Université de Franche-Comté
Département Informatique

Master 2ère année
Module AD

COMPTE RENDU DU PROJET :
« Gestion de l'accès à un tableau blanc »

BINOME		
Nom	**Prénom**	**Trigramme**
CHARLOT	Edouard	CE
NTUMBA WA NTUMBA	Patient	NTP

Année Universitaire 2015 - 2016

Table des matières

Introduction

Dans le cadre du module d'Algorithmique Distribuée, il nous est demandé de réaliser un projet d'accès à un tableau blanc distribué entre plusieurs processus reliés par un réseau quelconque. Les processus pouvant dessiner concurammnet sur le tableau, mais les actions de ces derniers doivent être synchronisées; afin d'assurer l'ordre d'affichage des actions effectuées sur le tableau blanc.
En effet si nous n'instaurons pas d'accès concurrent et que chaque processus peut ecrire sur le tableau à sa guise, il est possible que des processus ecrivent sur le tableau en même temps. Dans ce cas l'ordre des données n'est plus assuré et chaque processus ne verra pas la même chose sur son tableau. Ce problème existe dans de nombreuses situations et peut corrompre les données.

Pour atteindre les objectifs, il nous a été demandé d'utiliser les algorithmes distribués d'exclusion mutuelle étudiés en cours ci-dessous :

- ➔ **L'algorithme de Lelann** repose sur une communication en anneau entre les processus. Les processus se passent tour à tour un jeton qui leur donne le droit d'accéder à la ressource critique, qui ici est le tableau blanc.

- ➔ **L'algorithme de Ricart-Agrawala** permet d'ordonner les messages échangés en utilisant l'horloge de Lamport, ce qui permet que chaque processus puisse ordonner chaque message reçu en fonction de la date d'envoi et non de réception (qui peut etre différente pour chaque processus).

- ➔ **L'algorithme de Naimi-Tréhel** repose également sur un système de jeton, mais ici les processus désirant la section critique demandent l'autorisation au processus qui détient le jeton. Il devra ensuite attendre la réponse de celui-là avant de prendre la section critique.

Hors mis le fait que chaque algorithme a son propre mode de fonctionnement, il s'execute aussi sur une topologie de réseau bien spécifique, c'est à dire que l'algorithme de Lelan est adapté pour une topologie en anneau, celui de Ricart-Agrawala fonctionne sur un reseau complet et l'algorithme de Naimi-Tréhel est adapté pour une topologie en arbre.

Le projet devant se réaliser sur un réseau quelconque, nous devons donc également adapter les algorithmes au cas par cas afin qu'ils fonctionnent sur ces réseaux. Pour se faire, nous avons mis en place **une table de routage** qui permettra à chaque processus de ce réseau de pouvoir localiser tous les autres processus. De ce fait, nous pourrons donc créer des structures virtuelles pour faire fonctionner les différents algorithmes correctement.

La mise en oeuvre de ce projet est effectué sur le simulateur ViSidia, l'implémentation des algorithmes et de la logique de routage est réalisée en Java.

1) Mise en oeuvre

1.1) La table de Routage

➔ **Principe**

La table de routage est un tableau de taille égale au nombre des processus qui constituent ce réseau. Elle fait la correspondance entre le numéro d'un processus et la porte par laquelle le proprietaire de la table de routage est connecté à l'autre processus. **ROUTAGE[numero_processus] = numero_porte**, nous utilisons aussi une sentinelle nommée **COMPLETE** qui s'incrémente à chaque fois que la table de routage est mise à jour.

La construction de cette table commence premièrement par l'envoi d'un message **ROUTE** d'un processus à tous ses voisins, ensuite il attend de recevoir tous les messages **ROUTE** de ses voisins. A la reception de chacun de ces messages, il met à jour sa table de routage et incrémente la sentinelle.

Tant que la sentinelle **COMPLETE** n'est pas égale au nombre des processus du réseau, chaque processus envoi à tous ses voisins sa table de routage, (message **TABLE)**, et attend à son tour de recevoir les tables de routage de ses voisins,

A la reception de la table de routage de ses voisins, le processus ajoute les routes que son voisin connais tout en indiquant que pour atteindre les processus que son voisins connait, il faut passer par la porte de ce dernier **ROUTAGE[numero_processus] = numero_porte_voisin**, et sans oublier on incrémente la sentinelle dès que le processus ajoute une valeur dans la table de routage.

