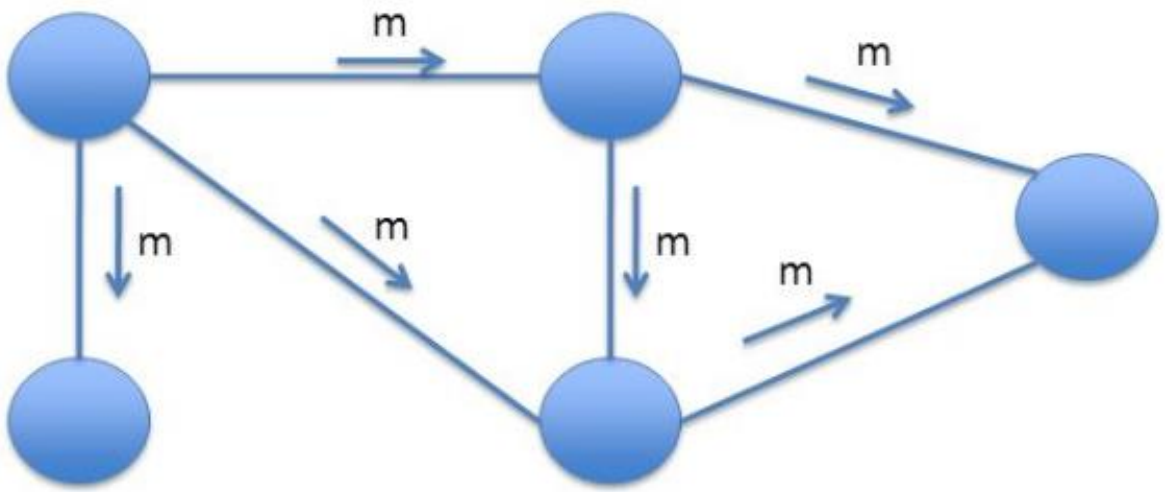
UNIVERSITÉ DE FRANCHE-COMTÉ

PROJET AD M2

Année 2016/2017

Algorithmique Distribuée
« Algorithme de Naimi-Tréhel »



Enseignant : Laurent PHILIPPE

Avec la Participitation de :
Pierre Wargnier

Sommaire

Introduction

Le but de ce projet est d'implémenter l'algorithme de Naimi-Tréhel en JAVA au-dessus d'un réseau quelconque à l'aide du simulateur ViSiDia. Nous avons vu en cours une implémentation de cet algorithme pour un réseau complet. Une première partie du travail sera d'implémenter l'algorithme, puis de le faire évoluer pour un réseau quelconque. Enfin, dans cet algorithme, le Leader est déterminé statiquement, nous devrons donc implémenter une élection basée sur le plus grand id de processus.

Initialisation

Dans un premier temps, Il nous a fallu implémenter Naimi-Tréhel. Nous disposions du fonctionnement algorithmique donner par notre professeur, mais l'implémentation en JAVA nous a demander d'adapter certaine partie.

Nous avons choisi de garder la structure de l'algorithme, et de greffer les composants nécessaires autour. Cependant, certaine règle ont été entièrement refaite ou séparer en plusieurs fonctions, ainsi nous avons choisi d'abandonner la structuration en règle pour une structuration par fonctionnement.

1. Chaque processus doit connaitre son voisin

Le premier ajout dû à l'implémentation en code est la création de message servant juste au processus à se connaitre entre eux.

```
//Partie gerant que chaque processus ce connaissent
    SyncMessage sm = new SyncMessage(MsgType.START, procId);
    printTraces(procId + " envoie le message START a ses voisins");
    sendAll(sm);

  /* try { this.wait(); } catch( InterruptedException ie) {ie.printStackTrace();}
   voisinReponse=0;*/
```

Ici, on envoi à tous ses voisins un message pour donner notre identité.
On écrit dans un fichier pour garder une trace de cette étape.
La partie commentée devait permet l'envoie de toutes les identités avant de continuer, cependant elle souffre d'un problème de synchronisation.

```
//permet de se souvenir du voisin
public void receiveSTART (int d, int proc){
    printTraces(procId + " reçoit le message START de " + d);
    map.put(proc,d);
    /*voisinReponse++;
    if(voisinReponse==this.getArity()){
    this.notifyAll();
    }*/
    displayState();
}
```

Ici, on gère la réception d'un message START, celui-ci enregistre dans une hashmap la Door et le processus associer. On écrit dans un fichier pour garder une trace de cette étape.
La partie commenté est la deuxième partie qui devait servir à notifier la première lorsque tous nos voisins avaient donné leur identité, mais ceci n'est fonctionnel que sur le principe. Cependant ça non-présence n'empêche pas le bon déroulement de l'algorithme.

