ESIREM

4A INFOTRONIQUE MODULE ITIL42

COMPTE-RENDU DE TRAVAUX PRATIQUES

Gestion distribuée des ressources

Vincent CANDAPPANE Valentin GRAVE

2 avril 2018

Table des matières

Introduction générale 1

1 TP 1 : Algorithme de stabilisation 2

1.1 Introduction 2

1.2 Élection 2

1.3 Ordre d'un graphe 2

1.4 Stabilisation de chaque noeud du graphe 3

1.5 Le rôle crucial de l'élu 3

1.6 Conclusion 3

2 TP2 : Détection d'une panne 4

2.1 Introduction 4

2.2 Problème 4

2.3 Solution 4

2.4 Conclusion 5

Conclusion générale 6

Annexe 7

2.5 Code source du TP1 : Algorithme de stabilisation 7

Table des figures

1.1 Simulation d'un algorithme de stabilisation sur un arbre 3

2.1 Simulation d'un algorithme de détection d'erreur dans un graphe 4

Introduction générale

Dans le cadre de nos travaux pratiques de Système Distribués et de parallélisme au sein de l'ESIREM, nous allons appliquer et résumer les connaissances acquises au cours du module ITIL42.

Ce rapport retrace notre démarche et l'utilisation d'un logiciel ViSiDiA lors de nos différentes séances de travaux pratiques.

ViSiDiA est un outil pour implémenter, simuler, tester et visualiser des algorithmes distribués. Il est guidé par les importants résultats théoriques sur l'utilisation des systèmes de réétiquetage des graphes pour encoder les algorithmes distribués et prouver leur corrélation.

L'encodage de haut niveau des algorithmes distribués sous forme de systèmes de réécriture permet une meilleure description et présentation de ces algorithmes. Le résultat est une approche formelle pour décrire et étudier les algorithmes distribués d'une manière simple et unifiée.

Cet outil peut être utilisé pour visualiser et expérimenter des algorithmes distribués spécifiques, et donc faciliter leur conception et leur validation.

Ayant déjà manipulé cet outil lors des TP de systèmes distribués (ITIS43), nous allons manipuler des méthodes encore plus approfondie de parallélisme, de stabilisation et de gestion de panne en LC1 et LC2.

1. TP 1 : Algorithme de stabilisation

1.1 Introduction

L'objectif de ce TP est de construire un algorithme de stabilisation. Un tel algorithme permettra de parcourir un arbre afin d'en vérifier sa stabilité et d'avoir une vision omnisciente de l'arbre dans son état stable. Nous utiliserons dans ce TP, la méthode de l'élection LC1.

1.2 Élection

Afin d'optimiser au maximum les performances informatiques, il est crucial de mettre en oeuvre des algorithmes spécialisés dans le parallélisme permettant de traiter des informations de manière simultanée. Cette technique a pour but de réaliser le plus grand nombre d'opérations en un temps le plus petit possible.

En parallèle de l'élection LC1, l'on devra déterminer l'ordre du graphe afin de le transmettre au noeud élu, ainsi l'élu va plafonner l'ordre du graphe final durant le déroulement du programme. Nous ajoutons également deux propriétés aux noeuds : **p** (propriété *"estStable"* dans notre programme) et **a** (propriété *"distancePointStable"* dans notre programme), tels que **p** est le stabilisateur local (initialisé à "faux" au début) et **a** est la distance de stabilité (initialisé à "-1" au début).

Un point est considéré "stable" dans 2 cas et chaque cas engendre une certaine réaction au niveau du point centre :

— Il est une feuille de l'arbre, c'est à dire lorsqu'il a un seul voisin à l'état **"N"**. L'état du point passe alors à **"F"** (= état final).

— Il est l'élu de l'arbre, lorsque tous ses voisins sont à l'état final **"F"**. L'état du point passe alors à **"E"** (= état élu) car ce cas ne se rencontre que si tous les autres points du graphe ont été parcourus (en isolant les feuilles une par une, en se concentrant progressivement vers le point élu).