Valeurs possibles de la table de routage à une position 'i':

- -1 : valeur par défaut, ça signifie qu'on a pas encore trouvé la porte pour atteindre le processus 'i'.
- -2 : Cela signifie que la case correspond au processus en cours.
- valeur positive : Le numéro de la porte associé au processus 'i'.

➔ **Règles**

Initialisation :

ROUTAGE[i] ← -1
ROUTAGE[mon_id] ← -2
COMPLETE ← 1

Regles 1 : Processus p envoi ROUTE

Pour tout q voisin de p
envoi <ROUTE> à q
Fin pour tout
Attendre <#ROUTE = #Voisin de p>

Regles 2 : Processus p reçoit ROUTE de q
ROUTAGE[q] = porte_de_q
COMPLETE ← COMPLETE + 1

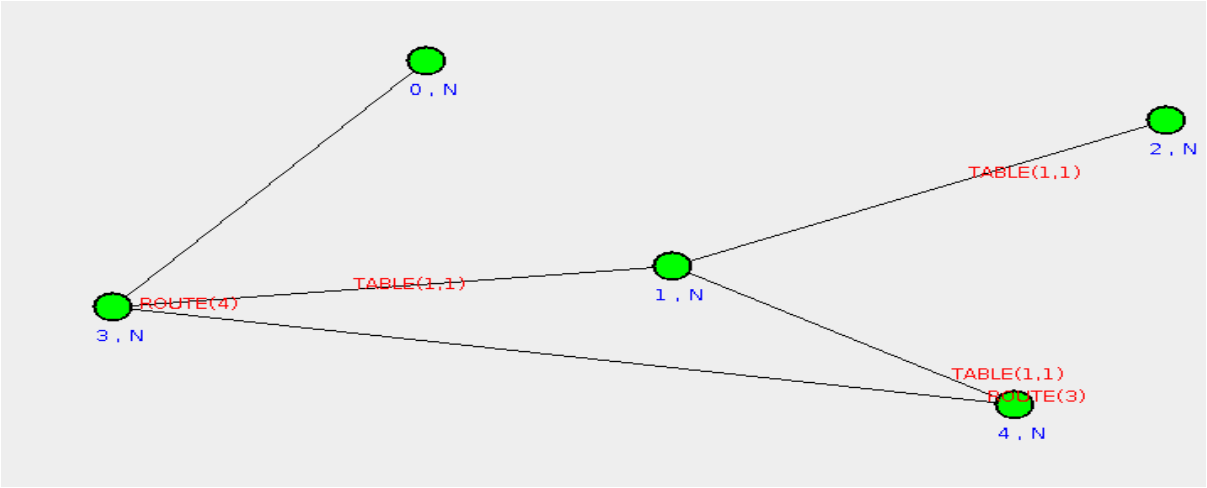
Regles 3 : Processus p envoi TABLE
Pour tout q voisin de p
envoi <TABLE> à q
Fin pour tout
Attendre <COMPLETE = #processus du reseau>

Regles 4 : Processus p reçoit TABLE de q
Pour tout i < #TABLE faire
si TABLE[i] > -1 ET ROUTAGE[i] = -1 alors
ROUTAGE[i] ← porte_de_q
COMPLETE ← COMPLETE + 1
finSi
Fin faire

Regles 5 : Si le table de routage est plein
Pour tout q processus du réseau
envoi <READY> à q
Fin pour tout
Attendre <#READY = COMPLETE>

➔ **Implémentation**

La table de routage est une classe qui contient un objet de type tableau sur lequel nous effectuons les mises à jour de la table de routage, nous utilisons aussi un entier pour assurer le mécanisme de la sentinelle. Outre celà nous avons aussi implémenté un getter et un setter de l'objet tableau pour y acceder. L'implémentation du processus de routage est réalisé grace Thread Java, les mécanismes de **attendre** sont gérés par le MUTEX java wait/notify. Le Thread s'arrete dès que le processus fini de remplir la table de routage. L'image ci-dessous présente le deroulement du processus de routage.



➔ **Avantage du principe**

Nous limitons le nombre des message lors du processus de routage pour les processus isolé c'est à dire le processus qui est connecté qu'à un seul processus. Ce processus n'envoi que le message ROUTE à son voisin et pendant toute la durée du routage il n'attend que les massages TABLE provenant de l'unique processus au quel il est connecté. Nous pouvons encore améliorer cette version en utilisant directement la table de routage de ce processus isolé que tous les processus du réseau sont atteignable qu'à partir de la porte de l'unique processus au quel il est connecté.

