versions. Une première version "NT_UDP" implémente l'algorithme en utilisant le protocole UDP pour l'envoi et la réception de messages sur le réseau, cela nous a permis dans un premier temps de nous concentrer sur l'algorithme en soit. Le protocole UDP étant plus simple à utiliser, car il n'oblige pas la connexion du client au serveur (connect), nous avons pu nous familiariser avec l'algorithme et visualiser assez rapidement un début de fonctionnement de l'algo.
C'est quand nous avons obtenu un code qui avait l'air de fonctionner correctement que nous avons décidé d'ajouter des fonctionnalités.
La deuxième version "NT_UDP_STACK" ne change pas beaucoup par rapport à la première, nous avons juste modifié la variable "next" d'un nœud pour qu'elle devienne une liste. Cela nous permet de ne pas effacer le "next" d'un nœud, dans le cas où on lui demande la racine alors qu'il n'a pas encore passé le jeton. D'une certaine façon, cette modification nous assure que si un nœud demande le jeton, il l'aura forcément à un moment donné.

Et pour la version finale “NT_TCP”, nous avons repris “NT_UDP_STACK” et fait des modifications pour utiliser le protocole TCP.
Le fichier “compillAll.sh” est un script qui exécute les makefiles des trois projets.

2.2 Organisation d’un projet

Organisation d’un projet

Chaque projet est organisé de la même façon, ils contiennent un makefile, qui permet de compiler le projet, ainsi que plusieurs fichiers.
Le fichier “bin” contient tous les exécutables générés par le makefile
Le fichier “exemple” contient les sources et les exécutables d’un exemple de fonctionnement de l’algo. Il contient aussi un script “tester.sh” qui permet de lancer 4 fois le programme dans des terminaux différents.
Le fichier “h” contient les fichier .h
Le fichier “src” contient le code.

Explication des programmes

3 Les structures

Les structures manipulées sont les mêmes pour tous les projets.

Message

```
int message{
    int type;
    /*
    type = 0 : Demande pour devenir la racine
    type = 1 : Envoi de token, pour entrer en section critique
    type = 2 : L'ex racine me prévient que je suis la nouvelle racine
    */
    struct sockaddr_in contenu;
    /*
    Qui suis-je ? Mon ip, et mon port d'écoute.
    */
};
```

Structure message

Cette structure représente les messages envoyés sur le réseau. L'entier "type" définit la nature du message, est-ce qu'il s'agit d'un envoi de jeton, d'une demande de racine.
La structure "sockaddr_in" contenue informe le receveur de l'identité de l'envoyeur du message.

Param

```
struct param{
    int* condBoucle;
    pthread_mutex_t* jeton;
    struct sockaddr_in *pere;
    struct sockaddr_in *next;

    int portEcoute;
};
```

Structure param

Cette structure représente les paramètres à faire passer dans le thread d'écoute de chaque nœud. L'entier "condBoucle" permet de mettre fin à la boucle d'écoute. Le mutex "jeton" représente le jeton, l'avoir permet d'entrer dans la section critique.
"Pere" et "Next" sont des pointeurs vers le père et le next du noeud.
Et "portEcoute" est le port sur lequel le thread doit écouter.

Sockaddr_in

```
struct sockaddr_in{
    short           sin_family;     //e.g. AF_INET
    unsigned short  sin_port;       //e.g. htons(3490)
    struct in_addr  sin_addr;       //see struct in_addr, below
    char            sin_zero[8];    //zero this if you want to
};
```

Structure sockaddrin

Cette structure identifie un nœud dans le réseau.