À noter qu'en parallèle de ces changements d'état du noeud, d'autres changements sont effectués :

— La propriété *"estStable"* passe à "true" (car le point a son état fixé)

— La propriété *"distancePointStable"* passe à "0" (car le point "est sûr" qu'il y a un point stable jusqu'à au moins une distance de 0 de lui, donc lui-même pour le coup).

1.3 Ordre d'un graphe

L'ordre d'un graphe est le nombre total de sommets dans un graphe. Pour pouvoir déterminer l'ordre de notre graphe, nous stockons une information supplémentaire dans chacun de nos noeuds. Cette information est un compteur initialisé à "1". Ensuite, à chaque fois que notre noeud passe à l'état final **"F"**, le compteur est additionné avec ceux des voisins directs du noeud central (via la méthode **"incrementOrder()"**). Une fois le parcours en profondeur terminé, le dernier noeud qui passera à l'état final **"F"**, plus exactement à l'état élu **"E"**, aura son compteur égal à l'ordre du graphe.

1.4 Stabilisation de chaque noeud du graphe

Une fois l'élection terminée et l'ordre du graphe obtenus, on considère la règle suivante pour calculer la distance de stabilité a : Si p est vrai alors a prend la valeur minimale présente parmi ses voisins et on incrémente de façon à ce que : $a = min(a)+1$. À noter que la distance de stabilité a ne peut pas dépasser la valeur de l'ordre du graphe au niveau du point "élu", donc si le a de ce point tente de s'incrémenter il est remis à la valeur de l'ordre du graphe, par défaut. Cependant, les autres distances a des autres points du graphe ne sont pas contrôlées de force comme au niveau du point "élu". Au lieu de cela, ils vont finir par se fixer par eux-mêmes. En effet, au fur et à mesure du déroulement de l'algorithme il vont avoir une valeur a qui va croître, mais au bout d'un certain temps les voisins directs du point "élu" auront comme plus petite valeur de voisins le a de ce point "élu", donc ils vont adopter cette valeur de $a+1$, puis les voisins de ces voisins auront comme plus petite valeur $a+1+1$, ..., et ainsi de suite. Donc, progressivement, les points auront une distance de stabilité qui va croître proportionnellement avec leur éloignement au point "élu".

1.5 Le rôle crucial de l'élu

L'élu joue un rôle crucial dans le calcul de stabilisation, en effet l'élu va plafonner son compteur et par effet de conséquences, ses voisins auront une distance de stabilité recalculée.

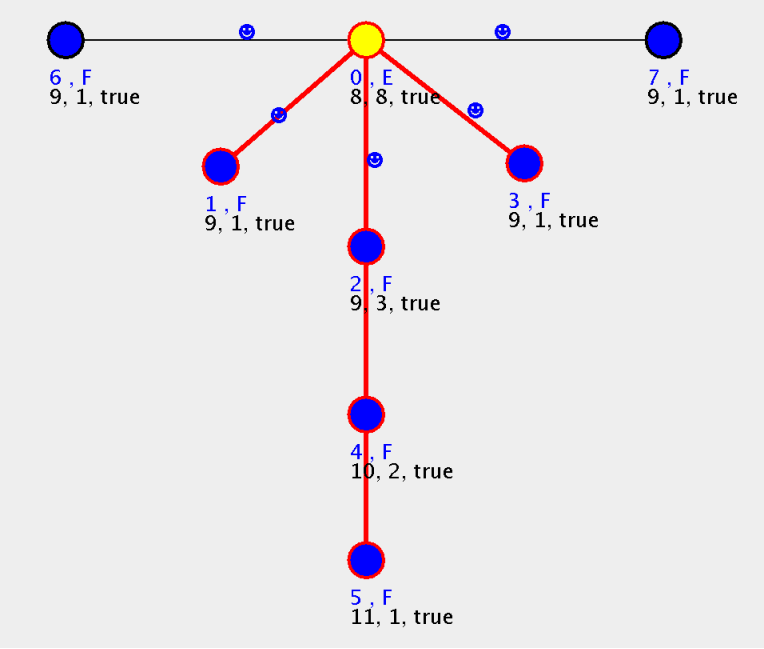

FIGURE 1.1 – Simulation d'un algorithme de stabilisation sur un arbre

1.6 Conclusion

Ce TP nous a permis de mieux appréhender la notion de stabilité d'un graphe. Cependant, il est difficile de calculer le diamètre d'un graphe. Nous avons donc substitué le diamètre du graphe par l'ordre du graphe afin de pouvoir visualiser comment fonctionne le graphe une fois la valeur maximale atteinte. Nous obtenons donc une vision omnisciente d'un arbre stable.