➔ **Difficulté**

La plus grande difficulté est d'arriver à bien générer quand est-ce que le processus arrete le thread de routage, nous arretons souvent brutalement le thread.

1.2) Générateur des formes

En ayant le tableau blanc fourni dans l'énoncé du projet, de notre part nous devrons mettre en place un mecanisme de génération des formes aléatoires, que chaque processus va devoir appeler pour avoir de quoi dessiner sur le tableau blanc.

Le générateur des formes est une classe Java qui nous genère le type de forme (point, ligne, rectangle, ellipse) et un premier point (abscise & ordonnée). Dans le cas où la forme requiert un deuxième point, donc si la forme est une ellipse un rectangle ou une ligne, nous générons un deuxième point en fonction du premier point, ce qui nous permet de ne pas obtenir de trop grosses structures. C'est en utilisant les éléments générés que le processus construit la forme à dessiner et la distribue grâce à un message de type **FORME** pour communiquer aux autre processus de son réseau la forme qu'il a dessiné. Ce message contient un objet de type forme contenant les primitives de la forme afin que chaque processus puisse retracer cette fome.

1.3) Intégration

L'intégration de la table de routage, le généteur de formes, le tableau blanc et l'Algorithme a été réalisé comme suit :

- ➔ Le processus commence par lancer le Thread de routage pour remplir sa table de routage;
- ➔ Dès que la table de routage est remplie, le processus lance concurremment d'une part le thread de réception des messages qui implémente les règles de l'algorithme à executer et d'autres parts il lance le thread qui gère la ressource partagée : le Tableau Blanc.
- ➔ Ensuite devons executer l'un des algorithmes implementés. Lorsqu'un processus veut dessiner une forme sur le tableau blanc et qu'il a accès à la section critique, il fait appel au générateur des formes et crée un message de type FORME qu'il enverra à tous les autres processus du réseau.

D'une manière plus detaillé nous allons présenter dans les paragraphes qui suit l'intégration pour chaque algorithme mis en place.

a. Intégration avec l'Algorithme de Lelan

L'algorithme de Lelan est basé sur un réseau en forme d'anneau avec un jeton qui circule d'un processus vers le suivant pendant toute la vie de l'algorithme.
Pour un réseau quelconque, grace à la table de routage nous pouvons définir un anneau virtuel dans un ordre ascendant, Pour un processus p, le prochain processus qui va recevoir le jeton sera p+1 et il sera facile de savoir à quelle porte envoyé le jeton en utilisant la table de routage.

Le message **FORME** transite aussi sur le réseau en suivant l'anneau virtuel, le message forme n'est pas lié au message jeton **TOKEN**, pour garantir une meilleur réactivité et empecher que le message FORME ne reste en attente avec le jeton pendant toute la durée d'utilisation de la section critique d'un processus. En effet avec un message **FORME** séparé du message **TOKEN** les deux peuvent se propager indépendament.

b. Intégration avec l'Algorithme de Ricart-Agrawalla

Dans cet algorithme le principe est sur un réseau complet, grace encore à la table de routage chaque processus peut atteindre tous les processus du réseau, nous pouvons donc simuler un réseau complet très facilement.

Pour implementer cet algorithme nous utilisons les message de type **REQ** pour démander la section critique avec l'horloge de Lamport et le message **REL** pour autoriser un processus P à acceder à la section critique. il y a aussi les message de type **FORME** que l'algorithme utilise pour envoyer sa forme générée à tous les intervenants dans le réseau.

La difficulté pour cet Algorithme est de savoir quand est-ce tous les processus sont pour pouvoir simuler l'accès à la section critique. Malgré le fait que nous utilisons le Mutex Java, il arrive à tout moment que la simulation ne commence pas car il y a croisement des messages, tel qu'un processus P ayant fini le routage fait une démande de section critique, pendant qu'un autre processus Q n'ayant pas encore fini à construire sa table de routage reçoit une message REQ au lieux d'un message TABLE, il genère une exception et ne sait pas avancer.

Cette difficulté provient d'une part du choix de la simulation, comme nous faisons une simulation automatique avec des endormissement des processus pend 5 sécondes en moyenne, facilement nous rencontrons ce problème. Une simulation manuelle (au click) sur le la ressource partagée ne génère pas cette problémantique.

Pour resoudre cette difficulté nous nous sommes assuré de bien gérer la reception des messages en faisant un thread qui gère les messages des règles de l'algorithme et doit aussi integrer la gestion des messages de routage.