2. Gestion de l'écriture dans le fichier

Pour garder une trace de l'exécution, nous avons choisi d'écrire dans des fichiers, un par processus, chaque action effectuer par l'algorithme. Cela nous permet de détecter des erreurs, ou d'optimiser des fonctionnements.

Pour ce faire, nous avons décidé d'utiliser des Filewriters. En effet, ceux-ci nous permettent, une fois le fichier créer, d'écrire facilement dedans et de les refermer aussitôt. Sachant que l'exécution se déroule en boucle et que nous somme généralement obliger d'arrêter notre simulateur brutalement, nous ne pouvons attendre d'avoir fini l'exécution pour fermer le fichier.

Ainsi, nous avons cette petite fonction qui nous permet d'écrire nos traces facilement.

```
//--------------------
// Partie Gestion de fichier
//--------------------

public void printTraces(String str){
    try{
        FileWriter writer = new FileWriter(f, true);
          // Writes the content to the file
          writer.write(str+"\n");
          writer.flush();
          writer.close();
    }
    catch (IOException exception){
        System.out.println ("Erreur lors de la lecture : " + exception.getMessage());
    }
}
```

3. Initialisation de l'élection

Dans l'algorithme tel qu'il nous est présenté, l'initiateur commence l'élection, cependant si on détermine statiquement qui sont les initiateurs, autant déterminer directement qui est le leader, de plus si le processus n'existe pas, cela pose problème. Nous avons donc choisi que tous les processus sont initiateurs.

```
//Partie gerant l'election
    SyncMessage sm2 = new SyncMessage(MsgType.LEADER, procId);
    printTraces(procId + " envoie le message a ses voisins'le LEADER est "+procId+"'");
    sendAll(sm2);

while( voisinReponseLeader < this.getArity() ) {
    displayState();
    try { this.wait(); } catch( InterruptedException ie) {}
}


//Partie initialisant le leader

if (leader == procId){
    token = true;
    owner = -1;
} else {
    owner = leader;
}
```

ateurs.

Ici, on envoie à nos voisins notre id pour l'élection, ensuite on en attend la fin, Puis on détermine qui a le Jeton.

Élection

Avant d'exécuté Naimi-Tréhel, nous devons d'abord déterminer qu'elle processus à le jeton. En effet, seul le processus possédant le jeton peut rentrer en section critique.
Nous allons donc procéder à une élection. Comme dit précédemment, tous les processus sont initiateurs.
Nous allons voir ici comment nous avons gérer la réception des messages liés à l'élection.

1. Réception du message LEADER

Lors de la réception de LEADER, nous recevons le leader actuel pour le processus qui nous envoie LEADER.

```
if(proc>leader){

    leader = proc;
    father = d;
    voisinReponse =1;

    for(int i=0;i<this.getArity();i++){
        // on envoi le leander à nos voisin
            SyncMessage sm = new SyncMessage(MsgType.LEADER, proc);
            printTraces(procId + " envoie le message a son voisin "+i+" 'le LEADER est "+proc+"'");
            sendTo(i, sm);
    }
```

Dans le cas où le Leader que l'on reçoit est supérieur à notre leader connu, sachant que celui-ci est initialisé avec le processus courant, alors le leader devient le nouveau leader, le facther prend la valeur de la Door, et on fixe le nombre de réponse reçu des voisins à 1. Ainsi on attend la confirmation de tout le monde avant de déclarer que c'est effectivement le leader. Ensuite on envoie notre leader à tout le monde.

```
}else if (proc == leader){
    voisinReponse++;
    if(voisinReponse == this.getArity()){
        if(leader == this.procId){
            for(int i = 0; i<this.getArity();i++){
                SyncMessage em = new SyncMessage(MsgType.IMLEADER, this.procId);
                printTraces(procId + " envoie le message IMLEADER a son voisin " + i);
                sendTo(i, em);
                voisinReponseLeader++;
            }
            notify();
        }

    }
```

Si tous les voisins nous envoient comme leader notre leader, et si ce leader correspond à nous-même, alors on envoi à tout le monde que nous somme le leader.

2. Réception du message IMLEADER

Lorsqu'un processus nous indique qu'il est le leader et que ce processus n'est pas nous, alors :

```
if(voisinReponseLeader  == 0){
    //si nombre de voisin egale a 1, on envoie directement à son voisin unique
    if(this.getArity() == 1){
        printTraces(procId + " envoie le message IMLEADER a " + d + " (le LEADER est "+proc+")");
        sendTo(d,sm);
    }
    // Sinon en envoie à tout ses voisins
    for (int i = 0; i< this.getArity();i++){
       //if (i != j) {
        printTraces(procId + " envoie le message IMLEADER "+proc+")");
            boolean sent = sendTo(i,sm);
        //}
    }
}
```

Si aucun voisin n'a encore déterminé le leader, alors on fait passer le message IMLEADER à tous nos voisins. Dès que tous nos voisins ont déterminer le leader, alors on enregistre le leader et on libère le processus.