2. TP2 : Détection d'une panne

2.1 Introduction

L'objectif de ce TP est d'ordre prévisionnel. Il s'agit de réaliser la détection de panne d'un noeud dans un système distribué de topologie quelconque. Nous recherchons donc une méthode afin de récupérer l'étiquette du noeud en panne et d'en informer tous ses voisins. Nous utiliserons dans ce TP, la méthode de synchronisation en LC2.

2.2 Problème

Afin de simuler cela, nous allons nous fixer un scénario illustrant le problème (bien évidemment valable pour d'autres cas de panne du même acabit) : nous étudions un graphe(de type cycle) de 12 noeuds aléatoirement interconnectés. Nous supposons qu'au bout de 3 synchronisations successives, le noeud ayant pour l'étiquette "W" s'arrête, cette étiquette possède 2 voisins.

2.3 Solution

Afin de résoudre le problème, chaque noeud devra stocker deux paramètres concernant ses voisins :

— Les identifiants de chacun de ses voisins
— Les étiquettes de tous ses voisins, sauf celui qui est sur le port 0.

Nous stockerons dans deux tableaux distincts ces informations (identifiants et étiquettes) si le port du voisin est actif. Afin de détecter l'arrêt d'un noeud, il faut faire en sorte que l'étiquette du noeud en panne soit connue par ses voisins. Voici une simulation d'une détection de panne au niveau du noeud "10" :

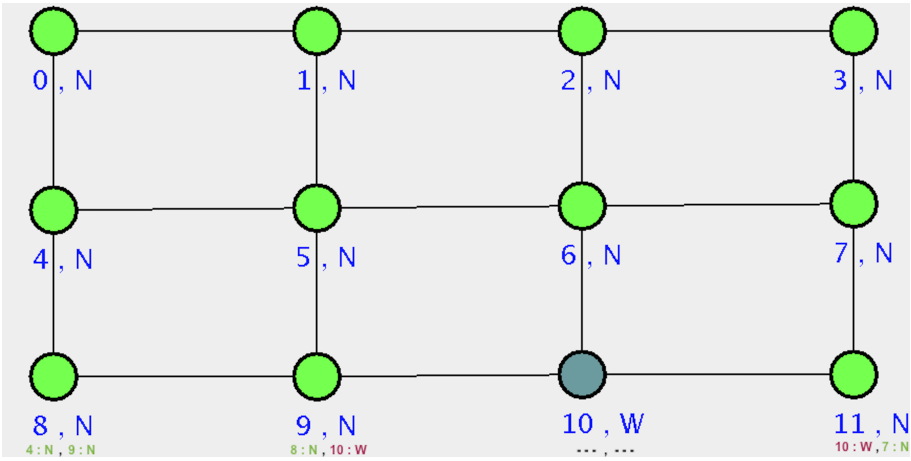

FIGURE 2.1 – Simulation d'un algorithme de détection d'erreur dans un graphe

Il s'agit d'une illustration de la détection d'erreur souhaitée. Si nous ne prenons en compte seulement les quatre derniers noeuds, nous pouvons voir vis-à-vis de leurs données stockées que le

noeud "10" est défaillant. Par conséquent, leurs voisins 9 et 11 ont pu le détecter. Ainsi, le noeud 8, étant lui même fonctionnel et ne possédant aucun noeud défaillant, n'indique pas de panne. Nous avions établi ici un système de détection de panne.