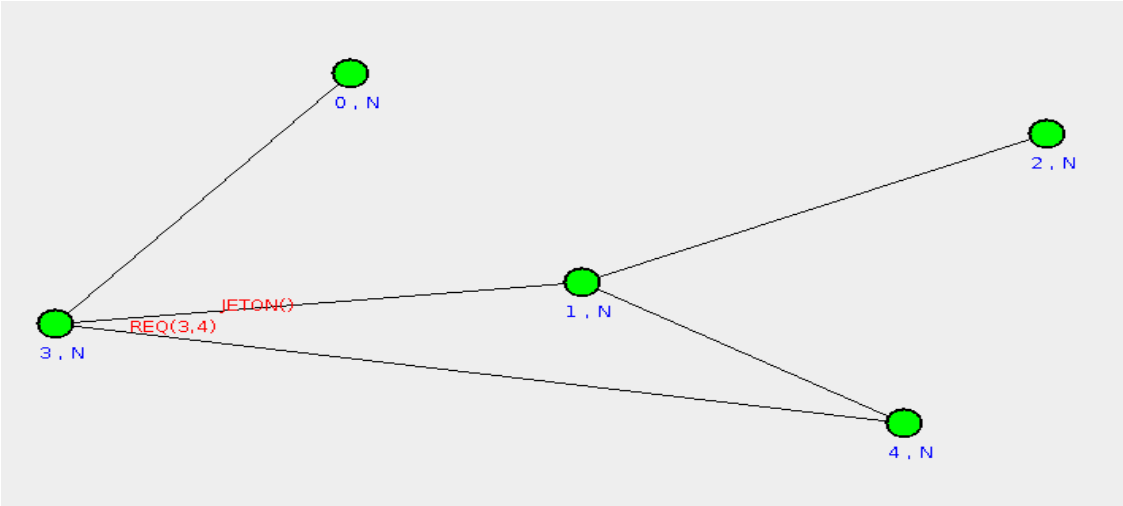
c. Intégration avec l'Algorithme de Naimi-Tréhel

L'Algorithme de Naimi-Tréhel fonctionne sur un réseau en forme d'arbre, encore une fois grâce à la mise en place de la table de routage nous pouvons créer une structure virtuelle adaptée.

Les messages échangés dans cet algorithme c'est le message de type **REQ**, pour la démande de la section critique, et le

message de type **TOKEN** pour autoriser l'accès à la section critique. Une fois qu'un processus peut acceder à la section critique il va générer une forme construit le message de type **FORME** et envoi sa formes aux autres processus.

La faiblesse que rencontre cette algorithme est la mise à jour du changement du proprietaire du jeton dans le réseau n'est pas en diffusion, celà entraine des fois une famine pour certain processus mais avec l'évolution et sans intervention humaine les processus en famine arrive à acceder à la section critique. Du coup nous pouvons dire que cet algorithme n'assure pas l'équité à tout moment. Ci-dessous une capture du déroulement de l'algorithme.



2) Résultats et Interprétation

D'après nos experiences, sur des réseaux complexes l'algorithme de Lelan est le plus performant, en effet c'est la structure d'anneau permet de garantir que chaque processus ait accès à la section critique chacun son tour et de ce fait on évite les famines.
Dans une situation où certains processus seraient parfois spéctateurs et parfois créateur l'algorithme de Ricart-Agrawalla serait plus adapté car la section critique n'est accordé qu'aux personnes la demandant. Si un processus veut rester spectateur, il ne la demande pas. Si au contraire un processus veut devenir producteur de données, il peut demander l'autorisation et attendre celle-ci.

Dans notre cas, nous voulons quelque-chose de réactif, en effet nous voulons éviter qu'une forme mette trop de temps à se propager au sein du réseau, nous voulons que la forme soit arrivée avant qu'un processus ne commence à déssiner sa propre forme.

Conclusion

Afin qu'un réseau distribué fonctionne correctement il faut trouver un moyen de synchronisation entre les différents processus de ce réseau. En l'absence de synchronisation l'intégrité des données n'est pas garanti et il est possible que certains processus aient des données corrompues.

En créant une table de routage il est possible de créer des structures virtuelles afin d'adapter les différents algorithmes de partage de section critique sur des réseaux quelconques.

Dans ce projet nous avons testé trois algorithmes différents avec des structures différentes. Chacun de ces algorithmes ont leur point fort et leur faiblesses, il en est au développeur de faire le choix d'implémentation en fonction des besoins et contraintes du système cible.