Naimi-Tréhel

Le principe que nous devons implémenter est :

- Un processus attend
- Il demande un accès en section critique
- Il attend d'avoir le jeton
- Il rentre en section critique
- Il sort de section critique
- Et l recommence

Nous avons traduit cela par :

```
while(true) {
    // Wait for some time
    int time = ( 3 + rand.nextInt(10)) * speed * 100;
    System.out.println("Process " + procId + " wait for " + time);
    try {
    Thread.sleep( time );
    } catch( InterruptedException ie ) {}

    // Try to access critical section
    waitForCritical = true;
    askForCritical();

    // Access critical
    waitForCritical = false;
    inCritical = true;

    displayState();

    // Simulate critical resource use
    time = (1 + rand.nextInt(3)) * 100;
    System.out.println("Process " + procId + " enter SC " + time);
    try {
    Thread.sleep( time );
    } catch( InterruptedException ie ) {}
    System.out.println("Process " + procId + " exit SC ");

    // Release critical use
    inCritical = false;
    endCriticalUse();
}
```

1. Gestion de la demande de section critique

Lorsqu'un processus demande l'accès en section critique, il passe sa variable d'attente de section critique à vrai, il envoie le message REQ au processus possédant le jeton, passe sa variable lui permettant de connaitre le possesseur du jeton à -1, ce qui lui permet de savoir que c'est lui qui va avoir le jeton (cela sera utile lorsqu'on le lui demandera) et attend l'arrivée du jeton.

```
//Pi ask for Critical Section
synchronized void askForCritical(){
    waitForCritical = true;
    if (this.owner != -1) {
        SyncMessage sm = new SyncMessage(MsgType.REQ, procId);
        printTraces(procId + " envoie le message REQ a " + map.get(owner));
        sendTo(map.get(owner), sm);
        owner = -1;
        displayState();
        while (token != true){
            try {
                wait();
            } catch (Exception e) { }
        }
    }
}
```

2. Réception du message REQ

Lors de la réception du message REQ, si le processus courant possède le jeton et qu'il n'a pas besoin d'aller en section critique, alors il l'envoi au demandeur. S'il a besoin d'aller en section critique, alors il sauvegarde la demande de jeton pour l'envoyer dès qu'il sort de section critique. S'il ne possède pas le jeton, il fait suivre la demande au dernier processus qui possède le jeton à sa connaissance.

```
public synchronized void receiveREQ (int k) {

    printTraces(procId + " reçoit le message REQ de " + k);

    if (owner == -1) {
        if (waitForCritical == true || inCritical == true) {
            next = k;
        } else {
            token = false;
            SyncMessage sm = new SyncMessage(MsgType.TOKEN, procId);
            printTraces(procId + " envoie le message TOKEN a " + map.get(k));
            sendTo(map.get(k),sm);
            displayState();
        }
    } else {
        SyncMessage sm = new SyncMessage(MsgType.REQ, k);
        printTraces(procId + " envoie le message REQ a " + map.get(owner) + "(transmet la demande de "+k+")");
        sendTo(map.get(owner), sm);
        displayState();
    }
    owner = k;
}
```

3. Sortie de section critique

A la sortie de section critique, le processus regarde s'il y a un processus en attente, si oui il lui envoi le jeton, sinon effectue l'action suivante de la boucle principale.

```
//Pi quit SC
void endCriticalUse() {
    waitForCritical = false;
    if (next != -1){
        SyncMessage sm = new SyncMessage(MsgType.TOKEN, procId);
        printTraces(procId + " envoie le message TOKEN a " + map.get(next));
        sendTo(map.get(next), sm);
        token = false;
        next = -1;
        displayState();
    }
}
```

4. Réception du Jeton

A la réception du Jeton, on indique au processus qu'il le possède et on le libère pour qu'il puisse passer en section Critique.

```
//Pi receive TOKEN from j
public synchronized void receiveTOKEN () {
    printTraces(procId + " reçoit le message TOKEN");
    token = true;
    notify();
}
```

Conclusion

Nous avons pu développer la majorité des fonctions demander pour le projet. Malheureusement, Le programme ne fonctionne que pour des réseaux complets. En effet nous n'avons pu trouver de solution satisfaisante qui ne soit pas fortement inspirer du système de table de routage. Une idée à l'aide de liste chainé a été envisager, ainsi qu'un parcours à l'aide des fathers. Cependant si informer de la demande de jeton est faisable, sa transmission a été plus complexe que prévu. Néanmoins, cela nous a permis d'explorer un bon nombre d'algorithme permettant le partage de donnée.