Si l'on devait réaliser un algorithme pouvant simuler l'exemple vu ci-dessus, nous mettrons en place le système suivant : si le nombre de voisins est différent du nombre de ports actifs, cela signifie que l'on a détecté une erreur ou un dysfonctionnement dans le graphe. Chaque port voisin sera parcouru afin de déterminer le port qui n'est pas dans la liste des ports actifs du noeud et l'on détermine donc de cette manière l'identifiant du port défaillant du réseau. Une fois cette étape réalisée, il nous reste simplement plus qu'à réaliser un transfert d'information aux voisins adéquats, ici l'information transférée sera l'étiquette du noeud défaillant.

2.4 Conclusion

Ce TP nous a permis de découvrir une approche permettant la détection d'un dysfonctionnement, d'une perturbation, d'une panne, ou même d'une erreur dans un système distribué grâce à une synchronisation en LC2. De cette manière, il est maintenant possible de prévenir les éventuelles intempéries dans un système distribué.

Conclusion générale

Ces TPs nous ont permis d'avoir une approche plus concrète et plus approfondie des notions théoriques abordées en cours. Ces séances de TPs ont été très enrichissantes, sur l'appréhension de la notion de stabilité et de récupération d'informations suite à une panne dans un réseau distribué.

De manière générale, il s'agissait d'une première approche nous permettant d'une part de mieux comprendre et maîtriser les notions de base importantes de ViSiDiA, et d'une autre, de se familiariser à la gestion distribuée des ressources, et au parallélisme.