4 L'interface "naimi.h"

Cette interface définit dans un premier temps les structures décrites plus haut. Elle définit aussi les fonctions importantes de notre programme :

```
int AttendreMessage(int socket, message *msg);
```

Cette méthode est bloquante, elle écoute sur le socket “socket” et attend un message, elle l’écrit dans le paramètre “msg”. Ce message pourra être lu par le thread d’écoute.

```
int EnvoyerMessage(int socket, struct sockaddr_in dest, message msg);
```

Cette méthode permet d’envoyer le message “msg” au destinataire “dest”. Elle utilise le socket “socket”.

```
int EnvoyerToken(pthread_mutex_t* jeton, int socket, struct sockaddr_in
    dest);
```

Un cas particulier de “EnvoyerMessage(..)” permet d’envoyer un message particulier au destinataire “dest”.

```
pthread_mutex_t initToken(struct sockaddr_in me, struct sockaddr_in pere);
```

Permet d’initialiser le jeton à l’initialisation.

```
void attendreToken(pthread_mutex_t* jeton);
```

Fonction bloquante, permet d’attendre le jeton.

```
struct sockaddr_in getSockAddr(char ip[], int port);
```

Elle permet de créer une structure “sockaddr_in” avec l’ip et le numéro de port passé en paramètre.

```
time_t getTime();
```

Retourne une valeur de temps, cela permet de visualiser une chronologie dans les affichages des différents terminaux.

5 Première version en UDP

Dans un premier temps, le programme crée toutes les variables nécessaires au fonctionnement : “pere”, “next”, initialise le jeton. On crée aussi les socket, d’envoi et d’écoute. Ensuite, on crée un thread, qui va se charger d’écouter sur le port d’écoute, il récupère des pointeurs sur les données à travers la structure “param”. Boucle tant que le programme principal ne termine pas et traite tous les messages entrant. Ce thread modifie les données du programme principal, “pere” et “next” en fonction des messages qu’il reçoit.

Et pour finir, dans le main, on écrit le programme, c’est-à- dire les attentes de jetons, la section critique et les calculs.

6 Version en UDP implémentant une pile pour le “next”

Ce code reprend exactement la version en UDP, simplement on vient ajouter un .h pour la pile.

```
1 struct Stack {
2     int top;
3     int capacity;
4     struct sockaddr_in* array;
5 };
```

La structure d’une pile, l’entier “top” représente l’indice dans le tableau “array” qui est en sommet de pile. L’entier “capacity” représente la taille du tableau “array”. C’est une pile de type FIFO.

```
1 struct Stack* createStack(unsigned capacity){
```

Permet de créer une pile de taille ”capacity”.

```
1 int isEmpty(struct Stack* stack){
```

Retourne 1 si la pile est vide.

```
1 void push(struct Stack* stack, struct sockaddr_in item){
```

Permet de pousser un élément dans la pile.

```
1 struct sockaddr_in pop(struct Stack* stack){
```

Permet de sortir le premier élément de la pile.

Grâce à ce code, la variable next peut maintenant contenir plusieurs éléments. On ne risque plus de perdre de l'information en écrasant un "next" déjà écrit.

7 Version finale en TCP

La version finale est basée sur la version UDP avec la pile, nous avons changé le protocole de UDP vers TCP et cela implique pas mal de changements.

La grosse modification est que chaque envoi de message doit être précédé d'une demande de connexion. Pour cela le thread d'écoute doit accepter toutes les demandes de connexions entrantes, et en même temps traiter les messages entrants. Pour cela nous avons décidé de mettre en place un multiplexage. De cette façon, nous pouvons traiter plusieurs "clients", sans créer de processus supplémentaires.

Utilisation

8 Comment compiler

Chaque projet contient un makefile qui compile les fichiers du dossier src/ et qui compile dans le fichier bin.
A noter qu'il faut être dans le répertoire du projet pour le compiler.

9 Comment exécuter notre exemple

Pour ensuite exécuter le projet nous avons créé un fichier bash qui lance plusieurs terminaux avec des nœuds.
En étant dans le répertoire du projet, exécuter la ligne de commande "./exemple/tester.sh" qui exécute le fichier bash et ouvre 4 terminaux qui représentent 4 nœuds de notre arbre. Chaque noeuds fait des demandes d'entrée en section critique.

Conclusion

En conclusion ce projet a été enrichissant car cela nous à permis de mélanger des connaissances théorique du cours (apprentissage de l'algorithme) avec des connaissances pratiques (utilisation de threads, TCP/IP, UDP).
Ce projet nous a aussi permis de voir plus en profondeur l'algorithme Naimi-Trehel et de cerner plus en détails ces subtilités.
Nous sommes plutôt fiers du résultat final même si avec plus de temps il aurait été possible de rajouter d'autre fonctionnalités.