Annexe

2.5 Code source du TP1 : Algorithme de stabilisation

```
import visidia.simulation.SimulationConstants.PropertyStatus;
import visidia.simulation.process.algorithm.LC0_Algorithm;
import visidia.simulation.process.algorithm.LC1_Algorithm;
import visidia.simulation.process.edgestate.MarkedState;
import java.util.ArrayList;

public class Gestion extends LC1_Algorithm{
        // Description de la classe s'affichant sous ViSiDiA
        @Override
        public String getDescription(){
                return "Election dans un arbre";
        }
        // Fonction s'exécutant avant le commencement de l'algorithme sous ViSiDiA
        @Override
        protected void beforeStart(){
        // Propriété permettant l'affichage d'un état (N, F ou E) sous un sommet (vertex)
        setLocalProperty("label", vertex.getLabel());
        // Propriété premettant l'affichage de l'état (stable ou non) du point
        setLocalProperty("estStable", false);
        // Propriété premettant l'affichage de la distance à laquelle se situe le point
         ↪  stable
        setLocalProperty("distancePointStable", -1);
        // Propriété stockant l'ordre du graphe
        setLocalProperty("ordre", 1);
        // Affichage de l'état "estStable"
        putProperty("estStable", getLocalProperty("estStable"), PropertyStatus.DISPLAYED);
        // Affichage de la "distancePointStable"
        putProperty("distancePointStable", getLocalProperty("distancePointStable"),
         ↪  PropertyStatus.DISPLAYED);
        // Affichage de l' "ordre"
        putProperty("ordre", getLocalProperty("ordre"), PropertyStatus.DISPLAYED);
        }
        @Override
        protected void onStarCenter(){
        // Détermination du diamètre du graphe
        if((Boolean)getLocalProperty("estStable") == true){
                // Récupère le chemin du port 0
                int min_a = (Integer)getNeighborProperty(0, "distancePointStable");
                // Parcours des chemins jusqu'alors générés par les voisins, afin d'en
                 ↪  sélectionner le plus court
                for(int i=1; i < getArity(); i++){
                        // Si une distance inférieure à la distance courante est trouvée,
                         ↪  l'ancienne distance est remplacée par la courante
                        if((Integer)getNeighborProperty(i, "distancePointStable") < min_a){
                                // Mise à jour de la distance
                                min_a = (Integer)getNeighborProperty(i,
                                 ↪  "distancePointStable");
                }
            }
```

```
                int a = min_a + 1;

                // Empêche l'incrémentation de la valeur du chemin lorsque l'ordre est
                 ↪  atteint (au niveau de l'élu)
                if (getLocalProperty("label").equals("E")){
                        if ( a > (Integer)getLocalProperty("ordre")){
                                a = (Integer)getLocalProperty("ordre");
                        }
                }

            setLocalProperty("distancePointStable", a);
            // Affichage de la "distancePointStable"
            putProperty("distancePointStable", getLocalProperty("distancePointStable"),
             ↪  PropertyStatus.DISPLAYED);
        }

        // N---N -> F---N (si le centre a un seul voisin à "N")
        if (getLocalProperty("label").equals("N") && hasOnlyOneNeighborToN()){
            // Changement du label
            setLocalProperty("label", "F");
            // Mise à l'état "stable" du point
            setLocalProperty("estStable", true);
            // Initialisation de la distance
            setLocalProperty("distancePointStable", 0);
            // Incrémentation de l'ordre au niveau du point
            incrementOrder();
            // Affichage de l'état "estStable"
            putProperty("estStable", getLocalProperty("estStable"),
             ↪  PropertyStatus.DISPLAYED);
            // Affichage de la "distancePointStable"
            putProperty("distancePointStable", getLocalProperty("distancePointStable"),
             ↪  PropertyStatus.DISPLAYED);
            // Affichage de l' "ordre"
            putProperty("ordre", getLocalProperty("ordre"), PropertyStatus.DISPLAYED);
        }
        // F---N---F -> F---E---F ("N" élu en "E" lorsque tous les autres points du graphe
         ↪  sont à l'état "F")
        else if (getLocalProperty("label").equals("N") && hasAllNeighborToF()){
            // Changement du label
            setLocalProperty("label", "E");
            // Mise à l'état "stable" du point
            setLocalProperty("estStable", true);
            // Initialisation de la distance
            setLocalProperty("distancePointStable", 0);
            // Incrémentation de l'ordre au niveau du point
            incrementOrder();
            // Affichage de l'état "estStable"
            putProperty("estStable", getLocalProperty("estStable"),
             ↪  PropertyStatus.DISPLAYED);
            // Affichage de la "distancePointStable"
            putProperty("distancePointStable", getLocalProperty("distancePointStable"),
             ↪  PropertyStatus.DISPLAYED);
            // Affichage de l' "ordre"
            putProperty("ordre", getLocalProperty("ordre"), PropertyStatus.DISPLAYED);
        }
}
```

```
// Clonage de la classe Election au niveau de chaque noeud
@Override
public Object clone(){
        return new Gestion();
}

// Test indiquant si un seul voisin est à N
public boolean hasOnlyOneNeighborToN(){
        // Compteur de voisins "N"
        int compteurDeN = 0;
        // Parcours des voisins du point central
        for(int i = 0; i < getArity(); i++){
                // Incrémentation du compteur si un voisin a l'état "N"
                if( getNeighborProperty(i, "label").equals("N")){
                        compteurDeN++;
                }
        }
        // Un seul voisin est à l'état "N"
        if(compteurDeN == 1)
                return true;
        // Aucun ou plus d'un voisin est à l'état "N"
        else
                return false;
}

// Test indiquant si tous les voisins sont à F
public boolean hasAllNeighborToF() {
        // Compteur de voisins "F"
        int compteurDeF = 0;
        // Parcours des voisins du point central
        for(int i = 0; i <  getArity(); i++){
                // Incrémentation du compteur si un voisin a l'état "F" (en récupérant
                 ↪  l'état via "label")
                if( getNeighborProperty(i, "label").equals("F")){
                        compteurDeF++;
                }
        }
        // Tous les voisins sont à l'état "F"
        if(compteurDeF == getArity()){
                return true;
        }
        // Tous les voisins ne sont pas à "F"
        else{
                return false;
        }
}

public void incrementOrder(){
        for(int i=0; i < getArity(); i++){
                if(getNeighborProperty(i,"label") == "F"){
                        setLocalProperty("ordre", (Integer)getLocalProperty("ordre") +
                         ↪  (Integer)getNeighborProperty(i,"ordre"));
                }
        }
    }
}
```